A METHODOLOGY FOR MICRO-POLICIES

Arthur Azevedo de Amorim

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2017

Para Susan, que sempre esteve lá comigo.

Acknowledgments

Many people provided encouragement, inspiration, and friendship during my time as a graduate student. It is only fair that I save some room for them here.

I thank my family for their incredible support; in particular, my parents Inês and Sebastião, my brothers, Bernardo and Pedro, and my grandfather, Alberto, who showed me the first diagonal argument that I have seen.

Penn has been a great place to be a student, and one where I met extraordinary people. Thanks to Justin Hsu, Rado Ivanov, Rasul Tutunov, and Emilio Jesús Gallego Arias for having been such wonderful roommates—and especially to Justin, who did not mind cold calling all the then just-arrived graduate students while looking for someone to share an apartment. To Steve Zdancewic and Stephanie Weirich, for opening their house and bringing everyone together on so many occasions. To Jean Gallier, for giving me an excuse to speak French, and to Val Breazu-Tannen and Mika Tsekoura, for enduring my Greek. To Lili Dworkin and to Laurel Emurian Mirarchi, for their matchmaking skills. To Nikos Vasilakis, for being an excellent travel guide. To Nicu Stiurca, for taking me to capoeira. To Maxime Dénès, for understanding Jacques Brel. To Shahin Jabbari, for early breakfast. To Delphine Demange, for knowing where to find good music in the basements of West Philadelphia. To Marco Gaboardi, for sharing his sake. To Christine Rizkallah, for karaoke. To André DeHon and Mayur Naik, for joining my thesis committee. To many others with whom I have crossed paths: Jocelyn Quaintance, João Sedoc, Venetia Pliatsika, Xiaocheng Huang, Antal Spector-Zabusky, Leonidas Lampropoulos, Jennifer Paykin, Robert Rand, Dmitri Garbuzov, Ben Karel, Joachim Breitner, Nadia Henninger, Hongbo Zhang, Richard Eisenberg, Benoît Montagu, Benoît Valiron, Peter-Michael Osera, Daniel Wagner, Michael Greenberg, Brent Yorgey, Vilhelm Sjöberg, Chris Casinghino, Loris D'Antoni, Jianzhou Zhao, Jamie Morgenstern, Rachel Cummings, Steven Wu, Matthew Weaver, Antoine Voizard, Kenny Foner, William Mansky, Daniel Winograd-

Cort, Li-yao Xia, Hengchu Zhang, Caleb Stanford, Nick Roessler, Alex Roederer, and Andrew King. And to Tim Griffin, Siome Klein Goldenstein, Davi Barbosa, Benjamin Werner, and Gilles Dowek, for having a crucial role in my decision to come to Penn.

Cătălin Hrițcu has been a good friend and an important collaborator. Thanks to him, I got involved in interesting projects and spent six months as an research intern at Inria, which led to a large portion of this thesis (and provided the perfect excuse for me to go to Paris). Thanks to Nick Giannarakis and Zoe Paraskevopoulou, who made our office especially fun, and to many other collaborators that I had the pleasure of working with during these years: Shin-ya Katsumata, Randy Pollack, David Pichardie, Yannis Juglaret, and Andrew Tolmach.

And finally, I thank my advisor Benjamin Pierce, who taught me the value of putting what is important at the end.

ABSTRACT

A METHODOLOGY FOR MICRO-POLICIES

Arthur Azevedo de Amorim

Benjamin C. Pierce

This thesis proposes a formal methodology for defining, specifying, and reasoning about *micro-policies*—security policies based on fine-grained tagging that include forms of access control, memory safety, compartmentalization, and information-flow control. Our methodology is based on a *symbolic machine* that extends a conventional RISC-like architecture with tags. Tags express security properties of parts of the program state ("this is an instruction," "this is secret," etc.), and are checked and propagated on every instruction according to flexible user-supplied rules. We apply this methodology to two widely studied policies, information-flow control and heap memory safety, implementing them with the symbolic machine and formally characterizing their security guarantees: for information-flow control, we prove a classic notion of termination-insensitive noninterference; for memory safety, a novel property that protects memory regions that a program cannot validly reach through the pointers it possesses—which, we believe, provides a useful criterion for evaluating and comparing different flavors of memory safety. We show how the symbolic machine can be realized with a more practical processor design, where a software monitor takes advantage of a hardware cache to speed up its execution while protecting itself from potentially malicious user-level code. Our development has been formalized and verified in the Coq proof assistant, attesting that our methodology can provide rigorous security guarantees.

Contents

List of Figures

Chapter 1

Introduction

Our society has a pressing need for stronger computer security. Large, interconnected computer systems are critical to our infrastructure (databases of medical records, mass transit networks, voting systems, nuclear power stations, …), yet infested with vulnerabilities that could be exploited by malicious agents. The general feeling seems to be that computer security is a hopeless pursuit, as nicely illustrated by this New York Times report on a recent wave of cyberattacks.

> But like any software, Scada [supervisory control and data acquisition] systems are susceptible to hacking and computer viruses. And for years, security specialists have warned that hackers could use remote access to these systems to cause physical destruction. [87]

Fortunately, reality is not as bad as it seems. There are many avenues for improving computer security, which involve solving basic issues that affect a wide variety of programs—such as the lack of memory safety in C and related languages—but also developing tools for enforcing application-specific policies. Among general solutions, reference monitors [3] are a popular option. Their job is to inspect the execution of a program, mediating and checking a range of operations to ensure that they conform to a policy of interest. If a policy violation is detected, the monitor halts execution or signals an error. Many computer systems execute under some form of monitoring, at different levels of granularity. For example, operating systems use access control to prevent a user from tampering with someone else's files, either inadvertently or maliciously. And managed languages like Java, Python, JavaScript, OCaml, and Haskell check the bounds of array accesses to prevent memory corruption.

In principle, reference monitors can prevent multiple vulnerabilities [95]—including many of those that haunt programmers today [1, 83, 84]. Unfortunately, they are often not adopted in practice because of their *cost*. Security is an important software requirement, but so are performance, flexibility, maintainability, and compatibility with legacy systems. If a security policy is significantly at odds with any of those requirements, it might be abandoned or replaced with an approximation that is still vulnerable to attack [23, 37, 44]. Buffer overflows are a good illustration of this issue. Though they can be prevented with simple solutions that have existed for decades, such as managed languages with bounds checking and garbage collection, they continue to pose a serious threat [77], because techniques for comprehensive safety are unsuitable for many memory-intensive applications.

An attractive option to make monitors more practical is the use of hardware support, especially in the era of cheap transistors we live in today. Besides significant performance improvements, specialized hardware could bring other benefits as well. For example, if the monitor is not tied to a particular programming language, it can more easily apply to legacy programs: they might need to be adapted to conform to the more restrictive execution model, but at least will not have to be rewritten from scratch. Many monitoring mechanisms have been integrated with hardware over the years, with virtual memory and page protection likely being the most widespread examples.

Though an interesting idea, implementing monitors with specialized hardware should be considered with care, because it incurs in development and deployment costs that are much higher than a pure software solution. As old features become obsolete and new ones are added, hardware manufacturers have strong incentives to maintain bloated, redundant designs, greatly exacerbating backwards compatibility issues. Page-based virtual memory, for example, was introduced in a time of scarce resources, where disk swapping was an important feature; today, it induces great performance penalties for applications that have widely different memory-access patterns [15]. The issue is particularly acute when considering security features, since their design involves trade offs between security, performance, and other factors that are fundamentally hard to evaluate in advance. For example, some hardware extensions provide support for spatial memory safety [30, 86], which is capable of preventing several popular, well-understood exploits, and forces attackers to find other techniques for subverting systems. It is not clear, however, how much this restricts attacker power effectively. Does the number of vulnerabilities that need to be patched decrease for systems that use these features? Or could attackers just as easily resort to double frees and other temporal safety violations to obtain the same end results?

We can avoid some of the risk associated with new hardware by focusing on generic, programmable mechanisms. Instead of committing in advance to a single, hardwired policy, the hardware should make it possible to experiment with different policy designs and their trade offs. If we find out that a policy was not strong enough, or perhaps too slow, not a problem: the hardware does not have to change, we just have to reprogram the policy. One possibility would be a mechanism for programming general reference monitors, but this might be *too* general to be made efficient: reference monitors can in principle be implemented with arbitrary programs, making it hard to imagine what kind of support we could provide that does not exist in processors already.

A clue for reconciling performance and flexibility is to observe that many monitors of interest follow a well-behaved pattern: they protect programs by pervasively checking and propagating metadata associated with their state. Typical Unix file systems let users impose rudimentary access control policies on their files by marking them with permission bits. Protection masks present in modern processors allow them to distinguish between writable and executable memory, thereby preventing a variety of devastating code-injection attacks. Systems for spatial memory safety enriching pointers with bounds information to detect invalid memory accesses [30, 73]. And systems for expressive information-flow control track the provenance of intermediate results in a computation to prevent public outputs to depend on secrets, directly or indirectly [6, 9, 10, 101]. The generality of this metaphor led researchers to explore efficient hardware support for programmable metadata checking and propagation. Extensions such as FlexiTaint [110], Harmoni [28], and the PUMP (*Programmable Unit for Metadata Processing*) [33, 34] augment the state of the program by attaching metadata tags to memory and registers, at the granularity of individual words. As the program runs, tags attached to the various operands and destinations are processed according to user-defined rules. Not only are these systems capable of encoding many policies of interest that have been studied in the literature, they also do so efficiently, with performance overheads running between 1 and 20% for typical policies and programs [28, 33, 110].

These preliminary experiments with hardware support for programmable tags are encouraging, and they suggest that it would be interesting to investigate tag-based policies from a more theoretical angle. As it is the case for other enforcement techniques, like general reference monitors or program rewriting, such a theory could help us understand the limits of what can be enforced using tags [14, 51, 95], point at new implementation strategies [36], or simply inform the use of these tools by security engineers. To develop such a theory of tag-based policies, it would help to first have a rigorous definition of what these policies *are*. The aim of this thesis is to answer this question by developing a programming model for tag-based policies—or, as we call them here, *micro-policies*.

Engaging in this exercise brings up fundamentally conflicting goals. Naturally, we would like a definition that is general enough to accommodate a wide variety of policies that intuitively fit this pattern, but also at an appropriate level of abstraction from the hardware mechanisms that have been proposed to support them. As usual, to facilitate reasoning about micro-policies at a high-level—and, in particular, about their security guarantees—we would like a programming model that hides a fair share of tedious implementation details, such as how tags are encoded, how they are checked, and where they are stored in the hardware. Given that hardware support for programmable tags is still in its infancy anyway, it would not make sense to overly commit to any particular existing mechanism, and keeping the model more abstract gives hardware freedom to evolve. Nevertheless, we would like the model to be somewhat connected to the hardware proposed in the literature, so that we can have a rough idea of performance when developing policies—we should not forget that efficiency was one of the main motivations for investigating tag-based policies in the first place.

Evaluating whether a programming model meets these goals is challenging, and the issues of convenience of reasoning and security guarantees are especially problematic. While there are simple criteria for judging performance and other aspects of a policy, there isn't anything quite as satisfying for security. This gap manifests itself when we try to weigh the trade offs associated with variations of a security policy—for example, different flavors of memory safety. It is not too hard to measure if one policy runs faster than another one, but how do we know if it is more secure? We can test how many known vulnerabilities each one is capable of preventing, but, as argued earlier, this provides little insight into protection guarantees against other attack techniques that might be tried in the future.

One idea to complement our assessment of security guarantees is to establish reasoning principles that we can apply to a program when it is running with a monitor turned on—for instance, we might want to reason about what parts of the system state can have an influence on the outputs observed by a certain user. We could proceed as with most programs, by careful inspection of the policy source code, or perhaps by proving rigorous mathematical results about it on paper. For most functional correctness properties, this approach is adequate, provided that it is backed by extensive testing. Though informal, pencil-and-paper reasoning about programs often turns out to be wrong, we can hope that the conditions under which the program is tested are not so different from when it ships, implying that the rate of defect would be similar on both scenarios. For security guarantees, however, this assumption simply cannot hold: if we reason incorrectly, there is a much higher chance that it will lead to a vulnerability in production, because it is hard to emulate in a development environment the threats that a program will face once deployed. Most bugs are the product of chance, but attacks result from premeditated, careful, and malicious analysis.

To address these shortcomings, we have been drawn to a new set of tools [72, 83]: *proof assistants*. Proof assistants are programs that allow users to write down precise mathematical definitions, specifications about them, and proofs that those specifications are met. The program checks every step in these proofs to ensure it is logically sound, giving us high confidence that the end result is correct. Recent years have seen a surge of maturity in this technology: proof assistants such as Coq [107], Isabelle, and ACL2 are being applied to progressively ambitious efforts, ranging from deep mathematical results, such as the proofs of the four-color theorem [45], the odd-order theorem [46], the Kepler conjecture [50], etc., to complex, realistic software, such as the CompCert C compiler [74], the seL4 micro-kernel [68], the Verasco program analyzer [62], and part of the LLVM infrastructure [117, 118], among others. Though verification with a proof assistant is fairly expensive, these programs exhibit a few characteristics that make this approach attractive: they are relatively easy to specify formally (compared to interactive programs like a video editor, for example), and their defects can have disastrous consequences for a large number of systems. And indeed, the effort pays off. The effectiveness of proof assistants is attested not only by pure reason, but also by solid empirical evidence. In a recent, thorough evaluation of bugs in C compilers by random testing, Yang et al. noted:

> The striking thing about our CompCert results is that the middleend bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users. [114]

Micro-policies share all of the aforementioned virtues, making them a good candidate for formal verification. By developing our programming model for micro-policies with a proof assistant, we would obtain a good way of judging its programming and security capabilities. Since proof assistants aim for high assurance, they tend to be rather constrained environments, subjecting definitions and proofs to draconian consistency checks. If we can argue about security in such a formal setting, we have good evidence for the usability of our model; furthermore, others could then use our development to build their own policies and reason about them.

Contributions In the remainder of this document, we develop a formal methodology for defining, specifying, and reasoning about micro-policies. In sum, we

- present a formal model for defining micro-policies;
- use the model for expressing policies that have received great attention in the literature, studying their security properties; and
- show how the model can be supported by a lower-level hardware model.

The technical results of this document have all been formalized in the Coq proof assistant; the formal development is available at `https://github.com/micro-policies/micro-policies-coq`. We have decided to work with the Coq proof assistant, which combines an expressive theory, with a mature code base, decent tool support, and active user community. We believe that other platforms (such as Isabelle, for example), would have been just as appropriate. As it is typical for many results of this kind, these proofs are challenging because of the large number of cases that needs to be handled, but tend to be somewhat tedious. Thus, we omit the proofs of most technical results in the thesis, referring the curious reader to our formalization instead.

The cornerstone of our methodology is the *symbolic machine* introduced in Chapter 2. The symbolic machine provides a programming model for both user programs and the micro-policies that monitor them. To user programs, the symbolic machine is a simple, conventional RISC processor. Programs run at a fairly low-level, by executing individual instructions that manipulate values stored in registers and memory. Micro-policies, on the other hand, are richer. They operate on metadata tags attached to every machine word in the program state, including the program counter. On each cycle, the policy decides whether the user code is allowed to execute by analyzing the tags on the operands on the program counter, current instruction, and each of that instruction's operands. Crucially, this analysis is not limited to a handful of operations: any function defined in Coq can be used as a micro-policy. Coq has a fairly expressive functional programming language that is particularly well suited for writing micro-policies. This design has two benefits. First is that it frees policy designers from low-level implementation details, allowing them to focus on high-level policy behavior. Second, it still makes it possible to discuss very low-level properties of the code that the policy protects, making it well-suited for applications of tagged monitors described in the literature.

After describing the basis of our methodology, we proceed to apply it to two policy use cases that have received attention in the literature: a policy for *information-flow control* (Chapter 3), which protects programs against certain breaches of confidentiality, and a policy for *heap memory safety* (Chapter 4), which prevents out-of-bound accesses and uses of a pointer after it has been freed. The policies demonstrate a wide range of features available for the model; for example, the information-flow policy relies on the ability to define policy-specific privileged services to implement a protected call stack, used to prevent user programs from revealing secret information through their control flow. We evaluate the security guarantees of both policies by proving formal theorems about their behavior. First, we define higher-level abstract machines where both policies are built in and show that the symbolic machine running each micro-policy accurately implements the corresponding abstract machine, in the sense of refining its behavior. These abstract machines provide a simpler, more self-contained description of each policy that does not rely on the tag-propagation model of the symbolic machine. We then use these refinement results as stepping stones to derive more end-to-end results about the guarantees of the policies. For the information-flow policy, we prove a standard termination-insensitive noninterference result [42], which guarantees that secret information stored in the machine state has no influence on its public observable behaviors, with the possible exception of termination and timing. For the memory-safety policy, we devise a new correctness criterion based on pointer reachability. Roughly, the criterion says that pointers in a memory-safe setting behave as capabilities, in the sense that they grant access to clearly defined parts of the program state. We prove another security property reminiscent of noninterference, which guarantees that a program's execution cannot affect or be affected by memory regions that it cannot reach via its pointers. To our knowledge, this is the first proposal of a correctness criterion for memory safety that tries to explore extensional (that is, related to program behavior) consequences of preventing errors such as buffer overflows. We further analyze the possible consequences that various pragmatically motivated relaxations of the memory-safety policy—for example, allowing pointer-to-integer casts—might have on these guarantees. We also show how the guarantees of memory safety apply to a simple imperative language with memory-safety checks, and use this setting to relate the guarantees of memory safety to the local reasoning principles of separation logic [90], a well-established framework for reasoning about heap-manipulating programs.

Finally, Chapter 5, we show how the symbolic machine can be implemented by a lower-level concrete machine. This concrete machine is meant to model more realistic hardware mechanisms that have been proposed in the literature for enforcing tag-based policies—specifically, the PUMP extension of Dhawan et al. [33], one of the most general of the proposed designs. Given that production-grade processors with built-in support for micro-policies are still underway [24], the concrete machine does not attempt to provide a faithful, detailed model of a hardware architecture; rather, it tries to capture implementation strategies that could be used to build such a system—specifically, combining software implementations of the monitors with a hardware cache for avoiding having to execute the monitor on every instruction. We prove a theorem saying that the concrete machine refines the symbolic one, assuming that the machine-code implementation of the micro-policy is correct. In particular, this guarantees that the mechanisms of the concrete machine are capable of isolating the monitor implementation from user-level code, thereby preserving its integrity.

Chapter 6 concludes and points to promising directions for future work. Appendix A briefly reviews basic mathematical concepts evoked in the text, and sets up notation.

The work reported here is mostly extracted from three conference papers written during my Ph.D. studies [8, 9, 11], one of which was subsequently extended and published in a journal [10]. This material is based upon work supported by the DARPA CRASH program through the United States Air Force Research Laboratory (AFRL) under Contract No. FA8650-10-C-7090. (The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.) This work has also been supported by the supported by National Science Foundation award 1513854, *Micro-Policies: A Framework for Tag-Based Security Monitors*.

Chapter 2

Symbolic Machine

The methodology we propose for specifying and verifying micro-policies is based on a simple abstract machine, here dubbed the *symbolic machine*. This chapter describes the symbolic machine, and gives some basic examples of micro-policies we can define with it.

In essence, the symbolic machine blends in two programming substrates. The bottom-most layer resembles the programming interface of typical RISC processors, such as an ARM or PowerPC, and is meant for running user-level applications. On each cycle, the machine looks up an instruction and manipulates values stored in the memory and in the register bank. Above this layer, the machine runs a monitor with a much more flexible execution model. Instead of operating on machine words, the monitor operates on structured tags that describe the data manipulated by the user-level program. Rather than analyzing these tags with assembly code, the other program is free to apply arbitrary mathematical functions on them. Crucially, both tags and the functions that analyze them are not fixed in advance, but determined by a policy designer for enforcing a certain security policy.

2.1 Tags and Machine State

All words stored in the state of the symbolic machine—including in its memory, registers, or even its program counter—have individual *metadata tags* attached to them.

Definition 2.1. The *tag parameters* of the symbolic machine are given by three sets Tag_{pc}, Tag_{r}, and Tag_{m}, used to tag the contents of the program counter (PC), registers, and memory. We use Tag to refer collectively to these three parameters. (In our formal development, these parameters can be given by arbitrary Coq data types, including user-defined ones.)

The machine imposes no arbitrary constraints on tags, and policy designers can choose them as they see fit. To maintain secrecy and integrity guarantees in a system, for example, we could choose from one-bit taint marks to much richer information-flow labels [78]. By tagging data with finite automata [69], we express rich declassification constraints allowing the release of information from a system. We can even enforce several policies simultaneously by packing their tags in a tuple.

Perhaps surprisingly, this flexibility can be supported with efficient, practical hardware [33], by interpreting word-sized tags as pointers to memory data structures. As these data structures become more complex, policy performance degrades; however, a policy designer is free to choose what overhead is adequate for the security guarantees they want to enforce.

Instead of having different tags for the PC, registers, and memory, we could have opted for a seemingly simpler design where all these tags are drawn from a single set. Our approach, however, makes it possible for policy designers to enforce tagging invariants through typing. This simplifies policy definitions and their proofs of correctness, which only need to consider combinations of tags that can arise during execution. The memory-safety policy of Chapter 4, for example, uses memory tags to keep track of allocation information. This information is not needed for the PC or data stored in registers, which can have much simpler tags.

Definition 2.2. Suppose we are given tag parameters Tag, and

- a set Word of all bit vectors of a given length,
- a finite set Reg of register names, and
- another set Extra.

We define a *symbolic machine state* as a tuple (m, rs, pc, e), where

- $m \in \mathsf{Word} \rightharpoonup \mathsf{Word} \times \mathsf{Tag}_m$ is the memory of the machine;
- $rs \in \mathsf{Reg} \rightharpoonup \mathsf{Word} \times \mathsf{Tag}_r$ is the register bank of the machine;
- $pc \in \mathsf{Word} \times \mathsf{Tag}_{pc}$ is the program counter (PC); and
- $e \in \mathsf{Extra}$ is the private monitor state.

We note State the set of system states. (Technically, this defines State as a function of four parameters, but we will leave those implicit in the notation. The e component of the state can be ignored for now; its role will become clear when we explain the semantics of the machine in Section 2.4.)

We'll often refer to pairs (x, t) comprising a value x tagged with tag t as *atoms*, and write them as $x@t$. The term "atom" hints that values in a system governed by a micro-policy cannot be dissociated from their tags. When the choice of tags is clear, we refer to atoms in $\mathsf{Word} \times \mathsf{Tag}_m$, $\mathsf{Word} \times \mathsf{Tag}_r$, and $\mathsf{Word} \times \mathsf{Tag}_{pc}$ as *memory*, *register*, and *PC* atoms. The exact choice of word size, as well as Reg, does not pay a major role in the security aspects of the policy, and we can assume for concreteness that they are set once and for all (for example, to a set of 32 registers and 64-bit words). We assume that Reg contains at least four registers: two registers r_{arg1} and r_{arg2} for passing arguments to a function, a register r_{ret} to return values to a function caller, and another register r_a for saving return addresses after a function call.

An unusual aspect of the above definition is that the register bank is a *partial* function: if a program attempts to access a register that is not defined in the bank, it causes the symbolic machine to halt. We can mostly ignore this choice at this point; the sole reason for allowing some registers to be undefined is that it simplifies the connection with the lower-level concrete machine introduced in Chapter 5. Typically, registers that are undefined at the symbolic level are defined at the concrete level, but store private state that is inaccessible to user programs. The four registers mentioned above, r_{arg1}, r_{arg2}, r_{ret}, and r_a, are meant to be manipulated by user programs, and thus should be defined in the register bank.

Before explaining how the machine executes and how it interacts with tags, we can give a rough idea of how policies work by describing a few simple tagging schemes.

Example 2.3. We can keep code and data separate in memory with the following tags.

$$\mathsf{Tag}_{pc} = \mathsf{Tag}_r = \{\mathtt{NONE}\} \qquad\qquad \mathsf{Tag}_m = \{\mathtt{C}, \mathtt{D}\}$$

Paraphrasing in words, these tags mark every word in memory as either "code" or "data," and do not attach any information to words stored in the register bank or the PC. We later program the symbolic machine to make data non executable and code non writable. This policy is somewhat redundant, given that many modern architectures (IA-64, x86-64, etc.) already provide this functionality; still, it provides a simple example of what can be done with tags.

Example 2.4. As a slightly more interesting example, we can use tags to implement *dynamic sealing*, a data protection mechanism similar to perfect symmetric encryption [79, 103]. Like for the simple policy sketched above, we set Tag_{pc} to the singleton $\{\mathtt{NONE}\}$, so PC tags of this policy carry no information. The other tags are defined as follows, using typical data-type notation.

$$\mathsf{Tag}_{m} = \mathsf{Tag}_{r} = \mathtt{UNSEALED} \mid \mathtt{SEALED}(k \in \mathbb{N}) \mid \mathtt{KEY}(k \in \mathbb{N})$$

The UNSEALED tag is used for ordinary data, $\mathtt{SEALED}(k)$ is used for words that have been sealed with the key k, and $\mathtt{KEY}(k)$ represents the key k. The idea is that we want to prevent most operations from being performed on sealed values, and only allow sealing and unsealing values when the code possesses a matching key. We chose to store the key value in the key tag to support an arbitrarily large number of keys; we could also have chosen to confine the set of possible keys to Word, and represent keys with atoms of the form $w@\mathtt{KEY}$.

2.2 Checking Tags

On each cycle, the machine fetches a word from the memory position referenced by its current PC, checks whether that word corresponds to a valid instruction $i \in \mathsf{Instr}$, and if so acts accordingly. Figure 2.1 summarizes the possible instructions. Excluding the interaction with tags, instructions behave essentially as in conventional processors, allowing programs to perform basic operations on registers, memory, and control flow. We assume there is a partial function $\mathsf{decode} \in \mathsf{Word} \rightharpoonup \mathsf{Instr}$ for converting words in memory to instructions. The operation of the machine will be detailed soon, but we must first explain what tags do.

The role of tags on execution is governed by another parameter supplied by the policy designer, called the *transfer function*, whose inputs and outputs are described in Figure 2.2. Before executing the instruction, the machine feeds into this function (1) the opcode of the current instruction, (2) the tag of the current PC, (3) the tag of the current instruction, and (4) tags on the arguments of the instruction (including any old tags on its destination). Policy designers then have two choices. One is to say that, based on these inputs, the current instruction does not violate the policy that they are trying to enforce, and should be allowed to execute. In this case, the transfer function must output a

Format	Description
Nop	No action
Const i r	Store constant i in r
Mov r_1 r_2	Move contents of r_1 to r_2
Binop$_\oplus$ r_1 r_2 r_3	Apply operation $\oplus$ to r_1 and r_2, store result in r_3
Load r_1 r_2	Load from memory address in r_1, put contents in r_2
Store r_1 r_2	Store contents of r_1 into memory address in r_2
Jump r	Unconditional absolute jump
Bnz i r	Conditional relative jump
Jal r	Call address in r, save return in the r_a register
Halt	Stop execution

$$\oplus \in \{+, -, \times, =, \leq, \wedge, \vee, ...\}$$

Figure 2.1: Instructions of symbolic machine. Here, r ranges over registers, and i ranges over some fixed set Imm of bit vectors. Binary operations $\oplus$ range over some set of basic logic and arithmetic operations.

tag to use on the next PC, and another tag for the instruction result, if any. On the other hand, policy designers could also decide that this combination of inputs is a sign that something dangerous is about to happen—perhaps the current instruction is trying to modify code stored in memory, for instance. In this case, the transfer function is undefined, forcing the machine to stop execution immediately. We could extend the model to handle violations differently—for example, by raising an exception that can be treated by user code—but we stick to the current one for simplicity. Note that the transfer function is pure: its behavior depends solely on its input tags, and cannot manipulate any stored state. The transfer function also ignores the payload portion of the atoms tagged by its inputs, making it easier to accelerate its execution with specialized hardware [33].

In our formal development, the transfer function is supplied by the policy designer as an arbitrary Coq function of the appropriate type. (As it is often the case, our development models partial functions as Coq functions that return an optional value; all functions in Coq are total.) Note that the exact combination of input and output tags of the transfer function depends on its opcode parameter. Coq's theory has good support for such dependent types, making it a nice platform to develop our methodology.

In this thesis, we'll often specify transfer functions using a *rule table* of the following form:

Opcode	PC	Instr	A1	A2	Dest	PC	Res
o	p	p	p	p	p	p	p

$$\mathsf{transfer} \in (o \in \mathsf{Opcode}) \times \mathsf{Tag}_{pc} \times \mathsf{Tag}_{m} \times T_1(o) \times T_2(o) \times T_d(o) \rightharpoonup \mathsf{Tag}_{pc} \times T_r(o)$$

Opcode	T_1	T_2	T_d	T_r
Nop				
Const			Tag_r	Tag_r
Mov	Tag_r		Tag_r	Tag_r
$\mathsf{Binop}_{\oplus}$	Tag_r	Tag_r	Tag_r	Tag_r
Load	Tag_r	Tag_m	Tag_r	Tag_r
Store	Tag_r	Tag_r	Tag_m	Tag_m
Jump	Tag_r			
Bnz	Tag_r			
Jal	Tag_r		Tag_r	Tag_r

Figure 2.2: Signature of transfer functions for each opcode. We note Opcode the set of opcodes. A blank cell indicates the absence of tags for the corresponding opcode. Notice that Tag_m only appears as an argument or result tag under the two instructions that manipulate the memory, Load and Store.

Opcode	PC	Instr	A1	A2	Dest	PC	Res
Nop	_	`C`				`NONE`	
Const	_	`C`	_		`NONE`	`NONE`	
Mov	_	`C`	_	_	`NONE`	`NONE`	
$\mathsf{Binop}_{\oplus}$	_	`C`	_	_	_	`NONE`	`NONE`
Load	_	`C`	_	`D`	_	`NONE`	`NONE`
Store	_	`C`	_	_	`D`	`NONE`	`D`
Jump	_	`C`	_			`NONE`	
Bnz	_	`C`	_			`NONE`	
Jal	_	`C`	_		_	`NONE`	`NONE`

Figure 2.3: Rules for non-executable data, non-writable code (NXD+NWC)

Each row on this table describes a pattern-matching branch: if we're currently executing an instruction with opcode o, and if the arguments match the patterns on the columns from "PC" to "Dest," then the instruction is allowed to execute, and the "PC" and "Res" columns on the right specify the tags that should be used in the next PC value and on the result; otherwise, we check if the next row matches. We sometimes attach a condition to a rule, a logical formula involving tags indicating that that rule should only be triggered when the condition is true. If no rules match a given tag combination, the transfer function is undefined. A missing pattern indicates that that tag is not used by that instruction, whereas _ is a wild-card pattern that matches anything. We will always provide rules that agree with the signature given in Figure 2.2.

Opcode	PC	Instr	A1	A2	Dest	PC	Res
Nop	_	UNSEALED				NONE	
Const	_	UNSEALED			_	NONE	UNSEALED
Mov	_	UNSEALED	t		_	NONE	t
$\mathsf{Binop}_{\oplus}$	_	UNSEALED	UNSEALED	UNSEALED	_	NONE	UNSEALED
Load	_	UNSEALED	UNSEALED	t	_	NONE	t
Store	_	UNSEALED	UNSEALED	t	_	NONE	t
Jump	_	UNSEALED	UNSEALED			NONE	
Bnz	_	UNSEALED	UNSEALED			NONE	
Jal	_	UNSEALED	UNSEALED		_	NONE	UNSEALED

Figure 2.4: Rules for dynamic sealing

As a simple illustration of rules, consider the ones given in Figures 2.3 and 2.4, which apply to the tagging schemes defined in Examples 2.3 and 2.4. The rules for non-executable data and non-writable code (Figure 2.3) cause the machine to halt whenever the tag on the current instruction is not C, and whenever we try to write to a memory location tagged as C. They also enforce the somewhat less conventional policy of preventing code from loading instructions into registers, as can be seen in the rule for Load. The rules for dynamic sealing (Figure 2.4) prevent code from doing anything with values tagged as SEALED(k) or KEY(k) besides moving them between registers and memory locations.

2.3 System Services

The rules for dynamic sealing show that something is missing: ways of sealing and unsealing values using a given key, and of generating new keys. We remedy this by allowing policy designers to specify a set of privileged *system services* that can be invoked by user code. These services allow users to change tags in controlled ways that are not possible with normal instructions; in particular, they make it possible to implement the missing operations for the sealing policy mentioned above.

System services are similar to typical procedures. They reside at fixed addresses in memory and can be invoked by jumping to those addresses with the Jal instruction. However, unlike typical procedures, system services in the symbolic machine are not implemented with machine code.

Definition 2.5. A *system service* for a given choice of parameters of the symbolic machine is a partial function $f \in \mathsf{State} \rightharpoonup \mathsf{State}$. A service table is a partial function service from Word to system services.

$$\frac{\textsc{MkKey}\quad rs(r_a) = w_{pc}@\texttt{UNSEALED} \qquad rs[r_{ret} \mapsto 0@\texttt{KEY}(k)] = rs' \qquad k \text{ is fresh}}{\mathsf{mkkey}(m, rs, \_, e) = (m, rs', w_{pc}@\texttt{NONE}, e)}$$

$$\frac{\textsc{Seal}\quad \begin{array}{c} rs(r_{arg1}) = w@\texttt{UNSEALED} \qquad rs(r_{arg2}) = \_@\texttt{KEY}(k) \\ rs(r_a) = w_{pc}@\texttt{UNSEALED} \qquad rs[r_{ret} \mapsto w@\texttt{SEALED}(k)] = rs' \end{array}}{\mathsf{seal}(m, rs, \_, e) = (m, rs', w_{pc}@\texttt{NONE}, e)}$$

$$\frac{\textsc{Unseal}\quad \begin{array}{c} rs(r_{arg1}) = w@\texttt{SEALED}(k) \qquad rs(r_{arg2}) = \_@\texttt{KEY}(k) \\ rs(r_a) = w_{pc}@\texttt{UNSEALED} \qquad rs[r_{ret} \mapsto w@\texttt{UNSEALED}] = rs' \end{array}}{\mathsf{unseal}(m, rs, \_, e) = (m, rs', w_{pc}@\texttt{NONE}, e)}$$

Figure 2.5: Sealing system services

Once again, our formal development allows policy designers to supply arbitrary Coq functions as system services. To illustrate the use of system services, let's consider how we can use them to implement the missing operations from the sealing micro-policy. We assume that Reg contains special registers r_a, r_{arg1}, r_{arg2}, and r_{ret}. The first one, r_a, is used to store the return address when calling a function through Jal; we will see how this works when presenting the rest of the operational semantics of the machine. The other three, r_{arg1}, r_{arg2}, and r_{ret}, are used for passing arguments to the services and receiving values from them.

Figure 2.5 defines the services for the sealing policy. The first service, mkkey, makes it possible to generate a new key. The key k is chosen in a way that is *fresh* with respect to the current state, by which we mean that there should not be any tags of the form KEY(k) or SEALED(k) stored anywhere. We can choose k according to any deterministic procedure that guarantees this property. Note that we make sure that the return address stored in r_a is not sealed. As we will see, it is possible for user code to call system services with the contents of r_a set to whatever they choose, by simply calling these services using other instructions than Jal. Thus, this check prevents user code from potentially leaking sealed information. Finally, the definition does not need to assume anything about the current PC. As we will see, each system service is invoked by a generic rule in the operational semantics which makes sure that the current PC matches the address that the service corresponds to.

The two other services allow code to use a key to seal and unseal values. Note that sealing fails if the value is not currently unsealed; it would be possible to consider nested sealed values, but this would slightly complicate our choice of tags. Similarly, we do not allow sealing keys.

With all the services we need in hand, we can finally complete the sealing micro-policy. We choose three addresses w_{mkkey}, w_{seal}, and w_{unseal}, and define a service table $\mathsf{service}_{sealing}$ that maps each one of them to the services mkkey, seal and unseal.

2.4 Operational Semantics

Now that we have described all the parameters that govern its function, we are ready to give a formal operational semantics to the symbolic machine. The rules in Figures 2.6 and 2.7 describe how each instruction is executed. There is no rule for Halt, since it simply causes the machine to stop execution. The Const rule implicitly performs a signed conversion of the immediate $w \in \mathsf{Imm}$ into a full-sized word. (We will tacitly adopt this convention when presenting other machines.) The Svc rule covers the invocation of system services. The machine checks whether any system call corresponds to the current PC in the service table; if so, it runs the service on the current state. (It also ensures that there are no instructions stored in memory at the location given by the PC. This is not fundamentally important, but it keeps the semantics deterministic.)

The rules show that system services are the only mechanism available for modifying the private state of the machine. Though we haven't made use of this feature yet, many policies require access to encapsulated mutable state to correctly enforce the security properties they were designed for. For example, the micro-policy for information-flow control of Chapter 3 uses the private state to implement a protected call stack, used to prevent illegal leaks through the control flow of the program. And the memory-safety policy of Chapter 4 uses the private state to keep track of which memory region are currently allocated and which ones are free.

Strictly speaking, private monitor state is not actually needed for implementing micro-policies, because of the generality of tags and system services. Instead of keeping a list of free regions, the memory-safety policy could simply inspect the entire machine state and its tags to decide where to allocate some chunk of memory requested by the user. In an extreme case, we could also imagine a micro-policy that piggybacks the entire monitor state on its PC tag. Although perfectly valid from a purely logical standpoint, these strategies would be too inefficient if implemented on lower-level mechanisms that have been proposed to support micro-policies, such as the concrete machine of Chapter 5. Private monitor state allows us to keep some performance concerns in mind when developing micro-policies, without overly complicating the model or giving up on convenience.

Another point worth noting is that the Svc rule is the only one that does not invoke the transfer function. We could consider a different design where, before calling a service, the machine consults the transfer function to determine whether the call is legal. This extension does not fundamentally change what system services can do, since they are already allowed to execute arbitrary checks and operations on the machine state; nevertheless, it may allow micro-policies in the symbolic machine to more closely correspond to a concrete implementation, which might use tags to control access to services for performance reasons.

The possibility of invoking the transfer function on system services was explored in the original model of the symbolic machine [11], where it was used to restrict access to system services in a control-flow-integrity [1] micro-policy. It can be implemented by adding a few parameters to the symbolic machine: a new set Tag_s of tags for system services, an extended service table that assigns a tag in Tag_s to each system call, and an extended transfer function that can be passed a new $\mathsf{Service}$ "opcode" that signals that a system call is about to be performed. The semantics of service invocation would then become:

Svc*

$$\frac{m(w_{pc}) = \bot \qquad \mathsf{service}\ w_{pc} = (f, t_s) \qquad \mathsf{transfer}(\mathsf{Service}, t_{pc}, t_s) \neq \bot \qquad f(m, rs, w_{pc}@t_{pc}, e) = s'}{(m, rs, w_{pc}@t_{pc}, e) \to s'}$$

2.5 Refinement

To ensure that micro-policies can protect programs, we would also like to state specifications for them and verify—ideally, using mechanized proofs on top of a formalization like the one proposed here—that these properties hold. Among the many properties that we could consider, one class is particularly useful: *refinement properties*.

When reasoning about a micro-policy, the generality of the symbolic machine can easily become a burden, especially in a formal setting like a proof assistant. Instead of explicitly reasoning about calls to the transfer functions and system services during execution, which may involve manually simplifying them, ignoring useless cases, etc., it is usually easier to relate the micro-policy to a more specific machine semantics where that micro-policy is built-in, by showing that every execution of

Nop

$$\frac{m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{Nop} \qquad \mathsf{transfer}(\mathsf{Nop}, t_{pc}, t_i) = t'_{pc}}{(m, r, w_{pc}@t_{pc}, e) \to (m, r, (w_{pc}+1)@t'_{pc}, e)}$$

Const

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{Const}\ w\ r \\ rs(r) = \_@t \qquad \mathsf{transfer}(\mathsf{Const}, t_{pc}, t_i, t) = (t'_{pc}, t') \qquad rs[r \mapsto w@t'] = rs' \end{array}}{(m, rs, w_{pc}@t_{pc}, e) \to (m, rs', (w_{pc}+1)@t'_{pc}, e)}$$

Mov

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{Mov}\ r\ r_d \qquad rs(r) = w@t \qquad rs(r_d) = \_@t_d \\ \mathsf{transfer}(\mathsf{Mov}, t_{pc}, t_i, t, t_d) = (t'_{pc}, t'_d) \qquad rs[r_d \mapsto w@t'_d] = rs' \end{array}}{(m, rs, w_{pc}@t_{pc}, e) \to (m, rs', (w_{pc}+1)@t'_{pc}, e)}$$

Binop

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{Binop}_{\oplus}\ r_1\ r_2\ r_d \\ rs(r_1) = w_1@t_1 \qquad rs(r_2) = w_2@t_2 \qquad rs(r_d) = \_@t_d \\ \mathsf{transfer}(\mathsf{Binop}_{\oplus}, t_{pc}, t_i, t_1, t_2, t_d) = (t'_{pc}, t'_d) \qquad rs[r_d \mapsto (w_1 \oplus w_2)@t'_d] = rs' \end{array}}{(m, rs, w_{pc}@t_{pc}, e) \to (m, rs', (w_{pc}+1)@t'_{pc}, e)}$$

Load

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{Load}\ r\ r_d \\ rs(r) = w@t \qquad m(w) = w'@t' \qquad rs(r_d) = \_@t_d \\ \mathsf{transfer}(\mathsf{Load}, t_{pc}, t_i, t, t', t_d) = (t'_{pc}, t'_d) \qquad rs[r_d \mapsto w'@t'_d] = rs' \end{array}}{(m, rs, w_{pc}@t_{pc}, e) \to (m, rs', (w_{pc}+1)@t'_{pc}, e)}$$

Store

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{Store}\ r_p\ r_s \\ rs(r_p) = w_p@t_p \qquad rs(r_s) = w_s@t_s \qquad m(w_p) = \_@t_d \\ \mathsf{transfer}(\mathsf{Store}, t_{pc}, t_i, t_p, t_s, t_d) = (t'_{pc}, t'_d) \qquad m[w_p \mapsto w_s@t'_d] = m' \end{array}}{(m, rs, w_{pc}@t_{pc}, e) \to (m', rs, (w_{pc}+1)@t'_{pc}, e)}$$

Jump

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{Jump}\ r \\ rs(r) = w'_{pc}@t \qquad \mathsf{transfer}(\mathsf{Jump}, t_{pc}, t_i, t) = t'_{pc} \end{array}}{(m, rs, w_{pc}@t_{pc}, e) \to (m, rs, w'_{pc}@t'_{pc}, e)}$$

Figure 2.6: Semantics of symbolic machine

$$\textsc{Bnz}\quad \frac{m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{Bnz}\ r\ n \qquad rs(r) = w@t \qquad \mathsf{transfer}(\mathsf{Bnz}, t_{pc}, t_i, t) = t'_{pc} \qquad w'_{pc} = \text{if } w = 0 \text{ then } w_{pc} + 1 \text{ else } w_{pc} + n}{(m, rs, w_{pc}@t_{pc}, e) \to (m, rs, w'_{pc}@t'_{pc}, e)}$$

$$\textsc{Jal}\quad \frac{m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{Jal}\ r \qquad rs(r) = w'_{pc}@t_1 \qquad rs(r_a) = \_@t_{ra} \qquad \mathsf{transfer}(\mathsf{Jal}, t_{pc}, t_i, t_1, t_{ra}) = (t'_{pc}, t'_{ra}) \qquad rs[r_a \mapsto (w_{pc} + 1)@t'_{ra}]}{(m, rs, w_{pc}@t_{pc}, e) \to (m, rs, w'_{pc}@t'_{pc}, e)}$$

$$\textsc{Svc}\quad \frac{m(w_{pc}) = \bot \qquad \mathsf{service}\ w_{pc} = f \qquad f(m, rs, w_{pc}@t_{pc}, e) = s'}{(m, rs, w_{pc}@t_{pc}, e) \to s'}$$

Figure 2.7: Semantics of symbolic machine (continued)

the symbolic machine instantiated with that micro-policy corresponds to a matching execution of the more specific machine. This is what we call a refinement result. By giving a more explicit definition of a policy, refinements help us understand what a micro-policy does; furthermore, it can serve as a stepping stone for proving more complex results, by showing them first on the more specialized machine, and then transferring them to the symbolic machine. Indeed, refinement will be an important ingredient when reasoning about our main case studies of micro-policies in Chapters 3 and 4. As a warm-up exercise, we present here a refinement result for the sealing micro policy described earlier.

Definition 2.6. The set Val of values of the sealing machine is given by the following data-type declaration:

$$\mathsf{Val} = w \in \mathsf{Word} \mid \mathsf{Sealed}(w, k) \mid k \in \mathbb{N}$$

The set of states of the sealing machine, written $\mathsf{State}_{sealing}$, is the set of triples (m, rs, pc), where

- $m \in \mathsf{Word} \rightharpoonup \mathsf{Val}$ is the memory;
- $rs \in \mathsf{Reg} \rightharpoonup \mathsf{Val}$ is the register bank; and
- $pc \in \mathsf{Word}$ is the program counter.

Thus, the state of the sealing machine is obtained by simplifying the state of the symbolic machine running the sealing micro-policy. We can easily define when a state of the symbolic machine represents a state of the sealing machine.

Definition 2.7. We say that a memory or register atom $w@t$ of the sealing micro-policy *refines* a value $v \in \mathsf{Val}$, noted $w@t \rhd v$, if one of the following holds:

- $t = \texttt{UNSEALED}$ and $v = w$;
- $t = \texttt{SEALED}(k)$ and $v = \mathsf{Sealed}(w, k)$; or
- $t = \texttt{KEY}(k)$ and $v = \mathsf{Key}(k)$.

We say that a state $s_1 = (m_1, rs_1, pc_1@t_{pc}, e_1)$ of the symbolic machine running the sealing micro-policy *refines* a state $s_2 = (m_2, rs_2, pc_2)$ of the sealing machine, written $s_1 \rhd s_2$, if their memories and registers are pointwise related by the value refinement relation above (cf. Definition A.1), and if $pc_1 = pc_2$. (Since the tag on the PC and the private state play no role for this policy, they are simply ignored.)

We can now play the same game with the stepping relation of the symbolic machine; the resulting relation is given for reference in Figures 2.8 and 2.9. The two semantics are related by the following result.

Theorem 2.8. *Let s_1 and s_1' be states of the symbolic machine instantiated with the sealing micro-policy, and let s_2 be a state of the sealing machine. Suppose that $s_1 \to^* s_1'$ and that $s_1 \rhd s_2$. There exists a state s_2' of the sealing machine such that $s_2 \to^* s_2'$ and $s_1' \rhd s_2'$.*

The proof of this result follows by induction on the number of steps taken by the symbolic machine, by considering all possible instructions that could have been executed at each point. A similar result stated for a slightly different sealing micro-policy has been formalized in our Coq development.

Refinement results such as Theorem 2.8 allow us to obtain safety guarantees for the symbolic machine by analyzing the behavior of its higher-level counterparts: if the higher-level machine fail-stops on a violation, then so does the symbolic machine. It would also be desirable to have a converse *liveness* result, saying that any execution of the higher-level machine can be realized by the symbolic one; however, we will not consider such properties here.

Nop

$$\frac{m(w_{pc}) = i \in \mathsf{Word} \qquad \mathsf{decode}\ i = \mathsf{Nop}}{(m, rs, w_{pc}) \to (m, rs, w_{pc} + 1)}$$

Const

$$\frac{m(w_{pc}) = i \in \mathsf{Word} \qquad \mathsf{decode}\ i = \mathsf{Const}\ w\ r \qquad rs[r \mapsto w] = rs'}{(m, rs, w_{pc}) \to (m, rs', w_{pc} + 1)}$$

Mov

$$\frac{m(w_{pc}) = i \in \mathsf{Word} \qquad \mathsf{decode}\ i = \mathsf{Mov}\ r\ r_d \qquad rs(r) = v \qquad rs[r_d \mapsto v] = rs'}{(m, rs, w_{pc}) \to (m, rs', w_{pc} + 1)}$$

Binop

$$\frac{\begin{array}{c} m(w_{pc}) = i \in \mathsf{Word} \qquad \mathsf{decode}\ i = \mathsf{Binop}_{\oplus}\ r_1\ r_2\ r_d \\ rs(r_1) = w_1 \qquad rs(r_2) = w_2 \qquad rs[r_d \mapsto w_1 \oplus w_2] = rs' \end{array}}{(m, rs, w_{pc}) \to (m, rs', w_{pc} + 1)}$$

Load

$$\frac{\begin{array}{c} m(w_{pc}) = i \in \mathsf{Word} \qquad \mathsf{decode}\ i = \mathsf{Load}\ r\ r_d \\ rs(r) = w \qquad m(w) = v \qquad rs[r_d \mapsto v] = rs' \end{array}}{(m, rs, w_{pc}) \to (m, rs', w_{pc} + 1)}$$

Store

$$\frac{\begin{array}{c} m(w_{pc}) = i \in \mathsf{Word} \qquad \mathsf{decode}\ i = \mathsf{Store}\ r_p\ r_s \\ rs(r_p) = w_p \qquad rs(r_s) = v \qquad m[w_p \mapsto v] = m' \end{array}}{(m, rs, w_{pc}) \to (m', rs, w_{pc} + 1)}$$

Jump

$$\frac{m(w_{pc}) = i \in \mathsf{Word} \qquad \mathsf{decode}\ i = \mathsf{Jump}\ r \qquad rs(r) = w'_{pc}}{(m, rs, w_{pc}) \to (m, rs, w'_{pc})}$$

Bnz

$$\frac{\begin{array}{c} m(w_{pc}) = i \in \mathsf{Word} \qquad \mathsf{decode}\ i = \mathsf{Bnz}\ r\ n \\ rs(r) = w \qquad w'_{pc} = \mathsf{if}\ w = 0\ \mathsf{then}\ w_{pc} + 1\ \mathsf{else}\ w_{pc} + n \end{array}}{(m, rs, w_{pc}) \to (m, rs, w'_{pc})}$$

Jal

$$\frac{m(w_{pc}) = i \in \mathsf{Word} \qquad \mathsf{decode}\ i = \mathsf{Jal}\ r \qquad rs(r) = w'_{pc} \qquad rs[r_a \mapsto w_{pc} + 1] = rs'}{(m, rs, w_{pc}) \to (m, rs', w'_{pc})}$$

Figure 2.8: Semantics of sealing machine

$$\textsc{MkKey}\quad \frac{m(w_{mkkey}) = \bot \qquad rs(r_a) = w_{pc} \qquad rs[r_{ret} \mapsto \mathsf{Key}(k)] = rs' \qquad k \text{ is fresh}}{(m, rs, w_{mkkey}) \to (m, rs', w_{pc})}$$

$$\textsc{Seal}\quad \frac{\begin{array}{c} m(w_{seal}) = \bot \qquad rs(r_{arg1}) = w \qquad rs(r_{arg2}) = \mathsf{Key}(k) \\ rs(r_a) = w_{pc}@ \qquad rs[r_{ret} \mapsto \mathsf{Sealed}(w, k)] = rs' \end{array}}{(m, rs, w_{seal}) \to (m, rs', w_{pc})}$$

$$\textsc{Unseal}\quad \frac{\begin{array}{c} m(w_{unseal}) = \bot \qquad rs(r_{arg1}) = \mathsf{Sealed}(w, k) \qquad rs(r_{arg2}) = \mathsf{Key}(k) \\ rs(r_a) = w_{pc} \qquad rs[r_{ret} \mapsto w] = rs' \end{array}}{(m, rs, w_{unseal}) \to (m, rs', w_{pc})}$$

Figure 2.9: Semantics of sealing machine (continued)

2.6 Discussion and Related Work

The symbolic machine described here was introduced by me in previous work with other collaborators [11], where it was used to specify and verify four micro-policies: the sealing policy we have seen here, heap memory safety (discussed in Chapter 4), control-flow integrity [1], and process-level compartmentalization, following the SFI model of Wahbe et al. [111]. It evolved from an earlier, simpler form [9, 10] that was used only as an intermediate refinement step in a proof of correctness for an information-flow monitor; only recently did it become a formalism for specifying generic micro-policies.

For some of the micro-policies studied by us [11], refinements were the final soundness results, guaranteeing that the micro-policy correctly implemented a higher-level abstract machine that embodied the notion of security that we were interested in. For others, refinement was simply used as a stepping stone to derive a higher-level property—for example, a trace property used by Abadi et al. [1] to characterize control-flow integrity, or noninterference for the IFC monitor developed for the PUMP's earlier incarnation [9, 10]. This double use is commonplace in formal security results: while the correctness of information-flow control in the PROSPER system was stated as a refinement result [21, 22, 66], work on the seL4 micro kernel [68] used refinement to transfer a noninterference result from the system's abstract specification to its C implementation [81, 82].

Other formalisms for defining and reasoning about security policies have been proposed in the literature. A particularly relevant one is the notion of *security automaton* introduced by Schneider [95] to describe properties that can be enforced by a dynamic monitor. A security automaton is similar to a non-deterministic finite automaton, and is given by

- a countable set Q of states;
- a set $Q_0 \subseteq Q$ of initial states;
- a countable set I of input symbols; and
- a transition relation $\delta \subseteq Q \times I \times Q$.

In this formalism, the execution of a system corresponds to a (possibly infinite) trace of input symbols, representing the states traversed by the system during execution, actions performed by the system, etc. Conceptually, the automaton maintains a set Q' of current states, initially set to Q_0, and it executes by successively analyzing input symbols produced by the system it monitors. When reading a symbol i, it transitions from Q' to $\bigcup_{q \in Q'} \{q' \mid (q, i, q') \in \delta\}$. If Q' ever becomes empty, the automaton considers that a policy violation occurred, and proceeds to terminate the monitored program.

One difference between security automata and our micro-policies is that the state of our monitors have a more fixed structure. Another important point of divergence is that execution of our monitors is more directly connected to the programs they supervise, whereas in the formalism of security automata the behavior of the monitor is almost completely decoupled from the underlying program. These differences are mostly to keep our model closer to the mechanisms that have been proposed to accelerate its execution.

Other formalisms for security policies were conceived with rather different use cases in mind. One example is the Legalease language of Sen et al. [96]. Legalease is a language for privacy policies for big-data systems that restrict how user data is handled, and for what purposes it can be used. Besides the more specific end application, Legalease policies are used to inspect programs statically, and rely on another analysis that attempts to map program concepts to the higher-level entities in Legalease.

Chapter 3

Information-Flow Control

One of the original motivations for the tagging capabilities of the PUMP, as they were initially developed in the context of the SAFE platform [25], was to provide efficient support for dynamic *information-flow control* (IFC) [29], a discipline for guaranteeing data confidentiality. In IFC systems, each piece of data is marked with a security *label* that describes what users or parts of the system are allowed to read that data. For example, a system could classify its data as either public or secret, stipulating that secret data can only be observed by users marked with high clearance, while placing no restrictions on how public data is used. This becomes challenging once we try not only to prevent users from reading data labeled in a certain way *directly*, but also from learning anything about that data *indirectly*, through the results of other computations that might depend on it. For example, if the hypothetical system described above stores the salaries of all users as secret information, it should not be possible for any information that is presented as public in the system to depend on those salaries—even the sum of all salaries or their average, which could in principle be computed as derived information somewhere in the system but inadvertently stored as public information.

Traditionally, IFC systems with formal claims of soundness relied on clever static analyses such as specialized type systems [52, 93]. Though many of the mechanisms for enforcing IFC were dynamic, especially those devised in the systems literature [35, 109], it was unknown whether they could be sound due to the possibility of *implicit flows*, whereby a secret can be leaked indirectly through the control flow of the program. This state of affairs changed rather recently, with works demonstrating that it was possible to enforce IFC dynamically and soundly [49], spurring great interest in dynamic mechanisms [6, 54, 94, 101]. Though dynamic mechanisms can be more costly than their static counterparts in terms of performance, they are also more permissive and well adapted to settings such as web browsers, where the evaluation of freshly generated code is commonplace. This chapter presents a micro-policy for information-flow control and discusses the formal guarantees it provides. This is essentially an adaptation of previous designs developed by me and other collaborators [9, 10, 55, 56] for a different, more structured machine model that mimicked special features present in the SAFE hardware—notably, a protected call stack, which is needed for correctly invoking procedures that manipulate sensitive data.

It is worth noting that a vast body of work in the IFC literature has demonstrated that its techniques go beyond pure confidentiality guarantees, being capable of enforcing integrity properties as well, controlling which parties in a system are allowed to influence certain data items. For example, we might want to prevent unprivileged users in a system from modifying the programs installed in that system. We will restrict our discussion to confidentiality properties in what follows, for simplicity.

3.1 Confidentiality Labels

To describe secrecy constraints, typical IFC systems impose an ordering $\sqsubseteq$ on the set L of data labels [29]. Intuitively, $l_1 \sqsubseteq l_2$ states that the label l_2 is "more restrictive" than l_1, meaning that data labeled with l_2 can be used in fewer contexts than data labeled with l_1. In a system that has only two classes of data, public and secret, the set of labels L would comprise two security levels P and S, ordered as $P \sqsubseteq S$. The idea is that P is used to label public data, whereas S labels secret data that requires special clearance to be observed (e.g., the user might have to be logged in as a system administrator). As a more elaborate example, we could take as elements of L sets of users of a system: each $l \in L$ then would describe which users are allowed to observe a given piece of data. In this case, we define $l_1 \sqsubseteq l_2$ as $l_1 \supseteq l_2$, the idea being that the fewer users can observe a value, the more secret that value is.

For our IFC micro-policy, we need to impose more structure on labels than simply an ordering: it should be possible to combine two labels l_1 and l_2 into a new label $l_1 \sqcup l_2$ that is as restrictive as l_1 and l_2. To see why, suppose that we have two pieces of data x_1 and x_2 that are used as inputs to some program f, and that their uses are governed by the labels l_1 and l_2. Then the use of the result $f(x_1, x_2)$ should combine the restrictions associated with x_1 and x_2, since in general it might allow observers to infer properties about both inputs; to track this dependency, we mark this result with the compound label $l_1 \sqcup l_2$. For this to make sense, we require some properties of the $\sqcup$ operation, summarized in the following definition.

Definition 3.1. A *semi-lattice* is an idempotent monoid $(L, \sqcup, \bot)$. Spelled out explicitly, this means that L is a set equipped with an element $\bot \in L$ and a binary operation $\sqcup$ that maps a pair of elements $l_1, l_2 \in L$ to an element $l_1 \sqcup l_2 \in L$, and that this data is required to satisfy the following properties:

Commutativity $l_1 \sqcup l_2 = l_2 \sqcup l_1$;

Associativity $l_1 \sqcup (l_2 \sqcup l_3) = (l_1 \sqcup l_2) \sqcup l_3$;

Identity $\bot \sqcup l = l$; and

Idempotence $l \sqcup l = l$.

We refer to $\bot$ as *bottom*, and to $l_1 \sqcup l_2$ as the *join* of the elements l_1 and l_2. We will often use the word *label* to refer generically to elements of a semi-lattice, reserving "tag" for the more general use in other policies.

This definition does not mention any notion of order on labels, but such an ordering is not necessary, as it can be fully recovered from the join operation: given a semi-lattice L and two labels $l_1, l_2 \in L$, we stipulate that $l_1 \sqsubseteq l_2$ if and only if $l_2 = l_1 \sqcup l_2$. Notice that this definition places the bottom element below all other labels, since $l = l \sqcup \bot$ by commutativity and identity. Under the security reading of the order relation given above, this means that bottom is the least restrictive security level. By similar algebraic manipulations, we can show that $\sqsubseteq$ is a *partial order*, meaning that it is

Reflexive $l \sqsubseteq l$ (because of idempotence);

Transitive $l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$ (because of associativity); and

Antisymmetric $l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$ (because of commutativity).

The interpretation that we gave of the join operation above, as computing a security level that is at least as restrictive as the one of its arguments, makes sense because

$$l_1 \sqsubseteq l_1 \sqcup l_2 \qquad\qquad l_2 \sqsubseteq l_1 \sqcup l_2.$$

Furthermore, we can show that $l_1 \sqcup l_2 \sqsubseteq l_3$ is equivalent to $l_1 \sqsubseteq l_3$ and $l_2 \sqsubseteq l_3$, which means that the join operation computes the smallest label that is as restrictive as its arguments. Thus, if we use $l_1 \sqcup l_2$ to label the result of a computation $f(x_1, x_2)$ performed on two values x_1 and x_2 labeled as l_1 and l_2, we are placing the least amount of restrictions on the result to correctly track the dependency on the inputs that were used to produce it.

To better understand these ideas, it is useful to analyze some examples of semi-lattices.

Example 3.2. Consider the first set of IFC labels mentioned above, with two security levels, $P \sqsubseteq S$. We can derive this order from a semi-lattice structure by taking $\bot$ to be P, and by defining the join operation as follows:

$$l_1 \sqcup l_2 = \begin{cases} S & \text{if } l_1 = S \text{ or } l_2 = S \\ P & \text{otherwise.} \end{cases}$$

Example 3.3. Let U be a set of users in a system. We might want to tag the data in a system with the set of users that are allowed to read that data. We can express this policy with the semi-lattice $(\mathcal{P}(U), \cap, U)$, where $\mathcal{P}(U)$ is the set of all subsets of U. Intuitively, the bottom label U is the least restrictive label because it allows every user in the system to make use of the data it classifies.

Example 3.4. The natural numbers form a semi-lattice by taking the join operation to be $\max$, and the bottom element to be 0.

In what follows, we will build an IFC micro-policy that is parameterized by an arbitrary semi-lattice L, used to describe data secrecy levels. Note that some IFC systems require more structure of labels than simply forming a semi-lattice. *Reactive information flow* [69], for example, uses as labels finite automata describing what users of the system are allowed to observe some information after a sequence of operations has been performed on it, enabling more permissive policies such as allowing users of an electronic voting system to learn about the outcome of an election, after all the votes having been tallied, while preventing them from observing choices cast in particular votes. Nevertheless, the order structure we employ is used as a basic building block even in these more complex systems, and the basic micro-policy we describe could certainly serve as a starting point to design new ones.

3.2 Information-Flow Rules

We begin by presenting a simple micro-policy with no system services, whose sole purpose is to correctly track the dependencies between data items that arise during execution. Having chosen some semi-lattice L, the labels used in this policy are as follows.

$$\mathsf{Tag}_{pc} = \mathsf{Tag}_{r} = L$$

$$\mathsf{Tag}_{m} = \mathtt{C} \mid \mathtt{D}(l : L)$$

Memory tags are used to distinguish between code and data, just as in the basic policy of Figure 2.3, but data items in memory carry an additional IFC label describing their security. Similarly, data stored in registers has its own labels, as well as the PC. Labeling the PC is commonplace in IFC systems, so that they can deal with *implicit flows*, where the control flow of the program may be used to leak information. As a simple example, consider the following piece of code:

```
if (secret) {
  leaked = 1;
} else {
  leaked = 0;
}
```

Suppose that a regular, unprivileged user has the ability to observe the value of the variable `leaked` above, and that we want to conceal the value of the variable `secret` from that user. It is clear that the value will depend on the value of the variable `secret`; however, with a naive flow analysis, we might not be able to detect this dependency, since the value of `secret` is not directly copied to `leaked`. The PC label is used precisely to remedy this problem: its value is an upper bound on the confidentiality restrictions of all values that have influenced the current control flow. By keeping track of this label, we can, for example, decide at the moment of the write to `leaked` that this might result in a data leak, and stop the program.

Figure 3.1 defines the transfer function for this policy. The PC label stays the same in most cases, except for branching instructions (Jump, Bnz and Jal), where it accounts for possible influences on the next PC value (the destination address, in the case of Jump and Jal, or the branch condition, in the case of Bnz). The result of each instruction is labeled by combining the security levels of the

Opcode	PC	Instr	A1	A2	Dest	PC	Res
Nop	l_{pc}	C				l_{pc}	
Const	l_{pc}	C			–	l_{pc}	$\bot$
Mov	l_{pc}	C	l_r		–	l_{pc}	l_r
Binop$_\oplus$	l_{pc}	C	l_1	l_2	–	l_{pc}	$l_1 \sqcup l_2$
Load	l_{pc}	C	l_{ptr}	$\mathtt{D}(l_{mem})$	–	l_{pc}	$l_{ptr} \sqcup l_{mem}$
Store	l_{pc}	C	l_{ptr}	l_{new}	$\mathtt{D}(l_d)$	l_{pc}	$\mathtt{D}(l_{ptr} \sqcup l_{new} \sqcup l_{pc})$ if $l_{ptr} \sqcup l_{pc} \sqsubseteq l_d$
Jump	l_{pc}	C	l_{dest}			$l_{pc} \sqcup l_{dest}$	
Bnz	l_{pc}	C	l_{cond}			$l_{pc} \sqcup l_{cond}$	
Jal	l_{pc}	C	l_{dest}		–	$l_{pc} \sqcup l_{dest}$	$\bot$

Figure 3.1: IFC transfer function

inputs that were used to compute it; for Const, since there are no inputs, this label is simply $\bot$. In a sense, we could see the current PC as part of these inputs, since it is responsible for choosing what instruction to execute. However, no instructions propagate the PC label to the label of results stored in registers, because this would not be necessary: the policy implicitly treats the entire register bank as having been influenced by the PC.

Assuming that the code does not try to access instructions as data or to execute data as instructions, there is nothing in the rules that can stop execution. The only exception is the Store instruction, which can only execute when $l_{ptr} \sqcup l_{pc} \sqsubseteq l_{old}$ holds—usually known in the literature as the *no-sensitive-upgrade* condition [6, 115]. The reason for this restriction is intuitively similar to the problem faced before with implicit flows: it should not be possible for a program to make the use of a memory location more restricted by writing to it from a high context, because it might change what an observer can infer about the state of the system. The issue will become clearer once we discuss the security property enforced by this policy.

3.3 Restoring the PC

A careful inspection of the rules of Figure 3.1 reveals an interesting problem with the micro-policy we have presented so far: since every rule makes the new PC label at least as restrictive as the previous one, it is impossible for that label to ever go down. This choice is too conservative, since as the computation evolves, different confidentiality levels creep into the PC label, restricting what values can be manipulated.

Typical IFC systems deal with this *label creep* problem by exploiting the control-flow structure of the program. Consider a program like the one we saw earlier, where we take advantage of the knowledge we obtain by being on each branch of the if statement to leak a secret. After the if statement, when the control flows of both branches meet, we are not under the influence of the secret condition we branched on, since the code to be executed is the same independently of which branch we took. This suggests that it should be possible to soundly restore the value of the PC to whatever it was before control branched [6, 54, 101].

Adapting this idea to work with machine code is challenging, given the unstructured nature of its control flow. The IFC policy for the SAFE platform [9, 10] attacked this problem by maintaining a stack of security labels managed with explicit call and return instructions. In this setting, if we want to restore the PC label after some branching construct—including conditionals and loops—we must compile that construct to a function call: before branching, we save the PC label on the stack, compute, and return after the merge point to restore the label. To guarantee soundness, it shouldn't be possible for user code to access this stack directly—otherwise, the PC label could be incorrectly downgraded on a return. The SAFE platform made it possible to implement this scheme by exposing a hardware protected stack for function calls; however, general micro-policies cannot directly rely on this feature, which is absent from conventional processors. The solution we propose is to protect this stack from user code by keeping it as part of the internal state of the micro-policy, exposing only special call and return *system services* that have the privilege to manipulate it. Specifically, the internal state of the IFC micro-policy is a stack of *call frames*, which are pairs of the form (pc, rs):

- pc is an atom $v_{pc}@l_{pc}$ comprising a PC value $v_{pc} \in \mathsf{Word}$ and a label $l_{pc} \in L$; and
- rs is a register bank (that is, a map $\mathsf{Reg} \rightharpoonup \mathsf{Word} \times L$), for saving previous register values, as explained below.

The system services for manipulating the stack, defined in Figure 3.2, are inspired by previous work on adapting our IFC policy to a machine with registers [56]. We assume that we have two designated registers r_{arg} and r_{ret} for passing arguments to system services, and that the services Call and Return are invoked by performing a Jal to addresses w_{call} and w_{return}.

It is worth taking some time to explain the mechanism of calls and returns, beginning with the contents of the call stack. We have already seen that we want to record the previous PC labels on the stack, so that we can restore them on every return. However, call frames also record the entire state

$$\textsc{Call}\quad \frac{rs(r_a) = pc_{caller}@l_{caller} \qquad rs(r_{arg}) = pc_{callee}@l_{callee} \qquad l'_{pc} = l_{callee} \sqcup l_{caller} \sqcup l_{pc} \qquad cf = (pc_{caller}@(l_{caller} \sqcup l_{pc}), rs)}{\mathsf{Call}(m, rs, w_{call}@l_{pc}, cs) = (m, rs, pc_{callee}@l'_{pc}, cf :: cs)}$$

$$\textsc{Return}\quad \frac{rs(r_{ret}) = w_{ret}@l_{ret} \qquad rs'[r_{ret} \mapsto w_{ret}@(l_{pc} \sqcup l_{ret})] = rs''}{\mathsf{Return}(m, rs, w_{return}@l_{pc}, (w'_{pc}@l'_{pc}, rs') :: cs) = (m, rs'', w'_{pc}@l'_{pc}, cs)}$$

Figure 3.2: System services for manipulating the IFC call stack

of the registers at the moment of each call, as seen in the rule for Call. The reason for this, intuitively, is that registers are not subject to the same restrictions on stores as the ones used for the memory. In order to prevent similar issues with implicit flows, we simply make such flows impossible, by restoring the state of almost all registers at the moment of a Return, when the PC label is lowered. The only exception is the special register r_{ret}, where the callee is allowed to leave a return value for the caller. To make this return mechanism sound, we must join to the label of that register the label on the PC that called the Return service, to correctly track the data that influenced that returned value.

A point worth noting about the semantics of function calls is that the call stack records not only the value of the current PC label, l_{pc}, but also the label of whatever PC performed that call, l_{caller}. When the Call service is invoked as usual, by performing a Jal to its address, it is necessarily the case that $l_{caller} \sqsubseteq l_{pc}$, so this update might not sound necessary. However, it could also be the case that somebody tries to trick the service into returning to a lower PC than the one that performed the Call, by storing that PC value directly in the specially designated r_a register used by Jal, and subsequently jumping to w_{call}. This adjusted level fixes these issues.

Performing calls and returns with special system services might raise performance concerns. Indeed, these services need to save and restore not only the current PC but the entire register bank, thus consuming several cycles to perform a single function call. Nevertheless, it is unclear how problematic this would be in practice, since we only need the special Call and Return services for restoring the PC label after the return, which may not be needed very often. Indeed, there are other systems built upon this assumption. For example, in concurrent LIO [99], an IFC library for Haskell, the only way to inspect secret information without raising the PC label is by forking a thread to perform the task, which should be at least as expensive as our approach.

3.4 Noninterference

Now that we have described the IFC micro-policy, it is time to analyze its security guarantees. Following standard practice, we will use as our correctness criterion the property of *noninterference*, originally introduced by Goguen and Meseguer [42]. Although noninterference can be formalized in a multitude of ways [52], the basic idea is simple: if two systems differ only in data that is secret with respect to some security level, then running the two systems should produce identical observations at that level.

To state a noninterference property, we begin by explaining what it means for two states to be indistinguishable.

Definition 3.5. Let l be a security label. We say that two register atoms $w_1@l_1$ and $w_2@l_2$ are *l-indistinguishable*, written $w_1@l_1 \approx_l w_2@l_2$, if

$$l_1 \sqsubseteq l \vee l_2 \sqsubseteq l \Rightarrow w_1@l_1 = w_2@l_2.$$

We say that two memory atoms $w_1@t_1$ and $w_2@t_2$ are l-indistinguishable if they are equal or if there are labels l_1 and l_2 such that $t_i = \mathtt{DATA}(l_i)$ and $w_1@l_1 \approx_l w_2@l_2$.

Two memories m_1 and m_2 (with IFC memory tags) are l-indistinguishable if they are pointwise related by l-indistinguishability for memory atoms (cf. Definition A.1).

Finally, two states s_1 and s_2 are l-indistinguishable if they agree on all components (except for their memories, which should be l-indistinguishable), and their output traces are empty; that is, $s_i = (m_i, rs, pc, cs, [])$ and $m_1 \approx_l m_2$.

Next, we slightly modify our policy by introducing another component in its internal state: a trace of *output events*. This output trace would probably not be stored in an actual implementation of this micro-policy; rather, it is a modeling device to represent the outputs that the machine has produced during its execution. Concretely, an output event is simply an atom $w@l$ that combines a word $w \in W$ and a label $l \in L$. The label l on this event models who is capable of reading the data; in a more realistic design, communication with the outside world would probably require some form

$$\textsc{Output}\quad \frac{rs(r_{arg}) = w@l \qquad rs(r_a) = w_{pc}@l'_{pc}}{\mathsf{Output}(m, rs, w_{output}@l_{pc}, cs, t) = (m, rs, w_{pc}@(l_{pc} \sqcup l'_{pc}), cs, t \cdot [w@(l \sqcup l_{pc})])}$$

Figure 3.3: Output system service. Here, $\cdot$ represents sequence concatenation.

of encryption, guaranteeing that only principals explicitly allowed by the label l have the ability of reading it.

Output is performed with a new Output system service, detailed in Figure 3.3. This is the only operation that can modify the output trace in the internal state; the other system services Call and Return are modified to leave the trace intact. Note that we must taint the label on the new PC in a similar fashion to the Call service.

Output traces also carry their own notion of indistinguishability.

Definition 3.6. Given a label l, we say that two traces of output events t_1 and t_2 are l-indistinguishable if the following condition holds. Let t'_i be the trace obtained by filtering out from t_i all the output events $w@l'$ that do not satisfy $l' \sqsubseteq l$. Then there should exist a trace t' such that $t'_1 = t'_2 \cdot t'$ or $t'_2 = t'_1 \cdot t'$. In other words, the filtered traces t'_1 and t'_2 should share a common maximal prefix.

Thus, if it is not the case that $t_1 \approx_l t_2$—that is, if the traces are distinguishable—that there must be an index i such that the filtered traces t'_1 and t'_2 are bigger than i but differ in their i-th element. The filtering of the traces represents the outputs that an observer with observation power bounded by l would receive.

Definition 3.7. We say that the symbolic machine has noninterference if the following property holds. Given a label l and a pair of indistinguishable states $s_1 \approx_l s_2$, suppose that $s_1 \to^* s'_1$ and $s_2 \to^* s'_2$. Then the traces generated during these executions (that is, the trace component of the internal state of s'_1 and s'_2) are l-indistinguishable.

This notion of noninterference does allow some secret information to leak. The number of execution steps between output events in two traces is irrelevant when defining indistinguishability, implying that secret data can influence computation time—that is, our notion is *timing insensitive*. Furthermore, since cropping a suffix of a trace yields an indistinguishable trace, varying a secret can cause an output trace to stop prematurely, by making the computation enter an infinite loop without observable outputs, or by triggering a run-time error. This is usually known in the literature as *termination-insensitive* (or *progress-insensitive* [5]) noninterference. We discuss the implications of this choice in Section 3.6.

3.5 Verifying Noninterference

Now that we have a micro-policy and a correctness criterion for it, it is time to verify it. The proof strategy mimics the one we used in previous work [9, 10]. First, we show that the symbolic machine running the IFC micro-policy simulates a morally equivalent machine where all the IFC checks and propagation rules are built in. Since this machine is tailored for IFC, it is much easier to prove noninterference for it directly, with a set of unwinding conditions [43] that can be used in an inductive proof. Once this is done, we can transfer the noninterference property to the symbolic machine via a preservation argument.

Definition 3.8. The state of the IFC machine is a tuple (m, rs, pc, cs), where

- $m \in \mathsf{Word} \rightharpoonup \mathsf{Instr} + \mathsf{Word} \times L$;
- $rs \in \mathsf{Reg} \rightharpoonup \mathsf{Word} \times L$;
- $pc \in \mathsf{Word} \times L$; and
- cs is a call stack, as used in the private state of the IFC micro-policy.

The step relation of the IFC machine is defined in Figures 3.4 and 3.5. Note that this is a 3-place relation: $s \to s'$ means that s steps to s' without producing any output events, whereas $s \to_o s'$ means that s steps to s' while producing an output $o \in \mathsf{Word} \times L$ (note that this can only occur in the rule for output; no other operations can produce events). We will sometimes write $s \to_e s'$ to indicate that s steps to s' either producing no output, when e is the silent event τ, or an output o.

To prove noninterference for this machine, we will need to generalize the notion of state equivalence so that it can be maintained throughout an inductive proof.

Definition 3.9. Two output events e_1 and e_2 are l-indistinguishable if they are both τ or both l-indistinguishable atoms.

Two call frames $f_1 = (w_1@l_1, rs_1)$ and $f_2 = (w_2@l_2, rs_2)$ are l-indistinguishable if $l_1 \sqsubseteq l \vee l_2 \sqsubseteq l \Rightarrow f_1 = f_2$.

Nop

$$\frac{m(w_{pc}) = \textsf{Nop}}{(m, r, w_{pc}@l_{pc}, cs) \rightarrow (m, r, (w_{pc}+1)@l_{pc}, cs)}$$

Const

$$\frac{m(w_{pc}) = \textsf{Const}\ w\ r \qquad rs[r \mapsto w@\bot] = rs'}{(m, rs, w_{pc}@l_{pc}, cs) \rightarrow (m, rs', (w_{pc}+1)@l_{pc}, cs)}$$

Mov

$$\frac{m(w_{pc}) = \textsf{Mov}\ r\ r_d \qquad rs(r) = w@l \qquad rs[r_d \mapsto w@l] = rs'}{(m, rs, w_{pc}@l_{pc}, cs) \rightarrow (m, rs', (w_{pc}+1)@l_{pc}, cs)}$$

Binop

$$\frac{\begin{array}{c} m(w_{pc}) = \textsf{Binop}_{\oplus}\ r_1\ r_2\ r_d \\ rs(r_1) = w_1@l_1 \qquad rs(r_2) = w_2@l_2 \qquad rs[r_d \mapsto (w_1 \oplus w_2)@(l_1 \sqcup l_2)] = rs' \end{array}}{(m, rs, w_{pc}@l_{pc}, cs) \rightarrow (m, rs', (w_{pc}+1)@l_{pc}, cs)}$$

Load

$$\frac{m(w_{pc}) = \textsf{Load}\ r\ r_d \qquad rs(r) = w@l \qquad m(w) = w'@l' \qquad rs[r_d \mapsto w'@(l \sqcup l')] = rs'}{(m, rs, w_{pc}@l_{pc}, cs) \rightarrow (m, rs', (w_{pc}{+}1)@t'_{pc}, cs)}$$

Store

$$\frac{\begin{array}{c} m(w_{pc}) = \textsf{Store}\ r_p\ r_s \qquad rs(r_p) = w_p@l_p \qquad rs(r_s) = w_s@l_s \qquad m(w_p) = \_@l_d \\ l_{pc} \sqcup l_p \sqsubseteq l_d \qquad m[w_p \mapsto w_s@t'_d] = m' \end{array}}{(m, rs, w_{pc}@l_{pc}, cs) \rightarrow (m', rs, (w_{pc}+1)@l_{pc}, cs)}$$

Jump

$$\frac{m(w_{pc}) = \textsf{Jump}\ r \qquad rs(r) = w'_{pc}@l}{(m, rs, w_{pc}@l_{pc}, cs) \rightarrow (m, rs, w'_{pc}@(l_{pc} \sqcup l'_{pc}), cs)}$$

Bnz

$$\frac{m(w_{pc}) = \textsf{Bnz}\ r\ n \qquad rs(r) = w@l \qquad w'_{pc} = \text{if } w = 0 \text{ then } w_{pc}+1 \text{ else } w_{pc}+n}{(m, rs, w_{pc}@l_{pc}, cs) \rightarrow (m, rs, w'_{pc}@(l_{pc} \sqcup l'_{pc}), cs)}$$

Jal

$$\frac{m(w_{pc}) = \textsf{Jal}\ r \qquad rs(r) = w'_{pc}@l_1 \qquad rs[r_a \mapsto (w_{pc}+1)@(l_{pc} \sqcup l_1)]}{(m, rs, w_{pc}@l_{pc}, cs) \rightarrow (m, rs', w'_{pc}@(l_{pc} \sqcup l_1), cs)}$$

Figure 3.4: Semantics of instructions for IFC machine

$$\frac{\begin{array}{c} m(w_{call}) = \bot \qquad rs(r_a) = pc_{caller}@l_{caller} \qquad rs(r_{arg}) = pc_{callee}@l_{callee} \\ l'_{pc} = l_{callee} \sqcup l_{caller} \sqcup l_{pc} \qquad cf = (pc_{caller}@(l_{caller} \sqcup l_{pc}), rs) \end{array}}{(m, rs, w_{call}@l_{pc}, cs) \to (m, rs, pc_{callee}@l'_{pc}, cf :: cs)}\ \textsc{Call}$$

$$\frac{m(w_{return}) = \bot \qquad rs(r_{ret}) = w_{ret}@l_{ret} \qquad rs'[r_{ret} \mapsto w_{ret}@(l_{pc} \sqcup l_{ret})] = rs''}{(m, rs, w_{return}@l_{pc}, (w'_{pc}@l'_{pc}, rs') :: cs) \to (m, rs'', w'_{pc}@l'_{pc}, cs)}\ \textsc{Return}$$

$$\frac{m(w_{output}) = \bot \qquad rs(r_{arg}) = w@l \qquad rs(r_a) = w_{pc}@l'_{pc}}{(m, rs, w_{output}@l_{pc}, cs) \to_{w@(l \sqcup l_{pc})} (m, rs, w_{pc}@(l_{pc} \sqcup l'_{pc}), cs)}\ \textsc{Output}$$

Figure 3.5: Semantics of services for IFC machine

Two call stacks cs_1 and cs_2 are l-indistinguishable if they have the same length, and at every position their call frames are l-indistinguishable. They are *weakly l-indistinguishable*, written $cs_1 \approxeq_l cs_2$, if they are l-indistinguishable after removing from them the largest prefix of call frames whose PC label is not below l; that is, if $crop(cs_1) \approx_l crop(cs_2)$, where

$$crop([]) = [] \qquad crop((pc@l_{pc}, rs) :: cs) = \begin{cases} (pc@l_{pc}, rs) :: cs & \text{if } l_{pc} \sqsubseteq l \\ crop(cs) & \text{otherwise.} \end{cases}$$

Two memory values $v_1, v_2 \in \mathsf{Instr} + \mathsf{Word} \times L$ are l-indistinguishable if they are both the same instruction, or if they are both l-indistinguishable atoms. Two memories are l-indistinguishable if they are pointwise l-indistinguishable, and similarly for two register banks.

Finally, two states $s_1 = (m_1, rs_1, w_1@l_1, cs_1)$ and $s_2 = (m_2, rs_2, w_2@l_2, cs_2)$ of the IFC abstract machine are l-indistinguishable if one of the following two conditions hold.

- $l_1 \sqsubseteq l$, $w_1@l_1 = w_2@l_2$, $m_1 \approx_l m_2$, $rs_1 \approx_l rs_2$, and $cs_1 \approx_l cs_2$; or
- $l_1 \not\sqsubseteq l$, $l_2 \not\sqsubseteq l$, $m_1 \approx_l m_2$, and $cs_1 \approxeq_l cs_2$.

Intuitively, this definition says that indistinguishable states should execute in lockstep as long as their PCs are below the level of observation l, with all their components being pairwise indistinguishable. However, as soon as their PC labels go beyond l, this condition becomes too strong to maintain: this happens when the machines execute a branching instruction that makes their next PC value depend on a secret, causing both machines to enter different execution paths. The no-sensitive-upgrade condition on stores is enough to guarantee that the memories stay indistinguishable during this execution; as there are no similar checks for the register banks, there is nothing we can guarantee for them. Finally, removing all the high frames stored on the call stack is all it takes to make the two match up.

The next set of *unwinding conditions* [43] describes the pattern that two parallel executions go through when started at indistinguishable states, and will play a crucial role in establishing noninterference by induction.

Lemma 3.10. *Let $s_1 \approx_l s_2$ be two indistinguishable states. Suppose that $s_1 \to_{e_1} s_1'$. Let l_1 and l_1' be the PC labels of s_1 and s_1'.*

1. *If $l_1 \sqsubseteq l$ and $s_2 \to_{e_2} s_2'$, then $s_1' \approx_l s_2'$ and $e_1 \approx_l e_2$.*
2. *If $l_1 \not\sqsubseteq l$ and $l_1' \not\sqsubseteq l$, then $s_1' \approx_l s_2$. Moreover, if $e = w@l_e$, then $l_e \not\sqsubseteq l$.*
3. *If $l_1 \not\sqsubseteq l$, $l_1' \sqsubseteq l$, $s_2 \to_{e_2} s_2'$, and the PC label of s_2' is below l, then $s_1' \approx_l s_2'$, and $e_1 = e_2 = \tau$.*

We can visualize these four cases in pictures; the dotted lines represent conclusions, the solid lines represent hypotheses.

(1)

$$\begin{tikzcd} s_1 \arrow[r, "e_1"] \arrow[d, dash, "\approx_l"'] & s_1' \arrow[d, dash, dashed, "\approx_l"] \\ s_2 \arrow[r, "e_2"'] & s_2' \end{tikzcd}$$

$e_1 \approx_l e_2$

(2)

$$\begin{tikzcd} s_1 \arrow[r, "e_1"] \arrow[d, dash, "\approx_l"'] & s_1' \arrow[d, dash, dashed, "\approx_l"] \\ s_2 \arrow[r, dash, "="'] & s_2 \end{tikzcd}$$

(3)

$$\begin{tikzcd} s_1 \arrow[r, "e_1"] \arrow[d, dash, "\approx_l"'] & s_1' \arrow[d, dash, dashed, "\approx_l"] \\ s_2 \arrow[r, "e_2"'] & s_2' \end{tikzcd}$$

$e_1 = e_2 = \tau$

The proof of these conditions is a straightforward (but tedious) case analysis on all possible kinds of steps that s_1 can take. With these conditions in hand, we can show that the IFC machine is noninterfering.

Lemma 3.11 (Noninterference for IFC machine)**.** *Let $s_1 \approx_l s_2$ be a pair of l-indistinguishable states of the IFC machine. Suppose that we have two executions $s_1 \rightarrow^*_{t_1} s'_1$ and $s_2 \rightarrow^*_{t_2} s'_2$, where the subscripts indicate the traces of non-silent output events produced during execution. Then $t_1 \approx_l t_2$.*

Proof. By induction on the sum of the number of steps taken by both executions, making use of Lemma 3.10 and of the fact that l-indistinguishability of states is a symmetric relation. □

To transfer this noninterference result to our IFC micro-policy, we need to relate the execution of both machine models. This can be done by simply abstracting the state of the symbolic machine into a matching IFC state.

Definition 3.12. We say that a symbolic state s_s *refines* a state s_a of the IFC machine, written $s_s \triangleright s_a$, if their register banks, PCs, and call stacks are equal, and if their memories m_s and m_a are pointwise related by the relation $\sim$ generated by the following rules (cf. Definition A.1):

$$\frac{\textsf{decode}\ i = instr}{i@\texttt{C} \sim instr} \qquad \frac{\textsf{decode}\ i = \bot}{i@\texttt{C} \sim \textsf{Halt}} \qquad \frac{}{w@\texttt{D}(l) \sim w@l}$$

(Relating words that are tagged as instructions but not decodable as such is just a trick to simplify the proof of noninterference later on, and has no major significance.)

Lemma 3.13. *Let $s_s \triangleright s_a$ be two refined states. Suppose that $s_s \rightarrow s'_s$. Then there exists a state s'_a of the IFC machine and an output event e such that $s_a \rightarrow_e s'_a$, $s'_s \triangleright s'_a$, and either*

- *$e = \tau$, and the output traces of s_s and s'_s are equal; or*
- *$e \in \textsf{Word} \times L$, and the trace of s'_s is equal to the trace of s_s with e added at the end.*

By iterating this lemma step by step, we can reconstruct an entire execution of the symbolic machine at the level of the IFC machine.

Corollary 3.14. *Let $s_s \triangleright s_a$ be two refined states. Suppose that $s_s \rightarrow^* s'_s$. There exists an output trace t such that the trace of the internal state of s'_s is obtained by adding t at the end of the corresponding component of s_s. Furthermore, there exists s'_a such that $s_a \rightarrow^*_t s'_a$.*

We have now all the ingredients we need to show our main result.

Theorem 3.15. *The symbolic machine running the IFC micro-policy has noninterference.*

Proof. Let $s_1 \approx_l s_2$ be a pair of indistinguishable states of the symbolic machine, and suppose that they yield two parallel executions $s_1 \to^* s_1'$ and $s_2 \to^* s_2'$. It is possible to construct a pair $s_{a1} \approx_l s_{a2}$ of indistinguishable states of the IFC machine such that $s_1 s_{a1}$ and $s_2 s_{a2}$. By Corollary 3.14, these executing these two states yields output traces t_1 and t_2, and these traces are exactly the traces of s_1' and s_2'. We conclude by Lemma 3.11. □

3.6 Discussion and Related Work

The IFC micro-policy presented here draws heavily from the earlier design of Hrițcu et al. [55] and its subsequent extension [56] to a richer register-based machine. The variant of noninterference used here and its proof were adapted from previous work published by me and other collaborators [9, 10].[1] Compared to these previous developments, the setup presented here uses the monitor private state to implement the protected call stack required to restore PC values, but it does not address some of the challenges these works tackled—notably, dynamic memory allocation, and the extension of the proofs of noninterference to a machine-code implementation of the policy verified with a custom Hoare logic. The introduction of dynamic allocation somewhat complicates the invariants used to prove noninterference (as seen in the work of Banerjee and Naumann [13], for instance), but this should have no major impact on proof mechanization. On the other hand, adapting the infrastructure for verifying machine code to this new setting (and, in particular, to the concrete machine described in Chapter 5) might be laborious due to the use of registers instead of a stack.

One feature we could consider adding to the policy described here is turning labels into first-class values [54, 101], allowing programs to extract the label of a value during execution, perform tests on this label (for example, "is this label below a certain threshold?"), and act depending on the result of these tests. One of our earlier verification works did feature a proof of noninterference for an abstract machine that supports public labels [56],[2] but it did not discuss how this policy could be implemented in terms of a more generic mechanism like the symbolic machine. One possible encoding would be to augment memory and register tags with elements of the form $\texttt{LABEL}(l, l')$, where l and l' are labels. The idea is that an atom with such a tag represents a label l that has itself

[1]The Coq proofs of the original work on verified IFC are available at `https://github.com/micro-policies/verified-ifc`.

[2]The Coq proofs are available at `https://github.com/QuickChick/IFC`.

an associated IFC label l'—a recurrent theme in systems supporting this feature. The payload of atoms with such tags would simply be ignored. The policy could then expose the primitives for manipulating labels as system services. Because first-class labels are stored in tag space (which cannot be arbitrarily accessed by user code), this scheme keeps the representation of tags opaque, which might be needed for avoiding information leaks in a system with dynamically generated labels. (As we will see in Chapter 4, such an issue is also crucial for guaranteeing confidentiality through memory safety.)

The variant of noninterference proved here protects code against illicit explicit and implicit information flows, but leaves open the possibility of leaks through side channels such as timing or energy consumption. Dealing with timing channels is potentially difficult in the context of the PUMP, since caching the transfer function might exacerbate the effects of secrets on these side channels. If such leaks are a practical concern, we could resort to quantitative variants of information-flow control to try to contain them [12, 98, 100, 116].

It was often claimed in the literature that termination-insensitive noninterference can leak at most one bit of secret information during execution. Askarov et al. [5] note that this guarantee applies only to batch-processing systems that do not produce intermediate outputs. In a mechanism like ours, where outputs are possible and the level of the PC can be lowered after inspecting a secret, an attacker can perform slightly better with a brute-force search, as follows.

```
i = 0;
while (1) {
  output(i);
  if (i == secret) loop();
  i = i + 1;
}
```

Nevertheless, Askarov et al. prove this is the best option the attack has, in the sense that reliably leaking a uniformly distributed secret of some size—for example, a private key—takes more than polynomial time in the size of the secret. The presence of concurrency complicates things [52], since a direct adaptation of our mechanism might make it possible to leak a secret in linear time, as the following example suggests:

```
test(bit, i) {
  if (secret[i] != bit) loop();
  output(i);
  fork(test(true, i + 1));
  fork(test(false, i + 1));
}

fork(test(true, 0));
fork(test(false, 0));
```

At its core, the issue results from lowering PC labels after computing on sensitive data. Concurrent LIO [99] attacks the problem by never restoring PC labels; to prevent label creep, users should fork new threads to manipulate sensitive information. Its approach seems worth considering, as LIO has been used in a number of programs, including web frameworks and applications.

The seL4 micro kernel [68] features a proof of information-flow security checked in the Isabelle/HOL proof assistant [81, 82]. Instead of the noninterference property studied here, they consider a variant of the finer notion of *intransitive noninterference* [92]. In the context of seL4, this property says that information can only flow between system partitions through a fixed list of communication channels that is set up once the system starts running. The proof of their property follows a strategy similar to ours: they first show intransitive noninterference to an abstract specification of the system, and then transfer this result by refinement to a more concrete implementation—in their case, C code.

Some formulations of noninterference are subject to the *refinement paradox* [59], where the abstract specification of a system may satisfy noninterference even though its refinements do not. This problem typically arises when the refinement removes some of the non-determinism present in the specification, allowing one to learn more about the state of a system by observing its execution. The refinement paradox does not apply in our case because the machines we consider are deterministic. Even though seL4 has a non deterministic specification, their proof also manages to avoid the refinement paradox by preventing sources of nondeterminism from being observed by partitions.

PROSPER [22] is a separation kernel for ARMv7 that was formally verified with HOL4. Unlike the verification efforts mentioned here, the final correctness result for PROSPER is not noninterference, but a trace equivalence between the machine code implementation of the system and a more abstract specification. In this specification, the system consists of two ARMv7 machines that run separate partitions communicating through asynchronous message passing. The rationale behind their choice is that, since message passing is the only mechanism of communication in the model, the same should hold for the concrete implementation. As a sanity check, they show a result asserting that, when running partitions that do not send messages, the partitions do not interference with each other.

Chapter 4

Heap Memory Safety

Many security holes in today's software arise from low-level memory errors such as buffer overflows and double frees in C and C++ [105]. They are hard to spot, easy to introduce inadvertently, and tend to break subtle invariants that programmers might hope to hold of their code. Even simple, intuitive desiderata for a program, such as preventing a function from accessing the local variables of its callers, cannot be enforced without inspecting our code in excruciating detail to rule out such errors.

Preventing and controlling memory errors is far from trivial. Simple solutions exist—for example, programming in memory-safe languages like OCaml, Haskell, Java, JavaScript, or Python—but they can break common low-level coding idioms, suffer from compatibility problems, force us to reimplement legacy software, and—worst of all—be prohibitively expensive in terms of performance for programs that make heavy use of the memory. Cheaper, more permissive alternatives for memory safety can thus have far-reaching consequences for the security of our software infrastructure, especially if they apply to legacy programs written in C or C++.

Practical challenges often lead to the development of security tools that trade in security guarantees for returns in run-time performance and other benefits. In the case of memory safety, some tools might choose to prevent only temporal violations [83], or might not try to detect overflows that occur between fields of an object, among other concessions. We have argued that a good method for evaluating this trade off would be to formulate the security guarantees that we care about as formal reasoning principles that can be applied to programs that conform to a policy. Unfortunately, there is no consensus on what these reasoning principles would be for memory safety. Popular characterizations of the concept are usually stated as the absence of a certain class of low-level errors during program execution. The Wikipedia article on the subject, for example, defines memory safety as "the state of being protected from various software bugs and security vulnerabilities when dealing with memory accesses" [112], a definition that appears in one form or another in many formal investigations of memory safety [72, 84]. Although conceptually simple and intuitive, some researchers have felt that this picture is incomplete from a formal perspective [53]. We believe this is due to a few reasons. First, there is often disagreement about which behaviors qualify as errors. For example, many C programs rely on unrestricted pointer arithmetic [76], even though it may lead to undefined behavior according to the language standard [58, §6.5.6]. Second, from the point of view of security, the problem is not the errors themselves, but rather the erratic behavior that they induce when they occur in unsafe languages like C. Indeed, in settings that are usually seen as memory safe, like Java, it is possible for programs to run into errors such as accessing an array out of bounds; however, instead of wild undefined behavior, such actions have a well-behaved, sensible semantics. Finally, knowing what low-level errors are prevented alone does not provide much insight into what kinds of threats a program is immune to. It allows us to rule out attacks that have already been documented, but it might leave the door open to other threads based on slightly different techniques that manage to bypass the mechanism. This gap between our intuitions about memory errors and the security guarantees of memory safety may help explain why it remains so elusive: though memory protection mechanisms keep coming up, many of them are quickly defeated by new, ingenious attack strategies.

The first goal of this chapter is to present a micro-policy for memory safety. It is based on a scheme originally due to Clause et al. [19], which uses tags on memory locations to track which memory region they belong to, and that has been subsequently expanded and ported to the PUMP by Dhawan et al. [33]. We prove that this micro-policy allows the symbolic machine to implement (that is, refine) a higher-level abstract machine that has memory-safety checks built in.

The second goal of this chapter is to propose a new characterization of memory safety, and show that our micro-policy conforms to it. Our characterization is based on the observation that typical memory errors arise when a pointer is used to access a resource that it was not meant to access, either because it lies outside of its bounds, or because its region has been freed. As has been noted previously, this discipline ensures that pointers in a memory-safe language behave as *capabilities* [53]. Thus, if memory errors in a piece of code are prevented or controlled, we can conclude that its execution is independent of memory regions it cannot access through the pointers it possesses. We formalize this idea as a *noninterference* result that we prove for both the symbolic machine running the memory-safety policy and its higher-level counterpart (Theorems 4.12 and 4.13). We show how these results carry over to a setting that is more familiar to programmers, a simple imperative language with dynamic allocation, and we use this setting to relate our characterization of memory safety to the reasoning principles of separation logic [90], a proof system for heap-manipulating programs. Finally, to demonstrate the generality of our characterization, we discuss how it is affected by common pragmatically motivated relaxations of the memory safety.

4.1 Policy Tags

We begin by describing the memory-safety micro-policy that forms the base of our discussion. The idea is to use tags to track which pointers may access which memory regions, and which region each memory location belongs to. When a memory access occurs, the policy checks that the pointer tag allows it to access the region corresponding to that location; if not, a fatal run-time error is triggered.

The micro-policy is based on two sets of tags: data and memory tags.

$$\mathsf{Tag}_r = \mathsf{Tag}_{pc} = \mathsf{Data} = \mathtt{NPTR} \mid \mathtt{PTR}(i \in \mathsf{Id})$$

$$\mathsf{Tag}_m = \mathtt{FREE} \mid \mathtt{D}(i \in \mathsf{Id}, t \in \mathsf{Data})$$

Here, Id is a set of *identifiers* used to distinguish memory regions allocated at different times; we assume it to be countably infinite.[3] We order Id according to some fixed enumeration of its elements, and note nextId the function that, given an identifier, returns the next identifier on that enumeration.

The Data tags attached to the PC and registers are used to distinguish pointer values from everything else; a word tagged as $\mathtt{PTR}(i)$ is a pointer that is only allowed to access region i, whereas everything else is grouped under the $\mathtt{NPTR}$ tag.

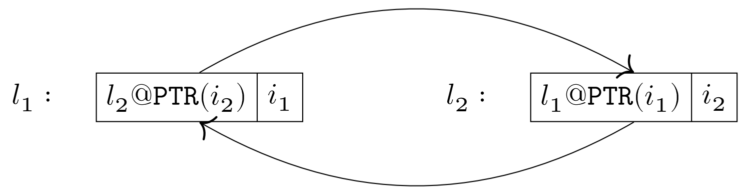
Memory tags are more complex. The $\mathtt{FREE}$ tag denotes a location that does not belong to any allocated region, whereas $\mathtt{D}(i,t)$ denotes a location that belongs to region i, and that furthermore currently stores a word described by the data tag t. A tag of the form $\mathtt{D}(i_1, \mathtt{PTR}(i_2))$, for example, represents a location of region i_1 that stores a pointer to a potentially different region i_2. As shown later, the allocator creates a new memory region by choosing a contiguous segment of memory

[3] This assumption is not strictly necessary, and does not appear in the Coq formalization, but it simplifies the memory-safe abstract machine introduced later, and the refinement result between it and the symbolic machine.

Opcode	PC	Instr	A1	A2	Dest	PC	Res
Nop	PTR(i_m)	D(i_m, NPTR)				PTR(i_m)	
Const	PTR(i_m)	D(i_m, NPTR)			_	PTR(i_m)	NPTR
Mov	PTR(i_m)	D(i_m, NPTR)	t		_	PTR(i_m)	t
Binop$_{\oplus}$	PTR(i_m)	D(i_m, NPTR)	NPTR	NPTR	_	PTR(i_m)	NPTR
Binop$_{+}$	PTR(i_m)	D(i_m, NPTR)	PTR(i)	NPTR	_	PTR(i_m)	PTR(i)
Binop$_{+}$	PTR(i_m)	D(i_m, NPTR)	NPTR	PTR(i)	_	PTR(i_m)	PTR(i)
Binop$_{-}$	PTR(i_m)	D(i_m, NPTR)	PTR(i)	NPTR	_	PTR(i_m)	PTR(i)
Binop$_{-}$	PTR(i_m)	D(i_m, NPTR)	PTR(i)	PTR(i)	_	PTR(i_m)	NPTR
Binop$_{=}$	PTR(i_m)	D(i_m, NPTR)	PTR(i)	PTR(i)	_	PTR(i_m)	NPTR
Load	PTR(i_m)	D(i_m, NPTR)	PTR(i)	D(i, t)	_	PTR(i_m)	t
Store	PTR(i_m)	D(i_m, NPTR)	PTR(i)	t	D(i, _)	PTR(i_m)	D(i, t)
Jump	PTR(i_m)	D(i_m, NPTR)	PTR(i)			PTR(i)	
Bnz	PTR(i_m)	D(i_m, NPTR)	NPTR			PTR(i_m)	
Jal	PTR(i_m)	D(i_m, NPTR)	t		_	t	PTR(i_m)

Figure 4.1: Memory-safety transfer function

tagged as FREE and changing its tags to D(i, NPTR), where i is some fresh identifier that has not been assigned to any currently existing region. Because of the simple type structure, nothing prevents the memory from storing cyclic data structures. Consider for instance the following situation: a memory location l_1 contains a pointer that references another location l_2, and vice versa. We could depict this situation as follows, assuming that these locations belong to regions marked as i_1 and i_2.



To represent such a cycle, it suffices to store l_2@D(i_1, PTR(i_2)) in l_1, and l_1@D(i_2, PTR(i_1)) in l_2.

Figure 4.1 defines the transfer function for this policy. It is worth taking some time to relate the rules shown there to checks commonly present in various systems that enforce memory safety.

The most basic property that this policy tries to enforce is that a memory access can only occur if performed with a pointer that has the appropriate permission to do so. The rules for Load and Store require (1) that the pointer argument supplied to the function is indeed tagged as a pointer, and (2) that the memory location being accessed is tagged as being currently allocated in a region whose identifier matches the pointer's. If those checks pass, the operation is allowed; otherwise, the policy signals that a violation occurred. Similarly, every instruction checks that the current PC is tagged as a pointer, and that the instruction that it is currently being executed belongs to a matching region.

Violations of these checks occur when we try to access memory using invalid pointers—that is, those that point past the bounds of their regions, or dangling pointers that reference regions that have already been freed. When explaining the behavior of the memory manager, we will see that freeing a memory region has the effect of changing all of its tags to FREE. Since no rule allows manipulating locations tagged this way, any subsequent access to that region will be caught and stopped. As for out-of-bounds accesses, they arise (as it is usually the case) through pointer arithmetic, which is allowed by the policy. The rule for addition, for example, allows us to add an integer to a pointer, the result of which is a pointer to the same memory region as before. Similarly, it is possible to subtract an integer from a pointer. Note that moving a pointer past its bounds is not an error by itself—in opposition to the C standard, for instance [58, §6.5.6]. However, if we try to access memory with such an invalid pointer, we will hit a memory location marked as free, or perhaps as belonging to a different memory region, leading the policy to signal a violation.

For these checks to make sense, the policy imposes a basic type discipline that completely separates pointers from other kinds of values. First, it is not possible to convert an arbitrary integer value into a pointer by changing its tag. All values tagged as pointers are obtained by performing operations on previously existing pointer values, and these operations preserve the region identifiers attached to them. We have seen that pointer arithmetic has this effect. Any instruction sets the tag on the next PC to match either the one on the previous PC, or the one on the target of a jump instruction. Finally, moving values between registers and memory locations preserves tags as well. The only exception is the allocation system service presented later, which has the privilege to fabricate pointers to previously unallocated regions. It is also not possible to change the region of a memory location, as we can see by inspecting the rule for Store, the only instruction of the symbolic machine that can change those tags.

Another aspect of our type distinction is that many integer operations are disallowed on pointers. This includes most binary operations, with the exception of addition, subtraction, and equality comparisons, which allow only a select number of well-behaved uses; for instance, we are cannot add two pointers together, or to test if pointers to different regions are equal. Moreover, we are not allowed to run pointers as instructions, as we can see by inspecting the "Instr" column of the rules. The reason for enforcing these restrictions is that treating pointers as integers may reveal information about their physical address, which could result in secrecy violations—we return to this point in Section 4.7.2. Note that the rules allow subtracting two pointers to the *same* memory region, as this only reveals the relative placement of the pointers, and nothing about their physical addresses.

Some of restrictions described here can be partially lifted: for example, by encapsulating equality tests as a system service, it is possible to compare pointers to different regions (which is, unfortunately, more costly, since it requires a function call instead of a single instruction).

4.2 System Services

The memory safety policy has four system services. Two of them, malloc and free, are named after the corresponding C functions, and used to allocate and free memory.

A base service is used to reset the offset of a pointer to its base. Though somewhat exotic, this service illustrates what kinds of operations can be performed on pointer values without giving up on the security guarantees analyzed later. We can use base and pointer subtraction to extract the offset of a pointer relative to its base as an integer; conversely, if we had a service for extracting offsets, it would be possible to encode base by subtracting a pointer's offset from the pointer itself. (The language of Section 4.5 explores this possibility.)

Finally, there is an eq service for soundly testing any two values for equality, including pointers to different memory regions. The crucial difference between this service and the analogous machine instruction is that the service is free to take region identifiers into account when comparing two pointers, whereas the instruction can only consider their physical addresses.

To allow programs to invoke system services, the rules for the Jal instruction do not force its destination to be a pointer. Programs can thus hard-wire the addresses of these services in their code, without the need for a special mechanism to load tagged pointers to these services in registers.

Real implementations of memory managers use auxiliary data structures to track which parts of memory are currently allocated and which ones are free. Given the generality of system services, it would have been possible for us to have a much higher-level algorithm that assigns new regions by sweeping the entire memory, looking for a segment of the appropriate size that is tagged `FREE`; however, this would be too inefficient to be adopted in practice. We chose to instead to stick with a more realistic design that tracks the status of memory regions with an explicit list of block descriptors.

Definition 4.1. The private state of the memory safety policy is a pair (i, bs), where $i \in \mathsf{Id}$ is the

Malloc

$$\frac{\begin{array}{c} 0 < w_{sz} \leq w'_{sz} \qquad rs(r_{arg1}) = w_{sz}@\mathtt{NPTR} \\ bs = bs_1 \cdot [(\bot, w_{bs}, w'_{sz})] \cdot bs_2 \qquad bs_1 \text{ minimal} \\ bs' = w_{sz} < w'_{sz} ? [(i, w_{bs}, w_{sz}); (\bot, w_{bs} + w_{sz}, w'_{sz} - w_{sz})] : [(i, w_{bs}, w_{sz})] \\ m[w_{bs}, ..., w_{bs} + w_{sz} - 1 \mapsto 0@\mathtt{D}(i, \mathtt{NPTR})] = m' \\ rs[r_{ret} \mapsto w_{bs}@\mathtt{PTR}(i)] = rs' \qquad r(r_a) = w_{pc}@\mathtt{PTR}(i_{pc}) \end{array}}{\mathsf{malloc}(m, rs, \_, i, bs) = (m', rs', w_{pc}@\mathtt{PTR}(i_{pc}), \mathsf{nextId}(i), bs_1 \cdot bs' \cdot bs_2)}$$

Free

$$\frac{\begin{array}{c} rs(r_{arg1}) = w@\mathtt{PTR}(i) \qquad bs(k) = (i, w_{bs}, w_{sz}) \qquad w_{bs} \leq w < w_{bs} + w_{sz} \\ m[w_{bs}, ..., w_{bs} + w_{sz} - 1 \mapsto 0@\mathtt{FREE}] = m' \\ bs[k \mapsto (\bot, w_{bs}, w_{sz})] = bs' \qquad rs(r_a) = w_{pc}@\mathtt{PTR}(i_{pc}) \end{array}}{\mathsf{free}(m, rs, \_, i', bs) = (m', rs, w_{pc}@\mathtt{PTR}(i_{pc}), i', bs')}$$

Base

$$\frac{\begin{array}{c} rs(r_{arg1}) = w@\mathtt{PTR}(i) \qquad (i, w_{bs}, \_) \in bs \qquad rs[r_{ret} \mapsto w_{bs}@\mathtt{PTR}(i)] = rs' \\ rs(r_a) = w_{pc}@\mathtt{PTR}(i_{pc}) \end{array}}{\mathsf{base}(m, rs, \_, i', bs) = (m, rs', w_{pc}@\mathtt{PTR}(i_{pc}), i', bs)}$$

Eq

$$\frac{\begin{array}{c} rs(r_{arg1}) = w_1@t_1 \qquad rs(r_{arg2}) = w_2@t_2 \qquad b = \text{if } w_1@t_1 = w_2@t_2 \text{ then } 1 \text{ else } 0 \\ rs[r_{ret} \mapsto b@\mathtt{NPTR}] = rs' \qquad rs(r_a) = w_{pc}@\mathtt{PTR}(i_{pc}) \end{array}}{\mathsf{eq}(m, rs, \_, i, bs) = (m, rs', w_{pc}@\mathtt{PTR}(i_{pc}), i, bs)}$$

Figure 4.2: System services for memory safety policy

first identifier that has not been assigned to a block, and bs is a list of *block descriptors*, which are elements of $(\mathsf{Id} \uplus \{\bot\}) \times \mathsf{Word} \times \mathsf{Word}$.

A block descriptor of the form $(\bot, w_{bs}, w_{sz})$ indicates that there is a free segment of memory of size w_{sz} starting at address w_{bs}. A block of the form (i, w_{bs}, w_{sz}), on the other hand, indicates that the segment of size w_{sz} starting at w_{bs} is allocated, and currently associated with the identifier i.

Figure 4.2 defines the system services. The most complex one, malloc, reads the size of the block to be allocated (w_{sz}), and looks for the first block descriptor that is currently marked as free and whose size w'_{sz} is large enough to fit the new block. If both sizes match ($w_{sz} = w'_{sz}$), we simply mark that block as allocated, assigning to it the fresh identifier i. If they don't match ($w_{sz} < w'_{sz}$), then malloc splits the block in two parts: the lower one, of size w_{sz}, and the upper one, of size $w'_{sz} - w_{sz}$. The new block is then initialized to $0@\mathtt{D}(i, \mathtt{NPTR})$, indicating that each of its positions belongs to block i, and we bump the next identifier in the private state. Finally, malloc stores a pointer to the beginning of the new block in r_{ret}, and returns to its caller. By keeping the invariant that the identifier i stored in the private state is "bigger" (that is, comes later in the enumeration of identifiers) than any other identifiers in the program state, we guarantee that malloc always produces a memory region that cannot be accessed with other available pointers. Furthermore, a piece of code that allocates a new memory region cannot use the pointer received from malloc to access other regions. We return to this point in Section 4.7.4.

The free service does the opposite of malloc: it finds the block descriptor that matches the identifier of its argument, marks it as free, and tags the contents of the block accordingly. Note, however, that free does not try to prevent fragmentation by merging consecutive free blocks. This would be an important optimization in practice, but does not affect the security guarantees discussed later.

4.3 Memory-safe Machine

Many of the restrictions that we have imposed on the use of memory thus far—preventing pointer forging, out-of-bounds accesses, etc.—are commonplace, and reflected in other mechanisms for enforcing memory safety. Nevertheless, the definition of the policy is complex, especially in what concerns the memory management services. We want to show that we can replace the memory-safety policy by a simpler, easier-to-understand abstract machine that inlines its checks, obviates the need for some of its tags, and uses a much higher-level allocator. Specifically, we prove a refinement result, Theorem 4.8, showing that every execution of the symbolic machine with the memory-safety micro-policy can be interpreted in terms of the abstract machine. Later, in Section 4.4, we prove that the abstract machine satisfies a form of noninterference, and use the refinement result to transfer noninterference to the symbolic machine.

Unlike the symbolic machine, where memory safety is programmed as a policy, the abstract machine has memory safety built in. It distinguishes pointers from other kinds of data at the hardware level, and has a structured memory segmented into independent blocks, as opposed to a flat memory like the symbolic machine does.

Definition 4.2. We define the set of values Val as

$$\mathsf{Val} = \mathsf{NPtr}(w \in \mathsf{Word}) \mid \mathsf{Ptr}(i \in \mathsf{Id}, w \in \mathsf{Word}).$$

A state of the memory-safe machine is a tuple (m, rs, pc), where

- $m \in \mathsf{Id} \rightharpoonup_{\mathrm{fin}} \mathsf{List}(\mathsf{Val})$;
- $rs \in \mathsf{Reg} \rightharpoonup \mathsf{Val}$; and
- $pc \in \mathsf{Val}$.

Values of the memory-safe machine are the analog of words tagged with Data tags in the memory-safety policy: they are either words $\mathsf{NPtr}(w)$ or a pair $\mathsf{Ptr}(i, w)$ of a block identifier and a word denoting an offset into that block. Like its policy counterpart, this machine distinguishes between independent memory regions by assigning them to different identifiers. However, here the identifiers are used for fetching the block in memory, instead of just being used to check the access was valid afterwards: to index into a block i at offset w, we check if $m(i)$ is defined, and if so look for the w-th position of its resulting frame. In this way, every such memory defines a partial function from $\mathsf{Id} \times \mathsf{Word}$ to Val, and we implicitly treat memories as such to simplify notations—for example, writing $m(i, w)$ to denote the above access. Accessing memory using out-of-bounds or freed pointers causes the machine to look up a location that is not in the domain of the partial function, which we will use to detect when memory-safety violations occur.

Since there are infinitely many identifiers, the memory in this machine can grow indefinitely, and allocation is a total operation. This is similar to the many memory models used in programming languages [27, 64, 74], but very different from the symbolic machine, which has only finitely many bit vectors to use as addresses and might run out of memory when executing. We will see that this difference has consequences for the security guarantees of the two computation models, since memory exhaustion can be used to leak secrets about the state of the machine.

Figures 4.3 and 4.4 define the machine semantics. The rule for Binop applies a binary operation to its arguments, but each binary operation is refined here into a partial operation that takes pointer values into account, integrating the checks of Figure 4.1:

$$\mathsf{NPtr}(w_1) + \mathsf{Ptr}(i, w_2) = \mathsf{Ptr}(i, w_1) + \mathsf{NPtr}(w_2) = \mathsf{Ptr}(i, w_1 + w_2)$$

$$\mathsf{Ptr}(i, w_1) - \mathsf{NPtr}(w_2) = \mathsf{Ptr}(i, w_1 - w_2)$$

$$\mathsf{Ptr}(i, w_1) - \mathsf{Ptr}(i, w_2) = \mathsf{NPtr}(w_1 - w_2)$$

$$(\mathsf{Ptr}(i, w_1) = \mathsf{Ptr}(i, w_2)) = \mathsf{NPtr}(w_1 = w_2)$$

The cases that combine numeric operands are defined by simply applying the corresponding operation, and all others—that is, those that have at least one pointer as their operand—fail.

In contrast to the allocator of the symbolic machine, which tries to be more realistic, the memory-safe machine uses a more abstract design that simply chooses a new block name by traversing the entire machine state, and then assigns a new frame to that identifier in memory (noted $m' = m \cup \{(i,k) \mapsto \mathsf{NPtr}(0) \mid k = 0, ..., sz-1\}$). In that rule, fresh refers to some function that maps a finite set of identifiers I to an identifier i that does not occur in I, and $\mathsf{supp}(s)$ (the *support* of s) is the set of identifiers that appears in the state s. We can characterize the support formally by seeing the set of states as a nominal set over Id [41]; these concepts are briefly reviewed in Appendix A.2. Note that all blocks begin at offset 0, whereas in the symbolic machine they usually begin at a non-zero address that points to the middle of the memory.

By avoiding the explicit manipulation of the auxiliary data structures used by the micro-policy, the memory-safe machine becomes much easier to understand. Unfortunately, this convenience comes with a price: proving that this machine is refined by the symbolic one is a daunting task. First, we need to show that the internal state of the memory-safety policy correctly tracks which regions are free. Second, since the two machines use different addressing schemes, we need to keep track of what addresses correspond to each other through a simplified form of *memory injections* [75].

Definition 4.3. A memory injection is a finite partial function $mi \in \mathsf{Id} \rightharpoonup_{\mathrm{fin}} \mathsf{Id} \times \mathsf{Word}$ such that $mi(i_1) = (i, w_1)$ and $mi(i_2) = (i, w_2)$ implies $i_1 = i_2$.

Memory injections map each identifier used to mark a block in a state of the symbolic machine to (1) its corresponding identifier in the abstract machine, and (2) the base of the corresponding block. We use them to define when values of the two machines correspond to each other, by simply adjusting the offsets and identifiers of pointers.

Definition 4.4. Let $v \in \mathsf{Val}$, $w \in \mathsf{Word}$, $t \in \mathsf{Data}$, and mi be a memory injection. We say that the atom $w@t$ refines v with respect to mi, noted $w@t \rhd_{mi} v$, if one of the following hold.

- $t = \texttt{NPTR}$ and $v = \mathsf{NPtr}(w)$; or
- $t = \texttt{PTR}(i)$, $mi(i) = (i', w_b)$, $v = \mathsf{Ptr}(i', w_b + w)$.

Since addresses on both machines do not correspond to each other anymore, we need a more complex notion of refinement for memories, instead of just pointwise correspondence.

Nop

$$\frac{m(i_{pc}, w_{pc}) = \mathsf{NPtr}(w_i) \qquad \mathsf{decode}\ w_i = \mathsf{Nop}}{(m, r, \mathsf{Ptr}(i_{pc}, w_{pc})) \to (m, r, \mathsf{Ptr}(i_{pc}, w_{pc} + 1))}$$

Const

$$\frac{m(i_{pc}, w_{pc}) = \mathsf{NPtr}(w_i) \qquad \mathsf{decode}\ w_i = \mathsf{Const}\ w\ r \qquad rs[r \mapsto \mathsf{NPtr}(w)] = rs'}{(m, rs, \mathsf{Ptr}(i_{pc}, w_{pc})) \to (m, rs', \mathsf{Ptr}(i_{pc}, w_{pc} + 1))}$$

Mov

$$\frac{m(i_{pc}, w_{pc}) = \mathsf{NPtr}(w_i) \qquad \mathsf{decode}\ w_i = \mathsf{Mov}\ r\ r_d \qquad rs(r) = v \qquad rs[r_d \mapsto v] = rs'}{(m, rs, \mathsf{Ptr}(i_{pc}, w_{pc})) \to (m, rs', \mathsf{Ptr}(i_{pc}, w_{pc} + 1))}$$

Binop

$$\frac{\begin{array}{c} m(i_{pc}, w_{pc}) = \mathsf{NPtr}(w_i) \qquad \mathsf{decode}\ w_i = \mathsf{Binop}_{\oplus}\ r_1\ r_2\ r_d \\ rs(r_1) = v_1 \qquad rs(r_2) = v_2 \qquad rs[r_d \mapsto v_1 \oplus v_2] = rs' \end{array}}{(m, rs, \mathsf{Ptr}(i_{pc}, w_{pc}) \to (m, rs', \mathsf{Ptr}(i_{pc}, w_{pc} + 1))}$$

Load

$$\frac{\begin{array}{c} m(i_{pc}, w_{pc}) = \mathsf{NPtr}(w_i) \qquad \mathsf{decode}\ w_i = \mathsf{Load}\ r\ r_d \\ rs(r) = \mathsf{Ptr}(p) \qquad m(p) = v \qquad rs[r_d \mapsto v] = rs' \end{array}}{(m, rs, \mathsf{Ptr}(i_{pc}, w_{pc})) \to (m, rs', \mathsf{Ptr}(i_{pc}, w_{pc} + 1))}$$

Store

$$\frac{\begin{array}{c} m(i_{pc}, w_{pc}) = \mathsf{NPtr}(w_i) \qquad \mathsf{decode}\ w_i = \mathsf{Store}\ r_p\ r_s \\ rs(r_p) = \mathsf{Ptr}(p) \qquad rs(r_s) = v \qquad m[p \mapsto v] = m' \end{array}}{(m, rs, \mathsf{Ptr}(i_{pc}, w_{pc})) \to (m', rs, \mathsf{Ptr}(i_{pc}, w_{pc} + 1))}$$

Jump

$$\frac{m(i_{pc}, w_{pc}) = \mathsf{NPtr}(w_i) \qquad \mathsf{decode}\ w_i = \mathsf{Jump}\ r \qquad rs(r) = \mathsf{Ptr}(p)}{(m, rs, \mathsf{Ptr}(i_{pc}, w_{pc}), e) \to (m, rs, \mathsf{Ptr}(p))}$$

Bnz

$$\frac{\begin{array}{c} m(i_{pc}, w_{pc}) = \mathsf{NPtr}(w_i) \qquad \mathsf{decode}\ w_i = \mathsf{Bnz}\ r\ n \\ rs(r) = \mathsf{NPtr}(w) \qquad w'_{pc} = \text{if } w = 0 \text{ then } w_{pc} + 1 \text{ else } w_{pc} + n \end{array}}{(m, rs, \mathsf{Ptr}(i_{pc}, w_{pc})) \to (m, rs, \mathsf{Ptr}(i_{pc}, w'_{pc}))}$$

Jal

$$\frac{\begin{array}{c} m(i_{pc}, w_{pc}) = \mathsf{NPtr}(w_i) \qquad \mathsf{decode}\ w_i = \mathsf{Jal}\ r \\ rs(r) = v \qquad rs[r_a \mapsto \mathsf{Ptr}(i_{pc}, w_{pc} + 1)] \end{array}}{(m, rs, \mathsf{Ptr}(i_{pc}, w_{pc})) \to (m, rs, v)}$$

Figure 4.3: Semantics of instructions for the memory-safe machine

$$\frac{\begin{array}{c}rs(r_{arg1}) = \mathsf{NPtr}(w) \qquad i = \mathsf{fresh}(\mathsf{supp}(m, rs, \mathsf{NPtr}(w_{malloc}))) \\ m' = m \cup \{(i,k) \mapsto \mathsf{NPtr}(0) \mid k = 0, ..., sz-1\} \\ rs[r_{ret} \mapsto \mathsf{Ptr}(i,0)] = rs' \qquad rs(r_a) = \mathsf{Ptr}(p)\end{array}}{(m, rs, \mathsf{NPtr}(w_{malloc})) \to (m', rs', \mathsf{Ptr}(p))}\ \textsc{Malloc}$$

$$\frac{\begin{array}{c}rs(r_{arg1}) = \mathsf{Ptr}(i,w) \qquad m(i) \neq \bot \qquad m' = \{(i',w') \mapsto m(i',w') \mid i' \neq i\} \\ rs(r_a) = \mathsf{Ptr}(p)\end{array}}{(m, rs, \mathsf{NPtr}(w_{free})) \to (m', rs, \mathsf{Ptr}(p))}\ \textsc{Free}$$

$$\frac{rs(r_{arg1}) = \mathsf{Ptr}(i,w) \qquad rs[r_{ret} \mapsto \mathsf{Ptr}(i,0)] = rs' \qquad rs(r_a) = \mathsf{Ptr}(p)}{(m, rs, \mathsf{NPtr}(w_{base})) \to (m, rs', \mathsf{Ptr}(p))}\ \textsc{Base}$$

$$\frac{\begin{array}{c}rs(r_{arg1}) = v_1 \qquad rs(r_{arg2}) = v_2 \qquad v = \mathsf{NPtr}(\text{if } v_1 = v_2 \text{ then } 1 \text{ else } 0) \\ rs[r_{ret} \mapsto v] = rs' \qquad rs(r_a) = \mathsf{Ptr}(p)\end{array}}{(m, rs, \mathsf{NPtr}(w_{eq})) \to (m, rs', \mathsf{Ptr}(p))}\ \textsc{Eq}$$

Figure 4.4: Semantics of system services for the memory-safe machine

Definition 4.5. Let mi be a memory injection, $m_a \in \mathsf{Id} \rightharpoonup_{\text{fin}} \mathsf{Val}$ be a memory of the memory-safe machine, and $m_s \in \mathsf{Word} \rightharpoonup \mathsf{Word} \times \mathsf{Tag}_m$ be a memory of the symbolic machine instantiated with the memory-safety policy. We say that m_s refines m_a with respect to mi, noted $m_s \rhd_{mi} m_a$, if the following condition holds. Suppose that $m_s(w_1) = w_2 @ \mathtt{D}(i,t)$ for $w_1, w_2 \in \mathsf{Word}$, $i \in \mathsf{Id}$, and $t \in \mathsf{Data}$. Then there exists an identifier i', a base $w \in \mathsf{Word}$, and $v \in \mathsf{Val}$ such that

- $mi(i) = (i', w)$;
- $m_a(i', w_1 - w) = v$; and
- $w_2 @ t \rhd_{mi} v$.

Paraphrasing, this definition says that every allocated address in the memory of the symbolic machine matches a location in the memory of the abstract machine, and that this matching is computed by offsetting the address according to the memory injection. Note that this only implies that blocks in the symbolic state have a corresponding block in the state of the memory-safe machine whose size is at least as large as its own. It would be possible to strengthen this invariant to require that their sizes be equal, which would allow us to include a system service for computing the size of a memory block.

To prove that the use of the internal state by the symbolic machine is valid, we need to ensure that the information stored there accurately describes the tags in memory. This leads to the following definition.

Definition 4.6. Let $m \in \mathsf{Word} \rightharpoonup \mathsf{Word} \times \mathsf{Tag}_m$ be a symbolic memory, and mi be a memory injection.

A block descriptor $b = (x, w_{bs}, w_{sz})$ is *well-formed* with respect to m and mi if $w_{sz} \neq 0$, $w_{bs} + w_{sz}$ does not overflow, and one of the following conditions hold.

- x is some identifier i, $mi(i) = (i', w_{bs})$, and for every $o \in \mathsf{Word}$, if $o < w_{sz}$, then there exists $w \in \mathsf{Word}$ and $t \in \mathsf{Data}$ such that $m(w_{bs} + o) = w@\mathtt{D}(i, t)$.

- $x = \bot$, and for every $o \in \mathsf{Word}$, if $o < w_{sz}$, then there exists $w \in \mathsf{Word}$ such that $m(w_{bs} + o) = w@\mathtt{FREE}$.

An internal state (i, bs) is well-formed with respect to m and mi if the following conditions hold.

Freshness $i > i'$ whenever $mi(i') \neq \bot$.

No overlap If two block descriptors $b_1 = (\_, w_{bs1}, w_{sz1})$ and $b_2 = (\_, w_{bs2}, w_{sz2})$ in bs have overlapping addresses (that is, if $w_{bs2} \leq w_{bs1} < w_{bs2} + w_{sz2}$ or $w_{bs1} \leq w_{bs2} < w_{bs1} + w_{sz1}$), then $b_1 = b_2$. Furthermore, bs contains no repetitions.

Covering If $m(w) \neq \bot$, there exists a block descriptor $(\_, w_{bs}, w_{sz}) \in bs$ such that $w_{bs} \leq w < w_{bs} + w_{sz}$.

Well-formedness Every block in bs is well-formed with respect to m and mi.

The freshness condition guarantees that the identifier stored in the internal state is indeed fresh. We can now show that the symbolic machine refines the memory-safe one when running the memory-safety policy.

Definition 4.7. Suppose we have states of the symbolic and memory-safe machines:

$$s_s = (m_s, rs_s, pc_s@tpc_s, i, bs) \qquad\qquad s_a = (m_a, rs_a, v_{pc}).$$

We say that s_s *refines* s_a, written $s_s \triangleright s_a$, if there exists a memory injection mi such that the following hold.

- $m_s \triangleright_{mi} m_a$;
- rs_s is pointwise related to rs_a by $\triangleright_{mi}$;
- $v_{pc} \triangleright_{mi} pc_s@tpc_s$;
- (i, bs) is well-formed with respect to mi and m_s; and
- if $mi(i_s) = (i_a, \_)$, then $i_a \in \mathsf{supp}(s_a)$.

Theorem 4.8. *Let s_s and s'_s be two states of the symbolic machine, and s_a be a state of the memory-safe machine. Suppose that $s_s \to s'_s$ and $s_s \triangleright s_a$. Then there exists a state s'_a of the memory-safe machine such that $s_a \to s'_a$ and $s'_s \triangleright s'_a$.*

The reverse direction, showing that a step of the memory-safe machine can be simulated by the symbolic machine, does not hold. Recall that the memory-safe machine never runs out of memory; therefore, there can be a situation where the symbolic machine gets stuck at an allocation because it ran out of memory, whereas the memory-safe machine continues successfully.

4.4 Characterizing Memory Safety

We now use the two execution models introduced above to discuss what security guarantees we expect to hold in a memory-safe setting. These guarantees take the form of a noninterference result: a program that executes with the memory-safety micro-policy is independent of memory blocks it cannot reach. The proof follows the same strategy we used with the information-flow micro-policy. We begin by proving noninterference for the abstract machine, which has a much simpler semantics. Once this is done, we transfer this result to the symbolic machine by refinement. We begin with the following warm-up property, which says that renaming block identifiers does not have a significant effect on execution.

Lemma 4.9 (Renaming). *Let s and s' be states of the memory-safe machine, and $\pi \in \mathsf{perm}(\mathsf{Id})$ be a permutation of identifiers. Suppose that $s \to s'$. Then, there exists another permutation π' such that $\pi \cdot s \to \pi' \cdot s'$.*

Here, $\pi \cdot s$ denotes the renaming of all identifiers that appear in s according to the permutation π. The formal definition of this operation can be obtained by describing s as a nominal set over Id [41], as reviewed in Appendix A.2.

Roughly speaking, Lemma 4.9 states that the choices of fresh identifiers performed by the allocator are irrelevant: they can only affect the identifiers that appear at the end of execution, and nothing else. This is a direct consequence of not allowing pointers to be treated as integers—specifically, of not allowing programs to observe the identifier of a pointer as an integer. Though Lemma 4.9 might not seem deep, it does have security implications: if it failed to hold, it could mean that programs can learn information about the memory layout of the program, which could lead to secrecy violations (cf. Section 4.7.2).

Lemma 4.9 immediately implies that renaming preserves errors. Suppose that a state s is stuck—that is, it is not the case that $s \to s'$ for any s'. Then $\pi \cdot s$ must also be stuck. For suppose that $\pi \cdot s \to s'$. By applying Lemma 4.9, we find π' such that $s = \pi^{-1} \cdot \pi \cdot s \to \pi' \cdot s$, contradicting the assumption that s was stuck. Similarly, Lemma 4.9 implies that renaming identifiers cannot change the values of integers stored in the final machine state. We will use this result later to argue that the behavior of the machine does not fundamentally change if the allocator is forced to perform other choices of fresh identifiers—specifically, because the state of the heap when it executed was slightly different.

We analyze how execution interacts with unreachable memory by partitioning the memory of a state into reachable and unreachable blocks, and considering what happens when the unreachable blocks are removed. To describe this partitioning, we need some notation. Let $s = (m, rs, pc)$ be a state of the memory-safe machine, and $m' \in \mathsf{Id} \rightharpoonup_{\mathrm{fin}} \mathsf{List}(\mathsf{Val})$ a memory. We define a new state $m' * s$ that is like s, except that we add all memory blocks that are defined in m'. Formally,

$$m' * s \triangleq (m' \mathbin{\vec{\cup}} m, rs, pc),$$

where $m' \mathbin{\vec{\cup}} m$ denotes the *left-biased* union of two partial functions:

$$(m' \mathbin{\vec{\cup}} m)(i) \triangleq \begin{cases} m'(i) & \text{if } m'(i) \neq \bot \\ m(i) & \text{otherwise.} \end{cases}$$

The following framing results describe what happens when we frame the initial state of an execution with unreachable memory. The first one says that adding unreachable memory to the beginning of a successful step yields another successful step.

Lemma 4.10 (Frame OK)**.** *Let s and s' be states of the memory-safe machine, and $m \in \mathsf{Id} \rightharpoonup_{\mathrm{fin}} \mathsf{List}(\mathsf{Val})$ a memory. Suppose that $s \to s'$ and that $\mathsf{supp}(s) \cap \mathsf{dom}(m) = \emptyset$.*[4] *Then there exists a permutation π such that $m * s \to m * (\pi \cdot s')$ and $\mathsf{supp}(\pi \cdot s') \cap \mathsf{dom}(m) = \emptyset$.*

The premise $\mathsf{supp}(s) \cap \mathsf{dom}(m) = \emptyset$ guarantees that the newly added memory is disjoint from the blocks already stored in s, but also unreachable through pointers in that state. It allows, however, pointers stored in m to reference blocks stored in s. The conclusion shows that executing on $m * s$ yields a final state that can still be partitioned into two components: the first one is m, which has been left intact, and the second one, $\pi \cdot s$, is almost the same as the original one, modulo a renaming of block identifiers. Moreover, m is still unreachable from $\pi \cdot s$.

Lemma 4.10 is what forces us to consider permutations in our results. The problem is that when stepping from s to s', the allocator might have chosen some fresh identifier i that already corresponds to a block in m—and that, therefore, is not fresh with respect to $m * s$. This means that the allocator will have to choose a different fresh identifier i' to use for the new block when executing on $m * s$.

The next frame result covers the other case—namely, when running the initial state yields an error.

Lemma 4.11 (Frame Error)**.** *Let s be a state of the memory-safe machine and $m \in \mathsf{Id} \rightharpoonup_{\mathrm{fin}} \mathsf{List}(\mathsf{Val})$ be a memory. Suppose that $\mathsf{supp}(s) \cap \mathsf{dom}(m) = \emptyset$ and that s is stuck. Then $m * s$ is also stuck.*

By combining these two results, we can relate executions of arbitrary length.

Theorem 4.12 (Noninterference)**.** *Let s be a state of the memory-safe machine, $m_1, m_2 \in \mathsf{Id} \rightharpoonup_{\mathrm{fin}} \mathsf{List}(\mathsf{Val})$ two memories, π a permutation, and $n \in \mathbb{N}$. Suppose that*

$$\mathsf{supp}(s) \cap \mathsf{dom}(m_1) = \emptyset \qquad \mathsf{supp}(\pi \cdot s) \cap \mathsf{dom}(m_2) = \emptyset.$$

Then one of the two possibilities hold.

[4]Considering the analogous result proved later for the memory-safe language (Lemma 4.15), it should be possible to weaken this hypothesis to $\mathsf{dom}(m') \cap \mathsf{dom}(m) = \emptyset$, where m' is the memory of the initial state s.

- *There exists a state s' and a permutation π' such that*

$$m_1 * s \to^n m_1 * s' \qquad \mathsf{supp}(s') \cap \mathsf{dom}(m_1) = \emptyset$$

$$m_2 * (\pi \cdot s) \to^n m_2 * (\pi' \cdot s') \qquad \mathsf{supp}(\pi' \cdot s') \cap \mathsf{dom}(m_2) = \emptyset.$$

- *There are no states s_1' and s_2' such that $m_1 * s \to^n s_1'$ and $m_2 * s \to^n s_2'$.*

Proof. By induction on n. The base case is trivial. For the induction step, suppose that the result is valid for n. First, suppose that there exists s' such that $s \to s'$. By Lemma 4.10, we can find π_1 such that $m_1 * s \to m_1 * (\pi_1 \cdot s')$ and $\mathsf{supp}(\pi_1 \cdot s') \cap \mathsf{dom}(m_1) = \emptyset$. By Lemma 4.9, we find π' such that $\pi \cdot s \to \pi' \cdot s'$. Again by Lemma 4.10, we find π_2 such that $m_2 * (\pi \cdot s) \to m_2 * (\pi_2 \cdot \pi' \cdot s')$, with $\mathsf{supp}(\pi_2 \cdot \pi' \cdot s') \cap \mathsf{dom}(m_2) = \emptyset$. Since $\pi_2 \cdot \pi' \cdot s' = (\pi_2 \cdot \pi' \cdot \pi_1^{-1}) \circ \pi_1 \cdot s'$, we can apply the induction hypothesis and close this case.

Now, suppose that s is stuck. By Lemma 4.9, we find that $\pi \cdot s$ is also stuck. By Lemma 4.11, this implies that both $m_1 * s$ and $m_2 * (\pi \cdot s)$ are stuck, thus allowing us to conclude. □

This formulation of noninterference is similar to the one used in the information-flow micro-policy (Definition 3.7). Call two states s_1 and s_2 indistinguishable if their reachable memory regions coincide modulo a permutation—that is, if there exists m_1, m_2, π and s such that $s_1 = m_1 * s$, $s_2 = m_2 * (\pi \cdot s)$, and $\mathsf{supp}(s) \cap \mathsf{dom}(m_1) = \mathsf{supp}(\pi \cdot s) \cap \mathsf{dom}(m_2) = \emptyset$. Then Theorem 4.12 says that running those states for n steps yields either indistinguishable final states or errors. This is a secrecy result: no information stored in unreachable memory can leak into the rest of the state, and not even influence the occurrence of errors during executions. There is also an integrity aspect to this result: it guarantees that unreachable memory cannot be modified.

Noninterference for the memory-safe machine yields an analogous result for the symbolic machine.

Theorem 4.13 (Noninterference for symbolic machine)**.** *Let s_1, s_1', s_2, and s_2' be states of the symbolic machine, s a state of the memory-safe machine, m_1 and m_2 two memories of the memory-*

safe machine, π a permutation of identifiers, and $n \in \mathbb{N}$. Suppose that

$$s_1 \rhd m_1 * s \qquad\qquad s_2 \rhd m_2 * (\pi \cdot s)$$

$$s_1 \to^n s_1' \qquad\qquad s_2 \to^n s_2'$$

$$\mathsf{supp}(s) \cap \mathsf{dom}(m_1) = \mathsf{supp}(\pi \cdot s) \cap \mathsf{dom}(m_2) = \emptyset.$$

Then there exists s' and π' such that

$$s_1' \rhd m_1 * s' \qquad\qquad s_2 \rhd m_2 * (\pi' * s')$$

$$\mathsf{supp}(s') \cap \mathsf{dom}(m_1) = \mathsf{supp}(\pi' \cdot s') \cap \mathsf{dom}(m_2) = \emptyset.$$

This result is stated directly using the notion of unreachability inherited from the memory-safe machine, which avoids having to formulate a similar notion for the symbolic machine. We could also have stated a version of this result that does not mention the memory-safe machine, although it would probably lead to a more complicated statement, because it would have to deal explicitly with the well-formedness of the private policy state. In Theorem 4.13, by contrast, these conditions are hidden in the refinement relation. Note that this result is weaker than Theorem 4.12, because it only applies if we know that both executions are capable of running for n steps without encountering a run-time error: if one of them gets stuck within n steps, there is nothing meaningful we can conclude about the other. This is a consequence of going from a machine with infinite memory to one where memory is finite: if we change the initial heap that a piece of code runs on, we could cause a successful execution to run out of memory and terminate earlier. We return to this point in Section 4.7.5.

The guarantees of memory safety described here are related, but not identical, to typical notions of type safety discussed in the programming languages literature. We have seen that our policy imposes a basic form of type safety, because it segregates pointers from other kinds of values. However, this separation is not enough to guarantee memory safety, in the sense of Theorem 4.13. For instance, if the memory-safety policy did not check pointer identifiers when accessing memory, then it would be possible to offset a pointer to access memory it should not be allowed to. We could also consider a hypothetical language with a primitive for iterating over all the references of a given type. This language might be type safe, in the sense that it satisfies sensible progress and preservation theorems, but it is not memory safe, because a program can manipulate a memory region even if it does not possess pointers to it.

Besides the limitations due to finite memory, we should keep in mind that the secrecy guarantees of memory safety are not absolute: they only protect programs against direct data leaks. It is still possible to extract secret information from memory-safe programs by other side channels, such as timing and energy consumption. If leaks via these side channels are a concern, then techniques from the information-flow literature might offer insight.

4.5 A Memory-safe Language

We propose that the framing properties and the noninterference result that we just proved for the two machines is a good criterion for characterizing the security guarantees of memory safety. In this section, we explore this idea by showing how the results carry over to a simple imperative language with memory-safety checks. Later, we show how this allows us to relate the guarantees of memory safety to separation logic [90]. Since these results are fairly independent from the rest of the micro-policy development, they have been formalized separately.[5]

4.5.1 Language Definition

The language's syntax, fairly standard, is summarized on Figure 4.5. The most unusual aspect is that it separates commands that have an effect on the heap from those that operate solely on local variables. The form $[e]$ represents an access to the memory location denoted by the expression e. Expressions include basic arithmetic, logic operations, and an offset operator for extracting the offset of a pointer. (Once again, we include offset to illustrate what operations can be performed on pointers; alternatively, we could have used a base operator, as we did for the micro-policy.) The set var is a countably infinite set of variables.

Programs of this language manipulate values ($\mathcal{V}$) stored in a state ($\mathcal{S}$) comprising two components: a *local store* $l \in \mathcal{L}$, a finite partial function from variables to values, and a *heap* $m \in \mathcal{M}$, a finite partial function from pointers to values. Like in the micro-policy presented earlier, pointers are pairs (i, n) of an identifier $i \in \mathsf{Id}$ and an offset $n \in \mathbb{Z}$; the only difference is that we allow offsets to be arbitrary integer values, as opposed to just machine words of fixed precision. Note that there is no expression for writing literal pointer values. Other values beside pointers include integers, Booleans,

[5]The Coq development is available at `https://github.com/arthuraa/memory-safe-language`.

Command c	Description
$x \leftarrow e$	Assign to local variable
$x \leftarrow [e]$	Read from heap location
$[e_1] \leftarrow e_2$	Assign to heap location
$x \leftarrow \mathsf{alloc}(e)$	Allocate heap block of size e
$\mathsf{free}(e)$	Free block starting at e
skip	Do nothing
$\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{end}$	Conditional
$\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{end}$	Loop
$c_1; c_2$	Sequencing

$$
\begin{aligned}
\mathsf{var} &= \{x, y, z, \cdots\} \\
e &= x \in \mathsf{var} \mid n \in \mathbb{Z} \mid b \in \mathbb{B} \mid \mathsf{nil} \mid e \oplus e \mid \neg e \mid \mathsf{offset}\ e \\
\oplus &\in \{+, -, \times, =, \leq, \wedge, \vee\}
\end{aligned}
$$

Figure 4.5: Syntax of memory-safe language

$$
\begin{aligned}
s \in \mathcal{S} &\triangleq \mathcal{L} \times \mathcal{M} && \text{(states)} \\
l \in \mathcal{L} &\triangleq \mathsf{var} \rightharpoonup_{\mathrm{fin}} \mathcal{V} && \text{(local stores)} \\
m \in \mathcal{M} &\triangleq \mathsf{Id} \times \mathbb{Z} \rightharpoonup_{\mathrm{fin}} \mathcal{V} && \text{(heaps)} \\
v \in \mathcal{V} &\triangleq \mathbb{Z} \uplus \mathbb{B} \uplus \{\mathsf{nil}\} \uplus \mathsf{Id} \times \mathbb{Z} && \text{(values)} \\
\mathcal{O} &\triangleq \mathsf{Res}(\mathcal{S}) \uplus \{\mathsf{error}\} && \text{(outcomes)}
\end{aligned}
$$

Figure 4.6: Program states and values for memory-safe language

and nil, used to represent an invalid pointer. These definitions are summarized in Figure 4.6; the set $\mathcal{O}$ represents the possible outcomes of a computation, and will be explained soon.

To detail how programs execute, we begin by defining expression evaluation. Given an expression e and a local store l, we define a value $[\![e]\!](l)$ as follows:

$$
\begin{aligned}
[\![x]\!](l) &\triangleq \begin{cases} l(x) & \text{if } l(x) \neq \bot \\ \mathsf{nil} & \text{otherwise} \end{cases} \\
[\![n]\!](l) &\triangleq n \\
[\![b]\!](l) &\triangleq b \\
[\![\mathsf{nil}]\!](l) &\triangleq \mathsf{nil} \\
[\![e_1 \oplus e_2]\!](l) &\triangleq [\![e_1]\!](l) \oplus [\![e_2]\!](l) \\
[\![\neg e]\!](l) &\triangleq \neg [\![e]\!](l) \\
[\![\mathsf{offset}\ e]\!](l) &\triangleq \begin{cases} n & \text{if } [\![e]\!](l) = (i, n) \\ \mathsf{nil} & \text{otherwise.} \end{cases}
\end{aligned}
$$

Note that $[\![e]\!]$ is a total function. We have lifted the definition of binary and unary operators to values to allow pointer arithmetic, and to assign nil to nonsensical combinations. (It would also have been possible to make expression evaluation a partial operation without affecting the results reported here.) The complete definition of these operations is included in Figure 4.7.

Now that we can evaluate expressions, we are ready to define how programs execute. To each program c, we associate a partial function $[\![c]\!] \in \mathcal{S} \rightharpoonup \mathcal{O}$, which takes an initial state s to an outcome $[\![c]\!](s) \in \mathcal{O}$ (cf. Figure 4.6). Contrary to the conventions we have been following thus far, partiality represents non-termination instead of a run-time error, which corresponds to the outcome value error. Crucially, in an implementation of this language, run-time errors should cause the program to terminate. This is rather different from languages like C, where many errors lead to erratic undefined behavior, which can manifest itself in a myriad of different ways during execution.

When a program successfully terminates, it yields a result $[I, s] \in \mathsf{Res}(\mathcal{S})$; intuitively, this final state is a pair of a state s and a finite set of identifiers I that has been used to refer to new blocks allocated during execution; however, we quotient this set of pairs to allow renaming newly allocated blocks, and dropping names of blocks that have been allocated and subsequently freed. This is a technical device to simplify the definition of program execution and the statement of our results, and can be mostly ignored by the casual reader; we refer to Appendix A.2 for the formal definition of

$$v_1 + v_2 \triangleq \begin{cases} n_1 + n_2 & \text{if } v_i = n_i \in \mathbb{Z} \\ (i, n_1 + n_2) & \text{if } v_i = (i, n_1) \text{ and } v_j = n_2 \text{, with } i \neq j \\ \mathsf{nil} & \text{otherwise} \end{cases}$$

$$v_1 - v_2 \triangleq \begin{cases} n_1 - n_2 & \text{if } v_i = n_i \in \mathbb{Z} \\ (i, n_1 - n_2) & \text{if } v_1 = (i, n_1) \text{ and } v_2 = n_2 \\ n_1 - n_2 & \text{if } v_1 = (i, n_1) \text{ and } v_2 = (i, n_2) \\ \mathsf{nil} & \text{otherwise} \end{cases}$$

$$v_1 \times v_2 \triangleq \begin{cases} n_1 \times n_2 & \text{if } v_i = n_i \in \mathbb{Z} \\ \mathsf{nil} & \text{otherwise} \end{cases} \qquad v_1 \leq v_2 \triangleq \begin{cases} n_1 \leq n_2 & \text{if } v_i = n_i \in \mathbb{Z} \\ \mathsf{nil} & \text{otherwise} \end{cases}$$

$$v_1 \wedge v_2 \triangleq \begin{cases} b_1 \wedge b_2 & \text{if } v_i = b_i \in \mathbb{B} \\ \mathsf{nil} & \text{otherwise} \end{cases} \qquad v_1 \vee v_2 \triangleq \begin{cases} b_1 \vee b_2 & \text{if } v_i = b_i \in \mathbb{B} \\ \mathsf{nil} & \text{otherwise} \end{cases}$$

$$\neg v \triangleq \begin{cases} \neg b & \text{if } v = b \in \mathbb{B} \\ \mathsf{nil} & \text{otherwise} \end{cases}$$

Figure 4.7: Lifting of operations to values of memory-safe language

this construction.

Figure 4.8 gives the complete definition of the language semantics. Just as in the memory-safe abstract machine, memory loads, stores, and frees lead to errors when accessing undefined memory. Note that most of the commands yield final states that have no new block identifiers in them; that is, the first component of the outcome is $\emptyset$. The only exceptions are allocation, sequencing, if, and iteration.

Just as we did for states of the abstract machines analyzed in this chapter, we can endow the set of states $\mathcal{S}$ with the structure of a nominal set. Recall that, given a state s, the support $\mathsf{supp}(s)$ represents the finite set of block identifiers that occurs somewhere in s. In the definition of allocation, (1) we pick some identifier i that does not occur anywhere in the program state, (2) initialize a new block corresponding to that identifier in memory with zeros, (3) load a pointer to the beginning of the block in the variable x, and (4) assign i to the set of new identifiers. Although we have not exactly specified what value of i should be chosen here, this choice does not matter: the nominal infrastructure guarantees that any choice of i leads to the same final outcome, guaranteeing that the

$$[\![x \leftarrow e]\!](l,m) \triangleq [\emptyset, (l[x \mapsto [\![e]\!](l)], m)]$$

$$[\![x \leftarrow [e]]\!](l,m) \triangleq \begin{cases} [\emptyset, (l[x \mapsto v], m)] & \text{if } [\![e]\!](l) = (i,n) \text{ and } m(i,n) = v \\ \mathsf{error} & \text{otherwise} \end{cases}$$

$$[\![[e_1] \leftarrow e_2]\!](l,m) \triangleq \begin{cases} [\emptyset, (l, m[(i,n) \mapsto v])] & \text{if } [\![e_1]\!](l) = (i,n) \text{ and } [\![e_2]\!](l) = v \\ \mathsf{error} & \text{otherwise} \end{cases}$$

$$[\![x \leftarrow \mathsf{alloc}(e)]\!](l,m) \triangleq$$

$$\begin{cases} [\{i\}, (l[x \mapsto (i,0)], m[(i,0) \mapsto 0, ..., (i,n-1) \mapsto 0])] & \text{if } i \notin \mathsf{supp}(l,m) \text{ and } [\![e]\!](l) = n > 0 \\ \mathsf{error} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{free}(e)]\!](l,m) \triangleq \begin{cases} [\emptyset, (l, \{(i',n) \mapsto m(i',n) \mid i' \neq i\})] & \text{if } [\![e]\!](l) = (i,0) \text{ and } m(i,0) \neq \bot \\ \mathsf{error} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{skip}]\!](l,m) \triangleq [\emptyset, (l,m)] \qquad [\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{end}]\!](l,m) \triangleq \begin{cases} [\![c_1]\!](l,m) & \text{if } [\![e]\!](l) = \mathsf{true} \\ [\![c_2]\!](l,m) & \text{if } [\![e]\!](l) = \mathsf{false} \\ \mathsf{error} & \text{otherwise} \end{cases}$$

$$[\![c_1; c_2]\!](l,m) \triangleq \begin{cases} [I \cup I', s'] & \text{if } [\![c_1]\!](l,m) = [I,s] \text{ and } [\![c_2]\!](s) = [I', s'] \\ \mathsf{error} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{end}]\!](s) \triangleq [\![\mathsf{if}\ e\ \mathsf{then}\ c; \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{end}\ \mathsf{else}\ \mathsf{skip}\ \mathsf{end}]\!](s)$$

Figure 4.8: Semantics of program execution in memory-safe language

semantics of programs is a partial function, and not an arbitrary relation. An important technical detail is that program execution is an *equivariant* operation, which guarantees that these equations are valid. As Lemma 4.9, equivariance means intuitively program execution does not depend on the choice of block identifiers. Formally:

Lemma 4.14. *Let c be a program, s be a state, and π be a permutation. Then*

$$[\![c]\!](\pi \cdot s) = \begin{cases} \bot & \textit{if } [\![c]\!](s) = \bot \\ \mathsf{error} & \textit{if } [\![c]\!](s) = \mathsf{error} \\ [\pi \cdot I, \pi \cdot s'] & \textit{if } [\![c]\!](s) = [I, s]. \end{cases}$$

Compare this result with Lemma 4.9, which required us to apply an additional permutation π' to adjust the result of execution. Here, this permutation is hidden by the construction of program outcomes.

Finally, we note that the semantics of iteration is not defined in terms the semantics of a smaller command, making it non-structurally decreasing. We can give a rigorous sense to this definition by applying Kleene's fixed-point theorem, reviewed in Appendix A.[6]

4.5.2 Guarantees of Memory Safety

All the results proved for the memory-safe abstract machine carry over to the language setting. Just like the equivariance result we proved (Lemma 4.14), their formulation is somewhat cleaner, thanks to the way we manage fresh block identifiers in program outcomes.

If $s_1 = (l_1, m_1)$ and $s_2 = (l_2, m_2)$ are program states, we define a new program state $s_1 \vec{\cup} s_2$ as follows:

$$s_1 \vec{\cup} s_2 \triangleq (l_1 \vec{\cup} l_2, m_1 \vec{\cup} m_2).$$

We also define a set $\mathsf{blocks}(l, m)$ containing all the identifiers corresponding to blocks stored in m:

$$\mathsf{blocks}(l, m) = \{i \mid \exists n.(i, n) \in \mathsf{dom}(m).\}$$

Finally, given a program c, we let $\mathsf{vars}(c)$ be the set of variables that are used in c.

To state the results, fix some program c, and two program states $s_1 = (l_1, m_1)$ and s_2. We suppose that we execute c on s_1, and we will analyze how execution is affected if we frame the initial state with extra data s_2. Throughout, we suppose that $\mathsf{vars}(c) \subseteq \mathsf{dom}(l_1)$, which guarantees that all the variables needed to execute c are defined in s_1.

First, an analogue of Lemma 4.10.

Lemma 4.15 (Frame OK)**.** *Suppose* $[\![c]\!](s_1) = [I, s_1']$, $\mathsf{blocks}(s_1) \cap \mathsf{blocks}(s_2) = \emptyset$, *and* $I \cap \mathsf{supp}(s_2) = \emptyset$. *Then* $[\![c]\!](s_1 \vec{\cup} s_2) = [I, s_1' \vec{\cup} s_2]$.

We assume that the execution of c on s_1 successfully terminated and that the heaps of s_1 and s_2 store disjoint memory regions. We also assume that the set I of block identifiers that was allocated

[6]Our Coq development adopts a slightly different definition, which takes an additional timeout parameter to bound the maximum number of iterations performed by the program; if this number is not enough, the program simply returns $\bot$.

during execution was chosen in a way that does not conflict with the identifiers present in s_2. (Such a choice is always possible.) Then, we conclude that executing c on a state framed with s_2 yields a successful final result that is framed similarly. Unlike the analog result obtained earlier, this one does not guarantee directly that s_1' did not somehow create a memory block that is stored in s_2. This is guaranteed by the following result, which is needed to prove Lemma 4.15.

Lemma 4.16. *Suppose that* $[\![c]\!](l, m) = [I, (l', m')]$ *and* $\mathsf{vars}(c) \subseteq \mathsf{dom}(l)$. *Then* $\mathsf{dom}(m') \subseteq \mathsf{dom}(m) \cup I$ *and* $\mathsf{dom}(l') = \mathsf{dom}(l)$.

Intuitively, this result guarantees (1) that the only blocks that can be stored in the final state of a program are those that were already allocated in the initial state, plus any blocks corresponding to the set of fresh identifiers I; and (2) that a program cannot change the set of variables defined on the local store. In the statement of Lemma 4.15, the two hypotheses about the identifiers that appear on s_2 allow us to conclude that the blocks of s_2 are not overwritten by those of s_1'.

A second framing result says that framing does not affect infinite loops. Its proof is similar to the one of Lemma 4.15.

Lemma 4.17 (Frame Loop)**.** *Suppose* $[\![c]\!](s_1) = \bot$ *and* $\mathsf{blocks}(s_1) \cap \mathsf{blocks}(s_2) = \emptyset$. *Then* $[\![c]\!](s_1 \vec{\cup} s_2) = \bot$.

Finally, we show that framing preserves memory errors, an analog of Lemma 4.11.

Lemma 4.18 (Frame Error)**.** *Suppose* $[\![c]\!](s_1) = \mathsf{error}$ *and* $\mathsf{supp}(s_1) \cap \mathsf{blocks}(s_2) = \emptyset$. *Then* $[\![c]\!](s_1 \vec{\cup} s_2) = \mathsf{error}$.

The proofs of these results follow by induction on the structure of the program, combined with fixed-point induction for iteration. Taken together, these three lemmas yield a noninterference result for our language.

Theorem 4.19 (Noninterference)**.** *Let c be a command and* $s_1 = (l_1, m_1)$, s_{21} *and* s_{22} *be program states. Suppose that*

- $\mathsf{vars}(c) \subseteq \mathsf{dom}(l_1)$*; and*
- $\mathsf{supp}(s_1) \cap \mathsf{blocks}(s_{2i}) = \emptyset$*, for* $i = 1, 2$.

Then one of the following three possibilities hold.

- *There exists I and s'_1 such that $I \cap \mathsf{supp}(s_{2i}) = \emptyset$ and $[\![c]\!](s_1 \cup s_{21}) = [I, s'_1 \vec{\cup} s_{2i}]$, for $i = 1, 2$.*

- $[\![c]\!](s_1 \vec{\cup} s_{2i}) = \bot$, *for* $i = 1, 2$.

- $[\![c]\!](s_1 \vec{\cup} s_{2i}) = \mathsf{error}$, *for* $i = 1, 2$.

4.6 Memory Safety and Separation Logic

A central theme in the properties that we have proposed is *locality of reasoning*. In order to analyze the behavior of a program, we do not have to consider the entire state on which it operates; it suffices to restrict our attention to the memory blocks that are reachable via the pointers it possesses. The strength of locality for reasoning about programs has been long recognized in the research literature, in particular in the development of *separation logic* [90], a proof system for reasoning about programs that manipulate a stateful heap. In this section, we will show that there is a strong connection between our properties and the view of locality proposed by separation logic; indeed, we show that our properties can be used to extend the proof rules of separation logic for taking advantage of reachability-based isolation.

In separation logic, as in Hoare logic and related formal systems, we are interested in proving program specifications that say that some property (the *postcondition*) holds of the final state of the program if we know that another property (the *precondition*) holds of the state when the program starts running. The logic possesses proof rules that allow us to compose simple results about the behavior of different code fragments to establish a potentially much more complex specification about the entire program.

Let's begin by developing a version of separation logic for the language of Section 4.5.

Definition 4.20. An *assertion* is any subset of program states $p \subseteq \mathcal{S}$. Let c be a program, and p and q be assertions. We say that a *specification* (or *triple*) $\{p\}\ c\ \{q\}$ is valid if the following condition holds. Consider an initial state (l, m) such that p holds and $\mathsf{vars}(c) \subseteq \mathsf{dom}(l)$. There can only be two possibilities.

- $[\![c]\!](l, m) = \bot$; or

- $[\![c]\!](l, m) \in \mathsf{Res}(\mathcal{S})$, and $\forall I\, s'. [\![c]\!](s) = [I, s'] \Rightarrow s' \in q$.

(Compared to usual formulations of separation logic, ours has an interesting subtlety: requiring that q be valid at the end of execution regardless of the choices of fresh identifiers made to represent that final state. A similar restriction is typically imposed by using a non-deterministic semantics, where the allocator is free to choose any available segment to store a new memory region, and by requiring that any successfully terminating execution validate the postcondition.)

For our purposes, the most important facet of this definition is that it precludes the occurrence of run-time errors: when run on an initial state that satisfies the precondition intended by its specification, a program can either run forever or terminate successfully. Before discussing this aspect, it is worth getting a taste of the basic system reviewing some of its basic proof rules. For instance,

$$\frac{}{\{p\}\ \mathsf{skip}\ \{p\}}$$

This rule says that we can show that the command skip always preserves its precondition. A more interesting rule allows us to prove results about the sequential composition of two programs.

$$\frac{\{p\}\ c_1\ \{q\} \qquad \{q\}\ c_2\ \{r\}}{\{p\}\ c_1; c_2\ \{r\}}$$

That is, to prove something about two commands run in sequence, we just have to prove a specification for each one separately, and check that the postcondition of the first one matches the precondition of the second. Besides these two simple rules, there are many others for dealing with each syntactic form for our simple imperative language. However, since our goal is merely analyzing the local reasoning facilities of separation logic, we will not consider these other rules in what follows.

Let us go back to the issue of errors in separation logic. Avoiding errors in specifications has two main benefits. First, it guarantees that the logic can apply to unsafe settings like C. In such settings, memory errors do not have a well-behaved semantics like they do in our language, but lead to wild undefined behavior. When this occurs, the ensuing behavior of the program is determined by obscure details of the language implementation, such as the layout of data structures in memory, making it difficult to obtain a clean proof system for reasoning about the program.

The second benefit of avoiding memory errors is enabling local reasoning. When a program does not run into memory errors, we force its execution to depend solely on parts of the heap described by its pre- and postconditions. This is captured by a proof principle known as the *frame rule* that, in our setting, can be stated as follows.

Definition 4.21. Let p and q be assertions. We define their *separating conjunction* $p * q$ as follows.

$$p * q \triangleq \{(l, m_1 \mathbin{\vec{\cup}} m_2) \mid (l, m_1) \in p, (l, m_2) \in q, \mathsf{blocks}(l, m_1) \cap \mathsf{blocks}(l, m_2) = \emptyset\}$$

We say that an assertion p is *independent* of a set of program variables V if the following condition holds.

$$\forall l_1\, l_2\, m. (\forall x \notin V. l_1(x) = l_2(x)) \wedge (l_1, m) \in p \Rightarrow (l_2, m) \in p$$

(Intuitively, this means that the values stored in the variables corresponding to V have no influence on whether an assertion holds of a state or not.) Finally, given a program c, we let $\mathsf{modvars}(c)$ be the set of variables that appear in the left-hand side of some assignment statement in c.

Theorem 4.22. *Let c be a program, and p, q, and r be assertions. Suppose that r is independent of* $\mathsf{modvars}(c)$. *The following rule is sound.*

$$\textsc{Frame}\quad \frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\}}$$

Let's dissect this statement. A separating conjunction $p * r$ roughly says that the heap of a program can be split into two disjoint parts, one of which satisfies p, and another one that satisfies r. Thus, the frame rule says that, given a program c with a valid specification $\{p\}\ c\ \{q\}$, we can always enlarge the initial heap of the program with a disjoint portion that satisfies another assertion r. When we do this, if the program terminates, it will yield a final state with a portion that satisfies the postcondition q, but we will also know that the portion corresponding to r will be left untouched. In other words, it guarantees that the integrity of that other portion is preserved. (It would be possible to find a formulation that captures secrecy guarantees as well, but we do not consider that, since it would require a less standard relational proof system.) This reading hints that the frame rule is closely related to the framing results that we have been proving so far, and this is indeed the case: we can prove Theorem 4.22 by invoking Lemmas 4.15 and 4.17.

We claimed that the frame rule is a consequence of precluding errors in specifications. Consider what would happen if we attempted to derive a frame rule for the following alternative reading of specifications.

Definition 4.23. Let c be a program, and p and q assertions. We say that the *weak specification* $\{p\}\ c\ \{q\}_e$ is valid if the following holds. Let s be an arbitrary state that satisfies p. If $[\![c]\!](s) = [I, s']$, then $s' \in q$.

Weak specifications are just like those of Definition 4.20, except that they do not prevent programs from terminating their execution with a run-time error. Unfortunately, this new notion of specification causes the frame rule to become unsound. Let us consider a counterexample. Let emp be the assertion that holds of states with an empty heap. If errors are allowed, we can show that the following specification vacuously holds:

$$\{\mathsf{emp}\}\ x \leftarrow [y]\ \{x = 0\}_e$$

By $x = 0$, we mean the postcondition that asserts that the local variable x holds the value 0. Why is this specification valid? If we start this program on an empty heap, trying to read from any memory location stored in y necessarily results in an error. Now, consider what happens if we frame this specification with the predicate $y \mapsto 1$, which asserts that the current heap contains only one memory cell referenced by the pointer stored in y, and that this memory cell stores the value 1. If the frame rule were valid, the specification

$$\{\mathsf{emp} * y \mapsto 1\}\ x \leftarrow [y]\ \{x = 0 * y \mapsto 1\}_e,$$

would be valid, which, after some simplification, is equivalent to the following one:

$$\{y \mapsto 1\}\ x \leftarrow [y]\ \{x = 0 \wedge y \mapsto 1\}_e$$

This triple, however, is *not* valid, because if we start executing the program in a state that satisfies the precondition, the program will terminate successfully, and the variable x will be set to 1, not 0.

Even if excluding errors is important, there is a high price to pay for enforcing this discipline: proving almost any nontrivial statement of a program in separation logic requires detailed, extensive reasoning about its behaviors. In some settings, this might be prohibitively expensive—for instance, in the case of large programs, programs with complex memory access patterns, such as image-processing algorithms, or programs that depend on third-party libraries or plugins, to whose source code we may have no access. Unfortunately, if any part of the program is not covered by the proof, there is nothing that separation logic allows us to say about its behavior, no matter how tiny that part is. Even the following seemingly vacuous rule is not valid in conventional separation logic.

$$\textsc{Taut}\quad \frac{}{\{p\}\ c\ \{\mathsf{true}\}}$$

For a counterexample, take p to be emp, and c to be $x \leftarrow [y]$, as above: running the program under these conditions results in an error, which is not allowed by specifications. The analog of this rule, naturally, is sound if errors are allowed:

$$\textsc{Taut}\quad \frac{}{\{p\}\ c\ \{\mathsf{true}\}_e}$$

Recall, however, that one of the motivations for separation logic was for it to apply to low-level settings like C, where the occurrence of memory errors is devastating. This raises the question of whether it is possible to salvage the frame rule somehow in a setting like ours, where memory errors lead to predictable behavior. The answer is yes, by replacing the separating conjunction with a stronger connective.

Definition 4.24. Let p and q be assertions. Define their *isolating conjunction* $p \vec{*} q$ as follows.

$$p \vec{*} q \triangleq \{(l, m_1 \vec{\cup} m_2) \mid (l, m_1) \in p, (l, m_2) \in q, \mathsf{supp}(l, m_1) \cap \mathsf{blocks}(l, m_2) = \emptyset\}$$

Theorem 4.25. *Let c be a program, and p, q, and r be assertions. Suppose that r is independent of* $\mathsf{modvars}(c)$. *The following rule is sound.*

$$\textit{SafeFrame}\quad \frac{\{p\}\ c\ \{q\}_e}{\{p \vec{*} r\}\ c\ \{q \vec{*} r\}_e}$$

The isolating conjunction is similar to the separating conjunction, but it additionally requires that the pointers that appear on the portion of the heap that satisfies the left assertion cannot reference regions of memory stored on the right. This assumption, crucial for proving Lemma 4.18, guarantees that framing cannot make the execution of a crashing program succeed, by ruling out the case where the original state referenced a dangling pointer that happened to be defined in the framed portion. Note, however, that pointers in the r portion of the heap are free to reference blocks stored in p.

The modified frame rule supported by our language is incomparable with the original variant. On the one hand, it requires a stronger notion of separation between state components; on the other, it can apply in settings where the program we are verifying can lead to memory errors. To see how this can be useful, imagine that we are writing an application that uses a JPEG decoder routine provided by a third-party library. By a simple check of the program text, we see that the only interaction of the decoder program with the heap is through its input and output buffers. Without heavy inspection of the program, assuming that it never frees or allocates memory, we might be able to show a specification of the form

$$\{p\}\ \mathsf{decoder}\ \{p\},$$

where p is an assertion guaranteeing that the input and output buffers are allocated. If we know that these buffers cannot reference a window object used by our program—which, in a realistic language, might be established with the help of additional information, such as typing—we can use the modified frame rule to conclude that the integrity of the window object is preserved across the call to the decoder routine.

The crucial difference between the two variants of the frame rule is the assumptions they make. The original variant, Theorem 4.22, is a local reasoning principle for programs that have detailed proofs about their behavior. They may run in unsafe languages like C, but the proofs guarantee that they never lead to memory errors. In this sense, we could see the frame rule as a good characterization of memory-safe *programs*. The variant of the frame rule considered here, Theorem 4.25, on the other hand, allows programs to be badly behaved, in the sense that they might *try* to misuse the memory, either intentionally or maliciously. However, it assumes that the run-time environment where these programs execute provides sensible guarantees for when these errors arise. Thus, we can see our variant of the frame rule as characterizing memory-safe *platforms and languages*.

Relaxation	Guarantees Integrity?	Possible leaks
Pointer forging	No	Everything
Observing physical addresses	Yes	Memory size and layout
Uninitialized memory	Yes	Contents of old memory
Dangling pointers	Usually not	Memory size and layout
Finite memory	Yes	Memory size

Figure 4.9: Possible security consequences of common memory-safety relaxations

4.7 Relaxing Memory Safety

The mechanisms reviewed here protect programs from various kinds of memory misuse. Unfortunately, the restrictions they impose are often relaxed in practical designs, usually for performance reasons: some platforms might allow programs to allocate memory without initializing it first, while others might allow them to extract the physical address of pointers as integers, etc. In this section, we discuss how some popular relaxations can affect the data-isolation guarantees we have analyzed. As depicted in Figure 4.9, these effects can be roughly grouped in two categories: integrity and secrecy violations. Most relaxations do not provide a means of changing the contents of memory regions they shouldn't access, thus guaranteeing their secrecy. Most of them, on the other hand, make it possible to extract information about the global state of the program, such as its total memory consumption. Naturally, this distinction should be taken with a grain of salt, since integrity depends on secrecy in many practical situations; for instance, if a system uses a password to control access to some resource, then a secrecy violation could be escalated into an integrity violation!

Our list of vulnerabilities only tries to predict possible avenues for attack, and is not backed by proof. In a real system, attacks can result from subtle interaction between multiple features, including some that are not considered here, such as undefined behavior arising from integer overflows in C.

4.7.1 Forging Pointers

The most basic restriction imposed by our policy is preventing pointer values from being fabricated out of thin air. The only way to access a memory region is by using a valid pointer to it, which must be obtained by allocation, or by retrieving it from some other part of the state. Thanks to this restriction, the set of identifiers that occur in a machine state serves as a bound on the memory regions that that state can modify. This would no longer hold if pointers would be fabricated out of thin air. Consider a typical system that allows casting integers to pointers. In general, it is hard to tell what part of the memory that integer could end up accessing, implying that no range of the address space can be properly isolated from code that uses this idiom. This is why such casts are disallowed in typical systems that provide comprehensive memory safety. We could argue that bounds checking—one of the most fundamental aspects of memory safety—is also a mechanism for preventing pointer forging, given that its absence allows using a pointer to access a resource it was not meant to.

Even if the dangers of fabricating pointers are well-known, similar problems arise in languages that are traditionally seen as memory safe. JavaScript, for example, allows programs to access global variables by indexing into a global associative array with arbitrary strings. In this sense, we could see global variables in JavaScript as "memory unsafe," since it is hard to determine which global variables a piece of code can interact with, which is known to enable several serious attacks [39, 106].

4.7.2 Observing Pointers

The memory-safety policy does not allow programs to extract physical addresses from pointer values. Typical memory-safe systems do not have a clear stance on this. Many practical tools for enforcing memory safety do not impose this restriction [84, 113], as many low-level C idioms require this functionality [76], and even traditional memory-safe languages like Java allow programs to inspect the physical placement of objects.

The problem with extracting physical addresses from pointers is that they reveal something about the entire state of the program, including memory regions that we cannot access. By inspecting addresses, an attacker might learn useful information about the memory layout of the program, or its total memory consumption—and, using this information, conclude something they were not supposed to. For example, Jana and Shmatikov [60] showed how an attacker can use the memory consumption of a web browser to find out what pages the user is visiting. Similar remarks would apply if the memory-safety policy allowed programs to extract integers from pointer identifiers.

Fortunately, observing physical addresses does not fundamentally break the guarantees of memory safety. For instance, if this were possible in the memory-safety policy, we should be able to show a variation of Theorem 4.13 where we additionally assume that if the two portions of unreachable memory differ only in what is stored in those memory locations, but not the size or placement of those memory regions. Intuitively, such a result would mean that the only information that this relaxation might reveal about unreachable regions is their placement and their size. In particular, if we want to prevent a component from learning a private key stored in an unreachable region, a system that allows pointer-to-integer casts might make this possible.

It is worth noting that preventing physical addresses from being observable may require more than simply not allowing pointer-to-integer casts. Recall that our policy does not allow using the equality comparison instruction on two pointers for different regions, and that this operator only takes payloads, not tags, into account. If this restriction were not enforced, an attacker could try to learn the physical address of a pointer by offsetting it and testing it for equality:

```
int *y = malloc(42);
if (x == y + 1729) something();
else somethingElse();
```

If the attacker knows the physical address of the pointer `x`—because it has detailed knowledge of the allocator implementation, for instance—then the above test might reveal information about the global state of live regions. The situation is even worse for order comparisons between pointers, which must be prevented even when the two pointers in question reference the same region. The problem is that we can extract the physical address of a pointer by observing when the result of such a comparison changes due to overflows, as the following snippet shows.

```
int *x = malloc(42);
unsigned int i = 1;
while (x < i + x) i++;
/* at this point, sizeof (int) * i + x == 0 */
unsigned int addr = - sizeof (int) * i;
```

Note, however, that it is perfectly valid for programs to extract the offset of the two pointers using the base service, and then compare their relative placement.

4.7.3 Uninitialized Memory

Every region allocated in the memory-safety policy has all of its locations initialized to zero, a behavior that is also true of many memory-safe platforms—Java for example. But this can degrade performance for some applications, leading other systems to expose allocation primitives that do not initialize memory beforehand, allowing code to inspect whatever stale data happened to be stored in the returned region. This includes mechanisms for memory safety in C [83, 84], but also languages that emphasize safety—for example, OCaml's `Bytes.create` returns an uninitialized array of bytes.

From an integrity perspective, uninitialized memory by itself is not a problem: if a program does not hold a pointer to a live region, it will not be able to change the contents of that region. The problem is that it may lead to secrecy breaches. Suppose that a piece of code should be kept isolated from data stored in some memory region. If that region is freed and reused when that piece of code allocates a new region, its contents will be made available, and isolation will be compromised.

It might be possible to prove a noninterference result for a system with uninitialized memory by exposing the state of free memory cells, and adding a hypothesis that says that the two initial states we run the program on agree on this free memory. Such a result could be useful in conjunction with common programming idioms that clear memory regions that contain sensitive data before those regions are freed.

4.7.4 Dangling Pointers

When we allocate a object in a language with garbage collection, we know it cannot be referenced by other pointers in the program state: the only pointers are those that point to live memory regions, and allocation returns objects that are disjoint from those. This discipline prevents forms of "passive" pointer forging, where a program component gains privilege to an object it was not meant to access, such as in the following snippet:

```
free(x);
/* ... */
int *y = malloc(42);
y[0] = secret;
/* ... */
printf("secret is %d\n", x[0]);
```

Even if the program did not mean to make the variable `secret` available to the piece of code that prints to the screen, that data may still be leaked if the memory region that `x` referenced happened to be reused in the call to `malloc`. Similarly, integrity would also be compromised in such a situation, as the code that controlled `x` could also have changed the contents of that new region arbitrarily.

Our policy controls dangling pointers by always marking new memory regions with fresh identifiers that do not occur anywhere else in the program state. Other systems adopt similar strategies, using some form of identifier to distinguish between regions allocated at different points of execution. Examples from the literature include the CETS compiler transformation [83], and the tagging scheme of Clause et al. [19], which inspired our policy.

The use of identifiers brings interesting practical issues. For efficiency, the two example systems given above, as well as the implementation of the memory-safety policy on the PUMP [33], can only support a finite number of identifiers. If a large enough number is supported, then the system behaves as if there were infinitely many available for all practical purposes. For instance, even if a new object were allocated on every cycle on a 3GHz machine, it would take approximately 200 years for CETS to run out of identifiers, which are 64-bits long. When this happens, to preserve strong guarantees, a system could simply stop execution, or in theory garbage-collect stale identifiers on its state. Another possibility is to simply reuse identifiers, hoping that a collision between a new block and a dangling pointer would be unlikely. However, there are systems that adopt a much smaller number of possible identifiers for performance reasons, in which case collisions become much more likely to be exploited by an attacker. Clause et al., for example, did experiments with their scheme using a much smaller number of colors (2, 4, or 16). It is also worth noting that some systems attempt to provide memory safety guarantees that choose not to protect programs against dangling pointers, treating these an orthogonal issues. SoftBound [84], for instance, focuses solely on bounds checking, and is not capable of preventing the issues described here by itself.

In addition to referencing unintended regions, the use of dangling pointers can have other disastrous consequences for program security. Freeing a dangling pointer can disturb allocator invariants, enabling many practical attacks [105].

4.7.5 Finite Memory

The two noninterference results that we have proved, Theorems 4.12 and 4.13, reveal an important difference between the security guarantees of the idealized memory-safe abstract machine and those of the micro-policy. Noninterference for the abstract machine guarantees that changing unreachable memory in arbitrary ways has no significant effect on program execution. On the symbolic machine, however, this result only holds for successful executions: if one of the executions leads to an error, there is nothing interesting we can say about the other. The reason for that is that modifying the amount of allocated memory at the initial execution state can cause a program to run out of memory and terminate execution.

We have seen a similar situation in the information-flow world: the termination-insensitive noninterference property that we proved in Chapter 3 only prevents information leaks if both executions terminate successfully. By analogy, we could call the variant of noninterference satisfied by the memory-safety policy *error-insensitive*. Many of the problems discussed in the context of information-flow control carry over to this setting [5]. Our version of error-insensitive noninterference can only leak one bit of secret information. However, this could change if outputs can be produced during execution, or in systems like Java, where memory exhaustion does not trigger a fatal error, but rather an exception that can be caught and treated. In any case, we could hope to show that the only information that can be leaked in this way is the total memory consumption of the process, and likely not anything about the contents of memory regions.

4.8 Discussion and Related Work

Memory safety has been one of the main applications for micro-policies and the PUMP. Earlier work presented policies for spatial and temporal protection of the heap [11, 33, 34] (inspired by an approach due to Clause et al. [19]), and other designs for stack protection are underway [91].

There is an extensive literature on hardware and software mechanisms for enforcing memory safety; we overview just a few recent ones that are particularly relevant for the micro-policy presented here. The SAFE platform [25], where the PUMP has its roots, provided memory safety with dedicated hardware fat pointers that compress base, bounds, and offset in 64 bits [73]. The mechanism obviates the need for marking memory locations and pointers with region identifiers, but it requires metadata to distinguish pointers from other kinds of values and to preserve the integrity of their base and bounds—only the highly privileged memory manager is capable of manipulating this information, similarly to our system services. HardBound [30] provides hardware support for fat pointers by storing base and bounds information in a separate address space. This metadata can be manipulated with special instructions provided by the architecture. These instructions can be executed by regular user code, and pointer integrity is guaranteed by having the compiler use them only at the right places. SoftBound [84] is a subsequent software implementation of a similar scheme. CHERI [113] is a RISC architecture with first-class support for capabilities. These capabilities are fat pointers with additional permission metadata that track which operations can be performed through that pointer—for example, read, write, or execute. They are manipulated by special instructions inserted by the compiler, and stored in conventional memory. To preserve capability integrity, the memory tracks where capabilities are stored, and declares them as invalid as soon as they are modified by conventional instructions. The micro-policy presented here sits somewhere in between hardware and software approaches: it takes advantage of specialized hardware support, but the hardware is much more generic than other mechanisms, and still needs to be customized to enforce memory safety.

Motivated by the critical security implications, many other works have proposed formal characterizations of memory safety, and used them to evaluate the security guarantees of mechanisms for enforcing it. These prior proposals mostly focus on guaranteeing that a particular list of erroneous operations never arise during execution. For example, SoftBound [84] guarantees that no memory access occurs out of bounds, while Cyclone [47], a language for safe manual memory management, prevents type errors and references to dangling pointers. In contrast, the characterization presented here enables reasoning about high-level data-isolation properties in programs. Related principles had already been recognized in other works. We have closely analyzed the case of separation logic [90], whose frame rule is similar to our results. However, as argued, separation logic was designed to apply to radically different settings from ours, where safety is established through proof, and where we can rely on little support from the language. By contrast, our characterization leverages the guarantees that are provided in memory-safe settings unconditionally.

There is also an extensive literature on reasoning principles for rich languages featuring isolation guarantees similar to the frame rule that apply unconditionally to all programs, such as L^3 [2] or LNL_D [70], phrased as systems of logical relations for reasoning about typed programs. However, these other systems tend to rely on much more expressive linguistic mechanisms than memory safety alone, such as linear type systems, or encapsulation features that control the operations that components of a program are allowed to perform on objects allocated in memory. Our characterization, on the other hand, is more directly applicable to the lower-level systems where the lack of memory safety is a concern.

We believe our characterization could serve as a criterion to guide possible strengthenings and variations of the micro-policy described here. A clear shortcoming of the current micro-policy is the isolation guarantees are too strong: once a region of memory becomes unreachable, there is no way the program can ever interact with it again. In practice, we want additional mechanisms for telling when and where a program is allowed to manipulate certain memory regions. For example, in the imperative language of Section 4.5, reachability is relative to the set of local variables mentioned by a program. If some part of the program does not mention some local variable x, then we know that that part of the program does not interfere with memory regions that can only be reached through x. Other parts of the program, however, are free to use this otherwise unreachable region. Concretely, we could consider strengthening the memory-safety micro-policy by including a protected stack [91], intra-object bounds checking (as done by SoftBound [84], for example), a basic distinction between code and data, or pointer narrowing for preventing code that receives a pointer from gaining access to the whole memory region it corresponds to. Though these additions would make the policy more complex, we believe that they may not have a huge impact on the proof on noninterference. On a different direction, we could replace the current memory allocator, which initializes every location in a new region to $0@\mathtt{D}(i, \mathtt{UNINIT})$, where i is a fresh identifier, to a different allocator that only sets the region tags to $\mathtt{D}(i, \mathtt{UNINIT})$, where UNINIT is a new data tag for uninitialized values. Given that many policies tag memory sparsely, it is conceivable that there could be architectural support for tagging large regions of memory at once, which could lead to a faster policy [91]. Furthermore, thanks to the UNINIT tag, it would still be possible to detect illegal uses of uninitialized memory, thereby preventing secrecy violations like the ones discussed above.

Chapter 5

Concrete Machine

The symbolic machine is a convenient tool for describing security policies for low-level code. Though we can deploy these policies by emulating the symbolic machine in software, the machine was designed to be backed by efficient hardware. We now discuss how this can be achieved through a low-level *concrete machine* that checks and propagates metadata tags by combining a hardware cache with software rules programmed in machine code. We prove that the concrete machine can refine the symbolic machine for an arbitrary choice of its parameters by instantiating it with correct implementations of the transfer function and system services and using the tagging mechanism itself to protect the monitor's integrity, isolating it from user code.

5.1 Machine State

To user code, there are barely any differences between the concrete machine and its symbolic counterpart: both are simple RISC processors with metadata tags accompanying every word. What distinguishes them is how tags are checked and propagated. While the symbolic machine is tailored for specification and reasoning, the concrete machine is closer to a practical processor with programmable tags. Its tags are not structured objects, but bare machine words. Instead of monitoring execution by feeding tags into an abstract transfer function, the concrete machine uses an implementation of the transfer function in machine code to decide whether an instruction is valid or not. To achieve good performance, the transfer function is not actually run on every instruction, but memoized using a hardware cache: after computing the result of the function for some input, this information is stored on the cache, so that it can be retrieved later without running the function again. This combination allows the concrete machine to retain the flexibility of the symbolic one without giving up on much performance.

Definition 5.1. A state of the concrete machine is a tuple of the form $(m, rs, pc, pc_f, cache)$, where

- $m \in \mathsf{Word} \rightharpoonup \mathsf{Word} \times \mathsf{Word}$ is the *memory*;
- $rs \in \mathsf{Reg} \rightharpoonup \mathsf{Word} \times \mathsf{Word}$ is the *register bank*;
- $pc \in \mathsf{Word} \times \mathsf{Word}$ is the *PC*;
- $pc_f \in \mathsf{Word} \times \mathsf{Word}$ is the *faulting PC*; and
- $cache \in \mathsf{IVec} \rightharpoonup \mathsf{OVec}$ is the *rule cache*.

The sets IVec and OVec contain *input and output tag vectors*. An *input vector* is a tuple of the form $(o, t_{pc}, t_i, t_{a1}, t_{a2}, t_d)$, where o is an opcode, and the other elements are words. An *output vector* is a pair of words (t_{pc}, t_{res}).

Since tags are full-sized words, there is a wide range of meanings that we can ascribe to them, from records of smaller bit vectors to pointers to complex data structures stored in memory. There is no arbitrary limit on how many input-output lines can be stored in the rule cache. This contrasts with the typical size constraints faced by real hardware, which might implement the rule cache with a much smaller (for instance, 512-line) near-associative hash table [32]. Since transfer functions are pure, we can use the concrete machine to implement most micro-policies without the need for evicting lines from the cache. Note that this may no longer hold if policies need to change the meaning of tags during execution for performance reasons—for example, to rearrange the memory data structures they point to.

In addition to the usual machine registers, there is a special-purpose register pc_f that saves the value of the PC on a cache miss, so that control can resume at the faulting point later. The concrete machine omits the extra state component of the symbolic machine, which must now be interpreted as data structures in memory manipulated by regular machine code.

Format	Description
AddRule	Add line to rule cache
JumpEpc	Transfer control to faulting instruction
GetTag r_s r_d	Extract tag from word
PutTag r_s r_{tag} r_d	Pack payload and tag into an atom

Figure 5.1: New instructions for concrete machine. From now on, any mentions to the sets Instr and Opcode implicitly include these new instructions (including Definition 5.1).

5.2 Semantics

The symbolic machine makes it impossible for user code to manipulate tags directly; every interaction is mediated by the system services and the transfer function of the monitor, which execute independently of the main processor. On the concrete machine, however, the monitor is just another program. The concrete machine simply provides basic functionality to isolate the monitor from user code, and new instructions for manipulating tags, summarized in Figure 5.1.

On each step, the machine constructs the input vector iv from the current instruction opcode and the relevant tags and looks it up in the cache. If a matching rule is found, the machine proceeds with the execution of the instruction, extracting the tags of its results from the rule. If no rule matches, then iv is saved in memory, the current PC value is saved in pc_f, and control is transferred to a fixed address w_{trap} where a *miss handler* should be loaded. The job of the miss handler is to analyze the input vector and decide whether that combination of tags is valid or not. If the combination is valid, the handler should update the cache, specifying which output vector should be used for them, and then transfer control back to the faulting instruction. If the combination is invalid, the miss handler should halt the machine.

The machine provides a special `MONITOR` tag to identify when the monitor is running (by using it to mark the PC), and to mark private monitor data to isolate it from user code. This tag influences the behavior of the cache through a set of *masks*.

Definition 5.2. An *input mask* is a quintuple of Booleans $(b_{pc}, b_i, b_{a1}, b_{a2}, b_d)$. An *output mask* is a pair (t_{pc}, t_r), where both components belong to $\{\bot, 1, 2, 3, 4, 5\}$.

The semantics of the machine is parameterized by function masks that maps a Boolean b and an opcode o to a pair $\mathsf{masks}(b, o)$ of an input mask and an output mask. The Boolean b represents whether the current PC is tagged `MONITOR`, allowing the cache behavior to change depending on

$$\mathsf{storeIVec}(m, o, t_{pc}, t_i, t_{a1}, t_{a2}, t_d) \triangleq m[0, ..., 5 \mapsto o@\mathtt{MONITOR}, ..., t_d@\mathtt{MONITOR}]$$

$$\mathsf{maskIn}(im, ts)_i \triangleq \begin{cases} ts_i & \text{if } im_i = \mathsf{false} \\ - & \text{otherwise} \end{cases}$$

$$\mathsf{maskOut}(ts, om, ov)_i \triangleq \begin{cases} ts_j & \text{if } om_i = j \neq \bot \\ ov_i & \text{otherwise} \end{cases}$$

$$\frac{\mathsf{masks}(ts_2 = \mathtt{MONITOR}, o) = (im, om) \qquad cache(o, \mathsf{maskIn}(im, ts)) = ov}{\mathsf{lookup}(cache, o, ts) = \mathsf{maskOut}(ts, om, ov)}$$

Figure 5.2: Concrete machine auxiliary functions

whether the monitor is currently running or not. The input mask is used to replace the fields of an input vector marked as "true" by a special dummy tag $-$. For instance an input mask $(0, 0, 1, 1, 0)$ turns the iv $(op, 1, 2, 3, 4, 5)$ into $(op, 1, 2, -, -, 5)$ before the cache lookup. This is an optimization to allow a single cache line to handle multiple input vectors. When an output mask for a field is defined (that is, different from $\bot$), the concrete machine uses the corresponding input tag as the result tag, instead of using those listed in the cache. For instance, an output mask of $(1, \bot)$ instructs the machine to assemble an output vector by combining the PC component of the iv with the second component of the output vector found in the cache.

In Section 5.3, we describe how to use to the mask mechanism to protect the monitor code, granting it privilege to use all data stored in the machine while preventing user code from directly interfering with the monitor's code and data.

The rules for the concrete machine use a set of auxiliary functions summarized in Figure 5.2. The storeIVec function stores an input vector into the first six positions of a memory, tagging all the locations with MONITOR. We assume that there is a fixed encoding of opcodes as words to make this possible. The maskIn and maskOut functions apply masks to input and output vectors; the indices in their definition refer to the corresponding positions in the vectors. Finally, lookup tries to find an input vector in the cache, while applying the necessary masks. Note the $ts_2 = \mathtt{MONITOR}$ test, to determine which set of masks to apply.

Figures 5.3 to 5.6 define the semantics of each instruction for the concrete machine; Figures 5.3 and 5.4 cover the cases of cache hits, while Figures 5.5 and 5.6 cover cache misses. When the instruction does not use some of tags of the input vector, those are filled in with the fixed dummy tag "$-$." Note that even the new instructions of Figure 5.1, which have the power to manipulate tags and the cache, undergo the same checks as regular instructions. The $\mathsf{AddRule}$ instruction updates the cache by reading the new cache line from the first eight positions in memory. Once again, this only allows adding inputs to the cache: there is no mechanism for deleting an input. The only interaction with the faulting PC register is done through cache misses, which set it to the current PC, and through the $\mathsf{JumpEpc}$ instruction, which uses it as the next PC.

5.3 Monitor

We now discuss how to use the concrete machine to implement the symbolic machine. Assuming that we have already fixed a policy for the symbolic machine that we want to encode, the first step is to find a way of representing symbolic tags as bare words for the concrete machine. Since we want to use the concrete machine to support arbitrary policies, we take this encoding as a parameter; however, we impose a few constraints on the encoding to ensure that the monitor is properly isolated from user code.

The tag encoding is specified by partial functions for interpreting words as symbolic tags.

Definition 5.3. The *decoding functions* of a policy implementation are three partial functions dec_M, dec_PC and dec_R

$$\begin{aligned} \mathsf{dec}_\mathsf{M} &\in (\mathsf{Word} \rightharpoonup \mathsf{Word} \times \mathsf{Word}) \times \mathsf{Word} \rightharpoonup \{\mathsf{User}(t) \mid t \in \mathsf{Tag}_m\} \cup \{\mathsf{Entry}\} \\ \mathsf{dec}_\mathsf{PC} &\in (\mathsf{Word} \rightharpoonup \mathsf{Word} \times \mathsf{Word}) \times \mathsf{Word} \rightharpoonup \mathsf{Tag}_{pc} \\ \mathsf{dec}_\mathsf{R} &\in (\mathsf{Word} \rightharpoonup \mathsf{Word} \times \mathsf{Word}) \times \mathsf{Word} \rightharpoonup \mathsf{Tag}_r. \end{aligned}$$

We require that each of these functions dec satisfy the following properties:

- $m(w) = x_1@t_1$, $\mathsf{dec}_\mathsf{M}(m, t_1) = \mathsf{User}(t_1')$, and $\mathsf{dec}_\mathsf{M}(m, t_2) = \mathsf{User}(t_2')$ imply $\mathsf{dec}(m[w \mapsto x_2@t_2], t) = \mathsf{dec}(m, t)$ for every m, x_1, x_2, t_1, t_2, t_1', t_2', w, and t; and
- $\mathsf{dec}(m, \mathtt{MONITOR}) = \bot$.

Each of these functions takes as arguments a memory m and a tag t, and return the symbolic tag t' that corresponds to t with respect to m; if no such tag exists, the decoding function is undefined.

$$\textsc{Nop}\quad \frac{m(w_{pc}) = i@t_i \qquad \textsf{decode}\ i = \textsf{Nop} \qquad \textsf{lookup}(cache, \textsf{Nop}, t_{pc}, t_i, -, -, -) = (t'_{pc}, \_)}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m, rs, (w_{pc}{+}1)@t'_{pc}, pc_f, cache)}$$

$$\textsc{Const}\quad \frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \textsf{decode}\ i = \textsf{Const}\ i\ r_d \qquad rs(r_d) = \_@t_d \\ \textsf{lookup}(cache, \textsf{Const}, t_{pc}, t_i, t_d, -, -) = (t'_{pc}, t'_d) \qquad rs[r_d \mapsto i@t'_d] = rs' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m, rs', (w_{pc}{+}1)@t'_{pc}, pc_f, cache)}$$

$$\textsc{Mov}\quad \frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \textsf{decode}\ i = \textsf{Mov}\ r\ r_d \qquad rs(r) = w@t \\ rs(r_d) = \_@t_d \qquad \textsf{lookup}(cache, \textsf{Mov}, t_{pc}, t_i, t, t_d, -) = (t'_{pc}, t'_d) \qquad rs[r_d \mapsto i@t'_d] = rs' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m, rs', (w_{pc}{+}1)@t'_{pc}, pc_f, cache)}$$

$$\textsc{Binop}\quad \frac{\begin{array}{c} m(w_{pc}) = i@t_i \\ \textsf{decode}\ i = \textsf{Binop}_{\oplus}\ r_1\ r_2\ r_d \qquad rs(r_1) = w_1@t_1 \qquad rs(r_2) = w_2@t_2 \qquad rs(r_d) = \_@t_d \\ \textsf{lookup}(cache, \textsf{Binop}_{\oplus}, t_{pc}, t_i, t_1, t_2, t_d) = (t'_{pc}, t'_d) \qquad rs[r_d \mapsto (w_1 \oplus w_2)@t'_d] = rs' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m, rs', (w_{pc}{+}1)@t'_{pc}, pc_f, cache)}$$

$$\textsc{Load}\quad \frac{\begin{array}{c} m(w_{pc}) = i@t_i \\ \textsf{decode}\ i = \textsf{Load}\ r\ r_d \qquad rs(r) = w@t \qquad m(w) = w'@t' \qquad rs(r_d) = \_@t_d \\ \textsf{lookup}(cache, \textsf{Load}, t_{pc}, t_i, t, t', t_d) = (t'_{pc}, t'_d) \qquad rs[r_d \mapsto w'@t'_d] = rs' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m, rs', (w_{pc}{+}1)@t'_{pc}, pc_f, cache)}$$

$$\textsc{Store}\quad \frac{\begin{array}{c} m(w_{pc}) = i@t_i \\ \textsf{decode}\ i = \textsf{Store}\ r_p\ r_s \qquad rs(r_p) = w_p@t_p \qquad rs(r_s) = w_s@t_s \qquad m(w_p) = \_@t_d \\ \textsf{lookup}(cache, \textsf{Store}, t_{pc}, t_i, t_p, t_s, t_d) = (t'_{pc}, t'_d) \qquad m[w_p \mapsto w_s@t'_d] = m' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m', rs, (w_{pc}{+}1)@t'_{pc}, pc_f, cache)}$$

$$\textsc{Jump}\quad \frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \textsf{decode}\ i = \textsf{Jump}\ r \\ rs(r) = w'_{pc}@t \qquad \textsf{lookup}(cache, \textsf{Jump}, t_{pc}, t_i, t, -, -) = (t'_{pc}, \_) \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m, rs', w'_{pc}@t'_{pc}, pc_f, cache)}$$

Figure 5.3: Semantics of concrete machine: successful cache look up

Bnz

$$\frac{m(w_{pc}) = i@t_i \qquad \textsf{decode}\ i = \textsf{Bnz}\ r\ n \qquad rs(r) = w@t \qquad \textsf{lookup}(cache, \textsf{Bnz}, t_{pc}, t_i, t, -, -) = (t'_{pc}, \_) \qquad w'_{pc} = \text{if } w = 0 \text{ then } w_{pc} + 1 \text{ else } w_{pc} + n}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m, rs, w'_{pc}@t'_{pc}, pc_f, cache)}$$

Jal

$$\frac{m(w_{pc}) = i@t_i \qquad \textsf{decode}\ i = \textsf{Jal}\ r \qquad rs(r) = w'_{pc}@t_1 \qquad rs(r_a) = \_@t_{ra} \qquad \textsf{lookup}(cache, \textsf{Jal}, t_{pc}, t_i, t_1, t_{ra}, -) = (t'_{pc}, t'_{ra}) \qquad rs[r_a \mapsto (w_{pc}{+}1)@t'_{ra}] = rs'}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m, rs', w'_{pc}@t'_{pc}, pc_f, cache)}$$

AddRule

$$\frac{m(w_{pc}) = i@t_i \qquad \textsf{decode}\ i = \textsf{AddRule} \qquad \textsf{lookup}(cache, \textsf{AddRule}, t_{pc}, t_i, -, -, -) = (t'_{pc}, \_) \qquad m(0, ..., 7) = (o, t_1, t_2, t_3, t_4, t_5, t_6, t_7) \qquad \forall iv, cache'(iv) = (\text{if } iv = (o, t_1, t_2, t_3, t_4, t_5) \text{ then } (t_6, t_7) \text{ else } cache(iv))}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m, rs', (w_{pc}{+}1)@t'_{pc}, pc_f, cache')}$$

JumpEpc

$$\frac{m(w_{pc}) = i@t_i \qquad \textsf{decode}\ i = \textsf{JumpEpc} \qquad \textsf{lookup}(cache, \textsf{JumpEpc}, t_{pc}, t_i, t_{pc_f}, -, -) = (t'_{pc}, \_)}{(m, rs, w_{pc}@t_{pc}, w_{pc_f}@t_{pc_f}, cache) \to (m, rs, w_{pc_f}@t'_{pc}, w_{pc_f}@t_{pc_f}, cache)}$$

GetTag

$$\frac{m(w_{pc}) = i@t_i \qquad \textsf{decode}\ i = \textsf{GetTag}\ r_s\ r_d \qquad rs(r_s) = w@t \qquad rs(r_d) = \_@t_d \qquad \textsf{lookup}(cache, \textsf{GetTag}, t_{pc}, t_i, t, t_d, -) = (t'_{pc}, t'_d) \qquad rs[r_d \mapsto t@t'_d] = rs'}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m, rs', (w_{pc}{+}1)@t'_{pc}, pc_f, cache)}$$

PutTag

$$\frac{m(w_{pc}) = i@t_i \qquad \textsf{decode}\ i = \textsf{PutTag}\ r_s\ r_{tag}\ r_d \qquad rs(r_s) = w@t_1 \qquad rs(r_{tag}) = t@t_2 \qquad rs(r_d) = \_@t_d \qquad \textsf{lookup}(cache, \textsf{PutTag}, t_{pc}, t_i, t_1, t_2, t_d) = (t'_{pc}, \_) \qquad rs[r_d \mapsto w@t] = rs'}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m, rs', (w_{pc}{+}1)@t'_{pc}, pc_f, cache)}$$

Figure 5.4: Semantics of concrete machine: successful cache look up (continued)

Nop-Miss

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{Nop} \\ iv = (\mathsf{Nop}, t_{pc}, t_i, -, -, -) \qquad \mathsf{lookup}(cache, iv) = \bot \qquad \mathsf{storeIVec}(m, iv) = m' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m', rs, w_{trap}@\mathtt{MONITOR}, w_{pc}@t_{pc}, cache)}$$

Const-Miss

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{Const}\ i\ r_d \qquad rs(r_d) = \_@t_d \\ iv = (\mathsf{Const}, t_{pc}, t_i, t_d, -, -) \qquad \mathsf{lookup}(cache, iv) = \bot \qquad \mathsf{storeIVec}(m, iv) = m' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m', rs, w_{trap}@\mathtt{MONITOR}, w_{pc}@t_{pc}, cache)}$$

Mov-Miss

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{Mov}\ r\ r_d \qquad rs(r) = w@t \qquad rs(r_d) = \_@t_d \\ iv = (\mathsf{Mov}, t_{pc}, t_i, t, t_d, -) \qquad \mathsf{lookup}(cache, iv) = \bot \qquad \mathsf{storeIVec}(m, iv) = m' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m', rs, w_{trap}@\mathtt{MONITOR}, w_{pc}@t_{pc}, cache)}$$

Binop-Miss

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \\ \mathsf{decode}\ i = \mathsf{Binop}_{\oplus}\ r_1\ r_2\ r_d \qquad rs(r_1) = w_1@t_1 \qquad rs(r_2) = w_2@t_2 \qquad rs(r_d) = \_@t_d \\ iv = (\mathsf{Binop}_{\oplus}, t_{pc}, t_i, t_1, t_2, t_d) \qquad \mathsf{lookup}(cache, iv) = \bot \qquad \mathsf{storeIVec}(m, iv) = m' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m', rs, w_{trap}@\mathtt{MONITOR}, w_{pc}@t_{pc}, cache)}$$

Load-Miss

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \\ \mathsf{decode}\ i = \mathsf{Load}\ r\ r_d \qquad rs(r) = w@t \qquad m(w) = w'@t' \qquad rs(r_d) = \_@t_d \\ iv = (\mathsf{Load}, t_{pc}, t_i, t, t', t_d) \qquad \mathsf{lookup}(cache, iv) = \bot \qquad \mathsf{storeIVec}(m, iv) = m' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m', rs, w_{trap}@\mathtt{MONITOR}, w_{pc}@t_{pc}, cache)}$$

Store-Miss

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \\ \mathsf{decode}\ i = \mathsf{Store}\ r_p\ r_s \qquad rs(r_p) = w_p@t_p \qquad rs(r_s) = w_s@t_s \qquad m(w_p) = \_@t_d \\ iv = (\mathsf{Store}, t_{pc}, t_i, t_p, t_s, t_d) \qquad \mathsf{lookup}(cache, iv) = \bot \qquad \mathsf{storeIVec}(m, iv) = m' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m', rs, w_{trap}@\mathtt{MONITOR}, w_{pc}@t_{pc}, cache)}$$

Jump-Miss

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{Jump}\ r \qquad rs(r) = w'_{pc}@t \\ iv = (\mathsf{Jump}, t_{pc}, t_i, t, -, -) \qquad \mathsf{lookup}(cache, iv) = \bot \qquad \mathsf{storeIVec}(m, iv) = m' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m', rs, w_{trap}@\mathtt{MONITOR}, w_{pc}@t_{pc}, cache)}$$

Figure 5.5: Semantics of concrete machine: cache miss

Bnz-Miss

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{Bnz}\ r\ n \qquad rs(r) = w@t \\ iv = (\mathsf{Bnz}, t_{pc}, t_i, t, -, -) \qquad \mathsf{lookup}(cache, iv) = \bot \qquad \mathsf{storeIVec}(m, iv) = m' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m', rs, w_{trap}@\texttt{MONITOR}, w_{pc}@t_{pc}, cache)}$$

Jal-Miss

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{Jal}\ r \qquad rs(r) = \_@t_1 \qquad rs(r_a) = \_@t_{ra} \\ iv = (\mathsf{Jal}, t_{pc}, t_i, t_1, t_{ra}, -) \qquad \mathsf{lookup}(cache, iv) = \bot \qquad \mathsf{storeIVec}(m, iv) = m' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m', rs, w_{trap}@\texttt{MONITOR}, w_{pc}@t_{pc}, cache)}$$

AddRule-Miss

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{AddRule} \\ iv = (\mathsf{AddRule}, t_{pc}, t_i, -, -, -) \qquad \mathsf{lookup}(cache, iv) = \bot \qquad \mathsf{storeIVec}(m, iv) = m' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m', rs', w_{trap}@\texttt{MONITOR}, w_{pc}@t_{pc}, cache)}$$

JumpEpc-Miss

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{JumpEpc} \qquad iv = (\mathsf{JumpEpc}, t_{pc}, t_i, t_{pc_f}, -, -) \\ \mathsf{lookup}(cache, iv) = (t'_{pc}, -) \qquad \mathsf{storeIVec}(m, iv) = m' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m', rs', w_{trap}@\texttt{MONITOR}, w_{pc}@t_{pc}, cache)}$$

GetTag-Miss

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \qquad \mathsf{decode}\ i = \mathsf{GetTag}\ r_s\ r_d \qquad rs(r_s) = w@t \qquad rs(r_d) = \_@t_d \\ iv = (\mathsf{GetTag}, t_{pc}, t_i, t, t_d, -) \qquad \mathsf{lookup}(cache, iv) = \bot \qquad \mathsf{storeIVec}(m, iv) = m' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m', rs', w_{trap}@\texttt{MONITOR}, w_{pc}@t_{pc}, cache)}$$

PutTag-Miss

$$\frac{\begin{array}{c} m(w_{pc}) = i@t_i \\ \mathsf{decode}\ i = \mathsf{PutTag}\ r_s\ r_{tag}\ r_d \qquad rs(r_s) = w@t_1 \qquad rs(r_{tag}) = t@t_2 \qquad rs(r_d) = \_@t_d \\ iv = (\mathsf{PutTag}, t_{pc}, t_i, t_1, t_2, t_d) \qquad \mathsf{lookup}(cache, iv) = \bot \qquad \mathsf{storeIVec}(m, iv) = m' \end{array}}{(m, rs, w_{pc}@t_{pc}, pc_f, cache) \to (m', rs', w_{trap}@\texttt{MONITOR}, w_{pc}@t_{pc}, cache)}$$

Figure 5.6: Semantics of concrete machine: cache miss (continued)

	t_{pc} = `MONITOR`			$t_{pc} \neq$ `MONITOR`			
Opcode	b_{a1}, b_{a2}, b_d	c_{pc}	c_r	b_{a1}	b_{a2}	b_d	c_{pc}, c_r
Nop	true	1	$\perp$	true	true	true	$\perp$
Const	true	1	$\perp$	false	true	true	$\perp$
Mov	true	1	3	false	false	true	$\perp$
Binop$_{\oplus}$	true	1	$\perp$	false	false	false	$\perp$
Load	true	1	4	false	false	false	$\perp$
Store	true	1	4	false	false	false	$\perp$
Jump	true	3	$\perp$	false	true	true	$\perp$
Bnz	true	1	$\perp$	false	true	true	$\perp$
Jal	true	3	1	false	false	true	$\perp$
JumpEpc	true	3	$\perp$	true	true	true	$\perp$
AddRule	true	1	$\perp$	true	true	true	$\perp$
GetTag	true	1	$\perp$	false	false	true	$\perp$
PutTag	true	1	$\perp$	false	false	false	$\perp$

Figure 5.7: Masks for standard monitor. The b_{pc} and b_i fields of the input vector are always set to false.

In the case of dec$_{\mathsf{M}}$, the resulting tag can be either User(t'), which corresponds to a real symbolic tag, or Entry, which is used to mark the entry points of system services. The memory argument is needed to allow the encoding to mention data structures stored in memory; the first condition above guarantees that the encoding does not depend on parts of the memory that are marked with user tags. The second condition prevents the implementation from using the special-purpose `MONITOR` tag for anything that is visible for user code.

In what follows, we assume that we have fixed set of decoding functions for the tags of the policy we want to implement. We can define other partial functions decIVec and decOVec that convert input and output vectors to the formats used by the symbolic transfer function, by essentially applying the decoding functions of the appropriate kind pointwise, and ensuring that no memory tags correspond to Entry.

We will soon detail the requirements we impose on concrete monitor code to behave correctly. Before that, however, we specify a set of masks to allow monitor code to execute (Figure 5.7). The masks for user code (that is, when the current PC is not tagged as `MONITOR`) are uninteresting: they do not mask any input tags beyond those that are not used by the instruction, and always use the tags provided by the output vector stored in the cache. In principle, specific user policies could take advantage of the masking mechanism for improved performance, but restricting our attention to the masks defined above does not limit expressive power. The masks for monitor code are more

Opcode	PC, Instr	Args	PC	Res
Nop	MONITOR	—	—	—
Const	MONITOR	—	—	MONITOR
Mov	MONITOR	—	—	—
$\mathsf{Binop}_{\oplus}$	MONITOR	—	—	MONITOR
Load	MONITOR	—	—	—
Store	MONITOR	—	—	—
Jump	MONITOR	—	—	—
Bnz	MONITOR	—	—	—
Jal	MONITOR	—	—	—
JumpEpc	MONITOR	—	—	—
AddRule	MONITOR	—	—	—
GetTag	MONITOR	—	—	MONITOR
PutTag	MONITOR	—	—	—

Figure 5.8: Ground rules for standard monitor

interesting, as they are set to allow the monitor to run with a limited number of rules installed in the cache. First, they mask out all tags on register and memory arguments, to ensure that monitor code is allowed to operate on any kind of argument. Second, they use output masks to propagate input tags directly into the output vector. In most cases, the new PC receives the same tag as the old one (that is, $c_{pc} = 1$), but in other cases, its tag comes from the instruction's argument (when c_{pc} is 3). For JumpEpc in particular, this has the effect of using whatever tag appeared on the faulting PC, allowing the cache miss handler to return to user code. For result tags, most instructions simply use whatever is specified by the cache, except instructions that move their arguments: Mov, Load, Store, and Jal. In these cases, the tags are set to use the tag in whatever atom was moved.

The masks shown above work in cooperation with a set of *ground rules* that are assumed to be always present in the cache (Figure 5.8); it is the job of the monitor to never overwrite such rules. Most rules simply allow the monitor to run following the behavior given by the masks; the only exceptions are Const, Binop, and GetTag, which always tag their results as MONITOR.

We are now ready to describe how the concrete machine implements the symbolic one. As usual, we define a refinement relation that explains when a concrete state represents a symbolic state. This relation comprises several components that say that the abstract data structures of the symbolic machine are correctly represented at the concrete level. The memories of the states are related if they agree on the contents that are visible to user code.

Definition 5.4. Let $m_s \in \mathsf{Word} \rightharpoonup \mathsf{Word} \times \mathsf{Tag}_m$ be a memory of the symbolic machine, and

$m_c \in \mathsf{Word} \rightharpoonup \mathsf{Word} \times \mathsf{Word}$ be a memory of the concrete machine. We say that m_c *refines* m_s, noted $m_c \rhd m_s$, if the following conditions hold.

- If $\mathsf{dec}_\mathsf{M}(m_c, t_c) = \mathsf{User}(t_s)$ and $m_c(w) = x@t_c$, then $m_s(w) = x@t_s$.
- If $m_s(w) = x@t_s$, there exists a concrete tag t_c such that $\mathsf{dec}_\mathsf{M}(m_c, t_c) = \mathsf{User}(t_s)$ and $m_c(w) = x@t_c$.

Because of the requirements on decoding functions, any memory location tagged as `MONITOR` is invisible to user code.

The notion of refinement for register banks is similar, except that the relation needs to take the concrete memory into account, because of the tag encoding.

Definition 5.5. Let $rs_s \in \mathsf{Reg} \rightharpoonup \mathsf{Word} \times \mathsf{Tag}_r$ and $rs_c \in \mathsf{Reg} \rightharpoonup \mathsf{Word} \times \mathsf{Word}$ be register banks, and $m_c \in \mathsf{Word} \rightharpoonup \mathsf{Word} \times \mathsf{Word}$ be a memory. We say that rs_c *refines* rs_s *with respect to* m_c, noted $rs_c \rhd_{m_c} rs_s$, if the following conditions hold.

- If $\mathsf{dec}_\mathsf{R}(m_c, t_c) = t_s$ and $rs_c(r) = x@t_c$, then $rs_s(r) = x@t_s$.
- If $rs_s(r) = x@t_s$, there exists t_c such that $\mathsf{dec}_\mathsf{R}(m_c, t_c) = t_s$ and $rs_c(r) = x@t_c$.

To allow the policy extra state to be implemented, we parameterize the refinement result by a *monitor invariant*. The invariant should be preserved by every monitor operation, and roughly states that the extra state is correctly represented in memory.

Definition 5.6. A *monitor invariant* is a relation I between concrete memories m_c, concrete register banks rs_c, caches $cache$, and symbolic extra states e that satisfies the following properties.

- The invariant does not depend on user memory: if $(m_c, rs_c, cache, e) \in I$, $m_c(w) = x@t_c$, $\mathsf{dec}_\mathsf{M}(m_c, t_c) = \mathsf{User}(t_s)$, and $m_c[w \mapsto x'@t'_c] = m'_c$, then $(m'_c, rs_c, cache, e) \in I$.
- The invariant does not depend on user registers: if $(m_c, rs_c, cache, e) \in I$, $rs_c(r) = x@t_c$, $\mathsf{dec}_\mathsf{R}(m_c, t_c) = t_s$, and $rs_c[r \mapsto x'@t'_c] = rs'_c$, then $(m_c, rs'_c, cache, e) \in I$.
- The invariant is stable under storing input vectors: if $(m_c, rs_c, cache, e) \in I$ and $\mathsf{storeIVec}(m_c, iv) = m'_c$, then $(m'_c, rs_c, cache, e) \in I$.

The concrete machine exposes system services by tagging their entry points accordingly.

Definition 5.7. Let m_c be a concrete memory. We say that its entry points are *well-formed* if the following holds. For every word w, $\mathsf{service}(w) \neq \bot$ if and only if there exists a word x and a concrete tag t such that $m_c(w) = x@t$ and $\mathsf{dec}_{\mathsf{M}}(m_c, t) = \mathsf{Entry}$.[7]

Finally, we need to guarantee that the answers given by the rule cache are correct according to the transfer function of the symbolic machine.

Definition 5.8. Let $cache \in \mathsf{IVec} \rightharpoonup \mathsf{OVec}$ be a cache, and $m_c \in \mathsf{Word} \rightharpoonup \mathsf{Word} \times \mathsf{Word}$ be a memory of the concrete machine. We say that *$cache$ is correct with respect to m_c* if the following holds. For any input vector iv and output vector ov, if $\mathsf{lookup}(cache, iv) = ov$ and $\mathsf{dec}_{\mathsf{PC}}(m_c, t_{pc}) \neq \bot$, where t_{pc} is the PC tag of iv, then there exists iv_s and ov_s such that $\mathsf{decIVec}(m_c, iv) = iv_s$, $\mathsf{decOVec}(m_s, ov) = ov_s$, and $\mathsf{transfer}(iv_s) = ov_s$.

This brings us to the notion of state refinement.

Definition 5.9. Let $s_c = (m_c, rs_c, pc_c@t_c, pc_f, cache)$ be a state of the concrete machine, and $s_s = (m_s, rs_s, pc_s@t_s, e)$ be a state of the symbolic machine. We say that s_c *refines* s_s, written $s_c \rhd s_s$, if and only if

- The PC is refined ($pc_c = pc_s$ and $\mathsf{dec}_{\mathsf{PC}}(m_c, t_c) = t_s$);
- $m_c \rhd m_s$;
- $rs_c \rhd_{m_c} rs_s$;
- $cache$ is correct with respect to m_c;
- the first six positions of m_c (that is, those that store the input vector on a cache miss) are tagged as `MONITOR`;
- the entry points of m_c are well-formed; and
- $(m_c, rs_c, cache, e) \in I$.

[7]The Coq formalization also requires $\mathsf{decode}\ x = \mathsf{Nop}$, to simplify the refinement proof by cutting down the number of possible input vectors to consider at the beginning of a system call. This is useful there because, since system services can delegate some of their checks to the transfer function, the behavior of the first instruction is exposed in the refinement statement.

To actually prove that the concrete machine refines the symbolic one, we must make a few assumptions about the behavior of the monitor code that implements the transfer function and system services. Roughly speaking, these conditions state that (1) upon a cache miss, if the implementation of the transfer function returns to user mode, then the cache of the final state is still correct, and the parts of its state accessible to the user have not changed; and (2) if the implementation of a system service returns to the user, then the corresponding symbolic system service produces a matching symbolic state.

Definition 5.10 (Monitor correctness)**.** We say that the monitor code is correct if the following conditions hold.

- Suppose that $(m_c, rs_c, \mathit{cache}, e) \in I$, $\mathsf{storeIVec}(m_c, iv) = m'_c$, and that cache is correct with respect to m_c. Suppose furthermore that there exists a state s_c whose PC is not tagged `MONITOR`, such that

$$(m'_c, rs_c, w_{trap}@\texttt{MONITOR}, pc_f, \mathit{cache}) \to^* s_c,$$

  and that all the intermediate execution states have PCs that are tagged as `MONITOR`. Let m''_c be the memory of s_c. Then, there exists iv_s and ov_s such that

  - $\mathsf{decIVec}(m_c, iv) = iv_s$;
  - $\mathsf{transfer}(iv_s) = ov_s$;
  - the cache of s_c is correct with respect to m''_c;
  - the first six positions of m''_c are all tagged as `MONITOR`;
  - user tags are unchanged: for all t_c, t_s, and any decoding function dec, $\mathsf{dec}(m''_c, t_c) = t_s$ if and only if $\mathsf{dec}(m_c, t_c) = t_s$;
  - user memory is unchanged: if $\mathsf{dec}_\mathsf{M}(m_c, t_c) = t_s$, then $m_c(w) = x@t_c$ if and only if $m''_c(w) = x@t_c$;
  - user registers are unchanged: if $\mathsf{dec}_\mathsf{R}(m_c, t_c) = t_s$, then $rs_c(r) = x@t_c$ if and only if $rs'_c(r) = x@t_c$, where rs'_c is the register bank of s_c;
  - the PC of s_c is equal to pc_f;
  - the entry points of m''_c are well formed; and

  - the monitor invariant I holds of m''_c, the register bank of s_c, its cache, and e.

- Let $s_c = (m_c, rs_c, pc@t_c, pc_f)$ be a concrete machine state, and $s_s = (m_s, rs_s, pc@t_s, e)$ be a symbolic machine state. Suppose that $(m_c, rs_c, cache, e) \in I$, $m_c \rhd m_s$, $rs_c \rhd_{m_c} rs_s$, $cache$ is correct with respect to m_c, the first six positions of m_c are tagged as `MONITOR`, $\mathsf{service}(pc) = f$, $\mathsf{dec}_{\mathsf{PC}}(m_c, t_c) = t_s$. Suppose furthermore that the cache of s_c currently allows a system service to run, in the sense that $\mathsf{lookup}(cache, iv) \neq \bot$, where iv is the input vector built by the concrete machine when trying to step s_c. Finally, suppose that $s_c \to^* (m'_c, rs'_c, pc'@t'_c, pc'_f, cache')$, where $t'_c \neq$ `MONITOR`, but all the intermediate execution steps go through states whose PCs are tagged `MONITOR`. Then, there exists m_s, rs_s, t'_s and e' such that

  - $f(m_s, rs_s, pc@t_s, e) = (m'_s, rs'_s, pc'@t'_s, e')$;
  - $\mathsf{dec}_{\mathsf{PC}}(m'_c, t'_c) = t'_s$;
  - $m'_c \rhd m'_s$;
  - $rs'_c \rhd_{m'_c} rs'_s$;
  - $cache'$ is correct with respect to m'_c;
  - the first six positions of m'_c are tagged as `MONITOR`;
  - the entry points of m'_c are well-formed; and
  - the monitor invariant I holds of m'_c, rs'_c, $cache'$, and e'.

Since we are only showing refinement in one direction, the first half of Definition 5.10 does not have to assert that the updated cache produced when the miss handler finishes executing actually maps the faulting input vector to the correct output vector; it merely says that, *if* the cache produces any result for that input vector, then the result is correct.

When the correctness conditions of the monitor are satisfied, we can show that the concrete machine refines the symbolic one.

Theorem 5.11. *Suppose that $s_c \rhd s_s$, that the monitor is correct (in the sense of Definition 5.10), and that $s_c \to^* s'_c$, where the PC of s'_c is not tagged as* `MONITOR`. *Then, there exists a symbolic state s'_s such that $s_s \to^* s'_s$ and $s'_c \rhd s'_s$.*

Proof. We decompose the execution of the concrete machine into segments that start and end in user states, and whose intermediate states are all running inside the monitor (that is, their PCs are tagged as `MONITOR`). By induction, it suffices then to prove the result for each segment separately. There are three cases to consider. The simpler cases are when the segments correspond to a cache miss or to the execution of a system service; then, it suffices to appeal to the correctness hypotheses concerning the monitor. The more complicated case is when we hit on the cache; then, we proceed as in the other refinement proofs, arguing that, because the cache is correct, the updated tags it produces match those of the symbolic machine. □

5.4 Discussion and Related Work

Tagged architectures are an old idea, dating back at least to the 1960s [38]. The rise of vulnerabilities in contemporary software has spurred renewed interest in the subject, due to the possibility of using of tags for pervasive and efficient enforcement of various security policies. Recent designs include hardware tagging schemes that are specialized for individual policies—such as memory safety [73, 84], capabilities [113], control data integrity [20], or taint tracking [102]—as well as more general ones that allow the behavior of tags to be programmed in software [16, 28, 97, 110]. The PUMP [25, 33, 34], which directly inspired the model presented here, is one of the most flexible of these designs, allowing policies to use arbitrary data structures as tags, and to tag more parts of the program state, including instructions and PC. While its first incarnation was designed for SAFE [25]—a clean-slate design with built-in support for inter-process communication, stack protection, bounds checking, and context switching, among many other features—recent versions can be layered on top of more conventional architectures [33], including an ongoing port of the PUMP for the RISC V processor [24]. As we have argued, it is still possible to provide formal security guarantees even in the absence of the special capabilities of the SAFE hardware. Despite its generality, the PUMP is fairly efficient: simulations show a run-time overhead of typically less than 10% when enforcing multiple policies simultaneously [33]. Furthermore, although earlier versions of the PUMP required 60% more energy and twice as much memory, ongoing research is working on shrinking those numbers substantially, by relying on a compression scheme that takes advantage of tag locality.

The results reported in this chapter were developed in two publications. The first [9, 10] proved that a low-level machine with PUMP-like infrastructure could be used to enforce information-flow control, and included a complete machine-code implementation of the monitor plus a proof of correctness for this implementation. The second [11] essentially corresponds to the development presented here: it generalizes the first by showing how to encode many other policies through tags, and is carried on a more realistic processor that does not include a protected stack, like the first one did. On the other hand, it does not include implementations of the policy monitors: as in Theorem 5.11, to obtain a complete, fully verified policy, we need to equip the concrete machine with correct implementations of a policy's transfer function and system services. Although verifying programs all the way down to running machine code is challenging, it is an active area of research [4, 18, 61, 67, 85], and the presence of metadata tags should not complicate the issue.

Chapter 6

Conclusion

We have reached the end of this thesis. We close with a brief summary of our contributions and possible directions for future work.

6.1 Summary

This thesis presented the symbolic machine, a new formalism for defining, specifying, and reasoning about micro-policies—expressive dynamic monitors based on fine-grained metadata tags. In his seminal work, Anderson [3] laid out three requirements that dynamic monitors should satisfy.

(a) They must be tamper proof.

(b) They must always be invoked.

(c) They must be small enough to be subject to analysis and tests to assure that they are correct.

Micro-policies for the symbolic machine can fulfill all of these requirements. As every modification to the monitor tags or internal state must be mediated by the transfer function or system services, the machine semantics guarantees the protection of the monitor from user code, freeing the policy designer from this responsibility. The transfer function is invoked on every instruction, guaranteeing that every operation performed by user code is properly inspected. Finally, not only are micro-policies generally quite small, but they can also be fully verified through mechanized proofs, greatly increasing our confidence that their design is correct.

Micro-policies are a good abstraction for programming security monitors. Monitors for a wide variety of purposes have been developed using metadata propagation, including memory safety, control-flow integrity, information-flow control, low-level compartmentalization, taint tracking, and web-session security [9–11, 17, 19, 33]. In addition to these fairly self-contained applications, recent work is demonstrating how micro-policies could be applied to more ambitions goals, such as secure compilation [63]. Thanks to its high-level view of tag propagation, and to the system-service mechanism, the symbolic machine provides a convenient platform for developing micro-policies. We have illustrated this by defining and conducting a detailed analysis of two advanced micro-policies: information-flow control and heap memory safety. We have shown that both policies could be expressed naturally on the symbolic machine, and have connected them to more abstract computation models where the policy is built in, by proving refinement results that show that every behavior of the monitor is also a valid behavior of the abstract models—or, taking the contrapositive, that the monitor halts whenever the abstract machine does so. We have derived high-level extensional properties that describe the security guarantees that apply to programs running under on these abstract machines, and used the refinement results to transfer these properties to the symbolic machine running with these policies installed.

For the information-flow policy, we proved a termination-insensitive noninterference result that we have adapted from the literature, which bounds the influence that secret data stored in the program state have on its outputs. One of the biggest challenges in the design of mechanisms for information-flow control is striking a good balance between soundness and permissiveness. On the one hand, we would like to avoid implicit flows that leak secrets through a program's control flow. On the other hand, we want to be able to determine with good precision what behaviors of the program can be influenced by a given secret, thus avoiding marking too many outputs as classified information. Dynamic information-flow control often addresses this issue by tracking what secrets have influenced control flow in a program-counter label, and by relying on control-flow merge points to downgrade this label. We have shown that this mechanism can be implemented on the symbolic machine with a protected call stack, which is stored in private monitor state and can be accessed by user code through specially designated services.

For the memory-safety policy, we have shown that the execution of a program is independent of memory regions it cannot validly reach through pointers it possesses. Similar properties had been studied previously in the literature, notably in the frame rule of separation logic [90] or systems of logical relations for reasoning about rich languages with encapsulation features [2, 31, 104]; however, none of these previous developments had proposed to use such properties to characterize systems like our monitor, which provide memory safety unconditionally for arbitrary programs and do not require richer mechanisms for controlling access to state, such as objects or capabilities. We believe that our proposed isolation properties form thus the first *extensional* characterization of memory safety. As argued previously, while other works have also proposed characterizations of memory safety, these have mostly been stated in terms of instances of memory misuse—buffer overflows, double frees, etc.—that are prevented by memory safety, without analyzing the consequences of the absence of these errors for reasoning about end-to-end program behavior. We believe that such extensional reasoning principles are a more robust way of evaluating the security guarantees of different mechanisms for enforcing memory safety, and thus of informing their design.

A fundamental aspect of the notion of micro-policy is that it is informed by recent hardware advances. We have formalized a simplified model of these features with a concrete machine that checks and propagates tags by combining a hardware cache and a software monitor. The hardware cache guarantees that the machine can run efficiently on the common case, while the software monitor allows the policy enforced by the concrete machine to be almost as flexible as those given by its symbolic counterpart: if we need to deploy a new monitor in a system to improve its security, it suffices to change this software component; the hardware does not need to change. We proved a result connecting the high-level symbolic machine to the low-level concrete machine, showing that every behavior of the concrete machine is also a valid behavior of the symbolic machine, provided that the concrete machine is instantiated with valid implementations of the micro-policy in machine code. This formal connection hints that many interesting policies can benefit from efficient hardware support without relying on overly specialized mechanisms. For example, while previous versions of the information-flow policy of Chapter 3 required a special-purpose protected stack in hardware, the version presented here demonstrates that it suffices to store the stack in generic monitor memory that is kept private. Although our concrete machine is mostly inspired by the PUMP extension [33], it should be easy to adapt the model to other similar schemes from the literature, like Harmoni [28] or FlexiTaint [110].

Micro-policies and Coq Our methodology was designed to serve as a rigorous, yet convenient platform for reasoning about micro-policies and their security guarantees. The Coq proof assistant

	Component	Definitions	Proofs
Generic	Symbolic Machine	2027	844
	Concrete Machine	619	117
	Miscellaneous	1565	282
Policies	CFI	1699	2973
	Compartmentalization	1809	3158
	Sealing	890	146
	Memory Safety	1533	1660
	IFC	655	706
	Total	10979	10179

Figure 6.1: Size of the components of micro-policy Coq development, measured in lines of code.

was instrumental for achieving this goal, allowing us to build our formalism and policies from first principles, and to be highly confident about their correctness with mechanized proofs. Our current development comprises 10797 lines of definitions and specifications, and 10179 lines of proofs, as measured by the `coqwc` tool. A more detailed count is summarized in Figure 6.1, including some policies that were not discussed in this thesis.

Reasoning at the level of rigor imposed by mechanized proofs is challenging, and managing this complexity requires care. Our most important decision in this respect was how to structure the development. In its earliest versions, micro-policies were programmed and verified directly on the concrete machine, but this soon proved to be unwieldy. This difficulty led us to introduce the symbolic machine as an intermediate step in this workflow, which allowed us to focus on the high-level tagging patterns of each policy and reason about their security while ignoring the caching mechanism of the concrete machine and the correctness of assembly code. A crucial ingredient was the use of Coq's highly expressive programming language to define policies. Besides typical functional programming features, such as algebraic data types and pattern matching, this enabled less conventional idioms, such as the use of dependent types to enforce a simple type discipline on tags, treating them differently depending on their location. Moreover, this generality facilitated reusing other Coq developments. For example, much of the micro-policies development (from its basic definitions to individual policies) built on the Mathematical Components library.[8] In some cases, the saved effort was substantial: the verification of the compartmentalization micro-policy suffered from severe performance problems, until we replaced its ad-hoc definition of finite sets with the one

[8]The Mathematical Components development was crucial a ingredient in recent breakthroughs in formal verification, including the Coq proofs of the four-color theorem [45] and the odd-order theorem [46]. The library is available at `https://math-comp.github.io/math-comp/`.

defined in the Mathematical Components library. The more complex policies become—for example, incorporating richer labels for information-flow control [69, 78]—the more they can benefit existing code bases. Our formalization also prompted us to develop generic, independent libraries, including a theory of finite maps supporting extensional equality, and a basic development of nominal sets, used to reason about the memory-safety policy; they are available at `https://github.com/arthuraa/coq-utils`.

6.2 Future Work

6.2.1 Expressiveness

The micro-policies presented in the literature are good empirical evidence of the expressiveness of the mechanism. To complement this picture, we would like to have a more solid theoretical understanding of what micro-policies can express.

Schneider initiated an extensive research program with collaborators to describe what classes of security properties can be enforced by certain mechanisms, such as execution monitoring or program rewriting [36, 51, 95]. In this framework, programs are modeled as processes that continuously generate sequences of actions as they execute. An execution monitor runs alongside this main program intercepting every action that it produces, and analyzing it to decide whether it is legal or not; in doing so, the monitor may also take previous actions into consideration, as well as the code of the program that is running. The actions model the monitor's observation power: they may include input and output events, access to system resources, or individual instructions, but need not characterize completely the program's execution—some of the program's operations may remain invisible to the monitor.

As a first step to characterize the power of tags, we can devise a scheme for implementing *arbitrary* monitors for an instance of the symbolic machine using only the tagging mechanism itself. The (admittedly inefficient) idea is to extend the parameters of the symbolic machine to include in the PC tag a trace of all states that have occurred during execution. Formally, suppose that we have some instance of the symbolic machine. Let M be a monitor for this instance of the symbolic machine that is allowed to inspect the entire state of the policy—that is, M is a computable predicate over finite sequences of elements of State that is closed under taking prefixes. We define a new set of

$$\mathsf{Tag}'_m = \mathsf{Tag}_m \qquad \mathsf{Tag}'_r = \mathsf{Tag}_r \qquad \mathsf{Extra}' = \mathsf{Extra} \qquad \mathsf{Tag}'_{pc} = \mathsf{Tag}_{pc} \times \mathsf{State}^*$$

$$\mathsf{transfer}'(o, (t_{pc}, ss \cdot [s]), t_r) = \begin{cases} ((t'_{pc}, ss \cdot [s; s']), t'_r) & \text{if } s \to s',\ ss \cdot [s; s'] \in M \\ & \text{and } \mathsf{transfer}(o, t_{pc}, t_r) = (t'_{pc}, t'_r) \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{service}'(w, s) = \begin{cases} (m, rs, pc@(t_{pc}, ss \cdot [s']), e) & \text{if } \mathsf{service}(w, s^-) = s' = (m, rs, pc@t_{pc}, e) \\ & \text{and } ss \cdot [s'] \in M \\ \bot & \text{otherwise.} \end{cases}$$

Figure 6.2: Encoding monitors for the symbolic machine.

parameters for the symbolic machine as depicted in Figure 6.2. There, State^* denotes the set of finite sequences of State, and s^- denotes the state of the original instance of the symbolic machine obtained by removing the trace of states from the PC tag. Paraphrasing in words, this policy simulates the entire execution history of the program on the PC tag, keeping the invariant that the last element of the sequence is equal to the current state with the trace removed from the PC tag. It extends the transfer function of the original policy by computing the next state of the machine (if any exists), and checking whether the extended trace is allowed by the monitor that we want to layer on top of the original machine. If so, it returns the results of the original transfer function, with the extended trace added to the PC tag. If not, it halts execution. The system services of the policy are modified in a similar fashion: it runs the original system service and checks whether the new trace is allowed by the monitor. For this scheme to work, we assume that there is some loader procedure that correctly sets up the initial PC tag based on the initial state of the original machine.

While this encoding might be of theoretical interest, it is too inefficient to be used in practice, as it can never benefit from the caching mechanisms of the PUMP or similar systems: every instruction produces a completely new PC tag. It seems that a coarse, extensional characterization such as those traditionally used to study these enforcement mechanisms is not detailed enough for analyzing the expressive power of the symbolic machine. Indeed, the main motivations for using the symbolic machine are its hardware acceleration model and its convenience for expressing certain policies; if a monitor cannot be optimized to run efficiently with a PUMP-like extension and cannot be easily expressed through tags, there is little reason for preferring an implementation for the symbolic machine over a pure software one. It would be interesting to explore possible refinements of Schneider's theory that account for more intensional or qualitative aspects of monitors, which could help understanding what security properties we would want to enforce using tags.

6.2.2 Realism

Micro-policies for the symbolic machine are powerful, but still not quite capable of protecting programs that run on real computers. We need a platform that can run programs and micro-policies alike—ideally a processor equipped with PUMP-like hardware, like the one currently being developed on top of the RISC V architecture [24], or a virtual machine that emulates a similar environment. For strong security guarantees, we would also need to adapt the symbolic and concrete machines to more closely model this platform. Naturally, it would be possible to port a verified monitor developed for the current version of the symbolic machine to run on the platform. However, the correctness proof of the monitor would simply prevent gross errors on its high-level design; other vulnerabilities could arise from errors introduced during the translation between the two implementations, and also from subtle mismatches between the platform and the machines discussed here.

Adapting our machines to a more realistic computer design should be doable given the current state of the art in formal verification. Relevant examples include the model of the PowerPC architecture used in the CompCert verified C compiler [74]; various models of the x86 architecture, such as the one used in the RockSalt SFI checker [80], and the one developed by Kennedy et al. [65]; and the work of [40] on the ARMv7 architecture. A possible challenge that we foresee is modeling operations that do not work at the granularity of machine words (such as decoding long instructions on x86 processors, or accessing individual bytes within machine words). More recent PUMP designs [24] allow monitor rules to take into account what part of a word is being accessed, but no formal investigation of this programming model has been carried yet.

For our guarantees to cover running machine code, we would also need to relate monitors written for the symbolic machine to machine-code implementations, thus filling in the assumptions needed to establish refinement with Theorem 5.11. In principle, these results could be established by combining a number of existing techniques and tools. Certified compilers like CompCert [74] or CakeML [71] allow us to code with higher-level programming languages instead of directly in assembly; these intermediate representations could then be linked to their specifications on the symbolic machine through refinement [4, 48, 67]. With a certified compiler for Coq, this approach would be even more attractive. (The CertiCoq project is currently building such a compiler.) Another option would be to generate and verify assembly code directly. We followed this approach by building a simple, custom infrastructure to verify an information-flow monitor [9, 10], but there are many other systems available intended for general use; these include Bedrock [18], Fiat [26], XCAP [85], and the work of Jensen, Benton, and Kennedy [61].

6.2.3 Policy Composition

Besides the deployment questions discussed above, there is another feature that would render our formalism more practical: a theory of policy composition. It would be useful to develop security monitors modularly by combining small, independent ones—for instance, enforcing information-flow control in the presence of memory allocation by running the policies of Chapters 3 and 4 "simultaneously." Naturally, for this to make sense, it should be possible to combine not only policies, but also their security guarantees: if a property A_1 holds of the symbolic machine when running with a policy P_1, and if A_2 holds when we use a policy P_2 instead, a sensible composite policy $P_1 \oplus P_2$ should enforce both A_1 and A_2—or a slightly modified version thereof, perhaps.

Thanks to its flexibility, the symbolic machine already supports a rudimentary model of policy composition. In this model, policies are composed *horizontally*, by attaching one tag of each policy to each word, and by running their transfer functions in parallel. The composite policy stops the machine as soon as one of its components decides to do so.

More formally, suppose that we have two micro-policies P_1 and P_2, such that policy P_i has sets of tags Tag^i_{pc}, Tag^i_{m}, and Tag^i_{r}, and a transfer function $\mathsf{transfer}_i$. We assume for now that policies have no private state or system services. We define tags and a transfer function for a policy

$P_1 \oplus P_2$—the horizontal composition of P_1 and P_2—as follows:

$$\begin{aligned}
\mathsf{Tag}_{pc} &= \mathsf{Tag}^1_{pc} \times \mathsf{Tag}^2_{pc} \\
\mathsf{Tag}_{m} &= \mathsf{Tag}^1_{m} \times \mathsf{Tag}^2_{m} \\
\mathsf{Tag}_{r} &= \mathsf{Tag}^1_{r} \times \mathsf{Tag}^2_{r} \\
\mathsf{transfer}(op, ts) &= \begin{cases} ((t^1_{pc}, t^2_{pc}), (t^1_r, t^2_r)) & \text{if } \mathsf{transfer}_i(op, ts.i) = (t^i_{pc}, t^i_r), i = 1, 2 \\ \bot & \text{otherwise,} \end{cases}
\end{aligned}$$

where $ts.i$ denotes the i-th components of the tuple of tags ts.

A pleasant feature of horizontal composition is that it preserves any single-trace properties enforced by the individual policies. (A similar observation had already been made by Schneider in his work on execution monitors [95].) Formally, suppose that we have a set S of program states. Let S^∞ be the set of finite or infinite sequences of elements of S. Suppose that we represent a system by a set E of possible execution traces. Then we say that the system satisfies a property $A \subseteq S^\infty$ when $E \subseteq A$. For instance, A could represent the set of traces that refine an execution of a higher-level machine; then, saying that the property A is satisfied is equivalent to stating refinement with respect to this higher-level machine.

Let E_1 and E_2 represent the execution traces of the symbolic machine instantiated with policies P_1 and P_2, and let E represent the execution traces of the composite policy. If $t \in E$ is an execution trace, let $\pi_i(t) \in E_i$ be the execution trace of the symbolic machine running P_i obtained by deleting all the tags corresponding to the other policy. We know that $\pi_i(t)$ is a valid execution trace of P_i because of the way $P_1 \oplus P_2$ is defined. Now suppose that P_i satisfies a property A_i. Then $P_1 \oplus P_2$ satisfies the property

$$A = \{t \mid \pi_1(t) \in A_1, \pi_2(t) \in A_2\}.$$

Roughly, this means that the traces of the composite policy are valid according to each one of the properties A_1 and A_2. In terms of refinement, suppose that each A_i states that P_i causes the symbolic machine to be the refinement of some higher-level abstract machine. Then A roughly says that $P_1 \oplus P_2$ refines both abstract machines simultaneously.

The problem with horizontal composition is that it does not provide a satisfactory account of system services. Given a system service associated with policy P_1, there is no canonical way to modify its definition to work on machine states that also contain tags from P_2. In general, there might be many different possible modifications, and some of them might break the security property that P_2 is trying to enforce. Consider for example the scenario mentioned earlier, where we try to enforce information-flow control and memory safety simultaneously. If not handled with care, dynamic allocation—which is implemented as a system service—can become a channel for laundering secrets. Though there are many solutions for combining these features [9, 10, 13, 56, 101], they all require clever design to enforce noninterference, and it is not clear if these mechanisms can be expressed by combining two simpler, independent policies in a generic way.

Another potential solution to the composition problem is to compose policies *vertically* instead of horizontally. In this model, a base policy is implemented directly on the symbolic machine, but in such a way that other policies can layered on top of it by specializing its behavior. In a sense, this pattern is an adaptation of the one we adopted for the monitor of the concrete machine. The only function of the monitor of the concrete machine is to wrap another monitor that implements a higher-level policy, isolating it from user-level code. This allows the concrete machine to implement the symbolic machine, which has this isolation built in, and exposes the choice of the high-level monitor in its parameters.

What would this pattern look like when replicated at the symbolic-machine level? Imagine that we wanted to enforce some policy P in the presence of memory safety, and that this policy works by checking and propagating metadata tags drawn from some set T. For instance, P might implement an abstract machine like the one of Hrițcu et al. [56], which combines information-flow control and memory safety. We could do this by adding elements of T to every tag of the memory-safety policy of Chapter 4, leading to the following sets of tags.

$$\mathsf{Tag}_r = \mathsf{Tag}_{pc} = \mathsf{Data} = \mathtt{NPTR}(t \in T) \mid \mathtt{PTR}(i \in \mathsf{Id}, t \in T)$$

$$\mathsf{Tag}_m = \mathtt{FREE} \mid \mathtt{D}(i \in \mathsf{Id}, t_d \in \mathsf{Data}, t_l \in T)$$

Note that free memory locations do not carry any extra information, as they are invisible to user code, and that locations in allocated memory regions carry two tags from T: one attached to the value item stored in the location, and another one attached to the location itself. Harmoni, another hardware accelerator for tag-based monitors, used a similar distinction [28].

To enforce P, we need to modify the behavior of the memory-safety transfer function to take the new tags into account. We could achieve this by calling a transfer function $\mathsf{transfer}_P$ that almost behaves like one for a regular micro-policy on the symbolic machine where all the tag sets are T. When checking the Nop instruction, for example, this modified transfer function could be defined as follows:

$$\begin{aligned}\mathsf{transfer}(\mathsf{Nop}, \mathtt{PTR}(i, t_{pc}), \mathtt{D}(i, \mathtt{NPTR}(t_d), t_l)) &= \mathtt{PTR}(i, t'_{pc}) \\ &\text{if } \mathsf{transfer}_P(\mathsf{Nop}, t_{pc}, t_d, t_l) = t'_{pc}.\end{aligned}$$

We might also define system services for P. Some could work independently of memory safety, but others could be specialized versions of those used for memory safety. For instance, we could have a parameterized malloc that uses some function given by P to decide what tags to put on the locations of the new region and on the returned pointer. This function might take into account tags on the argument passed to malloc, as well on its PC.

In addition to combining policy implementations, vertical composition might also provide limited support for reusing proofs of correctness, just like the general statement of correctness that we proved for the base monitor of the concrete machine (Theorem 5.11). We could modify the correctness criterion of the memory-safety policy to take the micro policy above into account. Instead of proving refinement with respect to a fixed abstract machine, we could generalize the abstract machine of Chapter 4 to make the choice of P visible, thus obtaining a new symbolic machine that has memory safety built in. We could even imagine proving generalizations of the noninterference result for memory safety (Theorem 4.12) that rely on assumptions about P.

Vertical composition would have interesting implications for the concrete machine. Consider a micro-policy for the symbolic machine that has been parameterized to support vertical composition. When fully instantiated, we could choose to implement the complete policy at the concrete level with a single, monolithic monitor—as long as the entire monitor code is fully verified, so that it satisfies the hypotheses needed to prove refinement between the two machines. In this case, the architecture of the monitor at the concrete level, which uses a single `MONITOR` tag to distinguish between trusted and untrusted code, would suffice. However, there could be other deployment scenarios where only part of the parameterized policy is verified. For instance, the code responsible for implementing the base policy could be fully verified, but policy layered on top of it might be implemented by unverified, user-supplied code. To preserve the guarantees of the base monitor, this scheme would require running the concrete machine with three protection domains: the end program, the user monitor, and the fully verified base monitor. The user monitor could have access to the end program, from which it is protected, but should not be allowed to tamper with the base monitor.

Although vertical composition seems to overcome some of the issues related to horizontal composition, the version sketched here it has its own quirks. It requires ordering micro-policies in advance to compose them, forcing a policy's definition to depend on all the others that precede it on the stack—thus, the later a policy comes on the stack, the less reusable it becomes. Furthermore, it is not clear how parameterized base policies should be. For example, the memory-safety policy above could have chosen to take not one, but several sets of tags as parameters, one for each place where tags can occur. There is naturally a trade off between expressiveness and complexity—for example, we could choose to take as parameters sets of tags for memory locations, data stored in those locations, pointers, non-pointer values… At the present state, the problem of composing micro-policies is far from solved.

6.2.4 Micro-policies beyond Assembly

Much of the original motivation for the symbolic machine came from the desire to enforce micro-policies directly at the assembly level. However, many micro-policies are hard to express for assembly programs, because they are formulated in terms of abstractions that exist only at much higher levels—for example, policies used in the back end of recent proposals for secure compilers [63], which may target a C-like execution environment, or policies for web-session security [17]. In such cases, it would make more sense to program the policies directly at an extension of the programming platform in question. For example, we could imagine having an extension of the C programming language with micro-policies, or consider the work of Calzavara et al. [17], who developed a core model of a web browser supporting micro-policies, and used it to enforce web-session security.

Besides the issue of finding the right way of attaching tags to some platform—choosing the granularity of tagging, the different types of tags, etc.—we would like to have a way of running these higher-level policies efficiently. In a sense, this could be solved with the idea of vertical composition discussed above, but taken to an extreme: the top-most layer is a relatively high-level execution model parameterized with tags, and it is refined by symbolic machine at the bottom-most layer, running a composite policy that implements the higher-level one, with possibly a wrapper policy around it.

Appendix A

Mathematical Preliminaries

We describe here basic concepts and notation that are used throughout the text.

A.1 Relations and Partial Functions

A *relation* between two sets X and Y is a subset $R \subseteq X \times Y$. When $X = Y$, we speak of a relation on the set X instead. We often use infix notations to denote relations, e.g. $x \sim y$ to mean that $(x, y) \in \sim$. A relation $\sim$ on set X can be:

Reflexive if $x \sim x$ for every $x \in X$;

Symmetric if $x \sim y$ implies $y \sim x$ for every $x, y \in X$;

Antisymmetric if $x \sim y$ and $y \sim x$ implies $x = y$ for every $x, y \in X$; and

Transitive if $x \sim y$ and $y \sim z$ implies $x \sim z$ for every $x, y, z \in X$.

Given a relation R between X and Y, and another relation S between Y and Z, we define the *composite* relation $S \circ R \subseteq X \times Z$ as $\{(x, z) \mid \exists y \in Y, (x, y) \in R \wedge (y, z) \in S\}$. The *identity relation* on a set X, noted 1_X, is defined as $\{(x, x) \mid x \in X\}$. Given a relation $R \subseteq X \times X$, we define R^n, the n-fold composition of a relation with itself by recursion, as follows:

$$R^0 = 1_X \qquad\qquad R^{n+1} = R \circ R^n = R^n \circ R.$$

The *reflexive transitive closure* of R, noted R^*, is given by the infinite union $\bigcup_{n \in \mathbb{N}} R^n$.

A relation f between two sets X and Y is a *partial function* if $(x, y) \in f$ and $(x, y') \in f$ implies $y = y'$. We say that f is *defined* at an element $x \in X$ if there exists y such that $(x, y) \in f$. We often note this by writing $f(x) \in Y$ or $f(x) \neq \bot$, and use $f(x)$ instead of y. The *domain* of a partial function f, written $\mathsf{dom}(f)$, is the set of input elements for which f is defined. We say that f is *undefined* at $x \in X$ if there isn't any $y \in Y$ such that $(x, y) \in f$; in this case, we write $f(x) = \bot$. We note $X \rightharpoonup Y$ the set of all partial functions from X to Y. A partial function $f \in X \rightharpoonup Y$ is *finite* if its domain is finite. We note $X \rightharpoonup_{\mathrm{fin}} Y$ the set of such partial functions. A partial function $f \in X \rightharpoonup Y$ is *total* (or simply a *function*) if $\mathsf{dom}(f) = X$.

Given a partial function $f \in X \rightharpoonup Y$ and elements $x \in X$ and $y \in Y$ such that $f(x) \neq \bot$, we define a partial function $f[x \mapsto y]$ as follows.

$$f[x \mapsto y] = \{(x', y') \in f \mid x' \neq x\} \cup \{(x, y)\}$$

In words, $f[x \mapsto y]$ agrees with f on every input x', except at x, where its value is y instead of $f(x)$. Whenever we use the notation $f[x \mapsto y]$, we tacitly assume that $f(x) \neq \bot$.

Definition A.1. Given two partial functions $f, g \in X \rightharpoonup Y$ and a binary relation $\sim$ on Y, we say that f and g are *pointwise related by* $\equiv$ if, for all $x \in X$, either $f[x] = g[x] = \bot$, or $f[x]$ and $g[x]$ are both defined, and $f[x] \sim g[x]$.

A.2 Nominal Sets

Nominal sets were introduced by Gabbay and Pitts [41] as a framework to discuss variable binding and name generation. We use the theory to simplify our treatment of dynamic memory allocation in Chapter 4. We only cover a few concepts of the theory here, referring interested readers to Pitts' book [88] for a thorough treatment.

When we allocate an object during the execution of a program, we have to reserve a new region of memory to store it, and return a pointer to that region to the program. In many memory-safe languages, this pointer value is immaterial, in the sense that the actual physical address where the object is stored has little influence on the behavior of the program; the only property that matters is that the new address is an unambiguous reference to the object that we just created, and cannot be used to refer to anything else. This suggests adopting an abstract view of the heap in which a pointer value is simply an element drawn from some infinite set that we can use to reference objects that we have allocated.

Definition A.2 (Identifiers)**.** We fix some countably infinite set $\mathsf{Id} = \{i_0, i_1, i_2, ...\}$ of identifiers. We order identifiers using some fixed enumeration of the set, and define a function $\mathsf{nextId} \in \mathsf{Id} \to \mathsf{Id}$ that maps each identifier to the next one in this enumeration: $\mathsf{nextId}(i_k) \triangleq i_{k+1}$.

In usual presentations of the theory of nominal sets, the nextId function is absent. We include it here because it allows expressing a more concrete implementation of an allocator for our memory-safety policy.

We are interested in the effect that the choice of different identifiers has on the execution of a program. We express these different choices by renaming the pointer identifiers that occur in the state of a program according to some *permutation*. A permutation (of identifiers) is a function $\pi \in \mathsf{Id} \to \mathsf{Id}$ that has a two-sided inverse; that is, there exists π^{-1} such that $\pi \circ \pi^{-1} = \pi^{-1} \circ \pi = 1_{\mathsf{Id}}$, where $1_{\mathsf{Id}} \in \mathsf{Id} \to \mathsf{Id}$ denotes the identity function on Id, and $\circ$ denotes function composition. We note $\mathsf{perm}(\mathsf{Id})$ the set of all such permutations. The identity function 1_{Id} is a permutation. Permutations are closed under function composition and inverses.

A *renaming operation* over a set X is a function that maps a permutation $\pi \in \mathsf{perm}(\mathsf{Id})$ and an element $x \in X$ to another element $\pi \cdot x \in X$. This function must satisfy the following properties.

$$1_{\mathsf{Id}} \cdot x = x$$

$$\pi_1 \cdot (\pi_2 \cdot x) = (\pi_1 \circ \pi_2) \cdot x.$$

We treat renaming as right associative: $\pi_1 \cdot \pi_2 \cdot x$ means $\pi_1 \cdot (\pi_2 \cdot x)$. Typically, elements of X are objects that contain identifiers; for example, a set of program states where identifiers are pointers to memory regions. A renaming operation makes a permutation π act on an element x by replacing every identifier i that occurs on x by $\pi(i)$. The above properties guarantee that it is always possible to undo a renaming, by renaming using the inverse permutation: $\pi^{-1} \cdot \pi \cdot x = x$. It would also be possible to work with renaming operations that take arbitrary functions on identifiers as parameters; however, this would complicate the theory, since it would often be necessary to state supplementary hypotheses saying that the function does not induce any aliasing between identifiers. With permutations, this comes for free.

Set	Id	Discrete (e.g. $\mathbb{N}$)
Renaming	$\pi \cdot i = \pi(i)$	$\pi \cdot x = x$
Support	$\mathsf{supp}(i) = \{i\}$	$\mathsf{supp}(x) = \emptyset$
Set	$X \times Y$	$X \uplus Y$
Renaming	$\pi \cdot (x, y) = (\pi \cdot x, \pi \cdot y)$	$\pi \cdot (k, z) = (k, \pi \cdot z)$
Support	$\mathsf{supp}(x, y) = \mathsf{supp}(x) \cup \mathsf{supp}(y)$	$\mathsf{supp}(k, z) = \mathsf{supp}(z)$
Set	$X \rightharpoonup_{fin} Y$	$\mathcal{P}_{\mathrm{fin}}(X)$
Renaming	$\pi \cdot f = \{(\pi \cdot x, \pi \cdot y) \mid (x, y) \in f\}$	$\pi \cdot X' = \{\pi \cdot x \mid x \in X'\}$
Support	$\mathsf{supp}(f) = \bigcup_{(x,y) \in f} \mathsf{supp}(x) \cup \mathsf{supp}(y)$	$\mathsf{supp}(X') = \bigcup_{x \in X'} \mathsf{supp}(x)$

Figure A.1: Standard nominal sets

Let X be a set endowed with a renaming operation. We say that a set $I \subset \mathsf{Id}$ *supports* an element $x \in X$ if the following condition holds:

$$\forall \pi, (\forall i \in I, \pi(i) = i) \Rightarrow \pi \cdot x = x.$$

An element $x \in X$ is *finitely supported* if there exists a *finite* set $I \subset \mathsf{Id}$ that supports x. Under these conditions, we can show that x possesses a smallest finite supporting set $\mathsf{supp}(x)$; that is, $\mathsf{supp}(x)$ supports x, and $\mathsf{supp}(x) \subseteq I$ whenever I is another finite set supporting x.

Definition A.3. A nominal set is a set X endowed with a renaming operation for which every element is finitely supported.

The support of an object intuitively corresponds to the set of identifiers that occur in that object: if a permutation does not affect any of those identifiers, it should not change the object when renaming it. Crucially, when the support of an element is finite, we can always find new identifiers that do not occur in it (because Id is infinite). This property is useful for defining functions on nominal sets that allocate fresh identifiers.

We use a few standard constructions to build nominal sets summarized in Figure A.1. Identifiers themselves form a nominal set by simply applying a renaming permutation to them; this nominal set forms a "base case" used to define others. We can see every set X as a nominal set under the discrete renaming operation given on the second column; we use this structure for sets like $\mathbb{N}$, whose elements do not have any identifiers in them. The other constructions simply traverse a complex object, applying a renaming operation pointwise to all of its sub-objects. In particular, we can picture the renaming operation on finite partial functions as acting on a table representation of those functions, where each row relates an input to its output.

The last construction of the theory of nominal sets that we need are *nominal restriction sets* [88], which we use to simplify the definition of computations that generate fresh names. We include the details of the construction here for formal completeness, but we believe that a thorough understanding of this material is probably not necessary for having a good intuitive grasp of the results of Chapter 4.

Definition A.4. A partial function $f \in X \rightharpoonup Y$ between two nominal sets is *equivariant* if $(x, y) \in f \Rightarrow (\pi \cdot x, \pi \cdot y) \in f$ for every permutation π, $x \in X$ and $y \in Y$. In particular, $f(x)$ is defined if and only if $f(\pi \cdot x)$ is. Furthermore, when this is the case, $f(\pi \cdot x) = \pi \cdot f(x)$.

Intuitively, an equivariant function is one such that, if applied to two objects that differ only in the choice of their identifiers, propagates this difference to its results without affecting them otherwise. Roughly speaking, any partial function on identifiers that does not mention any identifier constants, and that only manipulates identifiers by testing them for equality is equivariant.

Definition A.5. Let X be a nominal set. A *restriction operation* over X is an equivariant function that maps every every finite set of identifiers I and element $x \in X$ to an element $\nu I.x \in X$, in a way such that the following properties hold.

$$\begin{aligned} I \cap \mathsf{supp}(\nu I.x) &= \emptyset \\ \nu\emptyset.x &= x \\ \nu I.(\nu I'.x) &= \nu(I \cup I').x \\ \nu I.x &= \nu(I \cap \mathsf{supp}(x)).x. \end{aligned}$$

A *nominal restriction set* is a nominal set X equipped with a restriction operation.

Roughly speaking, in the nominal restriction sets that we will encounter, the restriction operation allows us to hide some of the identifiers that appear in an object, making their exact values immaterial. In particular, we will use restriction operations to choose fresh pointers in a language with dynamic allocation, since we want the exact values of fresh identifiers to be invisible. Because of equivariance, the first property implies that $\nu(\pi \cdot I).(\pi \cdot x) = \pi \cdot \nu I.x = \nu I.x$ whenever π fixes all the elements in

$\mathsf{supp}(x) \backslash I$. Thus, the atoms in $\mathsf{supp}(x) \backslash I$ are visible "to the outside world," and cannot be renamed, whereas all those in I can be changed at will. The second property says that hiding no names has no effect on an object. The third property says that we can combine several hiding operations into a single one. Finally, the last property says that only the names that appear in x matter for the restriction operation; hiding a name that does not appear in x has no effect on it.

The main example of restriction operation comes from *free nominal restriction sets* [7, 88, 108].

Definition A.6. Let X be a nominal set. The free nominal restriction set over X, noted $\mathsf{Res}(X)$, is defined as follows.

$$\begin{aligned}
\mathsf{Res}(X) &\triangleq L(X)/\!\sim \\
L(X) &\triangleq \{(I, x) \in \mathcal{P}_{\mathrm{fin}}(\mathsf{Id}) \times X \mid I \subseteq \mathsf{supp}(x)\} \\
(I_1, x_1) \sim (I_2, x_2) &\Leftrightarrow \exists \pi.(I_2, x_2) = \pi \cdot (I_1, x_1) \wedge \forall i \in \mathsf{supp}(x_1) \setminus I_1 . \pi(i) = i
\end{aligned}$$

Given a finite set of identifiers I and an element x of X, we note $[I, x]$ the equivalence class of the pair $(I \cap \mathsf{supp}(x), x)$ in $\mathsf{Res}(X)$. The set $\mathsf{Res}(X)$ can be endowed with the structure of a nominal set, given by

$$\pi \cdot [I, x] = [\pi \cdot I, \pi \cdot x] \qquad \qquad \mathsf{supp}([I, x]) = \mathsf{supp}(x) \setminus I.$$

Furthermore, there is a canonical restriction operation associated with $\mathsf{Res}(X)$, determined by

$$\nu I.[I', x] = [I \cup I', x].$$

Intuitively, an element of the form $[I, x]$ to the result of a computation that produces x as a final result and had to generate fresh identifiers in I during its execution. Since $[I, x] = \nu I.[\emptyset, x]$, this means that these fresh identifiers are hidden from the outside and can be renamed.

For an example of restriction in use, suppose that x is a partial function of the form $\{(i, 1)\}$, representing a heap that has exactly one allocated cell in it at address i, and that cell stores the value 1. An object of the form $[\emptyset, x]$ represents a computation that produced that heap in such a way that the location i can alias with objects defined "outside" of that computation. On the other hand, $[\{i\}, x]$ represents a similar computation that, instead of choosing the value i so that it aliased some preexisting location, chose i in a way that is *fresh* with respect to other identifiers appearing in the context of this computation. This freedom to rename identifiers has pleasant consequences; for instance, given an element $\bar{x} \in \mathsf{Res}(X)$ and a finite set of identifiers I', we can find another finite set I and an element $x \in X$ such that $\bar{x} = [I, x]$ and $I \cap I' = \emptyset$. In other words, we are always free to chose fresh identifiers in a way that does not conflict with *any* other finite set of identifiers that might have a special meaning somewhere else.

Besides $\mathsf{Res}(X)$, there are other nominal restriction sets that will be important for us. If X is a discrete nominal set (that is, one whose elements have empty support), then it carries a canonical restriction operation given by $\nu I.x = x$. Intuitively, if no identifiers occur in x, then there is no need to do anything when hiding identifiers in x. (As a matter of fact, this is the only restriction operation that can be defined on a discrete nominal set, since its properties force $\nu I.x = \nu(I \cap \mathsf{supp}(x)).x = \nu\emptyset.x = x$.) The other restriction operation that we use is on the disjoint union of two nominal restriction sets. If X and Y are nominal restriction sets, then we define a restriction operation on $X \uplus Y$ by setting $\nu I.(k, z) = (k, \nu I.z)$—that is, we just apply the renaming operation that corresponds to whatever side of the sum we are in.

By combining these two constructions, we can define a restriction operation on a nominal set of the form $\mathsf{Res}(X) \uplus \{\mathsf{error}\}$, where the right side of the sum is treated as a discrete nominal set. This restriction structure is used to define allocation for the memory-safe imperative language of Section 4.5.

We now summarize some results that we use to define functions on nominal restriction sets. The first result takes as input a computation f that depends on an element of X and a choice of a fresh identifier i. It then produces another computation $\hat{f}$ that calls f while choosing the fresh identifier that is fed into it.

Lemma A.7. *Let X be a nominal set, Y be a nominal restriction set, and $f \in X \times \mathsf{Id} \rightharpoonup Y$ an equivariant partial function. There exists an equivariant partial function $\hat{f} : X \rightharpoonup Y$ such that $\hat{f}(x) = \nu\{i\}.f(x, i)$ whenever $i \notin \mathsf{supp}(x)$. If $f(x, i)$ for any $i \notin \mathsf{supp}(x)$, then $\hat{f}(x)$ is also undefined.*

Note that, since the fresh identifier i is bound by the brackets, the choice of i performed by $\hat{f}$ is invisible, as i can be renamed to any other identifier that is fresh with respect to x.

The second result allows us to compose computations that generate fresh identifiers.

Lemma A.8. *Let X be a nominal set, Y be a nominal restriction set, and $f : X \rightharpoonup Y$ be an equivariant partial function. There exists an equivariant partial function $\tilde{f} : \mathsf{Res}(X) \rightharpoonup Y$ satisfying the following property. Let I be a finite set of identifiers and $x \in X$. If $f(x) = \bot$, then $\tilde{f}([I, x]) = \bot$. If $f(x)$ is defined, then $\tilde{f}([I, x]) = \nu I.f(x)$.*

We can interpret this result roughly as follows. Suppose that we have a computation f that takes an element of X as input and produces an element of Y. Now, suppose that we have the result $\bar{x} \in \mathsf{Res}(X)$ of a computation that produced an element of X while allocating some fresh identifiers. Then, we can apply f to $\bar{x}$ by materializing the hidden identifiers of $\bar{x}$ to any set of fresh values I, computing f, and in the end of the computation hiding the identifiers in I again.

Finally, we note a result that we use to define the semantics of iteration in the memory-safe language of Section 4.5. Instead of stating it in its most well-known (and general) form, we content ourselves with the following simplified form.

Theorem A.9 (Kleene's fixed-point theorem)**.** *Let X and Y be nominal sets. Let $X \rightharpoonup_{\mathrm{eq}} Y$ be the set of equivariant partial functions from X to Y. Suppose there is a function F from that set to itself that satisfies the following properties:*

Monotonicity *$f \subseteq g \Rightarrow F(f) \subseteq F(g)$; and*

Continuity *if $f_0 \subseteq f_1 \subseteq \cdots$ is an infinite increasing sequence of equivariant partial functions, then $F\left(\bigcup_i f_i\right) = \bigcup_i F(f_i)$.*

In this case, there exists $\mathsf{fix}(F) \in X \rightharpoonup_{\mathrm{eq}} Y$ such that $F(\mathsf{fix}(F)) = \mathsf{fix}(F)$ and $\mathsf{fix}(F) \subseteq f$ whenever $F(f) = f$.

Intuitively, $f \subseteq g$ means that g is defined on all the inputs that f is, and both functions give the same answers for those inputs. Kleene's fixed point allows us to build a solution to the recursive equation $F(f) = f$ that is the smallest possible—that is, defined on as few points as possible.

Bibliography

[1] Martín Abadi et al. “Control-flow integrity”. In: *12th ACM Conference on Computer and Communications Security*. ACM, 2005, pp. 340–353. isbn: 1-59593-226-7.

[2] Amal Ahmed, Matthew Fluet, and Greg Morrisett. “L^3: A Linear Language with Locations”. In: *Fundam. Inform.* 77.4 (2007), pp. 397–449.

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

[4] Andrew W. Appel. “Verified Software Toolchain - (Invited Talk)”. In: *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Ed. by Gilles Barthe. Vol. 6602. Lecture Notes in Computer Science. Springer, 2011, pp. 1–17. isbn: 978-3-642-19717-8.

[5] Aslan Askarov et al. “Termination-Insensitive Noninterference Leaks More Than Just a Bit”. In: *13th European Symposium on Research in Computer Security (ESORICS)*. Vol. 5283. LNCS. Malaga, Spain: Springer-Verlag, Oct. 2008.

[6] Thomas H. Austin and Cormac Flanagan. “Efficient purely-dynamic information flow analysis”. In: *Workshop on Programming Languages and Analysis for Security (PLAS)*. PLAS. ACM, 2009, pp. 113–124.

[7] Arthur Azevedo de Amorim. “Binding Operators for Nominal Sets”. In: *Electr. Notes Theor. Comput. Sci.* 325 (2016), pp. 3–27.

[8] Arthur Azevedo de Amorim, Cătălin Hrițcu, and Benjamin C. Pierce. “The Meaning of Memory Safety”. In: *CoRR* abs/1705.07354 (2017).

[9] Arthur Azevedo de Amorim et al. “A Verified Information-Flow Architecture”. In: *Proceedings of the 41st Symposium on Principles of Programming Languages*. POPL. ACM, Jan. 2014, pp. 165–178. isbn: 978-1-4503-2544-8.

[10] Arthur Azevedo de Amorim et al. “A verified information-flow architecture”. In: *Journal of Computer Security* 24.6 (2016), pp. 689–734.

[11] Arthur Azevedo de Amorim et al. “Micro-Policies: Formally Verified, Tag-Based Security Monitors”. In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 813–830. isbn: 978-1-4673-6949-7.

[12] Michael Backes, Boris Köpf, and Andrey Rybalchenko. “Automatic Discovery and Quantification of Information Leaks”. In: *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*. 2009, pp. 141–153.

[13] Anindya Banerjee and David A. Naumann. "Stack-based access control and secure information flow". In: *Journal of Functional Programming* 15.2 (2005), pp. 131–177.

[14] David A. Basin et al. "Enforceable Security Policies Revisited". In: *ACM Trans. Inf. Syst. Secur.* 16.1 (2013), p. 3.

[15] Arkaprava Basu et al. "Efficient virtual memory for big memory servers". In: *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*. Ed. by Avi Mendelson. ACM, 2013, pp. 237–248. ISBN: 978-1-4503-2079-5.

[16] J. Brown and Thomas F. Knight, Jr. *A Minimally Trusted Computing Base for Dynamically Ensuring Secure Information Flow*. Tech. rep. 5. Aries Memo No. 15. MIT CSAIL, Nov. 2001.

[17] Stefano Calzavara et al. "Micro-policies for Web Session Security". In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 2016, pp. 179–193. ISBN: 978-1-5090-2607-4.

[18] Adam Chlipala. "The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier". In: *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 2013, pp. 391–402.

[19] James A. Clause et al. "Effective memory protection using dynamic tainting". In: *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2007, pp. 284–292. ISBN: 978-1-59593-882-4.

[20] Jedidiah R. Crandall and Frederic T. Chong. "Minos: Control Data Attack Prevention Orthogonal to Memory Model". In: *37th Annual International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2004, pp. 221–232. ISBN: 0-7695-2126-6.

[21] Mads Dam, Roberto Guanciale, and Hamed Nemati. "Machine code verification of a tiny ARM hypervisor". In: *ACM Workshop on Trustworthy Embedded Devices*. Ed. by Ahmad-Reza Sadeghi, Frederik Armknecht, and Jean-Pierre Seifert. ACM, 2013, pp. 3–12. ISBN: 978-1-4503-2486-1.

[22] Mads Dam et al. "Formal verification of information flow security for a simple ARM-based separation kernel". In: *ACM Conference on Computer and Communications Security*. ACM, 2013, pp. 223–234. ISBN: 978-1-4503-2477-9.

[23] Lucas Davi et al. "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection". In: *23rd USENIX Security Symposium*. 2014, pp. 401–416.

[24] André DeHon et al. "DOVER: A Metadata-Extended RISC-V". In: *RISC-V Workshop*. Accompanying talk at `http://youtu.be/r5dIS1kDars`. Jan. 2016.

[25] André DeHon et al. "Preliminary Design of the SAFE Platform". In: *6th Workshop on Programming Languages and Operating Systems*. PLOS. Cascais, Portugal, Oct. 2011.

[26] Benjamin Delaware et al. "Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 689–700. ISBN: 978-1-4503-3300-9.

[27] Christian DeLozier et al. “Ironclad C++: a library-augmented type-safe subset of C++”. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. Ed. by Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes. ACM, 2013, pp. 287–304. ISBN: 978-1-4503-2374-1.

[28] Daniel Y. Deng and G. Edward Suh. “High-performance parallel accelerator for flexible and efficient run-time monitoring”. In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2012, pp. 1–12. ISBN: 978-1-4673-1624-8.

[29] D. E. Denning. “A lattice model of secure information flow”. In: *Communications of the ACM* 19 (5 May 1976), pp. 236–243. ISSN: 0001-0782.

[30] Joe Devietti et al. “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.

[31] Dominique Devriese, Frank Piessens, and Lars Birkedal. “Reasoning about object capabilities with logical relations and effect parametricity”. In: *1st IEEE European Symposium on Security and Privacy*. 2016.

[32] Udit Dhawan and André DeHon. “Area-Efficient Near-Associative Memories on FPGAs”. In: *International Symposium on Field-Programmable Gate Arrays, (FPGA2013)*. Feb. 2013.

[33] Udit Dhawan et al. “Architectural Support for Software-Defined Metadata Processing”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems*. 2015, pp. 487–502.

[34] Udit Dhawan et al. “PUMP – A Programmable Unit for Metadata Processing”. In: *Proceedings of the 3rd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP ’14. Minneapolis, USA: ACM, June 2014.

[35] Petros Efstathopoulos et al. “Labels and event processes in the Asbestos operating system”. In: *Proceedings of the Symposium on Operating Systems Principles*. SOSP. Brighton, United Kingdom: ACM, 2005, pp. 17–30. ISBN: 1-59593-079-5.

[36] Úlfar Erlingsson and Fred B. Schneider. “IRM Enforcement of Java Stack Inspection”. In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2000, pp. 246–255. ISBN: 0-7695-0665-8.

[37] Isaac Evans et al. “Missing the Point(er): On the Effectiveness of Code Pointer Integrity”. In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015, pp. 781–796.

[38] Edward A. Feustel. “On the Advantages of Tagged Architectures”. In: *IEEE Transactions on Computers* 22 (July 1973), pp. 644–652.

[39] Cédric Fournet et al. “Fully abstract compilation to JavaScript”. In: *40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2013, pp. 371–384. ISBN: 978-1-4503-1832-7.

[40] Anthony C. J. Fox and Magnus O. Myreen. "A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture". In: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Vol. 6172. Lecture Notes in Computer Science. Springer, 2010, pp. 243–258. ISBN: 978-3-642-14051-8.

[41] Murdoch J. Gabbay and Andrew M. Pitts. "A New Approach to Abstract Syntax with Variable Binding". English. In: *Formal Aspects of Computing* 13.3-5 (2002), pp. 341–363. ISSN: 0934-5043.

[42] Joseph A. Goguen and José Meseguer. "Security Policies and Security Models". In: *Symposium on Security and Privacy*. 1982, pp. 11–20.

[43] Joseph A. Goguen and José Meseguer. "Unwinding and Inference Control". In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1984, pp. 75–87.

[44] Enes Göktaş et al. "Out Of Control: Overcoming Control-Flow Integrity". In: *IEEE Symposium on Security and Privacy*. 2014.

[45] Georges Gonthier. *A computer-checked proof of the four colour theorem*. 2005.

[46] Georges Gonthier et al. "A Machine-Checked Proof of the Odd Order Theorem". In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 163–179. ISBN: 978-3-642-39633-5.

[47] Dan Grossman et al. "Region-Based Memory Management in Cyclone". In: *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*. Ed. by Jens Knoop and Laurie J. Hendren. ACM, 2002, pp. 282–293. ISBN: 1-58113-463-0.

[48] Ronghui Gu et al. "Deep Specifications and Certified Abstraction Layers". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 595–608. ISBN: 978-1-4503-3300-9.

[49] Gurvan Le Guernic et al. "Automata-Based Confidentiality Monitoring". In: *11th Asian Computing Science Conference*. Springer, 2006, pp. 75–89.

[50] Thomas C. Hales et al. "A formal proof of the Kepler conjecture". In: *CoRR* abs/1501.02155 (2015).

[51] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. "Computability classes for enforcement mechanisms". In: *ACM Transactions on Programming Languages and Systems* 28.1 (2006), pp. 175–205.

[52] Daniel Hedin and Andrei Sabelfeld. *A perspective on information-flow control*. Marktoberdorf Summer School. IOS Press. 2011.

[53] Michael Hicks. *What is memory safety?* 2014. URL: `http://www.pl-enthusiast.net/2014/07/21/memory-safety/` (visited on 02/07/2017).

[54] Cătălin Hriţcu et al. "All Your IFCException Are Belong To Us". In: *34th IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2013, pp. 3–17.

[55] Cătălin Hriţcu et al. “Testing Noninterference, Quickly”. In: *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Sept. 2013.

[56] Cătălin Hriţcu et al. “Testing noninterference, quickly”. In: *J. Funct. Program.* 26 (2016), e4.

[57] *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 2016. ISBN: 978-1-5090-2607-4.

[58] ISO. *ISO C Standard 1999*. Tech. rep. ISO/IEC 9899:1999 draft. ISO, 1999.

[59] J. Jacob. “On the Derivation of Secure Components”. In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1989, pp. 242–247.

[60] Suman Jana and Vitaly Shmatikov. “Memento: Learning Secrets from Process Footprints”. In: *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 2012, pp. 143–157. ISBN: 978-0-7695-4681-0.

[61] Jonas Braband Jensen, Nick Benton, and Andrew Kennedy. “High-level separation logic for low-level code”. In: *40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2013, pp. 301–314. ISBN: 978-1-4503-1832-7.

[62] Jacques-Henri Jourdan et al. “A Formally-Verified C Static Analyzer”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 247–259. ISBN: 978-1-4503-3300-9.

[63] Yannis Juglaret et al. “Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation”. In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 2016, pp. 45–60. ISBN: 978-1-5090-2607-4.

[64] Jeehoon Kang et al. “A formal C memory model supporting integer-pointer casts”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Steve Blackburn. ACM, 2015, pp. 326–335. ISBN: 978-1-4503-3468-6.

[65] Andrew Kennedy et al. “Coq: The World’s Best Macro Assembler?” In: *15th International Symposium on Principles and Practice of Declarative Programming*. ACM, 2013, pp. 13–24. ISBN: 978-1-4503-2154-9.

[66] Narges Khakpour, Oliver Schwarz, and Mads Dam. “Machine Assisted Proof of ARMv7 Instruction Level Isolation Properties”. In: *3rd International Conference on Certified Programs and Proofs*. Vol. 8307. Lecture Notes in Computer Science. Springer, 2013, pp. 276–291. ISBN: 978-3-319-03544-4.

[67] Gerwin Klein et al. “Comprehensive formal verification of an OS microkernel”. In: *ACM Transactions on Computer Systems* 32.1 (2014), p. 2.

[68] G. Klein et al. “seL4: Formal Verification of an OS Kernel”. In: *Proceedings of the Symposium on Operating Systems Principles*. ACM, 2009, pp. 207–220.

[69] Elisavet Kozyri et al. *JRIF: Reactive Information Flow Control for Java*. 2016.

[70] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. “Integrating Linear and Dependent Types”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 17–30. ISBN: 978-1-4503-3300-9.

[71] Ramana Kumar et al. “CakeML: a verified implementation of ML”. In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 179–192. ISBN: 978-1-4503-2544-8.

[72] Volodymyr Kuznetsov et al. “Code-Pointer Integrity”. In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’14, Broomfield, CO, USA, October 6-8, 2014*. Ed. by Jason Flinn and Hank Levy. USENIX Association, 2014, pp. 147–163.

[73] Albert Kwon et al. “Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2013, pp. 721–732. ISBN: 978-1-4503-2477-9.

[74] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Communications of the ACM* 52.7 (2009), pp. 107–115.

[75] Xavier Leroy and Sandrine Blazy. “Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations”. In: *Journal of Automated Reasoning* 41.1 (2008), pp. 1–31.

[76] Kayvan Memarian et al. “Into the depths of C: elaborating the de facto standards”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by Chandra Krintz and Emery Berger. ACM, 2016, pp. 1–15. ISBN: 978-1-4503-4261-2.

[77] MITRE. *Top 25 Common Weakness Enumeration*. `https://cwe.mitre.org/top25/`. 2013.

[78] Benoît Montagu, Benjamin C. Pierce, and Randy Pollack. “A Theory of Information-Flow Labels”. In: *26th IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2013, pp. 3–17. ISBN: 978-0-7695-5031-2.

[79] James H. Morris Jr. “Protection in programming languages”. In: *Communications of the ACM* 16.1 (1973), pp. 15–21. ISSN: 0001-0782.

[80] Greg Morrisett et al. “RockSalt: better, faster, stronger SFI for the x86”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2012, pp. 395–404. ISBN: 978-1-4503-1205-9.

[81] Toby C. Murray et al. “Noninterference for Operating System Kernels”. In: *Second International Conference on Certified Programs and Proofs (CPP)*. Vol. 7679. Lecture Notes in Computer Science. Springer, 2012, pp. 126–142.

[82] Toby C. Murray et al. “seL4: from General Purpose to a Proof of Information Flow Enforcement”. In: *34th IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 415–429.

[83] Santosh Nagarakatte et al. “CETS: compiler enforced temporal safety for C”. In: *9th International Symposium on Memory Management*. ACM, 2010, pp. 31–40. ISBN: 978-1-4503-0054-4.

[84] Santosh Nagarakatte et al. "SoftBound: highly compatible and complete spatial memory safety for C". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009, pp. 245–258. ISBN: 978-1-60558-392-1.

[85] Zhaozhong Ni and Zhong Shao. "Certified assembly programming with embedded code pointers". In: *Symposium on Principles of Programming Languages (POPL)*. ACM, 2006, pp. 320–333.

[86] Oleksii Oleksenko et al. "Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches". In: *CoRR* abs/1702.00719 (2017).

[87] Nicole Perlroth. "Hackers Are Targeting Nuclear Facilities, Homeland Security Dept. and F.B.I. Say". In: *The New York Times* (July 2017).

[88] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. New York, NY, USA: Cambridge University Press, 2013. ISBN: 1107017785, 9781107017788.

[89] Sriram K. Rajamani and David Walker, eds. *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM, 2015. ISBN: 978-1-4503-3300-9.

[90] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. LICS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 55–74. ISBN: 0-7695-1483-9.

[91] Nick Roessler and André DeHon. *Protecting the Stack with Metadata Policies and Tagged Hardware*. Under Preparation. 2017.

[92] John Rushby. *Noninterference, transitivity and channel-control security policies*. Tech. rep. 1992.

[93] A. Sabelfeld and A.C. Myers. "Language-based information-flow security". In: *IEEE Journal on Selected Areas in Communications* 21.1 (Jan. 2003), pp. 5–19. ISSN: 0733-8716.

[94] Andrei Sabelfeld and Alejandro Russo. "From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research". In: *Ershov Memorial Conference*. Springer, 2009, pp. 352–365.

[95] Fred B. Schneider. "Enforceable security policies". In: *ACM Transactions of Information Systems Security* 3.1 (2000), pp. 30–50.

[96] Shayak Sen et al. "Bootstrapping Privacy Compliance in Big Data Systems". In: *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society, 2014, pp. 327–342. ISBN: 978-1-4799-4686-0.

[97] Howard Shrobe, André DeHon, and Thomas F. Knight, Jr. *Trust-Management, Intrusion-Tolerance, Accountability, and Reconstitution Architecture (TIARA)*. Dec. 2009.

[98] Geoffrey Smith. "On the Foundations of Quantitative Information Flow". In: *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Luca de Alfaro. Vol. 5504. Lecture Notes in Computer Science. Springer, 2009, pp. 288–302. ISBN: 978-3-642-00595-4.

[99] Deian Stefan et al. "Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems". In: *Proceedings of the International Conference on Functional Programming (ICFP)*. ACM, 2012.

[100] Deian Stefan et al. “Eliminating Cache-Based Timing Attacks with Instruction-Based Scheduling”. In: *18th European Symposium on Research in Computer Security (ESORICS)*. Vol. 8134. Lecture Notes in Computer Science. Springer, 2013, pp. 718–735. ISBN: 978-3-642-40202-9.

[101] Deian Stefan et al. “Flexible dynamic information flow control in Haskell”. In: *4th Symposium on Haskell*. ACM, 2011, pp. 95–106.

[102] G. Edward Suh et al. “Secure Program Execution via Dynamic Information Flow Tracking”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems*. 2004, pp. 85–96.

[103] Eijiro Sumii and Benjamin C. Pierce. “A bisimulation for dynamic sealing”. In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 169–192.

[104] David Swasey, Deepak Garg, and Derek Dreyer. *Robust and Compositional Verification of Object Capability Patterns*. To appear at OOPSLA. Apr. 2017.

[105] Laszlo Szekeres et al. “SoK: Eternal War in Memory”. In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013, pp. 48–62. ISBN: 978-1-4673-6166-8.

[106] Ankur Taly et al. “Automated Analysis of Security-Critical JavaScript APIs”. In: *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. IEEE Computer Society, 2011, pp. 363–378. ISBN: 978-1-4577-0147-4.

[107] The Coq Development Team. *The Coq Reference Manual, version 8.5*. Available electronically at `http://coq.inria.fr/doc`. 2016.

[108] Christian Urban and Cezary Kaliszyk. “General Bindings and Alpha-Equivalence in Nominal Isabelle”. In: *Logical Methods in Computer Science* 8.2 (2012).

[109] N. Vachharajani et al. “RIFLE: An Architectural Framework for User-Centric Information-Flow Security”. In: *37th International Symposium on Microarchitecture*. 2004.

[110] Guru Venkataramani et al. “FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation”. In: *14th International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2008, pp. 173–184.

[111] Robert Wahbe et al. “Efficient Software-Based Fault Isolation”. In: *Proceedings of the Symposium on Operating Systems Principles*. SOSP. 1993, pp. 203–216.

[112] Wikipedia. *Memory safety — Wikipedia, The Free Encyclopedia*. 2017. (Visited on 08/29/2017).

[113] Jonathan Woodruff et al. “The CHERI capability model: Revisiting RISC in an age of risk”. In: *Proc. of the International Symposium on Computer Architecture (ISCA)*. June 2014, pp. 457–468.

[114] Xuejun Yang et al. “Finding and understanding bugs in C compilers”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. ACM, 2011, pp. 283–294.

[115] Stephan A. Zdancewic. “Programming Languages for Information Security”. PhD thesis. Cornell University, Aug. 2002.

[116] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. “Language-based control and mitigation of timing channels”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’12, Beijing, China - June 11 - 16, 2012*. Ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, 2012, pp. 99–110. ISBN: 978-1-4503-1205-9.

[117] Jianzhou Zhao. “Formalizing an SSA-based Compiler for Verified Advanced Program Transformations”. PhD thesis. University of Pennsylvania, 2013.

[118] Jianzhou Zhao et al. “Formalizing the LLVM intermediate representation for verified program transformations”. In: *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2012, pp. 427–440. isbn: 978-1-4503-1083-3.