Micro-Policies

A Framework for Verified, Hardware-Assisted Security Monitors

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

[1]University of Pennsylvania [2]INRIA Paris [3]NTU Athens [4]Portland State University

Abstract

A wide range of low-level security policies can be expressed as rules on metadata tags and enforced using a combination of a hardware rule cache and a software monitor. We present a generic framework for defining tag-based reference monitors (or *micro-policies*) on a simple tagged RISC processor, formalize this framework in Coq, and use it to define and verify micro-policies for dynamic sealing, control-flow integrity, memory safety, and compartmentalization; 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 hardware running a correctly implemented monitor embodies a high-level specification characterizing a useful security property.

1. Introduction

Today's computer systems are distressingly insecure, but many of their vulnerabilities can be avoided if low-level code is constrained to obey well-known safety and security properties such as type and memory safety, control flow integrity (CFI), "sealing" of sensitive information, and strong separation into least-privilege compartments. Ideally, such properties should be enforced statically, but for low-level code it is often more practical to detect violations dynamically using a *reference monitor* [3, 11, 26]. Reference monitors are sometimes implemented in software, but this can significantly degrade performance. Hardware implementation is thus an attractive alternative, especially in an era of cheap transistors.

Many designs for hardware monitors have been proposed, at first focusing on enforcing single hard-wired security policies and later evolving toward more programmable mechanisms allowing quicker adaptation to a shifting attack landscape. Our work is based on a flexible hardware/software mechanism, called the *Programmable Unit for Metadata Processing (PUMP)* [10], that can efficiently implement a wide range of different policies, singly or in combination.

The PUMP is designed as an add-on to a conventional RISC processor. Every word of data on the machine is associated with a piece of metadata (a full machine word) called a *tag*. The interpretation of tags is left entirely to software: the hardware simply propagates tags from operands to results according to software-defined *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* that, 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 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. The software components that can be changed to enforce a particular property are collectively called a *micro-policy*, or sometimes just *policy*. (These policies are "micro" in the sense that they enforce low-level security invariants such as spatio-temporal memory safety or CFI rather than user-level properties like "My web browser sends my bank details only to my bank"). This mechanism achieves the flexibility and adaptability of software with performance comparable to dedicated hardware. Hardware simulations suggest that the average runtime overhead for fairly complex micro-policies like memory safety and fine-grained CFI can be reduced to around 8–13% [14].

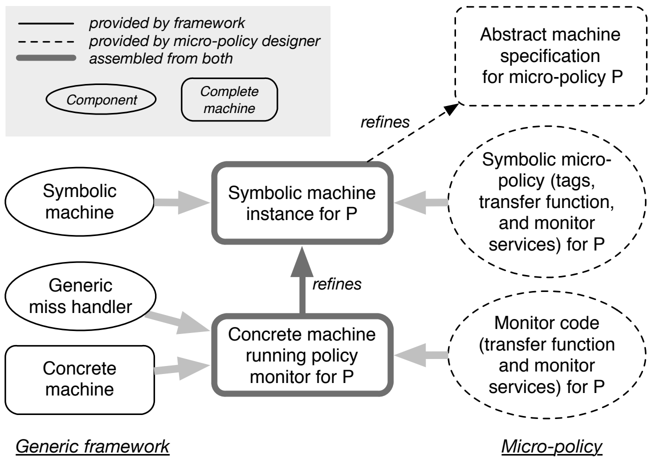
provided by framework
provided by micro-policy designer
assembled from both
Component
Complete machine
Abstract machine specification for micro-policy P
refines
Symbolic machine
Symbolic machine instance for P
Symbolic micro-policy (tags, transfer function, and monitor services) for P
refines
Generic miss handler
Concrete machine running policy monitor for P
Concrete machine
Monitor code (transfer function and monitor services) for P
Generic framework
Micro-policy

Figure 1. Overview

However, encoding desired properties as micro-policies that perform well in practice can be nontrivial. While a higher-level rule-based programming model helps in writing down micro-policies [14], it is still easy to get them wrong: at the end of the day, only formal verification can give complete confidence. This observation motivated a recent formal definition and correctness proof for PUMP-like tagging hardware running a specific monitor for an information-flow control (IFC) policy [4].

In this paper, we offer a *generic framework* for defining, implementing, and formally reasoning about micro-policies, and we use it to study a diverse collection of micro-policies. The framework is entirely implemented (and checked) in Coq [7].

Our investigation targets an idealized RISC ISA with a minimalist instruction set, extended with a PUMP—i.e., with word-size tags on words in memory, registers, and the *pc*, a rule cache, and a mechanism for trapping to software on cache misses. We call this hardware platform the *concrete machine* (see Figure 1).

The heart of our framework is a *symbolic machine* that serves both as a programming interface for the concrete machine—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. Unlike the concrete machine's tags, which are unstructured words, a symbolic tag might contain a list of principals for an IFC micro-policy, as in [4]. The designer also provides a *transfer function* that monitors program execution and determines how tags are propagated in each step. Finally, the micro-policy includes a set of *monitor services*, represented as partial functions from machine states to machine states, that can be directly invoked by user programs to control the monitor's behavior dynamically. For example, our dynamic sealing policy uses keys as tags and includes monitor services for creating new keys, sealing, and unsealing.

Each micro-policy is implemented at the concrete level by providing machine code for the transfer function and monitor services and 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 privileged instructions. We show that we can achieve this protection using only the tagging mechanism itself, by labeling the monitor's memory and registers with a *Monitor* tag that user programs are prevented from accessing by the monitor. In order to allow the monitor to access this tag or execute privileged instructions we pre-populate the cache with a set of non-evictable *ground rules*.

To demonstrate its expressive power, we instantiate this generic framework with a diverse set of security micro-policies: (a) dynamic sealing, a linguistic mechanism for protecting data [20]; (b) control-flow integrity (CFI), preventing code-reuse attacks such as return-oriented programming [1]; (c) memory safety, preventing temporal and spatial violations for heap-allocated data; and (d) compartmentalization, sandboxing untrusted code and allowing it to be run alongside trusted code [27]. The intended behavior of each micro-policy is specified by an *abstract machine* giving a clear characterization of the micro-policy's behavior as seen by a user-level programmer. In some cases (e.g., dynamic sealing and memory safety), the abstract machine definition itself is a sufficently clear explanation of the invariants it enforces. In other cases, it is useful to prove theorems that draw out key properties. 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 model (SFI) [27].

Our main technical results are *refinements* between the concrete and abstract machines for each policy. We show that every valid concrete behavior is a valid abstract behavior—hence, the concrete machine always fail-stops on policy violations. Using the symbolic machine as an intermediate point lets us factor out many commonalities among the refinement proofs of different policies. Indeed, we give a single generic refinement proof for the concrete and symbolic machines, parameterized by a proof of correctness of the policy-specific monitor code. For CFI, we additionally use this generic refinement to transfer the CFI property [1] of the abstract machine to the concrete level via a generic CFI-preservation theorem.

Our main contributions are as follows. First, we introduce a generic framework for defining (§2–§3), implementing (§5), and formally verifying a wide range of micro-policies for a simple PUMP-enhanced RISC processor. Second, we give a generic refinement proof (parameterized by crisp hypotheses about the behavior of policy-specific components) between our concrete hardware platform + generic miss handler and the symbolic machine; along the way, we show how to use the PUMP to protect its own policy monitor code (§6). And third, using this framework, we formally verify four security micro-policies: dynamic sealing (§4), control-flow integrity (§7), memory safety (§8), and compartmentalization (§9). We discuss related work in §10 (and, where appropriate, in §7–§9) and future work in §11. Our Coq development is included with the additional material for this submission.

Our work generalizes and formalizes previous work on PUMP-like hardware structures and PUMP-supported micro-policies. An important inspiration is [4], which formalizes an IFC micro-policy using a special-purpose symbolic machine over a PUMP-like tagging mechanism. Compared to [4], we base our work on a more realistic RISC architecture. Moreover, we use the PUMP's tagging facilities not only to implement user-level micro-policies but also to protect the instructions and data of the policy monitor itself. Our symbolic machine generalizes the one in [4] beyond IFC and adds arbitrary monitor services; the generic refinement proof between concrete and symbolic machines is also new. A second related paper [14] studies a programming model for the PUMP architecture and experimentally evaluates runtime overhead and characteristics like cache working set sizes for a set of micro-policies including the CFI and memory safety policies we investigate here, faster unsound variations thereof, and simple policies for taint tracking and low-level types. Our work here is complementary, focusing on *formal* specification and verification of micro-policies. Moreover, the micro-policies for compartmentalization and dynamic sealing are completely new. Finally, while the informal rule format used here to present symbolic transfer functions is based on [14], our Coq development uses Coq's internal language, Gallina.

Finally, one disclaimer on scope. There are two main steps in using our framework to implement a given micro-policy: (1) defining an abstract machine that directly characterizes some property of interest, devising symbolic tags, a symbolic transfer function, and symbolic monitor services that enforce the property using the features of the symbolic machine, and proving that this symbolic machine instance refines this abstract machine; and (2) implementing the transfer function and monitor services in machine code and proving these implementations correct with respect to the symbolic versions. We have completed only (1) for the policies described in §4 and §7–§9, both to manage the size of the verification effort and because this is where the main conceptual novelty of our framework lies. Our formal results assume the existence of correct monitor implementations as hypotheses. These implementations are generally straightforward, and verification of low-level code is a well-studied [4, 5, 15], if not yet fully settled, area.

2. Basic Machine

We begin by introducing a simplified RISC instruction set architecture, which forms a common core for all the machines throughout the paper. The machine has a fixed number of general-purpose registers plus a *pc* register. It offers a small collection of familiar instructions

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

where $\oplus \in \{+, -, \times, =, \leq, \mathit{and}, \mathit{or}, \mathit{xor}, \mathit{shru}, \mathit{shl}\}$ (*shru* = shift right unsigned, *shl* = shift left). *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 fixed-size word.

A *basic machine state* $(\mathit{mem}, \mathit{reg}, \mathit{pc})$ consists of a word-addressable memory (a partial function from words to words), a register file (a function from register names to words), and a *pc* (a word). Trying to address outside of the valid memory will halt the machine. The step rules of the basic machine are written like this:

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

The partial function *decode* maps binary words to the *instr* datatype defined above. The notation $\mathit{mem}[w_1{\leftarrow}w_2]$ is defined only when *mem*$[w_1]$ is; it then yields a partial function that maps w_1 to w_2 and behaves like *mem* on all other arguments.

Subroutine calls are implemented by the *Jal* instruction, which saves the return address to a general-purpose register *ra*. Returns from subroutines are just *Jump*s through the *ra* register.

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

3. Symbolic Machine

The symbolic machine is the key component of our framework, easing the micro-policy designer's work in several ways. First, it abstracts away from the hardware rule cache. Second, it allows policies to be expressed and reasoned about not only as chunks of machine code, but also as mathematical functions written in Gallina. This allows formal reasoning about security properties of micro-policies at an appropriate level of abstraction. And third, it comes with a generic proof of refinement between symbolic and concrete machine instances; all that needs to be supplied are correctness proofs for the individual code sequences with respect to their mathematical specifications.

The symbolic machine has the same general organization as the basic machine from §2. It is abstracted on several parameters: (1) A set T of *symbolic tags*, which are used to label words in memory, register contents, and the *pc*. (2) A partial function *transfer*, mapping an opcode and a 5-tuple of tags to a pair of tags, which is invoked on each step of the machine to check whether the current configuration is allowed by the current micro-policy and, if so, to calculate the tags on the next *pc* and the instruction's result (if any). (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 used to identify the service. (4) A type *EX* of *extra machine state* that can be used by the monitor services, plus an initial value. These parameters collectively form a *symbolic micro-policy*.

The tagged memory contents, register contents, and *pc* are called *symbolic atoms* and written $w@t$, where w (the "payload") is a machine word and t is a *symbolic tag*. The payload and tag parts are not separately addressable—they are treated as an indivisible unit except within the policy-specific transfer function and monitor services. Symbolic states, written (*mem*, *reg*, *pc*, *extra*), consist of a memory, registers, a *pc*, and a piece of extra state. The symbolic stepping rules call the transfer function to decide whether the step is allowed by the micro-policy; if not, the machine is stuck. (For simplicity, we assume that policy violations are fatal; various error recovery mechanisms could also be used.) The *transfer* function is passed a tuple containing the current opcode and the tags on the current *pc*, current instruction, and up to three inputs (depending on the opcode). It returns the new *pc* and result tags. For example:

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

Passing t_{old}, the tag on the current contents of the target memory cell, allows the transfer function to see what kind of data is being overwritten. This is used by the monitor protection policy in §6 to protect monitor memory from damage by user code.

In addition, there is one step rule for all monitor services, which applies when the *pc* is at a service entry point.

$$\frac{\begin{array}{c} get\_service\ pc = (f, t_i) \\ transfer(Service, t_{pc}, t_i, -, -, -) = (-, -) \\ f\ (mem, reg, pc@t_{pc}, extra) = (mem', reg', pc'@t'_{pc}, extra') \end{array}}{(mem, reg, pc@t_{pc}, extra) \to (mem', reg', pc'@t'_{pc}, extra')} \quad (\textsc{Svc})$$

The call to *transfer* checks that this particular service is permitted from the current machine state. The last three inputs to *transfer* are set to a fixed dummy value "$-$" , and the outputs are not used: we only care whether the operation is allowed or not.

4. Sealing Micro-Policy

For a first example, let us build a simple micro-policy for dynamic sealing [20], a linguistic mechanism related to perfect symmetric encryption. Abstractly, 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.

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 by 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 and Gallina implementations of the three monitor services. We prove that the symbolic sealing machine *refines* the abstract one. Finally, we build machine-code realizations of the symbolic transfer function and the three monitor services. A generic policy monitor (§6) wraps these four code blocks with boilerplate for interacting with the hardware trap facility and rule cache, and a generic refinement proof establishes (assuming the correctness of the micro-policy-specific code blocks with respect to their symbolic versions) that the concrete hardware (§5) running this policy monitor behaves the same as the sealing instance of the symbolic machine, and hence also the abstract sealing machine. We discuss just the first and second steps here.

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

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

The operations of generating keys, sealing, and unsealing are provided by monitor service routines located at specific addresses (*mkkey_addr*, *seal_addr*, and *unseal_addr*), 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 their arguments (if any) in general-purpose registers r_{arg1} and r_{arg2} and return their result in a general-purpose register r_{ret}. The step relation includes a rule for each service that applies when the *pc* is at the corresponding address (the omitted rule for sealing is analogous):

$$\frac{mkkey\_f\ ks{=}k \qquad reg'{=}reg[r_{ret}{\leftarrow}k] \qquad reg[ra]{=}pc'}{(mem, reg, mkkey\_addr, ks) \to (mem, reg', pc', k{::}ks)} \quad (\textsc{MkKey})$$

$$\frac{\begin{array}{cc} \mathit{reg}[r_{\mathit{arg1}}] = \{w\}_k & \mathit{reg}[r_{\mathit{arg2}}] = k \\ \mathit{reg}' = \mathit{reg}[r_{\mathit{ret}} \leftarrow w] & \mathit{reg}[\mathit{ra}] = \mathit{pc}' \end{array}}{(\mathit{mem}, \mathit{reg}, \mathit{unseal\_addr}, \mathit{ks}) \to (\mathit{mem}, \mathit{reg}', \mathit{pc}', \mathit{ks})} \quad (\textsc{Unseal})$$

The *pc* is restored from register *ra* after each of these steps. To invoke a monitor service, a user program simply performs a *Jal* to the corresponding address, which sets *ra* appropriately. Invoking services this way means that we can run exactly the same user code on this abstract machine as we do on the concrete machine in §5.

Symbolic Sealing Machine The abstract machine constitutes a specification of the sealing micro-policy. We next devise a *symbolic micro-policy* that implements this specification in terms of tags by representing each abstract value as a symbolic atom $w@t$, where symbolic tags t have the form *Data*, *Key* k, or *Sealed* k. Keys are represented as a dummy payload word tagged with a *symbolic key* k drawn from an ordered finite set *SK*. A word w tagged *Sealed* k represents the sealing of w under key k. The extra state type *EX* is just *SK*, and the extra state consists of a monotonic counter storing the next key. The initial extra state is the minimum key.

Except for the rules for monitor services, which are implemented directly as described below, all side conditions on the step relation involving the shapes of values are captured by the transfer function of the symbolic machine. In our formal development, transfer functions are written in Gallina; but for readability we will present examples as a collection of *symbolic rules* [14]

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

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

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

Similarly, the *Jal* rule ensures that the target register (OP_1) is tagged *Data* and tags the *ra* register (R') as *Data*:

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

The *get_service* function is $\{\mathit{mkkey\_addr} \mapsto (\mathit{mkkey}, \mathit{Data}), \mathit{seal\_addr} \mapsto (\mathit{seal}, \mathit{Data}), \mathit{unseal\_addr} \mapsto (\mathit{unseal}, \mathit{Data})\}$, where *mkkey* and *unseal* (*seal* is similar) are defined by:

$$\frac{\begin{array}{c} \mathit{reg}' = \mathit{reg}[r_{\mathit{ret}} \leftarrow \mathit{max\_word}@\mathit{Key}\ \mathit{nk}] \\ \mathit{nk} \neq \mathit{max\_key} \qquad \mathit{nk}' = \mathit{nk} + 1 \end{array}}{\mathit{mkkey}\ (\mathit{mem}, \mathit{reg}, \mathit{pc}, \mathit{nk})\ (\mathit{mem}, \mathit{reg}', \mathit{pc}, \mathit{nk}')} \quad (\textsc{MkKey})$$

$$\frac{\begin{array}{c} \mathit{reg}[r_{\mathit{arg1}}] = w@\mathit{Sealed}\ k \qquad \mathit{reg}[r_{\mathit{arg2}}] = w'@\mathit{Key}\ k \\ \mathit{reg}' = \mathit{reg}[r_{\mathit{ret}} \leftarrow w@\mathit{Data}] \end{array}}{\mathit{unseal}\ (\mathit{mem}, \mathit{reg}, \mathit{pc}, \mathit{nk})\ (\mathit{mem}, \mathit{reg}', \mathit{pc}, \mathit{nk})} \quad (\textsc{Unseal})$$

The constant *max_key* stands for the largest representable key, while *max_word* is used as a dummy payload. Note that *mkkey* can fail if all keys have been used up. By contrast, the abstract sealing machine uses an infinite set of keys, so it will never fail for this reason. This discrepancy causes no problems for our backward refinement proof, 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 always 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 not exposing the details of key allocation at the abstract level.

Refinement We formalize the connection between the abstract and symbolic sealing machines as a *backward* (from concrete to abstract) *refinement* property on traces. We state this here in general form and 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 *simulation relation* $\sim$ between low-level and high-level states if $s_1^L \sim s_1^H$ and $s_1^L \to^* s_2^L$ implies that there exists 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 refinement by showing *simulation* between individual execution steps. In the case of sealing, we can prove a strong 1*-backward simulation* theorem showing that each step of the symbolic machine is simulated by *exactly one* corresponding 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 $s_2^L \sim s_2^H$.

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

The relation $\sim_\psi^{SA}$ on states is itself defined in terms of the following relation between symbolic atoms and abstract values (plus some additional invariants described below):

$$\begin{array}{rcll} w@\mathit{Data} & \sim_\psi^{SA} & w' & = w = w' \\ w@(\mathit{Key}\ k^S) & \sim_\psi^{SA} & k^A & = \psi[k^A] = k^S \\ w@(\mathit{Sealed}\ k^S) & \sim_\psi^{SA} & \{w'\}_{k^A} & = w = w' \wedge \psi[k^A] = k^S \\ w@t & \sim_\psi^{SA} & v & = \mathit{false}\text{, otherwise} \end{array}$$

Since keys are dynamically allocated, the relation is parameterized by a partial function ψ from abstract to symbolic keys. This mapping is extended on each call to *mk_key* 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 (§8) from the high-level specification.

The simulation relation on states $\sim_\psi^{SA}$ is lifted "pointwise" from atoms, plus 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.

5. Concrete Machine

The concrete machine extends the basic machine with generic hardware for efficiently enforcing symbolic micro-policies, in the form of a *rule cache* and a software *miss handler*. Its memory, registers, and *pc* hold *concrete atoms* of the form $w@t$, where the *concrete tag* t is simply a machine word. The instruction set includes four additional instructions for use by low-level monitor code:

$$\mathit{AddRule} \mid \mathit{JumpEpc} \mid \mathit{GetTag}\ r_s\ r_d \mid \mathit{PutTag}\ r_s\ r_{\mathit{tag}}\ r_d$$

AddRule, described in detail below, inserts a new rule into the cache. *JumpEpc* jumps to the address in *epc*, 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 $t@\mathit{Monitor}$ 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 §6 ensures that these instructions can only be executed by monitor code.

Concrete states have the form $(\mathit{mem}, \mathit{reg}, \mathit{pc}, \mathit{epc}, \mathit{cache})$, where *cache* is a set of *concrete rules*, each of the form $(\mathit{iv}, \mathit{ov})$. Intuitively, each concrete rule encodes a single tuple in the graph of the transfer function. 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 result and the new *pc*. 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 $\mathit{cache} \vdash \mathit{iv} \mapsto \mathit{ov}$ below), 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* is saved in *epc*, and control is transferred to a fixed address where the miss handler is loaded (*trapaddr*).

Each rule in the step relation is split into 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*:

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

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

Addresses 0 to 5 are used 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.

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

Here $\uplus$ denotes map update, overwriting any previous value for $(\mathit{opcode}, t_1, t_2, t_3, t_4, t_5)$. We do not model cache eviction.

A final technical 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. We use them in the next section to ensure that the set of ground rules is finite.

6. Concrete Policy Monitor

It is not obvious that the simple mechanisms of the concrete machine are rich enough to implement the high-level model proposed by the symbolic machine. To demonstrate that they are, we present a generic framework for implementing micro-policies on the concrete machine and proving them correct in the sense of enjoying a two-way refinement with the corresponding symbolic machine instance. To use the framework, a policy designer must provide: (a) an encoding function *enc* that converts symbolic tags to concrete words (bit vectors)[1]; (b) a way of representing the policy-specific extra state in the memory of the concrete machine; (c) machine code for computing the micro-policy's transfer function on encoded tags; (d) machine code for implementing each monitor service; (e) correctness proofs for the above with respect to the corresponding symbolic components. We first present the framework, then show how to instantiate it to obtain a concrete implementation and a refinement proof for the sealing micro-policy of §4 (modulo the correctness assumptions for machine code, which we do not prove).

Tagging Scheme At the symbolic level, the interaction between user code and the tagging infrastructure is limited. In particular, it is impossible by construction to circumvent micro-policy enforcement. At the concrete level, however, some form of isolation for monitor memory is required. Moreover, on the symbolic machine the privileged instructions we added in §5 do not exist; on the concrete machine we need to ensure that only monitor code can execute them—otherwise, malicious or compromised code could, for instance, add arbitrary rules to the cache. We use the PUMP to simultaneously enforce the symbolic micro-policy, protect the monitor, and prevent user code from executing privileged instructions. Conceptually, our tags have the form *User st*, *Entry st*, or *Monitor*. They are represented concretely using two low-order bits to distinguish between the different cases: $\mathit{User}\ st \to \mathit{enc}\ st \cdot 0 \cdot 1$, $\mathit{Entry}\ st \to \mathit{enc}\ st \cdot 1 \cdot 0$, and $\mathit{Monitor} \to \bar{0}$, where $\cdot$ is bitstring concatenation and $\bar{0}$ is the all-zero bitstring. The concrete tag *User st* is used to label a user-level atom with (binary-encoded) symbolic tag *st*. *Monitor* is used to tag monitor memory as well as a few reserved monitor registers; at the symbolic level, these are undefined. The *pc* is also tagged *User* or *Monitor* to indicate which kind of code is running; the tag flips on cache misses (via Miss rules), returns from misses (via *JumpEpc*), returns from service routines (via *Jump ra*), and when an instruction tagged *Entry* is executed while the *pc* is tagged *User*, as explained below.

Miss Handler The miss handler has two jobs: enforcing the symbolic micro-policy and protecting the monitor from the user. Accordingly, we split the miss handler into a policy-specific transfer function implementation (provided by the policy designer) and a generic monitor protection wrapper. The generic wrapper reads the *iv* out of memory into a specific set of monitor registers, checks that the low-order bits of all the tags represent *User*, strips them off, runs the transfer function (which should halt the machine if *iv* violates the policy), re-wraps the tags in the resulting *ov*, stores it into the appropriate memory slots, calls *AddRule* to install it, and restarts the instruction that trapped by jumping through the *epc* register.

If the generic wrapper detects that user code is trying to manipulate or overwrite private monitor data, it halts the machine without invoking the transfer function. Its ability to do this relies crucially on the fact that tags from the "old contents" of registers and memory are included in the *iv*. The only exception is when the current instruction is tagged *Entry*, which happens when invoking monitor services. In this case, the wrapper does invoke the policy transfer function, after replacing the current opcode with the special value *Service*; this allows the transfer function to decide which services can be invoked from which user states. The wrapper also halts if user code tries to execute any of the privileged instructions.

We ensure that monitor code itself never faults (which could lead to inefficiency or infinite regress) by populating the rule cache in the initial machine state with a finite set of *ground rules*, one for each opcode, saying that this opcode can be executed when the *PC* and *CI* tags in the *iv* are *Monitor* and that the next *pc* and any result of the instruction are also tagged *Monitor*. A technical detail is that we use separate don't-care and copy-through masks when running

[1] For the moment, we *assume* the existence of correct encodings for all policies but dynamic sealing. Moreover, we assume that every symbolic tag can be encoded as a word; in reality, complex tags should be encoded as word-sized *pointers* to structures in memory. We know how to relax both these assumptions. For the former we already have the infrastructure in place, while for the latter we plan to follow [4]. This should require a few days of work.

monitor code (i.e., when the *pc* tag is *Monitor*), to ensure that the monitor does not fault when coming in contact with user tags (for instance when returning back to user mode).

Refinement We formalize the relation between the symbolic and concrete machines as a *two-way refinement* (forward and backward) between their step relations.[2] We consider a symbolic machine instance for an arbitrary micro-policy, assuming the policy designer has supplied an encoding scheme for symbolic tags in terms of machine words, together with compatible implementations for the transfer function and monitor services.

We provide generic backward and forward simulation proofs. The backward simulation statement and proof are, however, more complicated, because the steps of the concrete monitor cannot be mapped to any symbolic steps. Moreover, the concrete monitor will often temporarily break both the invariants and any strong correspondence with respect to some symbolic state, and only reestablish these before returning back to user code. To address these challenges, we define a *weak simulation relation* $\sim_I^{CS}$ between concrete and symbolic states in terms of a *strong simulation relation* $\approx_I^{CS}$. We prove backward simulation with respect to $\sim_I^{CS}$ and forward simulation with respect to $\approx_I^{CS}$. Both relations are parametric in I, an implementation-specific invariant that ensures that the policy extra state is correctly represented in memory.

Definition 6.1. We say that s^C weakly simulates s^S, written $s^C \sim_I^{CS} s^S$, if either the *pc* of s^C is tagged *User* and $s^C \approx_I^{CS} s^S$, or else if the *pc* of s^C is tagged *Monitor* and there exists another state s_U^C with a *pc* tagged *User* such that $s_U^C \approx_I^{CS} s^S$ and $s_U^C \to^* s^C$ and all states in this execution have the *pc* tagged *Monitor*.

The second case handles concrete states in which the monitor executes, and for which we do not have a direct correspondence with any symbolic state, but for which we need to remember enough information from the point in the past where the monitor was invoked to be able to reestablish strong simulation once execution returns to user mode. Strong simulation $\approx_I^{CS}$ is defined as follows:

$$\frac{\begin{array}{c} \mathit{reg}^C \sim \mathit{reg}^S \qquad \mathit{mem}^C \sim \mathit{mem}^S \\ \mathit{cache\_ok}(\mathit{cache}) \qquad \mathit{services\_ok}(\mathit{mem}^C) \\ \forall a, \mathit{get\_service}\ a = (\_, t) \iff \mathit{mem}^C[a] = \_@(\mathit{Entry}\ t) \\ \mathit{mem}^C[0..7] = [\_@\mathit{Monitor}, \ldots, \_@\mathit{Monitor}] \\ I_w(\mathit{mem}^C, \mathit{reg}^C, \mathit{cache}, \mathit{extra}) \qquad I(\mathit{mem}^C, \mathit{extra}) \end{array}}{\begin{array}{c} (\mathit{mem}^C, \mathit{reg}^C, \mathit{pc}@(\mathit{User}\ t), \mathit{epc}, \mathit{cache}) \\ \approx_I^{CS} (\mathit{mem}^S, \mathit{reg}^S, \mathit{pc}@t, \mathit{extra}) \end{array}}$$

where $\mathit{reg}^C \sim \mathit{reg}^S$ means $\forall r, x, t', \mathit{reg}^C[r] = x@(\mathit{User}\ t') \iff \mathit{reg}^S[r] = x@t'$, and similarly for $\mathit{mem}^C \sim \mathit{mem}^S$. The predicate *cache_ok* states that, whenever a rule with *pc* tag *User* 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 *Entry st*, where *st* is that service's tag. Finally, I_w is a generic wrapper invariant used to ensure that monitor code and data are tagged the appropriate way and located in the expected memory regions, and that the ground rules are good.

The two-way concrete to symbolic (*CS*) simulation theorem also relies on the following definition (1-forward simulation is just the dual of Definition 4.2, switching the roles of L and H):

[2] Only backward refinement is used in the rest of the paper—indeed, forward refinement doesn't hold at the symbolic-to-abstract level for most of the micro-policies we consider because their abstract machines abstract away from resource constraints that can cause the symbolic machines to fail. However, the forward direction may be interesting for other micro-policies. For example, we believe it holds for CFI.

Definition 6.2 ($\{0,1\}$-backward simulation)**.** If $s_1^L \sim s_1^H$ and $s_1^L \to s_2^L$ then $s_1^L \sim s_2^H$ or $\exists s_2^H$ such that $s_1^H \to s_2^H$ and $s_2^L \sim s_2^H$.

Theorem 6.3 (Two-way *CS*-simulation)**.** (1) The concrete machine $\{0,1\}$-backward-simulates the symbolic machine, with respect to $\sim_I^{CS}$. (2) The concrete machine 1-forward-simulates the symbolic machine, with respect to $\approx_I^{CS}$.

The proof assumes the correctness of the machine code provided by the policy designer. Specifically: (1) On a cache miss, if all the refinement invariants (including I) are satisfied at the faulting instruction, then the miss handler must successfully return to a user state iff the faulting tag combination is allowed by the transfer function. In that 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 iff the corresponding symbolic monitor service allows that execution. In this case, the resulting user state must be a refinement of the new symbolic state.

Example: Concrete Sealing Machine To implement the symbolic sealing machine on the concrete machine, we represent symbolic sealing tags as follows: $\mathit{enc}\ \mathit{Data} = \bar{0}$; $\mathit{enc}\ (\mathit{Key}\ k) = k \cdot 0 \cdot 1$; and $\mathit{enc}\ (\mathit{Sealed}\ k) = k \cdot 1 \cdot 1$. The key counter on the symbolic machine is represented concretely as a single word of monitor memory; the key inside is a sub-word though since it needs to fit in 28 bits of a *Key* user tag. Implementing the transfer function is just a matter of checking that all required tags are indeed *enc Data* and propagating tags that need to be preserved, following the symbolic rules presented in §4. Implementing the monitor services is also simple. The *mkkey* routine increments the key counter and remembers the old value k. It then tags the return register with $\mathit{User}\ (\mathit{Key}\ k)$ (with a dummy payload) and returns to user code. The *seal* routine checks (using *GetTag*) that its first argument has the form $x@(\mathit{User}\ \mathit{Data})$ and its second argument is tagged $\mathit{User}\ (\mathit{Key}\ k)$, assembles $x@(\mathit{User}\ (\mathit{Sealed}\ k))$ in r_{ret} using *PutTag*, and returns; *unseal* does the converse. All of these routines halt if the arguments do not have the required form.

Theorem 6.4 (Backward *CA*-refinement for sealing)**.** The concrete sealing implementation backward-refines (Definition 4.1) the abstract sealing machine, with respect to the simulation relation $\sim_{\mathit{true}}^{CS} \circ (\lambda\ s^S\ s^A.\ \exists \psi.\ s^S \sim_\psi^{SA} s^A)$, where $\circ$ stands for relation composition, $\sim_I^{CS}$ was defined above (we just choose the invariant I to be *true*), and $\sim_\psi^{SA}$ was explained at the end of §4.

The structure of the backward refinement proofs for the other micro-policies is very similar: we compose a policy-specific 1-backwards *SA*-simulation proof, with an instance of the generic $\{0,1\}$-backwards *CS*-simulation proof above (Theorem 6.3, part a).

7. Control-Flow Integrity Micro-Policy

We close our story with three more challenging micro-policies. The first targets control-flow hijacking attacks, in which an attacker exploits a low-level vulnerability (e.g. a buffer or integer overflow) to gain full control of a target program. As a first line of defense, we can use tags to make code non-writable (NWC [1]) and data non-executable (NXD [1]), preventing the injection and execution of an attacker payload. This useful defense appears in various forms in existing systems. However, it does not prevent code-reuse attacks (e.g., return- or jump-oriented programming), where the attacker chains together existing code snippets to induce arbitrary malicious behavior. We therefore use tags to ensure fine-grained *control-flow integrity (CFI)* [1] on top of basic NWC and NXD protection. Our CFI micro-policy dynamically enforces that all indirect control flows (computed jumps) adhere to a fixed control flow graph (CFG).

The main result of this section is a variant of the CFI property of Abadi *et al.* [1] for our concrete machine running a CFI monitor. For this we prove that CFI is preserved by $\{0, 1\}$-backward simulation, under certain additional assumptions. We then show that our instance of the concrete machine simulates an instance of the symbolic machine from §3, which in turn simulates an abstract machine that has CFI by construction.

CFI property and attacker model We give a generic CFI definition that will be instantiated to the three machines, adapting the original definition by Abadi *et al.* [1] to our setting. The two main technical differences are that (1) our 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 is not immediate; rather, it involves failing in the hardware rule cache and running the miss handler, which eventually halts. While these details are immaterial for security, 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 $\rightarrow_a$ (formally $\rightarrow = \rightarrow_n \cup \rightarrow_a$). 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 cannot directly subvert our NWC, NXD, or CFI protections.

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

Definition 7.1. We say that an execution trace $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$ *has CFI* if $(s_i, s_{i+1}) \in cfg\ J$ for all $i \in [0, \ldots, n)$.

Compared to Abadi *et al.* [1], this definition additionally requires that the steps that are in the intersection of $\rightarrow_a$ and $\rightarrow_n$ are in the CFG, which is helpful for proving CFI preservation. So if we required that all traces of a machine have CFI, we would obtain a definition slightly stronger than Abadi *et al.*'s. Instead we use the following incomparable definition, which allows a single violation in a trace, as long as the machine is "stopping" afterwards.

Definition 7.2 (CFI)**.** We say that the machine $(State, initial, \rightarrow_n, \rightarrow_a, cfg, stopping)$ 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 7.1, or else (2) there is some i such that $s_i \rightarrow_n s_{i+1}$, and $(s_i, s_{i+1}) \notin cfg\ J$, 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 ($\not\rightarrow_n$). We need this definition because 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 before the machine is fully halted; 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 instructions to be executed are fetched from this memory (NXD):

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

The machine state also contains 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 stop on the next step. 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).

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

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.

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

As for all other step rules we proceed only when the *ok* bit is *true*, which prevents monitor service calls outside the allowed CFG (i.e., it prevents jump-to-monitor-service code-reuse attacks).

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

$$\frac{dom\ dm = dom\ dm' \qquad dom\ reg = dom\ reg'}{(im, dm, reg, pc, true) \rightarrow_a^A (im, dm', reg', pc, true)}$$

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 7.3 (Abstract CFI)**.** This abstract machine has CFI.

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 and the *pc* are drawn from the set $Data \mid Instr\ addr \mid Instr\ \bot$ (registers are always tagged *Data*). To simplify the CFG conformance checks, instructions that are the source or target of indirect control flows are tagged with *Instr addr*, where *addr* is the address of the instruction in memory. For example, a *Jump* instruction stored at address 500 is tagged *Instr* 500. All other instructions are tagged *Instr* $\bot$. Only memory locations tagged *Data* can be modified (NWC), and only instructions fetched from locations tagged *Instr* can be executed (NXD). The symbolic rule for *Store* illustrates both these points:

$$Store \quad : \quad (Data, Instr\ \_, -, -, Data) \rightarrow (Data, -)$$

It requires the fetched *Store* instruction to be tagged *Instr* and the written location to be tagged *Data*. Performing a computed jump requires that the current instruction be tagged *Instr src* for an address *src*; it then copies *Instr src* to the *pc* tag.

$$Jal \quad : \quad (Data, Instr\ src, -, -, -) \rightarrow (Instr\ src, -)$$

Only on the next instruction do we have enough information about the destination in the tags to check that the jump is indeed allowed by J. For this we add a second rule for *Store*, dealing with the case where it is the target of a jump and thus the *pc* is *Instr src*.

$$\frac{(src, dst) \in J}{Store : (Instr\ src, Instr\ dst, -, -, Data) \rightarrow (Data, -)}$$

We add a similar rule for each instruction, including jumps, since the target of a computed jump can itself be another computed jump.

We capture the attacker at the symbolic level by the relation

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

where the $\rightarrow_a^S$ relation on memories and registers is the pointwise extension of the following inductive relation on atoms:

$$w_1@\mathit{Data} \rightarrow_a^S w_2@\mathit{Data} \qquad w@(\mathit{Instr}\ \mathit{id}) \rightarrow_a^S w@(\mathit{Instr}\ \mathit{id})$$

This allows attackers to change words tagged *Data* but prevents them from changing words tagged *Instr* or tags themselves.

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

Concrete machine As opposed to the other policies, which simply instantiate the generic refinement result from §6, for CFI we need to define all concepts appearing in the CFI property also for the concrete machine. 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 *epc*.

$$\frac{\mathit{mem} \rightarrow_a^C \mathit{mem}' \qquad \mathit{reg} \rightarrow_a^C \mathit{reg}'}{\begin{array}{c}(\mathit{mem}, \mathit{reg}, \mathit{cache}, \mathit{pc}@(\mathit{User}\ \mathit{ut}), \mathit{epc}) \\ \rightarrow_a^C (\mathit{mem}', \mathit{reg}', \mathit{cache}, \mathit{pc}@(\mathit{User}\ \mathit{ut}), \mathit{epc})\end{array}}$$

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

$$\frac{w_1@\mathit{ut}_1 \rightarrow_a^S w_2@\mathit{ut}_2}{w_1@(\mathit{User}\ \mathit{ut}_1) \rightarrow_a^C w_2@(\mathit{User}\ \mathit{ut}_2)}$$

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 of the tags. This attacker model relies on the correctness of the monitor-self protection mechanism from §6.

The initial steps 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 §6 (I_w) and that they respect the symbolic invariants.

A concrete trace is stopping if it has an (optional) prefix formed only of attacker steps between user states that are all stuck with respect to normal steps, followed by an (optional) suffix of kernel 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, when the machine is in monitor mode, all control flows are allowed. We assume that monitor code is correct, so we do not need to enforce CFI there.

Proof technique 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. This organization has significant advantages. Most importantly, it reduces the proof effort by allowing us to reuse the generic simulation result from §6 for relating the concrete and the symbolic machines. Secondly, using the symbolic machine as an intermediate step allows us to do most of the reasoning at the symbolic level, even for the proofs involving the concrete machine. Basically, the invariants we use at the concrete level come either from the generic framework in §6 or from symbolic invariants via the simulation relation; we do not need to reestablish them. Moreover, we use the symbolic invariants for proving both 1-backward *SA*-simulation and *SC*-CFI-preservation. Finally, the refinement with the correct-by-construction abstract machine provides additional assurance in the correctness of the micro-policy.

Theorem 7.4 (CFI Preservation)**.** Given a high-level machine $M^H = (\mathit{State}^H, \mathit{initial}^H, \rightarrow_n^H, \rightarrow_a^H, \mathit{cfg}^H, \mathit{stopping}^H)$, a low-level machine $M^L = (\mathit{State}^L, \mathit{initial}^L, \rightarrow_n^L, \rightarrow_a^L, \mathit{cfg}^L, \mathit{stopping}^L)$, 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 wrt $\sim$ for checked steps in $\rightarrow_n^L$;
A2. $\{0, 1\}$-backward simulation wrt $\sim$ for unchecked steps in $\rightarrow_n^L$;
A3. 1-backward simulation wrt $\sim$ for attacker steps ($\rightarrow_a^L$);
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 $\mathit{cfg}^H\ J = \mathit{cfg}^L\ J$; (the following 3 assumptions also have $s_1^L \sim s_1^H$ as a hypothesis)
A6. if $s_1^L \rightarrow_n s_2^L$ and $\neg\mathit{checked}\ s_1^L\ s_2^L$, then $(s_1^L, s_2^L) \in \mathit{cfg}^L\ J$;
A7. if $(s_1^H, s_2^H) \notin \mathit{cfg}^H\ J$ and $s_1^H \rightarrow_n s_2^H$ then $\neg(s_1^H \rightarrow_a s_2^H)$;
A8. and if *checked* $s_1^L\ s_2^L$, $(s_1^H, s_2^H) \notin \mathit{cfg}^H\ J$, and $s_1^H \rightarrow s_2^H$, 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 also *stopping* $(s_2^L :: t^L)$.

Assumption *A3* states that low-level attacker steps ($\rightarrow_a^L$) 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 7.5 (CFI Concrete)**.** The concrete machine running a CFI monitor satisfying the assumptions described in §6 has CFI.

Related Work Abadi *et al.* [1] proposed both the first CFI definition and a relatively efficient but coarse-grained enforcement mechanism based on binary analysis and rewriting, in which each node in the CFG is assigned to 1 out of 3 equivalence classes. This seminal work was extended in various directions, very often by trading off precision for low overheads and practicality [28]. Recent attacks against coarse-grained CFI [12] illustrate the security risks of imprecision. This has spurred interest in *fine-grained* CFI, sometimes called *complete* or *ideal* CFI [8, 25]; this, however, is often deemed "very expensive" [12]. The PUMP mechanism supports fine-grained CFI with average overheads around 8% [14]. Previous formal verification efforts for CFI include ARMor [29] and KCoFI [8]. Like most work on CFI, they use inline reference monitoring [11]; their verification targets a small-TCB component that validates that the right checks were inserted in the instrumented binary.

8. Memory Safety Micro-Policy

In this section we devise a micro-policy that prevents all spatial and temporal memory safety violations on heap-allocated data. Such violations are a common source of serious security vulnerabilities:

e.g., heap-based buffer overflows, confidential data leaks, use-after-free and double-free bugs, etc. The micro-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 memory-safety abstract machine presents a block-based memory model [4, 19]: it operates on values that are either 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 regular 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). *Store* first looks up b in the memory; if this block is currently allocated, it obtains a list of values vs, which it updates at index o (provided o is in bounds).

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

The *pc* is itself a pointer with a block and an offset; instruction fetches work the same as normal memory loads.

As with key generation in §4, the allocation and freeing monitor services are partially specified in terms of functions *alloc_f* and *free_f* that satisfy certain high-level properties. The *alloc_f* function 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.The *free_f* function takes a memory and an allocated block and returns a new memory in which the block is no longer allocated, keeping all other blocks the same.

Symbolic Machine The tag encoding of the memory-safety policy replaces 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. (The point of view here is a little different from what we saw in the symbolic sealing machine, where the tag on a memory cell was intuitively a tag on the *value* stored in the cell. Here, the tag is thought of as the color of the memory *location* itself; it persists when new values are stored there.)

We use different sets of tags for values in registers (and the *pc*) and in memory. The former are either pointers tagged with a *color* c or non-pointers tagged $\perp$; we use the variable t_v for value tags. 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 key idea of this tagging scheme is that when we read or write through a pointer, we check that its color is the same as the color on the memory cell it points to.

The *malloc* monitor service first searches its list of free memory blocks for one of at least the required size, cuts off the excess if needed, then generates a fresh color c (using a monotonic counter), initializes the new memory block with $0@(c, \perp)$, and returns the atom $w@c$ in a register, 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 adds it to the freelist for later re-use. The new tags prevent any remaining pointers to the deallocated block from being used to access it immediately after deallocation. Later, if another 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 is tagged with the same color c as the memory it points to.

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

We additionally require that the *pc* tag c_{pc} match the color of the block to which the *pc* points. On *Jump*s we change the color of the *pc* to that of the pointer

$$Jump \quad : \quad (c_{pc}, (c_{pc}, \perp), c, -, -) \rightarrow (c, -)$$

while for *Jal* we also use the color of the old *pc* to tag the *ra* register:

$$Jal \quad : \quad (c_{pc}, (c_{pc}, \perp), t_v, -, -) \rightarrow (t_v, c_{pc})$$

We also allow *Jal*s to values tagged $\perp$. This is needed for monitor services, which lie outside the accessible memory at this level of abstraction and so cannot be referenced by normal pointers.

All binary operations are allowed between values tagged $\perp$ (non-pointers), and they produce values tagged $\perp$:

$$Binop_{\oplus} : \ (c_{pc}, (c_{pc}, \perp), \perp, \perp, -) \rightarrow (c_{pc}, \perp)$$

Additional rules allow adding/substracting integers to pointers:

$$\begin{array}{l} Binop_{+,-} :(c_{pc}, (c_{pc}, \perp), c, \perp, -) \rightarrow (c_{pc}, c) \\ Binop_{+} : \ (c_{pc}, (c_{pc}, \perp), \perp, c, -) \rightarrow (c_{pc}, c) \end{array}$$

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* above prevent invalid accesses. Taking a pointer out of bounds is not a violation *per se* (indeed, it happens quite often in practice, e.g., at the end of loops). Subtraction can also compute the offset between two pointers to the same block:

$$Binop_{-,=} :(c_{pc}, (c_{pc}, \perp), c, c, -) \rightarrow (c_{pc}, \perp)$$

Pointers to the same block can also be compared for equality using $Binop_{=}$. However, comparing a pointer with a non-pointer or with a pointer to a different block is disallowed, to avoid producing a result that could be wrong with respect to the abstract machine defined below. (For instance, offsetting a concrete pointer can take it out of bounds and thus make it numerically equal to a pointer into a different block, which is impossible in the abstract machine.) Finally, we provide a monitor service for getting the base address of a pointer's block. We believe we can also provide a *size* service in the future but have not completed the refinement proof for that.

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

Theorem 8.1 (1-backward *SA*-simulation)**.** The symbolic memory safety machine backward-simulates the abstract machine, with respect to the simulation relation $\sim_\phi$.

The map ϕ relates the two memory models by sending colors to pairs of an abstract block identifier and the base address of the corresponding block on the symbolic machine. The simulation relation on states $\sim_\phi$ is defined in terms of the relation $\sim_{\phi,t_v}$ between abstract values and symbolic words tagged t_v :

$$\begin{array}{rccl} w' & \sim_{\phi,\perp} & w & = \ w = w' \\ (w+o) & \sim_{\phi,c} & (b,o) & = \ \phi(c) = (b,w) \\ w & \sim_{\phi,t_v} & v & = \ \mathit{false}\text{, otherwise} \end{array}$$

This relation is extended pointwise to register banks, whereas the relation between memories $mem_S \sim_\phi mem_A$ says that whenever a memory location can be accessed on the symbolic machine, the corresponding location on the abstract machine can also be and that the two values found in these locations are in refinement:

$$\begin{array}{l} mem_S[w_1] = w_2@(c, t_c) \Rightarrow \\ \phi(c) = (b, base) \wedge mem_A[b][base - w_1] = v \wedge w_2 \sim_{\phi,t_c} v \end{array}$$

Since we want to preserve both spatial and temporal safety, the map ϕ keeps track of both live and freed blocks. Hence, ϕ is left unchanged if the two machines take corresponding steps, except for allocation, which extends the domain of ϕ with the newly allocated block. The key property of ϕ is (left) injectivity: two colors that are mapped to the same block are identical.

The core of this proof consisted in verifying the allocator that we provide at the symbolic level. In particular, the allocator has to maintain a list of block descriptors keeping track of regions and colors, which is more than what a usual free list would give. We showed formally that some invariants are preserved, such as block descriptors cover the whole memory and are non-overlapping.

Related Work Our scheme is inspired by the metadata tainting technique of Clause *et al.* [6]. Similar ideas are used by Watchdog [22] (for temporal safety). These systems do not have formal proofs. Nagarakatte *et al.* have verified in Coq that the SoftBound pass in LLVM/Vellvm satisfies "spatial safety" [24] and that the CETS temporal safety extension to SoftBound is correct [23] in the sense of backward simulation. 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 full abstraction with respect to a high-level language semantics.

9. Compartmentalization Micro-Policy

The last of our micro-policies enforces isolation between program-defined "compartments," dynamically demarcated memory regions that by default cannot jump or write to each other. To allow controlled communication, each compartment has sets of *allowed jump targets* and *allowed store targets*. This model is based on Wahbe *et al.*'s software fault isolation (SFI) model [27], with a few differences discussed below. To prove isolation, we show that a symbolic-machine encoding using tags refines an abstract machine that enforces compartmentalization by construction. As a sanity check, we prove that this abstract machine satisfies a compartmentalization property drawn from Wahbe *et al.* [27].[3]

Abstract machine The abstract machine enforces compartmentalization directly by maintaining a set C of current compartments. It has no tags, and its values are plain words; compartmentalization is enforced by referring to C on each step to prevent one compartment from 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 the compartment is allowed to execute and write to; (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. As in Wahbe *et al.*'s model [27], all compartments are permitted to read all memory locations. We maintain the invariant that all compartments have disjoint address spaces (and some other invariants discussed later).

The machine state contains a flag $F \in \{\mathit{Jumped}, \mathit{Internal}\}$ that records whether or not the previous instruction was a *Jump* or a *Jal*, and a record of the previously-executing compartment, *prev*. This information is used to maintain compartmentalized execution and to allow monitor services to see which compartment called them.

At the abstract level, all instructions behave as in the basic machine in §2, modulo the addition of a compartmentalization check. Each step rule checks that the current instruction is executing inside some compartment and (using *prev*) that execution arrived at this instruction either (a) from the same compartment, or (b) with $F = \mathit{Jumped}$ and the current *pc* in the previous compartment's set of jump targets. Deferring detection of execution violations until one step *after* they have occurred (as we also do for CFI §7) is essential to our tag-based implementation strategy. Finally, *Store* has an additional check that its write is to either the current compartment or one of its store targets ($w_p \in A \cup S$):

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

The abstract machine also provides three monitor services. The core service is *isolate*, which creates a new compartment. At a high level, 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 at most as privileged as its parent. The argument sets are passed in as pointers to sets of words represented in memory.

By itself, *isolate* does not give the parent compartment access to the address space of the child, but it leaves the store and jump targets of the parent unchanged, and these can point to the child's address space. Before creating the child, the parent can use the monitor services *add_jump_target* and *add_store_target* to add addresses from its address space set to its set of jump and store targets.

In the initial configuration of the abstract machine, all defined addresses lie in one main compartment and each monitor service address (recall that these are undefined, at the abstract level) has its own unique compartment. 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.

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.

Symbolic machine To implement this abstract machine in terms of tags requires "dualizing" the representation of compartments: rather than maintaining global state recording which compartments exist and what addresses 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 additional state of the symbolic machine contains a monotonic counter *next* for the next available compartment id.

We use two kinds of symbolic tags: one for memory locations and one for the *pc* (registers are labeled with a dummy tag *Reg*). (1) A memory tag is a triple $\langle c, I, W\rangle$, where c is the id of the compartment to which this 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. (2) A *pc* tag is a pair $\langle F, c\rangle$, where the flag F has the same role as on the abstract machine and c is the ID of the compartment that was *previously* executing.

We use extra state cells t_I, t_{AJ}, and t_{AS} to store the tag on the monitor services entry points. We require that the compartment ids

[3] For this section, our Coq proofs are slightly incomplete. We've proved the correctness of the transfer function, but the formal proofs that the monitor services preserve refinement are missing a few steps.

of these tags be distinct and not be used as tags on defined addresses. Here are a few of the symbolic rules (the rest are very similar):

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

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

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

The side-condition on the first two rules guarantees compartment-safe execution. Recall that c is the previously-executing compartment; c', which tags 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 ($F = \mathit{Jumped} \wedge c \in I$). 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 current compartment as a writer ($c' \in W'$).

This encoding scheme scatters the information in the abstract jump tables across the various incoming components of tags; similarly, the store targets now live in the writers' components. The state maintained in the *pc* tag corresponds exactly to *prev* in the abstract machine, except that we use a compartment id rather than an abstract compartment.

The monitor services must now 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 the new compartment's address space, changing each tag from $\langle c, I, W\rangle$ into $\langle c_{\text{new}}, I, W\rangle$. (3) It retags the new compartment's set of jump targets, changing each 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 $\langle c_S, I_S, W_S \cup \{c_{\text{new}}\}\rangle$.

Abstract compartmentalization We start by proving that the abstract machine satisfies a high-level compartmentalization property drawn from [27]. We will establish a notion of "good" abstract states, such that we can prove the following:

Theorem 9.1 (Compartmentalization)**.** Let $(\mathit{mem}, \mathit{reg}, \mathit{pc}, C, F, \mathit{prev})$ be a good abstract machine state (explained below) such that the *pc* lies in the compartment (A, J, S). If this state steps to the state $(\mathit{mem}', \mathit{reg}', \mathit{pc}', C', F', \mathit{prev}')$, then (a) if this resulting state isn't stuck, then $\mathit{pc}' \in A \cup J$; and (b) for any address a such that $\mathit{mem}[a] \neq \mathit{mem}'[a]$, we have $a \in A \cup S$.

An abstract state $(\mathit{mem}, \mathit{reg}, \mathit{pc}, C, F, \mathit{prev})$ is *good* if: (1) $\mathit{prev} \in C$; (2) the address spaces of the compartments in C are pairwise non-overlapping; (3) all jump and store targets in C lie inside some address space in C; (4) each monitor service address lies in its own compartment; and (5) all non-monitor-service address spaces contain only defined addresses. We proved that goodness is preserved by the step relation, with particular care in the rules for *Jump*, *Jal*, and *Store* and in the definition of the monitor services.

Backward refinement Additionally, we prove backward refinement between the concrete machine running a correct implementation of our compartmentalization monitor and the abstract machine, by way of the symbolic compartmentalization machine:

Theorem 9.2 (backward *CA*-refinement)**.** The concrete compartmentalization implementation backward-refines the abstract machine, with respect to the simulation relation $\sim_I^{CS} \circ \sim^{SA}$.

The relation $\sim^{SA}$ is defined as follows: $(\mathit{mem}^S, \mathit{reg}^S, \mathit{pc}^S @\langle F^S, \mathit{prev}^S\rangle, \mathit{next}, t_I, t_{AJ}, t_{AS}) \sim^{SA} (\mathit{mem}^A, \mathit{reg}^A, \mathit{pc}^A, C, F^A, \mathit{prev}^A)$ when (1) $\mathit{reg}^S = \mathit{reg}^A$; (2) $\mathit{pc}^S = \mathit{pc}^A$; (3) $F^S = F^A$; (4) mem^S and mem^A agree on all values, and mem^S has only data tags; (5) prev^S simulates prev^A; (6) all compartments in C are simulated by something (explained below); (7) the symbolic state simulates C (explained below); (8) the addresses of system calls are distinct and are undefined in both mem^S and mem^A; and (9) the symbolic auxiliary state satisfies the appropriate invariants. This requires a notion of simulation between a compartment id and a single abstract compartment, as well as between a symbolic state and a set of abstract compartments. The former guarantees that a compartment id represents the same thing as a particular compartment; the latter guarantees that the set of all data tags captures the same thing as the original set of compartments.

A compartment id c^S simulates an abstract compartment (A, J, S), relative to a symbolic state s^S, when (1) all addresses in A have the compartment id c^S in their data tag in s^S; (2) the compartment id c^S occurs in the set of incoming compartments in the tag on all addresses in J in s^S; and (3) the compartment id c^S occurs in the set of writer ids in the tag on all addresses in S in s^S.

Simulation between a symbolic state s^S and a set C of abstract compartments is defined as follows. If a memory location p has a tag t in s^S, we require three things. First, t must have the form $\langle c^S, I, W\rangle$. Second, there must be some abstract compartment $(A, J, S) \in C$ with $p \in A$. Third, the tags of other addresses p' in s^S must have the form $\langle c'^S, I', W'\rangle$ and satisfy some additional constraints: $c^S = c'^S$ iff $p' \in A$; if $c^S \in I'$, then $p' \in J$; and if $c^S \in W'$, then $p' \in S$.

Related Work There are several verified SFI systems, including ARMor [29], RockSalt [21], and a portable one by Kroll *et al.* [16]. Our compartmentalization model, while based on Wahbe *et al.*'s original SFI work [27], differs from that in several important ways. Most importantly, our monitor is not based on binary rewriting, instead using the hardware / software mechanism of the PUMP architecture. Our model is also richer in that it provides a hierarchical compartment-creation mechanism, as opposed to a single trusted top-level program that can spawn one level of untrusted plugins. One feature Wahbe *et al.*'s model that we do not support is inter-compartment RPCs; we instead require programs to manually predeclare inter-compartment calls and returns.

10. Related Work on Micro-Policies

Work related to specific micro-policies has already been discussed. Here we focus on micro-policies in general.

The micro-policies framework and the PUMP architecture have their roots in SAFE, a clean-slate, security-oriented architecture [9]. In that context the PUMP was used only to implement dynamic IFC; various other special-purpose hardware mechanisms enforced properties like memory safety [17] and compartmentalization [9]. The PUMP was designed to be quite flexible because dynamic IFC is an active area of research, with various mechanisms and "label models" being proposed regularly. This use of the PUMP has been studied formally for a very simplified version of SAFE [4].

The present work, together with [14] and [10], aims to demonstrate the applicability of the PUMP beyond IFC and beyond SAFE. We consider a diverse set of micro-policies and a more conventional architecture, a (less) simplified RISC machine, with bit-strings as words instead of integers as in [4], 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. We show that one single hardware mechanism, the PUMP, is enough to obtain in software similar kinds of protection to what the concrete machine from [4] provided in hardware.

The general structure of our proofs is similar to [4]; in particular, that paper also proves refinement between a concrete machine and an abstract one, using a "symbolic IFC rule machine" as an intermediate step, and, as we do for CFI, it proves a generic preservation theorem that non-interference can be carried to the lowest level. The rule machine, however, is merely a reformulation of an IFC abstract machine to factor a "rule table" written in a simple DSL out of the semantics. In contrast, our symbolic machine is fully generic and is reused by all micro-policies. Moreover, with the exception of dynamic sealing, the symbolic machine instances we study are not just reformulations of (in a sense degenerate) abstract machines that expose low-level tags; indeed, devising these instances is by and large the hardest and most creative part of designing micro-policies, and the refinement between the symbolic and abstract machines is generally challenging. Our end-to-end abstract-to-concrete refinement proofs are obtained from a generic theorem for 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 their simple architecture. We chose here to focus on devising a generic micro-policy framework and on designing and verifying the symbolic machine instances for a diverse set of micro-policies. We did not verify (or indeed, with the exception of dynamic sealing, even write) machine code for the concrete monitors. In the future we hope to close this gap by porting our micro-policies framework to a conventional RISC architecture, for which verification infrastructure for low-level code [4, 5, 15] or a verified compiler [18] already exists (e.g., ARM), or at least where the payoff of building such infrastructure is worth the high upfront costs. This might also be a good target for property-based testing.

11. Conclusions and Future Work

We have presented a formal framework for specifying, implementing, and verifying micro-policies for a simple RISC machine enhanced with hardware for propagating and checking tags. Our Coq development runs to about 23.4k lines of code, out of which 8.8k lines are generic and the rest specific to our four micro-policies.

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 combine or compose micro-policies. Certain combinations are known to compose sensibly and some of them perform reasonably on practical workloads [14], but the general picture remains unclear. Another obvious target for future work is formalizing the informal symbolic rule language used to present examples here and in [14].

We used 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. We have not explicitly considered the role of the compiler or loader here, although in reality their support is sometimes crucial. 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 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 [14]) is to only protect user-level code, but this would generally lead to a much larger TCB.

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 interesting security properties: IFC, CFI, compartmentalization, and memory safety. For attacking the expressiveness question one can try to take inspiration in the work done by Schneider *et al.* [13, 26] for execution monitors and program rewriting.

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

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

[7] T. Coq Development Team. *The Coq Reference Manual, version 8.4*, 2012. Available electronically at `http://coq.inria.fr/doc`.

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

[9] 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.

[10] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, T. F. Knight, B. C. Pierce, and A. DeHon. PUMP – A Programmable Unit for Metadata Processing. *HASP*. 2014.

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

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

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

[14] C. Hriţcu, U. Dhawan, N. Vasilakis, S. Chiricescu, J. M. Smith, B. C. Pierce, and A. DeHon. Programming the PUMP: Hardware-assisted micro-policies for security. Under Review, May, 2014.

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

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

[17] 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.

[18] X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

[19] 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.

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

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

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

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

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

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

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

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

[28] 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.

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