2015 IEEE Symposium on Security and Privacy

Micro-Policies

Formally Verified, Tag-Based Security Monitors

Arthur Azevedo de Amorim[1,2] Maxime Dénès[1,2] Nick Giannarakis[2,3,4] Cătălin Hriţcu[2]
Benjamin C. Pierce[1] Antal Spector-Zabusky[1] Andrew Tolmach[5]

[1]University of Pennsylvania [2]Inria Paris-Rocquencourt [3]ENS Cachan [4]NTU Athens [5]Portland State University

***Abstract*—Recent advances in hardware design have demonstrated mechanisms allowing a wide range of low-level security policies (or *micro-policies*) to be expressed using rules on metadata tags. We propose a methodology for *defining* and *reasoning about* such tag-based reference monitors in terms of a high-level "symbolic machine," and we use this methodology to define and formally verify micro-policies for dynamic sealing, compartmentalization, control-flow integrity, and memory safety; in addition, we show how to use the tagging mechanism to protect its own integrity. For each micro-policy, we prove by refinement that the symbolic machine instantiated with the policy's rules embodies a high-level specification characterizing a useful security property. Last, we show how the symbolic machine itself can be implemented in terms of a hardware rule cache and a software controller.**

***Index Terms*—security; dynamic enforcement; reference monitors; low-level code; tagged hardware architecture; metadata; formal verification; refinement; machine-checked proofs; Coq; dynamic sealing; compartmentalization; isolation; least privilege; memory safety; control-flow integrity**

1 Introduction

Today's computer systems are distressingly insecure. However, many of their vulnerabilities can be avoided if low-level code is constrained to obey sensible safety and security properties. Ideally, such properties might be enforced statically, but for obtaining pervasive guarantees all the way to the level of running machine code it is often more practical to detect violations dynamically using a *reference monitor* [3], [13], [29]. Monitors have been used for many tasks, including enforcement of memory safety [27] or control-flow integrity (CFI) [1], taint tracking, fine-grained information-flow control (IFC), and isolation of untrusted code [33], [35]. They are sometimes implemented in software [13], but this can significantly degrade performance and/or cause designers to settle for rough approximations of the intended policy that are potentially vulnerable to attack [10], [14]. Hardware acceleration is thus an attractive alternative, especially in an era of cheap transistors. Many designs for hardware monitors have been proposed, with early designs focusing on enforcing single, hard-wired security policies [30] and later ones evolving toward more programmable mechanisms that allow quicker adaptation to a shifting attack landscape. Recent work has gone yet further in this direction by defining a generic, fully programmable hardware/software architecture for tag-based monitoring on a conventional processor extended with a *Programmable Unit for Metadata Processing (PUMP)* [11].

The PUMP architecture associates each piece of data in the system with a *metadata tag* describing its provenance or purpose (e.g., "this is an instruction," "this came from the network," "this is secret," "this is sealed with key k"), propagates this metadata as instructions are executed, and checks that policy rules are obeyed throughout the computation. It provides great flexibility for defining policies and puts no arbitrary limitations on the size of the metadata and the number of policies supported. Hardware simulations show [11] that an Alpha processor extended with PUMP hardware achieves performance comparable to dedicated hardware when simultaneously enforcing memory safety, CFI, and taint tracking on a standard benchmark suite. Monitoring imposes modest impact on runtime (typically under 10%) and power ceiling (less than 10%), in return for some increase in energy usage (typically under 60%) and chip area (110%).

Coding correct, efficient policies to run on the PUMP architecture can be nontrivial. Indeed, it is often challenging even to give a high-level *specification* for a policy of interest. In prior work, we showed how to address this challenge for one specific policy by giving a mechanized correctness proof for an information-flow control (IFC) policy running on an idealized machine incorporating PUMP-like hardware [4]. This proof is organized around three layers of machines sharing a common core instruction set: an *abstract* machine whose instruction semantics has a specific IFC policy built in; an intermediate *symbolic* machine that allows for different dynamic IFC mechanisms to be expressed using a simple domain-specific language; and a *concrete* machine, where the IFC policy is implemented by a software controller that interacts with low-level tag-management mechanisms of the hardware. A noninterference property is established at the abstract machine level and transferred to the other levels via two steps of *refinement*.

In this paper, we extend this IFC-specific proof to a *generic framework* for formalizing and verifying *arbitrary* policies enforceable by the PUMP architecture. We use the term *micro-policies* for such instruction-level security-monitoring mechanisms based on fine-grained metadata. We use this methodology to formalize and verify a diverse collection of micro-policies using the Coq proof assistant.

The heart of our methodology is a *generic symbolic machine* (middle layer in Figure 1) that serves both as a programming

© 2015, Arthur Azevedo de Amorim. Under license to IEEE.
DOI 10.1109/SP.2015.55

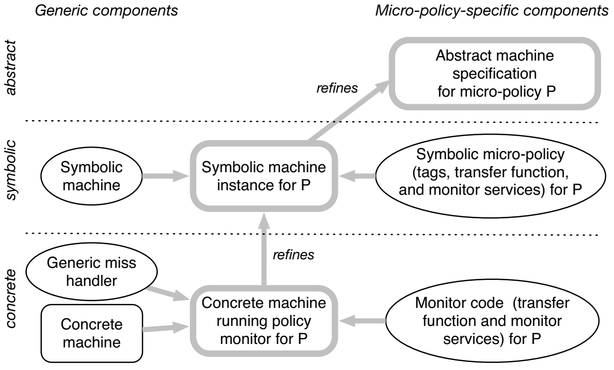
Generic components
Micro-policy-specific components
abstract
symbolic
concrete
Abstract machine specification for micro-policy P
refines
Symbolic machine
Symbolic machine instance for P
Symbolic micro-policy (tags, transfer function, and monitor services) for P
refines
Generic miss handler
Concrete machine
Concrete machine running policy monitor for P
Monitor code (transfer function and monitor services) for P

Figure 1. System overview

interface—abstracting away unnecessary implementation details and providing a convenient platform for micro-policy designers—and as an intermediate step in correctness proofs. This machine is parameterized by a *symbolic micro-policy* that expresses tag propagation and checking in terms of structured mathematical objects rather than low-level concrete representations. Each symbolic micro-policy consists of (i) sets of *metadata tags* that are used to label every piece of data in the machine's memory and registers (including the program counter); (ii) a *transfer function* that uses both the current opcode and the tags on the pc, on the current instruction, and on the instruction operands to determine whether the operation is permitted and, if it is, to specify how the pc and the instruction's result should be tagged in the next machine state; and (iii) a set of *monitor services* that can be invoked by user code. For example, in a micro-policy for *dynamic sealing* (a language-based protection mechanism in the style of perfect symmetric encryption [23], described below in §4) the set of tags used for registers and memory might be $\{\mathsf{Data}, \mathsf{Key}\ k, \mathsf{Sealed}\ k\}$, where Data is used to tag ordinary data values, $\mathsf{Sealed}\ k$ is used to tag values sealed with the key k, and $\mathsf{Key}\ k$ denotes a key that can be used for sealing and unsealing values. The transfer function for this micro-policy would allow, for example, arithmetic operations on values tagged Data but deny them on data tagged Sealed or Key. Monitor services are provided to allow user programs to create new keys and to seal and unseal data values with given keys.

We instantiate this symbolic machine with a diverse set of security micro-policies: (a) dynamic sealing [23], [31]; (b) compartmentalization, which sandboxes untrusted code and allows it to be run alongside trusted code [32]; (c) control-flow integrity (CFI), which prevents code-reuse attacks such as return-oriented programming [1]; and (d) memory safety, which prevents temporal and spatial violations for heap-allocated data [11]. The intended behavior of each micro-policy is specified by an *abstract machine* (top layer in Figure 1), which gives a clear characterization of the micro-policy's behavior as seen by a user-level programmer. The abstract machine enforces the invariants of the micro-policy by omitting insecure behaviors from its transition function: a program that violates the micro-policy gets stuck. Where appropriate, we prove that the abstract machine for a micro-policy satisfies standard properties from the literature. For example, for the CFI micro-policy we prove a variant of the original CFI property proposed by Abadi *et al.* [1], while for our compartmentalization micro-policy we prove a single-step property drawn from Wahbe *et al.*'s original software fault isolation (SFI) model [32]. For each micro-policy, we prove *backward refinement* between the abstract and symbolic machines, i.e., every possible symbolic machine behavior is a valid abstract behavior—hence, the symbolic machine always fail-stops on policy violations.

Finally, we extend this methodology to the hardware level by showing how instances of the symbolic machine can be realized on a low-level *concrete machine*, a minimalist RISC ISA extended with the key mechanisms of the PUMP hardware architecture [11] (bottom layer in Figure 1). Every word of data in this machine is associated with a piece of metadata called a *tag*—itself a full machine word that can, in particular, hold a pointer to an arbitrary data structure in memory. The interpretation of tags is left entirely to software; the hardware simply propagates tags from operands to results according to software-defined *concrete rules*. To propagate tags efficiently, the processor is augmented with a *rule cache* that operates in parallel with instruction execution. On a rule cache miss, control is transferred to a trusted *miss handler* which, given the tags of the instruction's arguments, decides whether the current operation should be allowed and, if so, computes appropriate tags for its results. It then adds this set of argument and result tags to the rule cache so that when the same situation is encountered in the future, the rule can be applied without slowing down the processor.

Each micro-policy can be implemented at the concrete level by providing machine code for the transfer function and monitor services, along with a concrete bit-encoding for symbolic tags. This *monitor code* can make use of a handful of privileged instructions of the concrete machine, allowing it to inspect and change tags and to update the cache. For all micro-policies, it is obviously necessary to protect the integrity of the monitor's code and data, and to prevent user programs from invoking the privileged instructions. We show that we can achieve this protection using only the tagging mechanism itself (no special kernel protection modes, page table tricks, etc.). We also give a generic proof of backward refinement between the symbolic and concrete machines, modulo some assumptions characterizing the behavior of the micro-policy-specific concrete code. Composing this refinement with the abstract-symbolic refinement described above gives a proof that the concrete machine always fail-stops on policy violations. For CFI, we additionally show that the corresponding higher-level property [1] is preserved by refinement, allowing us to transfer it to any valid implementation of the micro-policy.

Our focus throughout is on proving *safety* properties, which we formalize as backward refinements: the observable behaviors of the lower-level machine are also legal behaviors of the higher-level machine, and in particular the lower-level machine fail-stops whenever the higher-level machine does. *Liveness*, or forward refinement (the lower-level machine only fail-stops when the higher-level one does), is also a desirable property; indeed, a completely inert machine (i.e., one that never steps) at the symbolic or concrete level would satisfy backward refinement but would be of no use. However, full forward refinement doesn't always hold for our micro-policies. In particular, resource constraints that we prefer to ignore at the abstract level (e.g., word size and memory capacity) become visible at the symbolic or concrete level when tags and monitor data structures are made explicit. Fortunately, in practice it is reasonable to check that the lower-level machines are "live enough" by testing.

Our main contributions are as follows. First, we introduce a generic symbolic machine (§2-§3) for conveniently defining and effectively verifying a wide range of micro-policies for a simple RISC processor. Second, we use the symbolic machine to give formal descriptions and verified tag-based implementations of four security micro-policies: dynamic sealing (§4), compartmentalization (§5), control-flow integrity (§6), and memory safety (§7). Third, we define a concrete machine incorporating a PUMP cache (§8) and sketch how to construct concrete monitors implementing symbolic micro-policies. And finally (§9), we give a generic construction showing how tags can be used to protect the concrete monitor itself from attack, together with a generic proof (parameterized by some assumptions about the micro-policy-specific monitor code) that this construction is correct. We discuss related work specific to each micro-policy in the relevant section (§4–§7), saving more general related work on micro-policies and reference monitors for §10. We outline future work in §11. The appendices present additional technical details; further details can be in a long version, available electronically.

2 Basic Machine

We begin by introducing the simplified RISC instruction set architecture that forms the common core of all the machines throughout the paper. This basic machine has a fixed word size and a fixed set of general-purpose registers plus a program counter (pc) register. It features a small collection of familiar instructions

$$\begin{array}{l} inst ::= \mathsf{Nop} \mid \mathsf{Const}\ i\ r_d \mid \mathsf{Mov}\ r_s\ r_d \mid \mathsf{Binop}_\oplus\ r_1\ r_2\ r_d \\ \qquad \mathsf{Load}\ r_p\ r_d \mid \mathsf{Store}\ r_p\ r_s \mid \mathsf{Jump}\ r \mid \mathsf{Jal}\ r \mid \mathsf{Bnz}\ r\ i \mid \mathsf{Halt} \end{array}$$

where $\oplus \in \{+, -, \times, =, \leq, \ldots\}$. Const i r_d puts a constant i into register r_d. Mov r_s r_d copies the contents of r_s into r_d. Jump and Jal (jump-and-link) are unconditional indirect jumps, while Bnz r i branches to a fixed offset i (relative to the current pc) if register r is nonzero. Each instruction is encoded in a word.

A *basic machine state* is a tuple (mem, reg, pc) of a word-addressable memory *mem* (a partial function from words to words), a register file *reg* (a function from register names to words), and a *pc* value (a word). Note that the memory is a *partial* function; trying to address outside of the valid memory (by trying to fetch the next instruction from an invalid address, or loading from or storing to one) halts the machine. A typical step rule for this machine is written like this:

$$\frac{\begin{array}{c} mem[pc] = i \quad decode\ i = \mathsf{Binop}_\oplus\ r_1\ r_2\ r_d \\ reg[r_1]{=}w_1 \quad reg[r_2]{=}w_2 \quad reg'{=}reg[r_d{\leftarrow}w_1 \oplus w_2] \end{array}}{(mem, reg, pc) \to (mem, reg', pc{+}1)} \quad (\textsc{Binop})$$

Let's read this rule in detail. Looking up the memory word at address *pc* yields the word i, which should correspond to some instruction (i.e., an element of the *inst* set defined above) via partial function *decode*. In this case, that instruction is $\mathsf{Binop}_\oplus\ r_1\ r_2\ r_d$. Registers r_1 and r_2 contain the operands w_1 and w_2. The notation $reg[r_d{\leftarrow}w_1 \oplus w_2]$ denotes a partial function that maps r_d to $w_1 \oplus w_2$ and behaves like *reg* on all other arguments. The next machine state is calculated by updating the register file, incrementing the pc, and leaving the memory unchanged.

The step rule for the Store instruction is similar. (The notation $mem[w_p{\leftarrow}w_s]$ is defined only when $mem[w_p]$ is defined; i.e., it fails if w_p is not a legal address in *mem*. This ensures that the set of addressable memory locations remains fixed as the machine steps.)

$$\frac{\begin{array}{c} mem[pc] = i \quad decode\ i = \mathsf{Store}\ r_p\ r_s \\ reg[r_p]{=}w_p \quad reg[r_s]{=}w_s \quad mem'{=}mem[w_p{\leftarrow}w_s] \end{array}}{(mem, reg, pc) \to (mem', reg, pc{+}1)} \quad (\textsc{Store})$$

Subroutine calls are implemented by the Jal instruction, which saves the return address to a general-purpose register $\mathsf{r_a}$. Returns from subroutines are just Jumps through the $\mathsf{r_a}$ register.

$$\frac{\begin{array}{c} mem[pc] = i \quad decode\ i = \mathsf{Jal}\ r \\ reg[r] = pc' \quad reg' = reg[\mathsf{r_a}{\leftarrow}pc{+}1] \end{array}}{(mem, reg, pc) \to (mem, reg', pc')} \quad (\textsc{Jal})$$

3 Symbolic Machine

The symbolic machine is the keystone of our methodology, embodying our micro-policy programming model. It allows micro-policies to be expressed and reasoned about in terms of high-level programs written in Gallina, Coq's internal functional programming language, abstracting away irrelevant details about how they are implemented on concrete low-level hardware and providing an appropriate level of abstraction for reasoning about their security properties. In this section, we give just the bare definition of the symbolic machine; §4 illustrates how its features are used.

The symbolic machine shares the same general organization as the basic machine from §2. Its definition is abstracted on several parameters that are provided by the micro-policy designer, collectively forming a *symbolic micro-policy*: (1) A collection of *symbolic tags*, which are used to label instructions and data. (2) A partial function *transfer*, which is invoked on each step of the machine to propagate tags from the current machine state to the next one. (3) A partial function *get_service* mapping addresses to pairs of a *symbolic monitor service* (a partial function on machine states) and a symbolic tag that can be used to restrict access to that service. (4) A type *EX* of *extra machine state* for use by the monitor services, plus an initial value for this extra state.

Symbolic states $(mem, reg, pc, extra)$ consist of a memory, a register file, a program counter, and a piece of extra state.

The memory, registers, and pc hold *symbolic atoms* written $w@t$, where w (the "payload") is a machine word and t is a symbolic tag. The tag parts are not separately addressable; they are only accessible to the transfer function and monitor services, not to user programs.[1]

The symbolic step rules call the transfer function to decide whether the step is allowed by the micro-policy and, if so, how the tags should be updated in the next machine state. The transfer function is passed a 6-tuple containing the current opcode plus the tags from the current pc, current instruction, and the inputs to the current instruction (up to three, depending on the opcode). It returns a pair containing a tag for the next pc and a tag for the instruction's result, if any. For example, here is the symbolic rule for the Binop instruction:

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

As in the basic machine, looking up the memory word at address w_{pc} (the payload part of the current pc value) yields the atom $i@t_i$; decoding its payload part yields the instruction $\mathsf{Binop}_{\oplus}\ r_1\ r_2\ r_d$. Registers r_1 and r_2 contain the operands w_1 and w_2, with tags t_1 and t_2, and the current tag on the result register r_d is t_d. The payload part of the current contents of r_d doesn't matter, since it's about to be overwritten; we indicate this with the wildcard pattern _. Passing all these tags to the transfer function yields tags t'_{pc} and t'_d for the next pc and the new contents of r_d. Since *transfer* is a partial function, it may not return anything at all for a given 6-tuple of opcode and tags. If it doesn't—i.e., if the next step would cause a policy violation—then none of the step rules will apply and the current machine state will be stuck. (For simplicity, we assume here that policy violations are fatal; various error-recovery mechanisms could also be used [16], [21].) Passing t_d, the tag on the old contents of the target register, to the transfer function allows it to see what kind of data is being overwritten. This information is useful for implementing dynamic information-flow policies like "no sensitive upgrade" [4].

To illustrate how a transfer function might behave, consider how the symbolic machine might be used to implement a very simple *taint-propagation micro-policy*. Symbolic tags are drawn from the set $\{\top, \bot\}$, representing tainted and untainted values. The transfer function is written to ensure that, on each step of the machine, any result that is influenced by tainted values is itself tainted. E.g., it might include this clause for the Binop opcode

$$transfer(\mathsf{Binop}_{\oplus}, t_{pc}, -, t_1, t_2, -) = (t_{pc}, t_1 \vee t_2)$$

where $t_1 \vee t_2$ denotes the max of t_1 and t_2, where $\bot < \top$. For this policy, the t_i and t_d tags don't matter, which we indicate by writing a dummy value "$-$".

The generic symbolic rule for Store is similar:

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

The symbolic machine's step relation includes a similarly augmented version of each of the step rules of the basic machine (see §B for a complete listing). In addition, there is one new step rule for handling calls to monitor services, which applies when the pc is at a service entry point—i.e., an address for which the *get_service* function is defined.

$$\frac{\begin{array}{c} get\_service\ w_{pc} = (f, t_i) \\ transfer(\mathsf{Service}, t_{pc}, t_i, -, -, -) = (-, -) \\ f(mem, reg, w_{pc}@t_{pc}, extra) = (mem', reg', pc', extra') \end{array}}{(mem, reg, w_{pc}@t_{pc}, extra) \to (mem', reg', pc', extra')} \quad (\textsc{Svc})$$

Here, *get_service* returns the monitor service itself (the function f), plus a tag t_i. The call to *transfer* checks that this service is permitted, given the tag on the current pc. The last three inputs to *transfer* are set to the dummy value "$-$", and the outputs are not used, since we only care whether the operation is allowed or not. The Service "opcode" is a special value that is just used for querying the transfer function about service routines. Finally, the rule invokes f to carry out the actual work of the service routine. The behavior of f itself is now completely up to the micro-policy designer: it is given the complete symbolic machine state as argument, and it returns an entire new machine state as result. In particular, some of the service routines for the policies described below will modify the tags in the memory. Also, user code will typically get to the service routine entry point by executing a Jal, and the service routine is responsible for resetting the pc to the return address stored in register $\mathsf{r_a}$ by the Jal. Allowing service routines to be arbitrary partial functions from machine states to machine states reflects the fact that, at the concrete level, service routine code runs with a high level of privilege.

To streamline proofs about the micro-policies in later sections, we divide the symbolic tags into four distinct sets that are used in different parts of the symbolic machine: tags from the set T_m are used for labeling words in memory, T_r for registers, T_{pc} for the pc, and T_s for monitor services. The definition of the transfer function must conform to these conventions; for example, when propagating tags for Binop, the three last arguments t_1, t_2, and t_d should belong to T_r and the result tag t'_d should also be in T_r. This separation allows some policy invariants to be maintained "by typechecking," obviating the need to maintain them explicitly in proofs and easing the burden of formal policy verification.

As we will see in the following sections, each micro-policy has complete freedom to treat tags as if they are associated with the *contents* of memory locations or registers (pc included) or as if they are associated with the memory locations or registers themselves. Both points of view are valid and useful: micro-policies like dynamic sealing and taint tracking associate metadata only with contents, while CFI uses tags to distinguish the memory locations containing instructions and the sources and targets of indirect jumps, while using the pc tag to track execution history (the sources of indirect jumps). In fact, some micro-policies mix the two points of view: IFC associates tags with memory and register contents, but the pc tracks execution history (implicit flows), while memory safety tags memory locations with compound tags that contain a part associated with the contents and a part associated with the location.

---

[1] We use the term "user code" for all code in the system that is not part of the micro-policy monitor, including OS-level code such as schedulers and device drivers.

4 Sealing Micro-Policy

Now it's time to build micro-policies! As a warm-up, we begin with a simple micro-policy for dynamic sealing [23], a linguistic mechanism analogous to perfect symmetric encryption. Informally, we extend the basic machine with three new primitives (presented as monitor services): *mkkey* creates a fresh sealing key; *seal* takes a data value (a machine word) and a key and returns an opaque "sealed value" that can be stored in memory and registers but not used in any other way until it is passed (together with the same key that was used to seal it) through the *unseal* service, which unwraps it and returns the original raw word.

We proceed in three steps. First, we define an *abstract sealing machine*, a straightforward extension of the basic machine from §2 that directly captures the "user's view." Second, we show how the abstract machine can be emulated on the *symbolic* machine by providing an appropriate encoding of abstract-machine values (words, sealed values, and keys) as symbolic atoms, together with a transfer function (written as a program in Gallina) and Gallina implementations of the three monitor services. We prove that the symbolic sealing machine refines the abstract one. Third, we build *concrete* machine-code realizations of the symbolic transfer function and the three monitor services, which can be executed (together with user code) on a concrete processor with PUMP hardware extensions. We carry out the first two parts in this section and sketch the third in §9.

Abstract Sealing Machine To define an abstract machine with built-in sealing, we replace the raw words in the registers and memory of the basic machine with *values* v drawn from the more structured set $\mathit{Val} ::= w \mid k \mid \{w\}_k$, where w ranges over machine words, k ranges over an infinite set *AK* of *abstract sealing keys*, and $\{w\}_k$ is the sealing of payload w under key k. To keep the example simple, we disallow nested sealing and sealing of keys: only words can be sealed. We enrich basic machine states with a set *ks* of previously allocated keys, and we assume there is some total function *mkkey_f* that, given a set of keys *ks*, chooses a fresh key not in the set.

The rules of the basic step relation are modified to use this richer set of values. Most instructions will only work with raw words—e.g., attempting to compare sealed values or jump to a key will halt the machine. Load and Store require a word as their first argument (the target memory address) but they place no restrictions on the value being loaded or stored; similarly Mov copies arbitrary values between registers.

$$\frac{\begin{array}{c}\mathit{mem}[\mathit{pc}] = i \qquad \mathit{decode}\ i = \mathsf{Store}\ r_p\ r_s \\ \mathit{reg}[r_p]{=}w_p \qquad \mathit{reg}[r_s]{=}v_s \qquad \mathit{mem}'{=}\mathit{mem}[w_p{\leftarrow}v_s]\end{array}}{(\mathit{mem}, \mathit{reg}, \mathit{pc}) \to (\mathit{mem}', \mathit{reg}, \mathit{pc}{+}1)}\ (\textsc{Store})$$

The operations of generating keys, sealing, and unsealing are provided by monitor service routines located at addresses mkkey_addr, seal_addr, and unseal_addr, all of which lie outside of accessible memory at the symbolic and abstract levels (at the concrete level, the code for the services will begin at these addresses). By convention, these routines take any arguments in general-purpose registers $\mathsf{r}_{\mathsf{arg1}}$ and $\mathsf{r}_{\mathsf{arg2}}$ and return their result in a general-purpose register $\mathsf{r}_{\mathsf{ret}}$. The definition of the step relation includes a rule for each service that applies when the pc is at the corresponding address. For example:

$$\frac{\mathit{mkkey\_f}\ \mathit{ks}{=}k \qquad \mathit{reg}'{=}\mathit{reg}[\mathsf{r}_{\mathsf{ret}}{\leftarrow}k] \qquad \mathit{reg}[\mathsf{r}_{\mathsf{a}}]{=}\mathit{pc}'}{(\mathit{mem}, \mathit{reg}, \mathsf{mkkey\_addr}, \mathit{ks}) \to (\mathit{mem}, \mathit{reg}', \mathit{pc}', k{::}\mathit{ks})}\ (\textsc{MkKey})$$

This rule applies when the machine's pc is mkkey_addr. The first premise uses *mkkey_f* to generate a fresh key k. The second premise updates the result register $\mathsf{r}_{\mathsf{ret}}$ with k. The third premise restores the pc from the register r_{a}. To invoke this service, a user program performs a Jal to the address mkkey_addr, which sets r_{a} appropriately and causes this rule to fire on the next step. Invoking services this way means that we can run exactly the same user code on this abstract machine as we do on the symbolic machine (described below) and the concrete machine (§8-§9). The rule for the unsealing service is similar (as is the one for the sealing service, which we omit):

$$\frac{\begin{array}{c}\mathit{reg}[\mathsf{r}_{\mathsf{arg1}}] = \{w\}_k \qquad \mathit{reg}[\mathsf{r}_{\mathsf{arg2}}] = k \\ \mathit{reg}' = \mathit{reg}[\mathsf{r}_{\mathsf{ret}}{\leftarrow}w] \qquad \mathit{reg}[\mathsf{r}_{\mathsf{a}}] = \mathit{pc}'\end{array}}{(\mathit{mem}, \mathit{reg}, \mathsf{unseal\_addr}, \mathit{ks}) \to (\mathit{mem}, \mathit{reg}', \mathit{pc}', \mathit{ks})}\ (\textsc{Unseal})$$

The first two premises extract the sealed value $\{w\}_k$ from the first argument register and check that second argument register contains the same key k. (If the first register doesn't contain a sealed value or the key doesn't match the second register, the rule fails to fire and the machine gets stuck.) The third premise writes the raw value w into the result register, and the last premise extracts the return address from r_a.

Symbolic Sealing Machine The abstract machine described above constitutes a specification—an application programmer's view—of the sealing micro-policy. The next piece of the micro-policy definition is a *symbolic micro-policy* that implements this abstract specification in terms of tags. Since the pc is just a bare word in the sealing abstract machine, and there are no restrictions on when monitor services can be called, we can take the pc and service tag sets T_{pc} and T_s to be just the singleton set $\{\bullet\}$. T_r and T_m, on the other hand, will be used to represent the values of the abstract machine: their elements have the one of the forms Data, Key k, or Sealed k, where k is a *symbolic key* drawn from an ordered finite set *SK*. Raw words are tagged Data. Keys are represented as a dummy payload word (say, 0) tagged Key k for some k. A word w tagged Sealed k represents the sealing of w under key k. The extra state type *EX* is just *SK*—i.e., the extra state is a single monotonic counter storing the next key to be generated. The initial extra state is the minimum key.

Outside of monitor services, all the propagation and checking of tags is performed by the transfer function of the symbolic machine. In our formal development, transfer functions are written in Gallina, but for readability here we will present examples as collections of *symbolic rules* of the form

$$opcode : (PC, CI, OP_1, OP_2, OP_3) \to (PC', R')$$

where the metavariables PC, CI, etc. range over symbolic expressions, including variables plus a dummy value "$-$" to indicate input or output fields that are ignored. For example, the fact that Store requires an unsealed word in its pointer register (OP_1) and copies the tag of the source register (OP_2) to the result is captured by the following symbolic rule:

$$\mathsf{Store} : (\bullet, \mathsf{Data}, \mathsf{Data}, t_{src}, -) \to (\bullet, t_{src})$$

Similarly, the Jal rule ensures that the target register (OP_1) is tagged Data:

$$\mathsf{Jal} : (\bullet, \mathsf{Data}, \mathsf{Data}, -, -) \to (\bullet, \mathsf{Data})$$

(The symbolic machine step rule for Jal is in §B.)

As we described in §3, the symbolic machine handles all monitor services with a single rule that uses a function *get_service* (provided as part of the micro-policy definition) to do the actual work; given a memory address, *get_service* returns either nothing or a pair of a Gallina function defining the service's behavior and a symbolic tag that is passed to the transfer function so that it can check whether the call to this service is legal from this machine state. For the sealing micro-policy, we define *get_service* to map mkkey_addr to (*mkkey*, $\bullet$), seal_addr to (*seal*, $\bullet$), and unseal_addr to (*unseal*, $\bullet$), where *mkkey* and *unseal* (*seal* is similar) are defined by:

$$\frac{\mathit{reg}[\mathsf{r_a}] = w_{pc'}@\_ \qquad \mathit{reg}' = \mathit{reg}[\mathsf{r_{ret}} \leftarrow 0@(\mathsf{Key}\ \mathit{nk})] \\ \mathit{nk} \neq \mathsf{max\_key} \qquad \mathit{nk}' = \mathit{nk} + 1}{\mathit{mkkey}\ (\mathit{mem}, \mathit{reg}, \mathit{pc}, \mathit{nk}) \mapsto (\mathit{mem}, \mathit{reg}', w_{pc'}@\bullet, \mathit{nk}')}$$

$$\frac{\mathit{reg}[\mathsf{r_{arg1}}] = w@(\mathsf{Sealed}\ k) \qquad \mathit{reg}[\mathsf{r_{arg2}}] = w'@(\mathsf{Key}\ k) \\ \mathit{reg}[\mathsf{r_a}] = w_{pc'}@\_ \qquad \mathit{reg}' = \mathit{reg}[r_{ret} \leftarrow w@\mathsf{Data}]}{\mathit{unseal}\ (\mathit{mem}, \mathit{reg}, \mathit{pc}, \mathit{nk}) \mapsto (\mathit{mem}, \mathit{reg}', w_{pc'}@\bullet, \mathit{nk})}$$

The constant max_key stands for the largest representable key, and 0 is used as a dummy payload for fresh keys.

Note that *mkkey* is a partial function: it can fail if all keys have been used up. This models the fact that, on the concrete machine, keys will be implemented as fixed-width machine words. By contrast, the abstract sealing machine uses an infinite set of keys, so it will never fail for this reason. This discrepancy is not an issue for the backward refinement property, which only requires us to show that *if* the symbolic machine takes a step then a corresponding step can be taken by the abstract machine. (Forward refinement, on the other hand, does not hold: the symbolic machine will fail to simulate the abstract one when it runs out of fresh keys. Giving up forward refinement is the price we pay for choosing not to expose low-level resource constraints at the abstract level.)

Refinement We formalize the connection between the abstract and symbolic sealing machines as a backward (i.e., from symbolic to abstract) refinement property on traces. We state the property here in a general form so that we can instantiate it repeatedly throughout the paper.

Definition 4.1 (Backward refinement). We say that a low-level machine $(\mathit{State}^L, \to^L)$ *backward refines* a high-level machine $(\mathit{State}^H, \to^H)$ with respect to a *simulation relation* $\sim$ between low- and high-level states if, whenever $s_1^L \sim s_1^H$ and $s_1^L \to^* s_2^L$, there is some s_2^H such that $s_1^H \to^* s_2^H$ and $s_2^L \sim s_2^H$.

Following standard practice, we prove this general multi-step refinement property by establishing a correspondence between individual execution steps. In the case of sealing, we prove a strong 1-*backward simulation* theorem showing that each step of the symbolic machine is simulated by *exactly one* step of the abstract one.

Definition 4.2 (1-backward simulation). If $s_1^L \sim s_1^H$ and $s_1^L \to s_2^L$ then there exists s_2^H such that $s_1^H \to s_2^H$ and $s_2^L \sim s_2^H$.

For sealing, since keys are dynamically allocated, our simulation relation is parameterized by a partial map ψ relating abstract and symbolic keys. We begin by defining an auxiliary relation $\sim_\psi^{SA}$ showing how abstract atoms relate to symbolic ones (*SA* stands for Symbolic-to-Abstract):

$$\begin{aligned} w@\mathsf{Data} &\sim_\psi^{SA} w' && \text{when } w = w' \\ w@(\mathsf{Key}\ k^S) &\sim_\psi^{SA} k^A && \text{when } \psi[k^A] = k^S \\ w@(\mathsf{Sealed}\ k^S) &\sim_\psi^{SA} \{w'\}_{k^A} && \text{when } w = w' \wedge \psi[k^A] = k^S. \end{aligned}$$

The relation $\sim_\psi^{SA}$ does not hold otherwise. Then, we define the simulation relation on states, also noted $\sim_\psi^{SA}$, by lifting the previous relation "pointwise" to all atoms, and adding these invariants: (a) all abstract keys in the domain of ψ are in the set of currently allocated keys in the abstract state; (b) all symbolic keys in the range of ψ are strictly smaller than the current value of the monotonic counter; and (c) ψ is injective. We then get the following result:

Theorem 4.3 (1-backward SA-simulation for sealing). The symbolic machine instantiated with the sealing micro-policy 1-backward-simulates the sealing abstract machine with respect to the simulation relation $\lambda s^S\, s^A.\ \exists \psi.\ s^S \sim_\psi^{SA} s^A$.

Notice that, in the above statement, the key map parameter ψ is existentially quantified and not fixed, since it must be updated on each call to *mkkey* to maintain the correspondence between the newly generated keys, which are drawn from different sets at the two levels. This setup allows us to elide irrelevant details of key allocation from the abstract machine. This is only a minor convenience for sealing, but the idiom becomes quite important in other micro-policies for hiding complex objects like memory allocators (§7) from the high-level specification.

5 Compartmentalization Micro-Policy

We next describe a micro-policy for enforcing isolation between program-defined "compartments," dynamically demarcated memory regions that, by default, cannot jump or write to each other. This model is based on Wahbe *et al.*'s work on software fault isolation (SFI) [32], with a few differences discussed below. To demonstrate isolation, we show that a symbolic-machine instance refines an abstract machine that enforces compartmentalization by construction.

Abstract Machine The abstract machine for this micro-policy enforces compartmentalization directly by maintaining, alongside the usual machine state, a set C of current compartments that is consulted on each step to prevent one compartment from improperly transferring execution to or writing to another. Each *abstract compartment* in C is a triple (A, J, S) containing (1) an *address space* A of addresses that belong to the compartment, i.e., where its instructions and data are stored; (2) a set of *jump targets* J, additional addresses that it is allowed to jump to; and (3) a set of *store targets* S, additional addresses that it is allowed to write to. Compartments are not limited to contiguous regions of memory. Also, as in Wahbe *et al.*'s model [32], reading from memory is not constrained: code in one compartment is permitted to read from addresses in any other. (Adding a set of "read targets" to each abstract compartment would be a straightforward extension.) The machine maintains a number of invariants, of which the most important is that all compartments have disjoint address spaces.

The abstract machine state includes a flag $F \in \{\mathsf{Jumped}, \mathsf{FallThrough}\}$ that records whether or not the previous instruction was a Jump or a Jal, together with the previously executing compartment, $\mathit{prev} = (A_{\text{prev}}, J_{\text{prev}}, S_{\text{prev}})$. This information is used to prevent illegal pc changes (on both jumps and ordinary steps) and to allow monitor services to see which compartment called them.

At the abstract level, all instructions behave as in the basic machine (§2), with the addition of a compartmentalization check. For example, here is the rule for Store:

$$\frac{\begin{array}{c} \mathit{mem}[\mathit{pc}] = i \qquad \mathit{decode}\ i = \mathsf{Store}\ r_p\ r_s \\ \mathit{reg}[r_p] = w_p \qquad \mathit{reg}[r_s] = w_s \qquad \mathit{mem}' = \mathit{mem}[w_p \leftarrow w_s] \\ (A, J, S) \in C \qquad \mathit{pc} \in A \qquad w_p \in A \cup S \\ (A, J, S) = (A_{\text{prev}}, J_{\text{prev}}, S_{\text{prev}}) \vee (F = \mathsf{Jumped} \wedge \mathit{pc} \in J_{\text{prev}}) \end{array}}{\begin{array}{c} (\mathit{mem}, \mathit{reg}, \mathit{pc}, C, F, (A_{\text{prev}}, J_{\text{prev}}, S_{\text{prev}})) \\ \to (\mathit{mem}', \mathit{reg}, \mathit{pc}+1, C, \mathsf{FallThrough}, (A, J, S)) \end{array}} \ (\textsc{Store})$$

Most of the new features here are common to the step rules for all the instructions: each rule checks that the current instruction is executing inside some compartment ($(A, J, S) \in C$ and $\mathit{pc} \in A$) and (using *prev*) that execution arrived at this instruction either (a) from the same compartment, or (b) with $F = \mathsf{Jumped}$ and the current pc in the previous compartment's set of jump targets (the final line of the precondition). In the new machine state, we update the previous compartment to be the compartment the pc currently lies in. And we set F to FallThrough (the rules for Jump and Jal set it to Jumped). Besides these generic conditions, the Store rule has an additional check that its write is either to the current compartment or to one of its store targets ($w_p \in A \cup S$).

Deferring detection of illegal pc changes until one step *after* they have occurred is the key trick that makes this tag-based implementation at the symbolic level work; we will use a similar approach for CFI in §6.

The compartmentalization abstract machine also provides three monitor services. The core service is *isolate*, which creates a new compartment. It takes as input the description of a fresh compartment (A', J', S') and adds it to C, also removing the addresses in A' from the address space of the parent compartment. Before allowing the operation, the service checks, relative to the parent compartment (A, J, S), that $A' \subseteq A$, that $J' \subseteq A \cup J$, and that $S' \subseteq A \cup S$. This ensures that the new compartment is no more privileged than its parent. The argument sets are passed to the service as pointers to in-memory data structures representing sets of addresses.

The other two services modify the target sets of the current compartment. If the current compartment is (A, J, S), *add_jump_target* takes as input an address $a \in A$ and modifies the current compartment to $(A, J \cup \{a\}, S)$; *add_store_target* does the same thing for store targets. Note that although *isolate* removes the child's address space from the parent, it leaves the store and jump targets of the parent unchanged, and these can overlap with the child's address space. Thus, the parent can preserve access to an address in its child's memory by calling *add_jump_target* or *add_store_target* with that address before invoking *isolate*.

In the initial configuration of the abstract machine, all defined addresses lie in one big compartment and each monitor service address has its own unique compartment (i.e., these locations are special and live outside of addressable memory). The main compartment has the addresses of the monitor services in its set of jump targets, allowing it to call them; the monitor service compartments have all defined addresses in their set of jump targets, allowing them to return to any address. Since, in order to call a monitor service, its address must lie in the calling compartment's set of jump targets, a parent compartment can choose to prevent a child it creates from calling specific services by restricting the child's jump table.

Before returning, each monitor service checks that the compartment it is returning to is the same as the one it was called from. This detail is needed to prevent malicious use of monitor services to change compartments: otherwise, calling a service from the last address of a compartment would cause execution to proceed from the first address of a subsequent compartment, even if the original compartment was not allowed to jump there.

As a sanity check on the abstract machine, we prove that it satisfies a *compartmentalization property* based on the informal presentation by Wahbe *et al.* [32]. We first prove that the machine maintains invariants ensuring that each defined memory location lies in exactly one compartment. We use this to prove that, on every step, (a) if the machine isn't stuck, then the new pc is either in the initial pc's compartment or in its set of jump targets; and (b) if a memory location was changed, then its address was either in the initial pc's compartment or in its set of store targets.

Symbolic Machine Our method for implementing this abstract machine in terms of tags involves "dualizing" the representation of compartments: rather than maintaining global state recording which compartments exist and what memory locations they are allowed to affect, we instead tag memory locations to record which compartments are allowed to affect *them*. Compartments are represented by unique ids, and the extra state of the symbolic machine contains a monotonic counter *next* for the next available compartment id.

For this policy, T_m contains triples $\langle c, I, W\rangle$, where c is the id of the compartment to which a tagged memory location belongs, I is the set of *incoming compartment ids* identifying which other compartments are allowed to jump to this location, and W is the set of *writer ids* identifying which other compartments are allowed to write to this location. T_{pc} contains pairs $\langle F, c\rangle$, where the flag F has the same role as on the abstract machine and c is the id of the compartment from which the *previous* instruction was executed. Since registers do not play a role in this micro-policy, T_r contains just the dummy value $\bullet$. The extra state contains three tags, t_I, t_{AJ}, and t_{AS}, corresponding to the tags on the monitor services' entry points in the abstract machine. We cannot use the symbolic machine's monitor service tags, as those are immutable; the compartmentalization policy thus maintain its mutable monitor service tags in the extra state. (This limitation is not fundamental, but does not impact any other micro-policies.)

Here are a few of the symbolic rules (the rest are similar):

$$\frac{c = c' \vee (F = \mathsf{Jumped} \wedge c \in I)}{\mathsf{Nop} : (\langle F, c\rangle, \langle c', I, W\rangle, -, -, -) \to (\langle \mathsf{FallThrough}, c'\rangle, -)}$$

$$\frac{c = c' \vee (F = \mathsf{Jumped} \wedge c \in I)}{\mathsf{Jump} : (\langle F, c\rangle, \langle c', I, W\rangle, \bullet, -, -) \to (\langle \mathsf{Jumped}, c'\rangle, -)}$$

$$\frac{c = c' \vee (F = \mathsf{Jumped} \wedge c \in I) \qquad c' = c'' \vee c' \in W'}{\begin{array}{l}\mathsf{Store} : \quad (\langle F, c\rangle, \langle c', I, W\rangle, \bullet, \bullet, \langle c'', I', W'\rangle) \\ \qquad\quad \to (\langle \mathsf{FallThrough}, c'\rangle, \langle c'', I', W'\rangle)\end{array}}$$

The first side-condition on all the rules guarantees that all pc changes are legal: c, taken from the pc tag, is the previously-executing compartment; and c', from the tag on the current instruction, is the current compartment. An execution step is allowed if it is in the same compartment ($c = c'$), or if it follows a jump from a permitted incoming compartment ($F = \mathsf{Jumped} \wedge c \in I$). Similarly, the extra side-condition for Store checks that the write is to a location in the currently-executing compartment ($c' = c''$) or to a location that accepts the current compartment as a writer ($c' \in W'$).

From the rules, we can see that this encoding breaks up the jump targets of each compartment, scattering the jumping compartment's id into the destination component in the tag on each individual jump target; the store target are similarly scattered across the writers component. The state maintained in the pc tag corresponds exactly to the extra state maintained by the abstract machine (i.e., F and *prev*), except that we use a compartment id rather than an abstract compartment.

The monitor services must also be rephrased in terms of tags. The *add_jump_target* service simply modifies the tag on the given address; if the previous tag was $\langle c, I, W\rangle$ and the current compartment is c', then the new tag will be $\langle c, I \cup \{c'\}, W\rangle$. The *add_store_target* service is analogous. The *isolate* service does four things: (1) It gets a fresh compartment id c_{new} (from the counter, which it then increments). (2) It retags every location in the new compartment's address space, changing its tag from $\langle c, I, W\rangle$ into $\langle c_{\text{new}}, I, W\rangle$. (3) It retags every location in the new compartment's set of jump targets, changing its tag from $\langle c_J, I_J, W_J\rangle$ into $\langle c_J, I_J \cup \{c_{\text{new}}\}, W_J\rangle$. (4) It retags the new compartment's set of store targets, changing each tag from $\langle c_S, I_S, W_S\rangle$ into $\langle c_S, I_S, W_S \cup \{c_{\text{new}}\}\rangle$.

Refinement To prove that the symbolic compartmentalization machine is correct, we prove backward simulation with respect to the abstract compartmentalization machine:

Theorem 5.1 (1-backward SA-simulation). The symbolic compartmentalization machine backward-simulates the abstract compartmentalization machine.

The bulk of the work in this proof lies in showing that when we pass from a global set of compartment information to our "dualized" tag-based approach, that we indeed retain the same information: we must prove that the compartment IDs are assigned consistently and that the jump targets/store targets correspond to the incoming/writers. In other words, our symbolic tags must "refine" our abstract compartments. This difficulty shows up for the monitor services in particular: since the effects of the monitor services are specified in terms of the abstract representation (e.g., *add_jump_target* must add a jump target, but there is no such thing at the symbolic level), the proof that the effects of the symbolic implementations on the tags do in fact correspond to the more direct abstract implementations is particularly complicated. For the single-instruction steps of the symbolic machine, we are able to capture most of the tag-based complexity in a single lemma proving that the standard symbolic check ($c = c' \vee (F = \mathsf{Jumped} \wedge c \in I)$), along with well-formedness and refinement constraints, suffices to prove that the standard abstract checks ($(A, J, S) \in C$, $pc \in A$, and $(A, J, S) = (A_{\text{prev}}, J_{\text{prev}}, S_{\text{prev}}) \vee (F = \mathsf{Jumped} \wedge pc \in J_{\text{prev}})$) hold for the corresponding compartment.

Related Work Fine-grained compartmentalization is usually achieved by software fault isolation [33]. There are several verified SFI systems, including ARMor [35], RockSalt [24], and a portable one by Kroll *et al.* [18]. Our compartmentalization model is based on Wahbe *et al.*'s original SFI work [32] but differs from it in several ways. Most importantly, our monitor is not based on binary rewriting, but instead uses the hardware/ software mechanism of the PUMP architecture. Our model is also richer in that it provides a hierarchical compartment-creation mechanism instead of a single trusted top-level program that can spawn one level of untrusted plugins. While Wahbe *et al.*'s model produces safe (intra-compartment) but arbitrary effects on compartmentalization violations, we detect such violations and halt the machine. One feature Wahbe *et al.*'s model that we do not currently support is inter-compartment

RPCs; we instead require programs to manually predeclare inter-compartment calls *and returns*.

6 Control-Flow Integrity Micro-Policy

We next outline a micro-policy enforcing fine-grained *control-flow integrity (CFI)* [1] as well as providing basic non-writable code (NWC) and non-executable data (NXD) protection. It dynamically enforces that all indirect control flows (computed jumps) adhere to a fixed control flow graph (CFG). (This CFG might, for example, be obtained from a compiler.) This prevents control-flow-hijacking attacks, in which an attacker exploits a low-level vulnerability to gain full control of a target program. A more detailed description of this micropolicy is in §A.

Our main result is a proof that a variant of the CFI property of Abadi *et al.* [1] holds for the symbolic machine when instantiated with our CFI micro-policy. For this, we first prove that the CFI property is preserved by backward simulation. We then use this preservation result to show that the symbolic CFI machine has CFI, by proving that it simulates an abstract machine that has CFI by construction. The CFI definition relies on a formal overapproximation of the attacker's capabilities, allowing the attacker to change *any* data in the system except for the code, the pc, and the tags (if the machine has them). This models an attacker that can mount buffer-overflow attacks but cannot subvert the monitor; this is a reasonable assumption since we assume that any implementation of the monitor will be able to protect its integrity. For the same reason we assume that in monitor mode all control flows are allowed. Since we assume that monitor code is correct, we do not need to dynamically enforce CFI there.

Abstract CFI Machine The abstract machine has separate instruction and data memories; the instruction memory is fixed (NWC), and instructions are fetched only from this memory (NXD). Indirect jumps are checked against an allowed set J of source-target pairs; if the control flow is invalid the machine stops. The attacker can make arbitrary changes to the data memory and registers at any time.

Symbolic CFI Machine At the symbolic level, code and data are stored in the same memory, and we use tags to distinguish between the two. Tags on memory are drawn from the set $\mathsf{Data} \mid \mathsf{Code}\ addr \mid \mathsf{Code}\ \bot$ and for the pc from $\mathsf{Code}\ addr \mid \mathsf{Code}\ \bot$ (other registers are always tagged $\bullet$). For the CFG conformance checks, instructions that are the source or target of indirect control flows are tagged with $\mathsf{Code}\ addr$, where $addr$ is the address of the instruction in memory. For example, a Jump instruction stored at address 500 is tagged $\mathsf{Code}\ 500$. The CFI policy does not need to keep track of where other instructions are located, so they are all tagged $\mathsf{Code}\ \bot$. (This keeps the number of distinct tags small, which would reduce cache pressure when executing this micro-policy on the concrete machine described in §8.) Only memory locations tagged Data can be modified (NWC), and only instructions fetched from locations tagged Code can be executed (NXD). The symbolic rule for Store illustrates both these points:

$$\mathsf{Store} : (\mathsf{Code}\ \bot, \mathsf{Code}\ \_\ , -, -, \mathsf{Data}) \to (\mathsf{Code}\ \bot, \mathsf{Data})$$

It requires the fetched Store instruction to be tagged Code and the written location to be tagged Data. On the other hand, the Jal instruction's rule requires that the current instruction be tagged $\mathsf{Code}\ src$; it then copies $\mathsf{Code}\ src$ to the pc tag:

$$\mathsf{Jal} : (\mathsf{Code}\ \bot, \mathsf{Code}\ src, -, -, -) \to (\mathsf{Code}\ src, -)$$

Only on the next instruction do we get enough information from the tags to check that the destination of the jump is indeed allowed by J. For this we add a second rule for each instruction, dealing with the case where it is the target of a jump and thus the pc tag is $\mathsf{Code}\ src$, e.g.:

$$\frac{(src, dst) \in J}{\mathsf{Store} : (\mathsf{Code}\ src, \mathsf{Code}\ dst, -, -, \mathsf{Data}) \to (\mathsf{Code}\ \bot, \mathsf{Data})}$$

We add such rules even for jump instructions, since the target of a computed jump can itself be another computed jump:

$$\frac{(src, dst\text{-}src) \in J}{\mathsf{Jal} : (\mathsf{Code}\ src, \mathsf{Code}\ dst\text{-}src, -, -, -) \to (\mathsf{Code}\ dst\text{-}src, -)}$$

Proof Organization Our proofs are structured around a generic CFI preservation result that states that CFI is preserved by backward simulation under some additional assumptions (§A). As mentioned, we use this to transport the CFI property from the abstract machine to the symbolic machine.

This approach allows us to structure our proofs in a modular way. More importantly, the reusable nature of the preservation theorem provides an easy way to transfer the CFI property from the symbolic machine to a concrete machine that correctly implements the CFI micro-policy while keeping most of the reasoning about the properties of the micro-policy at a higher level. We were able to do this and transport the CFI property to the instance of the concrete machine presented in §8; details are presented in §A.

Finally, we prove that the symbolic machine backward-simulates the correct-by-construction abstract machine, which—in combination with our CFI preservation result—proves the correctness of the micro-policy.

Related Work Abadi *et al.* [1] proposed both the first CFI definition and a reasonably efficient, though coarse-grained, enforcement mechanism based on binary analysis and rewriting, in which each node in the CFG is assigned to one of three equivalence classes. This seminal work was extended in various directions, often trading off precision for low overheads and practicality [34]. However, recent attacks against coarse-grained CFI [10], [14] have illustrated the security risks of imprecision. This has spurred interest in *fine-grained* CFI [8], [22], [28], sometimes called *complete* or *ideal* CFI; however, this has been deemed "very expensive" [14]. Several proposed hardware mechanisms are directly targeted at speeding up CFI [5], [9]; here we achieve CFI using a generic hardware mechanism in a formally verified way. The PUMP mechanism supports fine-grained CFI with modest runtime overhead [11]. Previous formal verification efforts for CFI include ARMor [35] and KCoFI [8]. Like most work on CFI, they use inline reference monitoring [13]; their verification targets a small-TCB component which validates that the right checks were inserted in the instrumented binary.

7 Memory Safety Micro-Policy

Last, we describe a micro-policy that enforces safe access to heap-allocated data, by preventing both spatial safety violations (e.g., accessing an array out of its bounds) and temporal safety violations (e.g., referencing through a pointer after the region has been freed). Such violations are a common source of serious security vulnerabilities such as heap-based buffer overflows, confidential data leaks, and exploitable use-after-free, and double-free bugs. The policy we study here only guards heap-allocated data, for which calls to the *malloc* and *free* monitor services tell us how to set up and tear down memory regions; we leave stack allocation and C-like unboxed structs as future work.

Abstract Machine The abstract machine presents a block-based memory model to the programmer [4], [20]: it operates on values that are either ordinary machine words w or pointers p. A pointer is a pair (b, o) of a block identifier b (drawn from an infinite set) and an offset o (a machine word). The memory is a partial function from block identifiers to lists of values; its domain is the set of allocated blocks. Load and Store require pointer values (b, o). They first look up the block id b in the memory; if this block is currently allocated, they obtain a list of values vs, which they read or update at index o (provided o is in bounds).

$$\frac{\begin{array}{c} mem[b_{pc}] = vs_{pc} \quad vs_{pc}[o_{pc}] = i \quad decode\ i{=}\mathsf{Store}\ r_p\ r_s \\ reg[r_p]{=}(b,o) \quad reg[r_s]{=}v \\ mem[b]{=}vs \quad vs'{=}vs[o{\leftarrow}v] \quad mem'{=}mem[b{\leftarrow}vs'] \end{array}}{(mem, reg, (b_{pc}, o_{pc})) \to (mem', reg, (b_{pc}, o_{pc}{+}1))} \ (\textsc{Store})$$

The pc itself contains a pointer with a block and an offset; instruction fetching works the same as normal memory loads.

As with sealing key generation (§4), the allocation and freeing monitor services are parameterized by two functions, *alloc_f* and *free_f*, that are assumed to satisfy certain high-level properties: *alloc_f* takes a memory and a size, and returns a block that was not already allocated and a new memory in which this block is mapped to a frame; *free_f* takes a memory and an allocated block and returns a new memory where the block is no longer allocated, keeping all other blocks the same.

Symbolic Machine In the symbolic part of the memory-safety micro-policy, we replace the block-structured memory of the abstract machine by a flat memory where each cell is tagged with a *color* representing the block to which it belongs. Pointers are also tagged with colors, and when a pointer is dereferenced we check that its color matches the color of the memory cell it points to.

More precisely, we use different sets of tags for values in registers (and the pc) and in memory. Value tags t_v are either pointers tagged with a *color* c or non-pointers tagged $\bot$. Allocated memory locations are tagged with a pair (c, t_v), where c is the color of the encompassing block and t_v is the tag of the stored value. Unallocated memory is tagged with the special tag F (free). We use t_m to range over memory tags. The extra state for this policy is a list of *block descriptors* recording which memory regions have been allocated (with the corresponding base and bounds) and which colors correspond to them, plus a counter for generating new colors.

The *malloc* monitor service first searches the list of block descriptors for a free block of at least the required size, cuts off the excess if needed, generates a fresh color c, initializes the new memory block with $0@(c, \bot)$, and returns the atom $w@c$, where w is the start address of the block.

The *free* monitor service reads the pointer color, deallocates the corresponding block, tags its cells with F, and updates the block descriptors. The F tags prevent any remaining pointers to the deallocated block from being used to access it after deallocation. If a later allocation reuses the same memory, it will be tagged with a different (larger) color, so these dangling pointers will still be unusable.

The symbolic rules for Load and Store check that the pointer and the referenced location have the same color c.

$$\begin{array}{lcl} \mathsf{Load} & : & (c_{pc}, (c_{pc}, \bot), c, (c, t_v), -) \to (c_{pc}, t_v) \\ \mathsf{Store} & : & (c_{pc}, (c_{pc}, \bot), c, t_v, (c, t_v')) \to (c_{pc}, (c, t_v)) \end{array}$$

We additionally require that the pc tag c_{pc} matches the color of the block to which the pc points. This ensures that the pc cannot be used to leak information about inaccessible frames by loading instructions from them. On Jumps we change the color of the pc to the color c of the pointer, while for Jal we also use c_{pc} to tag the $\mathsf{r_a}$ register:

$$\begin{array}{ll} \mathsf{Jump} & : (c_{pc}, (c_{pc}, \bot), c, -, -) \to (c, -) \\ \mathsf{Jal} & : (c_{pc}, (c_{pc}, \bot), t_v, -, -) \to (t_v, c_{pc}). \end{array}$$

We also allow Jals to words tagged $\bot$, since monitor services lie outside the accessible memory at this level of abstraction and so cannot be referenced by normal pointers.

Binary operations are allowed between values tagged $\bot$ (non-pointers), and they produce values tagged $\bot$:

$$\mathsf{Binop}_{\oplus} : (c_{pc}, (c_{pc}, \bot), \bot, \bot, -) \to (c_{pc}, \bot)$$

We also allow adding and subtracting integers from pointers:

$$\begin{array}{ll} \mathsf{Binop}_{+,-} & : (c_{pc}, (c_{pc}, \bot), c, \bot, -) \to (c_{pc}, c) \\ \mathsf{Binop}_{+} & : (c_{pc}, (c_{pc}, \bot), \bot, c, -) \to (c_{pc}, c) \end{array}$$

The result of such pointer arithmetic is a pointer with the same color c. The new pointer is not necessarily in bounds, but the rules for Load and Store will prevent invalid accesses. (Computing an out-of-bounds pointer is not a violation *per se*—indeed, it happens quite often in practice, e.g., at the end of loops.) Moreover, subtraction can compute the integer offset between two pointers to the same block:

$$\mathsf{Binop}_{-,=} : (c_{pc}, (c_{pc}, \bot), c, c, -) \to (c_{pc}, \bot)$$

Pointers to the same block can also be compared for equality using Binop$_=$. But comparing a pointer and a non-pointer or comparing two pointers to different blocks must be disallowed (because two out-of-bounds pointers to different blocks can be numerically equal at the symbolic level, whereas they cannot be equal at the abstract level). While the transfer function can detect this situation, it cannot alter the results of instructions; thus, we can only preserve refinement by having the transfer function stop execution. Intuitively, all instructions on pointers must be expressible at the abstract level independent of base addresses. (Subtraction works as presented because equal base addresses cancel out, even in the presence of overflows.) Monitor services do not have this restriction, though, so we can—if one is required—provide a total equality service that returns false in those cases where the equality instruction would be disallowed.

Refinement We prove a backward-simulation theorem similar to the one for sealing (§4):

Theorem 7.1 (1-backward SA-simulation). The symbolic memory safety machine backward-simulates the abstract machine.

The main technical difficulty lies in formalizing the correspondence between memory addresses at the symbolic and abstract levels, and showing that this correspondence is preserved throughout execution. Specifically, each color c used at the symbolic machine should map to a block identifier b at the abstract level, in such a way that the memory region tagged with c matches the block pointed to by b; we consider that an address x marked with color c should correspond to memory location $(b, x - x_{base})$ at the abstract level, where x_{base} is the base address of the corresponding region. Additionally, we must maintain the invariant that symbolic block descriptors faithfully describe how memory regions are tagged.

Showing that memory operations and monitor services (in particular, the allocator) preserve the refinement relation involves explicitly manipulating the address mappings described above and a fair amount of low-level reasoning about address segments and arithmetic, which consumes almost half of the complete proof.

Related Work Our scheme is inspired by the metadata tainting technique of Clause *et al.* [7]. Similar ideas have been used by Watchdog [25] (for temporal safety), though these systems do not have formal proofs. Nagarakatte *et al.* have verified in Coq that the SoftBound pass in LLVM/Vellvm satisfies "spatial safety" [27] and that the CETS temporal safety extension to SoftBound is correct in the sense of backward simulation[26]. These proofs are with respect to correct-by-construction special-purpose machines. Abadi and Plotkin [2] show that address space layout randomization can be used to prevent low-level attacks, including memory safety violations, by proving a probabilistic variant of full abstraction with respect to a high-level language semantics.

8 Concrete Machine

Having explored four examples of how the symbolic machine can be instantiated to enforce a variety of micro-policies, we turn to the question of how its behavior can be realized on a *concrete* machine that incorporates PUMP-like hardware [11] in idealized form. The concrete machine differs from the symbolic one in several key ways. (1) Its memory, registers, and pc hold *concrete atoms* of the form $w@t$, where the *concrete tag* t is simply a machine word (possibly interpreted as a pointer into memory). (2) It propagates and checks tags using a *cache* of *concrete rules*, each encoding a single tuple from the graph of the (concrete) transfer function. (3) The cache initially contains a finite set of *ground rules*; it is further populated as needed by a software *miss handler*, which embodies the transfer function. (4) Extra machine state is represented by ordinary in-memory data structures. (5) Each monitor service is implemented as an (almost) ordinary software subroutine, whose starting address coincides with the service's entry point at the abstract and symbolic levels. An instance of the symbolic machine for a specific micro-policy is realized on the concrete machine by defining a concrete encoding for tags and any extra state, providing a miss handler that implements the symbolic transfer function, and providing implementations of any monitor services. In §9, we describe a generic approach to constructing and verifying such realizations. In the remainder of this section, we formalize the concrete machine itself as an extension of the basic machine (§2). A practical PUMP implementation [11] would add similar extensions to a real-world RISC ISA. The details in this section are not needed to follow §9 and can be skimmed if desired.

The concrete machine adds four new instructions for monitor code:

$$\mathsf{AddRule} \mid \mathsf{JumpFpc} \mid \mathsf{GetTag}\ r_s\ r_d \mid \mathsf{PutTag}\ r_s\ r_{tag}\ r_d$$

AddRule, described in detail below, inserts a new rule into the cache. JumpFpc is used to return from the miss handler: it jumps to the address in the fpc ("fault pc"), a new special-purpose register that holds the address of the faulting instruction after a cache miss. GetTag r_1 r_2 takes the tag t from the atom $w@t$ stored in r_1 and returns it as the payload part of a new atom $t@\mathsf{Monitor}$ in r_2, where Monitor is a fixed concrete tag used by monitor code. PutTag r_1 r_2 r_3 does the converse: if r_1 and r_2 contain $w_1@t_1$ and $w_2@t_2$, it stores $w_1@w_2$ into r_3. The monitor self-protection mechanism described in §9 ensures that these instructions can only be executed by monitor code.

Concrete states have the form (*mem ,reg ,pc ,fpc ,cache*), where *cache* is a set of *concrete rules*, each of the form (iv, ov). The input vector *iv* represents the key for rule cache lookups and contains the instruction opcode, the tag of the current instruction, the tag of the pc, and up to three operand tags. The output vector *ov* provides the tags of the new pc and of the result. On each step, the machine constructs *iv* from the current instruction opcode and the relevant tags and looks it up in the cache. If a matching rule is found (written $cache \vdash iv \mapsto ov$), the instruction is allowed and the next state is tagged according to *ov*. If no rule matches ($\mathit{cache} \vdash \mathit{iv} \uparrow$), then *iv* is saved in memory, the current pc value is saved in the fpc, and control is transferred to a fixed address where the miss handler should be loaded (trapaddr). Accordingly, each rule in the step relation comes in two variants—one for when we hit in the cache and one for when we trap to the miss handler. For example, here are the rules for Store:

$$\frac{\begin{array}{c} \mathit{mem}[w_{pc}] = i@t_i \qquad \mathit{decode}\ i = \mathsf{Store}\ r_p\ r_s \\ \mathit{reg}[r_p]{=}w_p@t_p \qquad \mathit{reg}[r_s]{=}w_s@t_s \qquad \mathit{mem}[w_p]{=}\_@t_d \\ \mathit{cache} \vdash (\mathsf{Store}, t_{pc}, t_i, t_p, t_s, t_d) \mapsto (t'_{pc}, t'_d) \\ \mathit{mem}' = \mathit{mem}[w_p{\leftarrow}w_s@t'_d] \end{array}}{\begin{array}{c} (\mathit{mem}, \mathit{reg}, w_{pc}@t_{pc}, \mathit{fpc}, \mathit{cache}) \\ \to (\mathit{mem}', \mathit{reg}, (w_{pc}{+}1)@t'_{pc}, \mathit{fpc}, \mathit{cache}) \end{array}} \quad (\textsc{Store})$$

$$\frac{\begin{array}{c} \mathit{mem}[w_{pc}] = i@t_i \qquad \mathit{decode}\ i = \mathsf{Store}\ r_p\ r_s \\ \mathit{reg}[r_p]{=}w_p@t_p \qquad \mathit{reg}[r_s]{=}w_s@t_s \qquad \mathit{mem}[w_p]{=}\_@t_d \\ \mathit{cache} \vdash (\mathsf{Store}, t_{pc}, t_i, t_p, t_s, t_d) \uparrow \\ \mathit{mem}' = \mathit{mem}[0..5 \leftarrow (\mathsf{Store}, t_{pc}, t_i, t_p, t_s, t_d)] \end{array}}{\begin{array}{c} (\mathit{mem}, \mathit{reg}, w_{pc}@t_{pc}, \mathit{fpc}, \mathit{cache}) \\ \to (\mathit{mem}', \mathit{reg}, \mathsf{trapaddr}@\mathsf{Monitor}, w_{pc}@t_{pc}, \mathit{cache}) \end{array}} \quad (\textsc{Store-Miss})$$

Addresses 0 to 5 are used, by convention, to store the current *iv* for use by the miss handler in the final premise of the second rule. The miss handler computes the result tags, stores them at addresses 6 and 7, and uses the AddRule instruction to insert the new rule into the cache.

$$\frac{\begin{array}{c} \mathit{mem}[w_{pc}] = i@t_i \qquad \mathit{decode}\ i = \mathsf{AddRule} \\ \mathit{cache} \vdash (\mathsf{AddRule}, t_{pc}, t_i, -, -, -) \mapsto (t'_{pc}, -) \\ \mathit{mem}[0..7] = (\mathit{opcode}, t_1, t_2, t_3, t_4, t_5, t_6, t_7) \\ \mathit{cache}' = \mathit{cache} \uplus ((\mathit{opcode}, t_1, t_2, t_3, t_4, t_5) \mapsto (t_6, t_7)) \end{array}}{\begin{array}{c} (\mathit{mem}, \mathit{reg}, w_{pc}@t_{pc}, \mathit{fpc}, \mathit{cache}) \\ \to (\mathit{mem}, \mathit{reg}', (w_{pc}{+}1)@t'_{pc}, \mathit{fpc}, \mathit{cache}') \end{array}} \quad (\textsc{AddRule})$$

Here $\uplus$ is map update, overwriting any previous value for $(\mathit{opcode}, t_1, t_2, t_3, t_4, t_5)$. For simplicity, we assume that the cache's size is unlimited, avoiding the need to model eviction; limiting the cache size would require more complicated code for handling cache misses, of course, but would not change the specification of handler correctness (see §9).

A final detail is that the machine can be configured on a per-opcode basis to mask out (i.e., set to a predefined "don't care" tag) selected fields of the *iv* before matching against the cache. This is easy to implement in hardware, and it permits a single cache entry to match many different *iv* tuples. The machine can also be configured on a per-opcode basis to "copy through" a specified *iv* tag to either of the *ov* tag fields. These features allow more compact representation of transfer functions as concrete rules. The machine uses a special pair of don't-care and copy-through masks when running monitor code (i.e., when the pc tag is Monitor); we use this to ensure that the set of ground rules is finite (see §9) and that the monitor does not fault when it comes in contact with user tags (for instance when returning back to user mode).

9 Concrete Micro-Policy Monitor

The last piece of our story is the realization of symbolic micro-policies on the concrete machine. Although symbolic micro-policies vary widely in details, concrete micro-policy implementations share several important characteristics: (1) concrete tags (and extra state) must faithfully encode symbolic tags (and extra state); (2) the concrete miss handler and monitor services code must implement the symbolic machine's Coq specifications; (3) control transfers between user and monitor code must obey a clear protocol; and (4) tags and monitor code and data must be protected from malicious or compromised user code. To take advantage of these commonalities, we have built a generic framework for organizing the construction of concrete micro-policies and proved a theorem stating that they refine a corresponding symbolic machine instance. Since symbolic micro-policies can be specified using the full power of Gallina, the details of concrete tag and extra state representation plus the actual code for the miss handler and monitor services must all be provided by the micro-policy designer. This code might be handwritten or generated from a high-level language by a compiler; the details are unimportant. The proof of correctness of the generic framework is parameterized on correctness proofs for these policy-specific components.[2]

Tag Representation A micro-policy has four sets of symbolic tags (T_m, T_r, T_{pc} and T_s) that must be represented as word-sized bit vectors on the concrete machine. Concrete tags on the register file and the pc will be used to represent symbolic ones drawn from the corresponding sets—namely, T_r and T_{pc}. Since monitor services are implemented by code that lives in memory, a tag in memory will either represent something in T_m (in which case it marks a memory location that is visible at the symbolic level), or something in T_s (in which case it marks the address of a monitor service).

Formally, this representation scheme is specified by partial *decoding* functions $\mathit{dec}_k(\mathit{mem}^C, t^C)$ that take a concrete machine memory and concrete machine tag word as inputs, and produce symbolic tags as output.[3] Here, $k \in \{\mathsf{M}, \mathsf{R}, \mathsf{P}\}$ specifies which kind of concrete tag—memory, register or pc—we are decoding, so that we know what kind of symbolic tag to produce. Hence, $\mathit{dec}_\mathsf{M}(\mathit{mem}^C, t^C) \in \{\mathsf{User}\ t^S \mid t^S \in T_m\} \uplus \{\mathsf{Entry}\ t^S \mid t^S \in T_s\}$, while $\mathit{dec}_\mathsf{R}(\mathit{mem}^C, t^C) \in T_r$ and $\mathit{dec}_\mathsf{P}(\mathit{mem}^C, t^C) \in T_{pc}$. We say that a concrete tag is a *valid user-level tag* (given some memory) when it can be decoded into a symbolic tag. For simplicity, from here on we will refer to such tags by their symbolic decodings. We require that $\mathit{dec}_k(\mathit{mem}^C, \mathsf{Monitor})$ be undefined—i.e., that no symbolic tag be represented by it.

Monitor Self-Protection At the symbolic level, it is impossible for user code to interfere with the internal state of the micro-policy. At the concrete level, however (unlike in some of our own previous work [4], [11]), monitor code and data live in ordinary memory and registers, which user code must somehow be prevented from accessing. Moreover, we need to ensure that only monitor code can execute the special instructions AddRule, PutTag, GetTag, and JumpFpc. Fortunately, we can use the PUMP itself both to implement the symbolic micro-policy and, at the same time, to enforce the restrictions above (which we call *monitor self-protection*). To achieve this, we use the special Monitor tag to mark all of the monitor's code and data, allowing the miss handler to detect when untrusted code is trying to tamper with it, as explained below.

---

[2] To manage the size of our verification effort and focus attention on the more novel parts, we *assume* the existence of correct monitor implementations as hypotheses. We expect the actual implementations to be straightforward, and verification of this kind of low-level code is a well-studied area [4], [6], [17].

[3] For simple micro-policies, symbolic tags can be accurately represented in a single machine word, so *dec* does not depend on the memory argument. More complex micro-policies may use the memory argument to represent tags as data structures in memory—e.g., the compartmentalization micro-policy of §5 uses this feature.

Monitor Code and Ground Rules On the concrete machine, every instruction causes a rule cache lookup, which results in a fault if no corresponding rule is present. Since the machine has no special "privileged mode," this applies even to monitor code. To ensure that monitor code can do its job, we set up cache ground rules (one for each opcode) saying that the machine can step whenever the *PC* and *CI* tags in the *iv* are tagged Monitor; in this case, the next pc and any result of the instruction are also tagged Monitor. Monitor code must never change or override these rules.

In addition, the fact that the machine uses a special pair of don't-care and copy-through masks when running monitor code lets us ensure that the monitor does not fault when coming in contact with user tags. For example, while policies will usually check the tag on the target pc every time user code performs a Jump (which may cause faults), such checks are not needed for monitor code, since we assume that it behaves correctly. To allow for both behaviors, we program the monitor-specific masks to always use the tag of a Jump target as the new pc tag, while disabling this bypass in the normal masks. Aside from the *PC* and *CI* tags, all other positions in the *iv* are marked as don't-care for all opcodes. Copy-throughs are used for keeping the same pc tag in most instructions and for copying the pc tag from a register tag in the case of Jump, Jal, and JumpFpc. Mov, Load, Store, and Jal also use copy-through for the result tag.

Miss Handler Since ground rules ensure that monitor code never faults, the miss handler is only invoked for monitoring user-level code. The job of the miss handler is thus twofold: (i) implement the symbolic transfer function of a micro-policy; and (ii) enforce monitor self-protection. For the latter, the miss handler just needs to ensure that the faulting opcode is not a privileged instruction (e.g., AddRule), and that the Monitor tag does not occur anywhere in the faulting *iv* (which, crucially, includes the tags on the "old contents" of any registers and memory locations that the instruction overwrites). If these checks fail, the miss handler halts the machine. (In a real system, the miss handler would instead tell the scheduler to halt just the offending process.) Otherwise, the miss handler can compute the transfer function on the *iv*, halting the machine if it violates the micro-policy. If the instruction is allowed, the miss handler should store the resulting *ov* into the appropriate memory slots, call AddRule to install it, and restart the instruction that trapped by jumping through the fpc register.

Besides correctly implementing its symbolic counterpart, the concrete transfer function is also responsible for setting the next pc to Monitor whenever a valid monitor-service call is made (i.e., when the instruction tag is Entry t^S). This ensures that monitor services can execute with the appropriate privilege.

Refinement We formalize the relation between the symbolic machine (instantiated with the symbolic parts of some micro-policy) and the concrete machine (instantiated with the concrete parts of the same micro-policy) as a *backward refinement* between their step relations.[4] The proof of backward refinement relies on some lemmas relating the symbolic and concrete parts of the specific policy; the proofs of these must also be supplied by the micro-policy designer.

At the heart of our refinement result lies the following *strong simulation relation*, which describes how a symbolic machine state is represented at the concrete level.

Definition 9.1. Strong simulation $\approx^{CS}$ is defined as follows (its pieces are discussed below):

$$\frac{\begin{array}{c} \mathit{reg}^C \sim_{\mathit{mem}^C} \mathit{reg}^S \qquad \mathit{mem}^C \sim \mathit{mem}^S \\ \mathit{cache\_ok}(\mathit{mem}^C, \mathit{cache}) \qquad \mathit{services\_ok}(\mathit{mem}^C) \\ \mathit{mem}^C[0..7] = [\_@\mathsf{Monitor}, \ldots, \_@\mathsf{Monitor}] \\ I(\mathit{mem}^C, \mathit{reg}^C, \mathit{cache}, \mathit{extra}) \qquad \mathit{dec}_{\mathsf{P}}(\mathit{mem}^C, t^C) = t^S \end{array}}{\begin{array}{c} (\mathit{mem}^C, \mathit{reg}^C, \mathit{pc}@t^C, \mathit{fpc}, \mathit{cache}) \\ \approx^{CS} (\mathit{mem}^S, \mathit{reg}^S, \mathit{pc}@t^S, \mathit{extra}) \end{array}}$$

This relation is implicitly parameterized over policy-specific decoding functions (dec_P, etc.) plus an invariant I, which should be chosen to ensure that (i) the symbolic machine's extra state component is correctly represented in the concrete machine memory; (ii) the monitor's code and data are tagged appropriately; and (iii) the cache contains the ground rules needed by the monitor. A concrete register file simulates a symbolic one ($\mathit{reg}^C \sim_{\mathit{mem}^C} \mathit{reg}^S$) when they agree on register values and the tags decode to the corresponding symbolic tags:

$$\forall r, x, t^S, (\exists t^C, \mathit{reg}^C[r] = x@t^C \wedge \mathit{dec}_{\mathsf{R}}(\mathit{mem}^C, t^C) = t^S) \iff \mathit{reg}^S[r] = x@t^S$$

The relation $\mathit{mem}^C \sim \mathit{mem}^S$ is defined similarly; notice that the concrete machine can contain more registers or memory locations than the symbolic machine, as long as the concrete tags on these registers or locations do not encode any valid user tag. The predicate *cache_ok* states that, whenever a rule with a valid user-level pc tag is found in the cache, the rule's result matches that of the symbolic transfer function, modulo the tag encoding. The predicate *services_ok* states that each location in the concrete memory that corresponds to a monitor service is tagged with Entry t^S, where t^S is that service's tag.

---

[4] Our formal Coq development also gives sufficient conditions for proving *forward* refinement between the implementation of some policy on the concrete machine and the corresponding symbolic machine instance. Since this is not the focus of the present work—and since, in any case, forward refinement between the symbolic and abstract machines doesn't hold for all policies—we omit the details here, referring the reader to the formal development for more information.

Once again, we would like to construct a backward refinement inductively by using a backward simulation (Definition 4.2). However, we can't use Definition 9.1 for this right away, since backward refinement doesn't hold for it because steps taken by the concrete machine while inside the monitor are not mapped to any steps of the symbolic machine. Moreover, the concrete monitor will often need to temporarily break both the invariants and the strong correspondence with respect to some symbolic state. To address these points, we use Definition 9.1 to define a *weak simulation relation* $\sim^{CS}$:

Definition 9.2. $s^C \sim^{CS} s^S$ if either (i) the pc of s^C has a valid user-level tag and $s^C \approx^{CS} s^S$, or else (ii) the pc of s^C is tagged Monitor and there exists a state s^C_U with a pc that is a valid user-level tag such that $s^C_U \approx^{CS} s^S$ and $s^C_U \to^* s^C$, where all states in this execution have the pc tagged Monitor.

Case (ii) handles concrete states where the monitor is executing, giving us a way to remember enough information from the point where the monitor was invoked to be able to reestablish strong simulation once execution returns to user mode.

Definition 9.3 ($\{0,1\}$-backward simulation). We say that a low-level machine $(\mathit{State}^L, \to^L)$ *$\{0,1\}$-backward simulates* a high-level machine $(\mathit{State}^H, \to^H)$ with respect to a relation $\sim$ if, whenever $s^L_1 \sim s^H_1$ and $s^L_1 \to s^L_2$, either $s^L_2 \sim s^H_1$ or there exists s^H_2 such that $s^H_1 \to s^H_2$ and $s^L_2 \sim s^H_2$.

Theorem 9.4 (Backward CS-simulation). The concrete machine $\{0,1\}$-backward-simulates the symbolic machine, with respect to $\sim^{CS}$.

The proof assumes the correctness of the monitor machine code provided by the micro-policy designer: (1) On a cache miss, if all the invariants are satisfied at the faulting instruction, then the miss handler must successfully return to a user state only if the faulting tag combination is allowed by the transfer function. In this case, the resulting user state must be a refinement of the original symbolic state, and the cache must be updated to allow execution to proceed. (2) When executing a monitor service, the concrete machine returns to user code only if the corresponding symbolic monitor service allows that execution. In this case, the resulting user state must be a refinement of the new symbolic state. (3) Monitor data structures and invariants are not affected by updates to user memory.

Therefore, for any policy implemented in terms of abstract and symbolic machines, we can get end-to-end refinement by composing Theorem 9.4 and the policy-specific symbolic-abstract simulation, relating the abstract machine to the concrete machine instantiated with a correct monitor implementation.

Example: Concrete Sealing Machine To implement the sealing micro-policy from §4 on the concrete machine, we can represent symbolic sealing tags as follows on a concrete machine with 32-bit words, assuming that $|\mathit{SK}| \leq 2^{28}$ and the Monitor tag is represented by 0.[5]

$$\begin{aligned} f(0^{28} \cdot 0 \cdot 0) &= \mathsf{Data} & \mathit{dec}_{\mathsf{P}}(m, \_ \cdot 1) &= \bullet \\ f(k \cdot 0 \cdot 1) &= \mathsf{Key}\ k & \mathit{dec}_{\mathsf{R}}(m, t \cdot 0 \cdot 1) &= f(t) \\ f(k \cdot 1 \cdot 1) &= \mathsf{Sealed}\ k & \mathit{dec}_{\mathsf{M}}(m, t \cdot 0 \cdot 1) &= \mathsf{User}\ f(t) \\ & & \mathit{dec}_{\mathsf{M}}(m, \_ \cdot 1 \cdot 0) &= \mathsf{Entry}\ \bullet \end{aligned}$$

Here $\cdot$ is bitstring concatenation and k is a 28-bit binary representation of a symbolic key. (Notice that our sealing tags can be represented in a single word, so *dec* does not depend on the machine memory.) The key counter on the symbolic machine is represented concretely as a single word of monitor memory. Implementing the transfer function is easy: we just need to prevent certain operations (e.g., Binop) from being performed on sealed values and keys, following the symbolic rules presented in §4. Implementing the monitor services is also simple. The *mkkey* routine increments the key counter (halting if it would overflow) and remembers the old value k. It then tags the return register with Key k (with a dummy payload) and returns to user code. The *seal* routine checks (using GetTag) that its first argument has the form x@Data and its second argument is tagged Key k, assembles x@Sealed k in $\mathsf{r_{ret}}$ using PutTag, and returns; *unseal* does the converse. All of these routines halt if the arguments do not have the required form.

10 Related Work

We have already discussed work related to specific micro-policies. Here we focus on work related to micro-policies and reference monitors in general.

Micro-Policies The micro-policies framework and the PUMP architecture have their roots in SAFE, a clean-slate, security-oriented architecture [12]. There, the PUMP was used only to implement dynamic IFC; other special-purpose hardware mechanisms enforced properties such as memory safety [19] and compartmentalization [12]. Still, the PUMP design in the SAFE system was made quite flexible, since dynamic IFC is an active area of research, with various mechanisms and "label models" being proposed regularly, making baked-into-hardware solutions unattractive. A simple IFC micro-policy was studied formally for an idealized version of the SAFE processor [4].

The present work aims to demonstrate the applicability of the PUMP beyond IFC and beyond clean-slate hardware. We consider a diverse set of micro-policies and a more conventional architecture—a simplified RISC machine, with bit-strings as words (instead of integers), with registers (instead of a hardware stack), and with no separate instruction memory, no call-stack or memory protection, no special monitor mode with access to protected memory, and no special monitor invocation instruction. Despite giving up these multiple specialized hardware protection features, we obtain similar kinds of protection through more extensive use of the PUMP's tagging features.

The general structure of our proofs is similar to [4]; in particular, that work also proves refinement between a concrete machine and an abstract one, using a "symbolic IFC rule machine" as an intermediate step; also, as we do here for CFI, it proves a generic preservation theorem that non-interference can be carried to the lowest level. The rule machine in [4], however, is merely a reformulation of an IFC abstract machine to factor a "rule table" (written in a simple IFC-specific domain-specific language) out of the semantics. In contrast, our symbolic machine is generic and is reused by all micro-policies. On the other hand, the proofs in [4] include the verification of an IFC monitor at the machine code level using a framework for structured code generators and a verified DSL compiler, both specially crafted for the simple architecture used there. Here, we chose to focus on designing a generic micro-policy framework and on building and verifying the symbolic machine instances for a diverse set of micro-policies, leaving concrete-level implementation and verification for later.

---

[5] Disclaimer: We have implemented and tested this concrete sealing micro-policy as a sanity check on our framework, but we have not formally proved the sealing-specific assumptions supporting Theorem 9.4.

Another paper on the PUMP [11] proposes implementation optimizations for its hardware architecture and experimentally evaluates their overhead for a set of micro-policies including CFI and memory safety, plus a taint-tracking micro-policy. Our work here is complementary, focusing on *formal* specification and verification of micro-policies. Also, the micro-policies for compartmentalization and dynamic sealing, as well as the mechanisms for monitor services and monitor self-protection described here, are new.

Reference Monitors Reference monitors have been around since the early seventies [3]. However, building low-overhead enforcement mechanisms for a broad set of policies has proved challenging. Besides low overhead, Anderson's seminal work mentions a set of security requirements for reference monitors: *"(a) The reference monitor must fully mediate all operations relevant to the enforced security policy. (b) The integrity of the reference monitor must be protected, either by the reference monitor itself or by some external means. (c) The correctness of the reference monitor must be assured, in part by making the reference monitor be small enough to analyze and test."*

Micro-policies meet all these requirements: (a) they provide complete mediation at the level of individual instructions; (b) they include mechanisms for using tags to protect the integrity of monitor code and data structures (§9); and (c) the TCB of micro-policy monitors is generally quite small. Moreover, we achieve high confidence in the their correctness by formal verification of symbolic policies in Coq; in the future we hope also to verify low-level concrete implementations. Finally, while precisely characterizing the class of properties that can be expressed as micro-policies and efficiently enforced by the PUMP is an interesting open problem, we know for sure that it includes very important security properties: IFC, CFI, compartmentalization, and memory safety. Inspiration for attacking the expressiveness question formally may come from the work by Schneider *et al.* [15], [29] on execution monitors and program rewriting.

11 Conclusions and Future Work

We have presented a generic verification framework for a rich class of low-level, hardware-accelerated, tag-based security enforcement mechanisms. The micro-policies we verify target a wide range of critical security properties, illustrating the power of a simple but flexible hardware mechanism.

Our Coq development runs to about 17.7k lines of code, out of which 4.8k lines are generic (2.8k for the symbolic machine and the generic symbolic-concrete refinement proof) and the rest specific to our four micro-policies (4.9k for compartmentalization, 4.7k for CFI, 1.9k for memory safety, and 1.2k for dynamic sealing). Our Coq development is available at https://github.com/micro-policies/micro-policies-coq.

We are currently working on a micro-policy for call-stack protection, as well as extensions of the current policies, such as memory protection for stack-allocated data and unboxed `structs`. An obvious question at the level of the framework itself is how to compose micro-policies. Certain micro-policies are known to compose sensibly, and micro-architectural optimizations ensure that they perform well on practical workloads [11], but the general picture remains unclear. Another obvious target for future work is formalizing the symbolic rule language that we used informally here for exposition.

Our framework currently targets a simplified ISA with a limited instruction set, a single core, no hardware concurrency or interrupts, etc. An interesting challenge is to scale our formalization to a more realistic RISC architecture such as MIPS, Alpha, RISC-V, or ARM extended with a PUMP. Also, we have not explicitly considered the role of the compiler or loader here, although in reality their support is crucial for some policies. For example, CFI relies on having a control-flow graph, which would naturally come from a compiler, and on the initial tags on instructions, which would have to be added or at least vetted by the loader. We have not formalized the operating system or its interaction with micro-policy monitoring. Indeed, micro-policies might even live below an OS, and could then help protect the OS itself from attacks. Another alternative (discussed in [11]) is to only protect user-level code, but this would lead to a larger TCB.

Acknowledgments We are grateful to Delphine Demange, Udit Dhawan, André DeHon, Greg Morrisett, Steve Zdancewic, and the anonymous reviewers for helpful discussions and thoughtful feedback on earlier drafts. This material is based upon work supported by the DARPA CRASH program through the US 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. Tolmach was partly supported by a Digiteo Chair at Laboratoire de Recherche en Informatique, Université Paris-Sud.

Appendix

A Details of the CFI Micro-Policy

CFI Property and Attacker Model We give a generic CFI definition that can be instantiated to all three levels of machine, adapting the original definition by Abadi *et al.* [1] to our setting. The two main technical differences are that (1) at the symbolic level, the tag-based mechanism detects a CFI violation on the step *after* it has occurred (i.e., when checking the instruction following an illegal control transfer, rather than the instruction that caused the transfer) and (2) at the concrete level, detecting a violation and halting the machine is a nontrivial process involving missing in the hardware rule cache and running the software miss handler, which eventually halts. These differences do not affect security (at both levels, the machine halts before it does anything externally visible), but they lead to a slightly more complex CFI definition.

As usual [1], the definition is given with respect to an extended step relation $\rightarrow$, which is the union of normal machine steps $\rightarrow_n$ and attacker steps $\rightarrow_a$.The $\rightarrow_n$ and $\rightarrow_a$ relations are parameters of the general CFI definition. The $\rightarrow_a$ relation represents an overapproximation of the attacker's capabilities, allowing the attacker to change *any* user-level data in the system but none of the code. At the concrete and symbolic levels, the attacker will also be prevented from directly changing the tags. This models an attacker that can mount buffer-overflow attacks but has no built-in capability for subverting our NWC, NXD, or CFI protections (e.g., no hardware backdoor).

We start by defining when a trace has CFI with respect to a set of allowed indirect jumps J (a binary relation on code addresses). From J we can construct the complete CFG, a relation on machine states written $\mathit{cfg}\ J$. This involves adding all direct CFG edges that are clear in the code (e.g., a Nop or Bnz can reach the next instruction; a Bnz can reach its target).

Definition A.1. We say that an execution trace $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$ *has CFI* if $(s_i, s_{i+1}) \in \mathit{cfg}\ J$ for all $i \in [0, \ldots, n)$ such that $s_i \rightarrow_n s_{i+1}$.

Compared to Abadi *et al.* [1], this definition additionally requires that an attacker step which happens to be a valid normal step must also be in the CFG, which is helpful for proving CFI preservation. This definition is slightly stronger than Abadi *et al.*'s; however, we instead use the following incomparable definition, which allows a single violation in a trace, as long as the machine is "stopping" afterwards.

Definition A.2 (CFI). We say that the machine $(\mathit{State}, \mathit{init}, \rightarrow_n, \rightarrow_a, \mathit{cfg}, \mathit{stopping})$ has CFI with respect to the set of allowed indirect jumps J if, for any execution starting from initial state s_0 and producing a trace $s_0 \rightarrow \ldots \rightarrow s_n$, either (1) the whole trace has CFI according to Definition A.1, or else (2) there is some i such that $s_i \rightarrow_n s_{i+1}$, and $(s_i, s_{i+1}) \notin \mathit{cfg}\ J$, where the sub-traces $s_0 \rightarrow \ldots \rightarrow s_i$ and $s_{i+1} \rightarrow \ldots \rightarrow s_n$ both have CFI and the sub-trace $s_{i+1} \rightarrow \ldots \rightarrow s_n$ is stopping.

At the abstract and symbolic levels a trace is *stopping* if it is formed only of attacker steps ($\rightarrow_a$) between states that are all stuck with respect to normal steps ($\not\rightarrow_n$). This definition expresses the fact that the attacker can take steps even after a violation has occurred and the machine has halted with respect to normal steps. At the concrete level the attacker can even take steps while the machine is halting; this is discussed together with the concrete machine for CFI.

Abstract CFI Machine The abstract machine has CFI by construction. It has separate instruction and data memories (*im* and *dm*); the instruction memory is fixed (NWC), and all executed instructions are fetched from this memory (NXD):

$$\frac{\begin{array}{c}\mathit{im}[\mathit{pc}] = i \quad \mathit{decode}\ i = \mathsf{Store}\ r_p\ r_s \quad \mathit{reg}[r_p] = p \\ \mathit{reg}[r_s] = w \quad \mathit{dm}' = \mathit{dm}[p{\leftarrow}w]\end{array}}{(\mathit{im}, \mathit{dm}, \mathit{reg}, \mathit{pc}, \mathsf{true}) \rightarrow_n (\mathit{im}, \mathit{dm}', \mathit{reg}, \mathit{pc}+1, \mathsf{true})} \quad (\textsc{Store})$$

Machine states contain an additional bit *ok*. The machine executes instructions only when this bit is true; otherwise it gets stuck with respect to normal steps (the attacker can take steps at any time). Indirect jumps are checked against the allowed set J; if the control flow is invalid the jump is taken but the violation is recorded by setting *ok* to false so that the machine will now be stuck with respect to normal steps. This behavior is designed to match rule-based enforcement at lower levels, thus simplifying the proofs (we can prove a 1-backward *SA*-simulation instead of a $\{0, 1\}$ one).

$$\frac{\begin{array}{c}\mathit{im}[\mathit{pc}] = i \quad \mathit{decode}\ i = \mathsf{Jal}\ r \quad \mathit{reg}[r] = \mathit{pc}' \\ \mathit{reg}' = \mathit{reg}[\mathsf{r_a}{\leftarrow}\mathit{pc}+1] \quad \mathit{ok} = (\mathit{pc}, \mathit{pc}') \in J\end{array}}{(\mathit{im}, \mathit{dm}, \mathit{reg}, \mathit{pc}, \mathsf{true}) \rightarrow_n (\mathit{im}, \mathit{dm}, \mathit{reg}', \mathit{pc}', \mathit{ok})} \quad (\textsc{Jal})$$

While the CFI micro-policy does not provide any monitor services itself, the abstract machine fully exposes ("paravirtualizes") the lower-level monitor service mechanism—that is, the *abstract* machine can be instantiated with an arbitrary set of monitor services.

$$\frac{\begin{array}{c}\mathit{get\_service}\ \mathit{pc} = (f, t_i) \\ f(\mathit{im}, \mathit{dm}, \mathit{reg}, \mathit{pc}, \mathsf{true}) = (\mathit{im}, \mathit{dm}', \mathit{reg}', \mathit{pc}', \mathsf{true})\end{array}}{(\mathit{im}, \mathit{dm}, \mathit{reg}, \mathit{pc}, \mathsf{true}) \rightarrow_n (\mathit{im}, \mathit{dm}', \mathit{reg}', \mathit{pc}', \mathsf{true})} \quad (\textsc{Svc})$$

As in all other step rules, we proceed only when the *ok* bit is true, which prevents monitor service calls outside the allowed CFG.

The step rules above capture the intuition of a machine that has CFI by construction. With the exception of the rule for Load, the remaining rules are straightforward; we show just the ones for Load and Bnz:

$$\frac{\begin{array}{c}\mathit{im}[\mathit{pc}] = i \quad \mathit{decode}\ i = \mathsf{Load}\ r_p\ r_s \\ \mathit{reg}[r_p]{=}w_p \quad \mathit{im}[w_p]{=}w \vee \mathit{dm}[w_p]{=}w \\ \mathit{reg}' = \mathit{reg}[r_s{\leftarrow}w]\end{array}}{(\mathit{im}, \mathit{dm}, \mathit{reg}, \mathit{pc}, \mathsf{true}) \rightarrow_n (\mathit{im}, \mathit{dm}, \mathit{reg}', \mathit{pc}+1, \mathsf{true})} \quad (\textsc{Load})$$

$$\frac{\begin{array}{c}\mathit{mem}[\mathit{pc}] = i \quad \mathit{decode}\ i = \mathsf{Bnz}\ r\ n \quad \mathit{reg}[r]{=}w \\ \mathit{pc}' \leftarrow \mathit{if}\ w = 0\ \mathit{then}\ \mathit{pc}{+}1\ \mathit{else}\ \mathit{pc}{+}n\end{array}}{(\mathit{im}, \mathit{dm}, \mathit{reg}, \mathit{pc}, \mathsf{true}) \rightarrow_n (\mathit{im}, \mathit{dm}, \mathit{reg}', \mathit{pc}', \mathsf{true})} \quad (\textsc{Bnz})$$

Notice that the Load rule allows loading a word from either the instruction or the data memory, capturing the intuition that instructions can be loaded as data. The disjointness of the two memories (and thus the fact that the rule is deterministic) is guaranteed by the simulation relation between the symbolic and the abstract machine. The step rule for Bnz demonstrates the fact that we only check indirect jumps, not direct ones, for control-flow violations.

Proving CFI for this abstract machine is easy. We capture the attacker's capabilities by the following relation:

$$\frac{\mathit{dom}\ \mathit{dm} = \mathit{dom}\ \mathit{dm}' \quad \mathit{dom}\ \mathit{reg} = \mathit{dom}\ \mathit{reg}'}{(\mathit{im}, \mathit{dm}, \mathit{reg}, \mathit{pc}, \mathit{ok}) \rightarrow_a^A (\mathit{im}, \mathit{dm}', \mathit{reg}', \mathit{pc}, \mathit{ok})}$$

This allows the attacker to arbitrarily change the data memory and registers at any time. Finally, the only requirement on initial states is that the *ok* bit starts out true.

Theorem A.3 (Abstract CFI). This abstract machine has CFI.

Symbolic CFI Machine The symbolic micro-policy for CFI was described in §6. For completeness, we present the rest of the symbolic rules. The case for Jump is identical to Jal. For the other operators (besides Jump, Jal or Store), we again have one rule to deal with the case of a jump target...

$$\frac{(src, dst) \in J}{op : (\mathsf{Code}\ src, \mathsf{Code}\ dst, -, -, -) \to (\mathsf{Code}\ \bot, -)}$$

... and a different one when execution did not take a jump:

$$\frac{}{op : (\mathsf{Code}\ \bot, \mathsf{Code}\ \_\ , -, -, -) \to (\mathsf{Code}\ \bot, -)}$$

We capture the symbolic-level attacker by the relation

$$\frac{mem \to_a^S mem' \qquad reg \to_a^S reg'}{(mem, reg, w_{pc}@t_{pc}) \to_a^S (mem', reg', w_{pc}@t_{pc})}$$

where the relation on memories and registers is the pointwise extension of the following inductive relation on atoms:

$$w_1@\mathsf{Data} \to_a^S w_2@\mathsf{Data} \qquad w@(\mathsf{Code}\ id) \to_a^S w@(\mathsf{Code}\ id)$$

This allows attackers to change words tagged Data but not words tagged Code and not the tags themselves.

Two properties are invariant under execution: all words in memory tagged Code *addr* are indeed located at address *addr*, and all sources and destinations in J are tagged Code *addr*. A symbolic machine state is *initial* if it satisfies these invariants and the pc is tagged Data (no jump in progress).

Concrete Machine To obtain a useful result about CFI on the concrete machine, it is not enough to simply instantiate the generic refinement result from §9, as we do for other policies; we must first define concrete versions of each concept used in the statement of the CFI property. The concrete attacker is only allowed to take steps when the machine is in user mode. It can change memory and registers but not the contents of the cache, the pc, or the fpc.

$$\frac{mem \to_a^C mem' \qquad reg \to_a^C reg'}{(mem, reg, cache, pc, fpc) \to_a^C (mem', reg', cache, pc, fpc)}$$

The attacker relation for memories and registers directly extends the symbolic one to the additional low-level tags

$$\frac{w_1@ut_1 \to_a^S w_2@ut_2 \qquad dec_{\mathsf{R}}(m, ct_i) = ut_i}{w_1@ct_1 \to_a^C w_2@ct_2}$$

and similarly for dec_{M}. This allows the concrete attacker to change atoms tagged User *ut* for some symbolic tag *ut* under the same conditions as at the symbolic level, but prevents it from changing any other atoms (in particular monitor code, data, and registers) or any tags. This attacker model relies on the correctness of the monitor self-protection mechanism from §9.

The initial states at the concrete level are defined as the image under $\approx_I^{CS}$ of symbolic initial states that additionally satisfy our symbolic invariants. This ensures that concrete initial states satisfy both the generic low-level conditions from §9 (I_w) and that they respect the symbolic invariants.

A concrete trace is stopping if it has a (possibly empty) prefix formed only of attacker steps between user states that are all stuck with respect to user steps, followed by a (possibly empty) suffix of monitor states. This captures either immediately getting stuck with respect to normal steps or missing in the rule cache, faulting into the monitor, and eventually halting without returning from monitor mode. This definition also deals with the fact that we allow the attacker to take steps even after a violation has occurred but before the machine is halted (right before the fault into the miss handler).

The *cfg* function is defined so that in monitor mode all control flows are allowed. We assume that monitor code is correct, so we do not need to enforce CFI there.

Formal Results We prove CFI for the concrete machine running a CFI monitor by transporting CFI from the abstract machine to the symbolic and then to the concrete one using a general CFI-preservation result.

Theorem A.4 (CFI Preservation). Given a high-level machine $M^H = (State^H, initial^H, \to_n^H, \to_a^H, cfg^H, stopping^H)$, a low-level machine $M^L = (State^L, initial^L, \to_n^L, \to_a^L, cfg^L, stopping^L)$, a simulation relation between states $s^L \sim s^H$, a predicate *checked* $s_1^L\ s_2^L$ indicating which low-level steps need to be checked for CFI, and a set of allowed indirect jumps J, if M^H has CFI, then M^L also has CFI under the following additional assumptions:

A1. 1-backward simulation with respect to $\sim$ for checked steps in $\to_n^L$;

A2. $\{0, 1\}$-backward simulation with respect to $\sim$ for unchecked steps in $\to_n^L$;

A3. 1-backward simulation with respect to $\sim$ for attacker steps ($\to_a^L$);

A4. if *initial* s^L, then $\exists s^H$ so that *initial* s^H and $s^L \sim s^H$;

A5. if $s_1^L \sim s_1^H$, $s_2^L \sim s_2^H$, *checked* $s_1^L\ s_2^L$, then $cfg^H\ J = cfg^L\ J$;

A6. if $s_1^L \sim s_1^H$, $s_1^L \to_n s_2^L$ and $\neg checked\ s_1^L\ s_2^L$, then $(s_1^L, s_2^L) \in cfg^L\ J$;

A7. if $s_1^L \sim s_1^H$, $(s_1^H, s_2^H) \notin cfg^H\ J$, and $s_1^H \to_n s_2^H$ then $\neg(s_1^H \to_a s_2^H)$;

A8. and if $s_1^L \sim s_1^H$, *checked* $s_1^L\ s_2^L$, $(s_1^H, s_2^H) \notin cfg^H\ J$, and $s_1^H \to s_2^H$, and the trace $s_2^L :: t^L$ refines the trace $s_2^H :: t^H$ with respect to simulation relation $\sim$, and *stopping* $(s_2^H :: t^H)$, implies that *stopping* $(s_2^L :: t^L)$.

Assumption *A3* states that low-level attacker steps ($\to_a^L$) are simulated by corresponding high-level attacker steps, which ensures that the low-level attacker is at most as strong as the high-level one. *A4* enforces that all low-level initial states can be mapped to related high-level initial states. *A5* ensures that for checked low-level steps the two *cfg* functions completely agree. *A6* states that all unchecked low-level steps are allowed by cfg^L (e.g., monitor steps are allowed by the CFG). *A7* states that CFG violations are not simultaneously attacker steps. Finally, *A8* ensures that a high-level stopping trace can only be mapped by the simulation relation to a stopping low-level trace.

Theorem A.5 (CFI Concrete). The concrete machine running a CFI monitor satisfying the assumptions in §9 has CFI.

B Additional Symbolic Machine Rules

$$\frac{\begin{array}{c} \mathit{mem}[w_{pc}] = i@t_i \quad \mathit{decode}\ i = \mathsf{Nop} \\ \mathit{transfer}(\mathsf{Nop}, t_{pc}, t_i, -, -, -) = (t'_{pc}, -) \end{array}}{(\mathit{mem}, \mathit{reg}, w_{pc}@t_{pc}, \mathit{extra}) \to (\mathit{mem}, \mathit{reg}, (w_{pc}{+}1)@t'_{pc}, \mathit{extra})} \quad \text{(\textsc{Nop})}$$

$$\frac{\begin{array}{c} \mathit{mem}[w_{pc}] = i@t_i \quad \mathit{decode}\ i = \mathsf{Const}\ w\ r \\ \mathit{reg}[r]=\_\ @t \\ \mathit{transfer}(\mathsf{Const}, t_{pc}, t_i, t, -, -) = (t'_{pc}, t') \\ \mathit{reg}' = \mathit{reg}[r \leftarrow w@t'] \end{array}}{(\mathit{mem}, \mathit{reg}, w_{pc}@t_{pc}, \mathit{extra}) \to (\mathit{mem}, \mathit{reg}', (w_{pc}{+}1)@t'_{pc}, \mathit{extra})} \quad \text{(\textsc{Const})}$$

$$\frac{\begin{array}{c} \mathit{mem}[w_{pc}] = i@t_i \quad \mathit{decode}\ i = \mathsf{Mov}\ r\ r_d \\ \mathit{reg}[r]=w@t \quad \mathit{reg}[r_d]=\_\ @t_d \\ \mathit{transfer}(\mathsf{Mov}, t_{pc}, t_i, t, t_d, -) = (t'_{pc}, t'_d) \\ \mathit{reg}' = \mathit{reg}[r_d \leftarrow w@t'_d] \end{array}}{(\mathit{mem}, \mathit{reg}, w_{pc}@t_{pc}, \mathit{extra}) \to (\mathit{mem}, \mathit{reg}', (w_{pc}{+}1)@t'_{pc}, \mathit{extra})} \quad \text{(\textsc{Mov})}$$

$$\frac{\begin{array}{c} \mathit{mem}[w_{pc}] = i@t_i \quad \mathit{decode}\ i = \mathsf{Load}\ r\ r_d \\ \mathit{reg}[r]=w@t \quad \mathit{mem}[w] = w'@t' \quad \mathit{reg}[r_d]=\_\ @t_d \\ \mathit{transfer}(\mathsf{Load}, t_{pc}, t_i, t, t', t_d) = (t'_{pc}, t'_d) \\ \mathit{reg}' = \mathit{reg}[r_d \leftarrow w'@t'_d] \end{array}}{(\mathit{mem}, \mathit{reg}, w_{pc}@t_{pc}, \mathit{extra}) \to (\mathit{mem}, \mathit{reg}', (w_{pc}{+}1)@t'_{pc}, \mathit{extra})} \quad \text{(\textsc{Load})}$$

$$\frac{\begin{array}{c} \mathit{mem}[w_{pc}] = i@t_i \quad \mathit{decode}\ i = \mathsf{Jump}\ r \\ \mathit{reg}[r] = w'_{pc}@t \\ \mathit{transfer}(\mathsf{Jump}, t_{pc}, t_i, t, -, -) = (t'_{pc}, -) \end{array}}{(\mathit{mem}, \mathit{reg}, w_{pc}@t_{pc}, \mathit{extra}) \to (\mathit{mem}, \mathit{reg}', w'_{pc}@t'_{pc}, \mathit{extra})} \quad \text{(\textsc{Jump})}$$

$$\frac{\begin{array}{c} \mathit{mem}[w_{pc}] = i@t_i \quad \mathit{decode}\ i = \mathsf{Bnz}\ r\ n \\ \mathit{reg}[r] = w@t \\ \mathit{transfer}(\mathsf{Bnz}, t_{pc}, t_i, t, -, -) = (t'_{pc}, -) \\ w'_{pc} = \text{if } w = 0 \text{ then } w_{pc} + 1 \text{ else } w_{pc} + n \end{array}}{(\mathit{mem}, \mathit{reg}, w_{pc}@t_{pc}, \mathit{extra}) \to (\mathit{mem}, \mathit{reg}', w'_{pc}@t'_{pc}, \mathit{extra})} \quad \text{(\textsc{Bnz})}$$

$$\frac{\begin{array}{c} \mathit{mem}[w_{pc}] = i@t_i \quad \mathit{decode}\ i = \mathsf{Jal}\ r \\ \mathit{reg}[r] = w'_{pc}@t_1 \quad \mathit{reg}[\mathsf{ra}] = \_\ @t_{ra} \\ \mathit{transfer}(\mathsf{Jal}, t_{pc}, t_i, t_1, t_{ra}, -) = (t'_{pc}, t'_{ra}) \\ \mathit{reg}' = \mathit{reg}[\mathsf{ra} \leftarrow (w_{pc}{+}1)@t'_{ra}] \end{array}}{(\mathit{mem}, \mathit{reg}, w_{pc}@t_{pc}, \mathit{extra}) \to (\mathit{mem}, \mathit{reg}', w'_{pc}@t'_{pc}, \mathit{extra})} \quad \text{(\textsc{Jal})}$$

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *TISSEC*, 13(1), 2009.

[2] M. Abadi and G. D. Plotkin. On protection by layout randomization. *ACM TISSEC*, 15(2):8, 2012.

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

[4] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. *POPL*. 2014.

[5] M. Budiu, Ú. Erlingsson, and M. Abadi. Architectural support for software-based protection. *ASID*. 2006.

[6] A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. *ICFP*. 2013.

[7] J. A. Clause, I. Doudalis, A. Orso, and M. Prvulovic. Effective memory protection using dynamic tainting. *ASE*. 2007.

[8] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. *IEEE S&P*, 2014.

[9] L. Davi, P. Koeberl, and A.-R. Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. *DAC*. 2014.

[10] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. *USENIX Security*, 2014.

[11] U. Dhawan, C. Hriţcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing. ASPLOS, 2015.

[12] U. Dhawan, A. Kwon, E. Kadric, C. Hriţcu, B. C. Pierce, J. M. Smith, A. DeHon, G. Malecha, G. Morrisett, T. F. Knight, Jr., A. Sutherland, T. Hawkins, A. Zyxnfryx, D. Wittenberg, P. Trei, S. Ray, and G. Sullivan. Hardware support for safety interlocks and introspection. *AHNS*, 2012.

[13] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. *IEEE S&P*. 2000.

[14] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. *IEEE S&P*, 2014.

[15] K. W. Hamlen, J. G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *TOPLAS*, 28(1):175–205, 2006.

[16] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. *IEEE S&P*. 2013.

[17] J. B. Jensen, N. Benton, and A. Kennedy. High-level separation logic for low-level code. *POPL*. 2013.

[18] J. Kroll, G. Stewart, and A. Appel. Portable software fault isolation. *CSF*. 2014.

[19] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. *CCS*. 2013.

[20] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR*, 41(1):1–31, 2008.

[21] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *IJISS*, 4(1-2):2–16, 2005.

[22] A. J. Mashtizadeh, A. Bittau, D. Maziéres, and D. Boneh. Cryptographically enforced control flow integrity. arXiv:1408.145.

[23] J. H. Morris, Jr. Protection in programming languages. *CACM*, 16(1):15–21, 1973.

[24] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: better, faster, stronger SFI for the x86. *PLDI*. 2012.

[25] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Hardware-Enforced Comprehensive Memory Safety. *IEEE Micro*, 33(3):38–47, 2013.

[26] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. *PLDI*. 2009.

[27] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. *ISMM*. 2010.

[28] B. Niu and G. Tan. Modular control-flow integrity. *PLDI*. 2014.

[29] F. B. Schneider. Enforceable security policies. *TISSEC*, 3(1):30–50, 2000.

[30] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *ASPLOS*, 2004.

[31] E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. *TCS*, 375(1-3):169–192, 2007.

[32] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SOSP*, 1993.

[33] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: a sandbox for portable, untrusted x86 native code. *CACM*, 53(1):91–99, 2010.

[34] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity & Randomization for Binary Executables. *IEEE S&P*, 2013.

[35] L. Zhao, G. Li, B. D. Sutter, and J. Regehr. ARMor: fully verified software fault isolation. *EMSOFT*. 2011.