A Verified Information-Flow Architecture

Arthur Azevedo de Amorim[1] Nathan Collins[2] André DeHon[1] Delphine Demange[1]
Cătălin Hriţcu[1,3] David Pichardie[3,4] Benjamin C. Pierce[1] Randy Pollack[4] Andrew Tolmach[2]

[1]University of Pennsylvania [2]Portland State University [3]INRIA [4]Harvard University

Abstract

SAFE is a clean-slate design for a highly secure computer system, with pervasive mechanisms for tracking and limiting information flows. At the lowest level, the SAFE hardware supports fine-grained programmable tags, with efficient and flexible propagation and combination of tags as instructions are executed. The operating system virtualizes these generic facilities to present an information-flow abstract machine that allows user programs to label sensitive data with rich confidentiality policies. We present a formal, machine-checked model of the key hardware and software mechanisms used to control information flow in SAFE and an end-to-end proof of noninterference for this model.

Categories and Subject Descriptors D.4.6 [*Security and Protection*]: Information flow controls; D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords security; clean-slate design; tagged architecture; information-flow control; formal verification; refinement

1. Introduction

The SAFE design is motivated by the conviction that the insecurity of present-day computer systems is due in large part to legacy design decisions left over from an era of scarce hardware resources. The time is ripe for a complete rethink of the entire system stack with security as the central focus. In particular, designers should be willing to spend more of the abundant processing power available on today's chips to improve security.

A key feature of SAFE is that every piece of data, down to the word level, is annotated with a *tag* representing policies that govern its use. While the tagging mechanism is very general, one particularly interesting use of tags is for representing *information-flow control (IFC)* policies. For example, an individual record might be tagged "This information should only be seen by principals `Alice` or `Bob`," a function pointer might be tagged "This code is trusted to work with `Carol`'s secrets," or a string might be tagged "This came from the network and has not been sanitized yet." Such tags representing IFC policies can involve arbitrary sets of principals, and principals themselves can be dynamically allocated to represent an unbounded number of entities within and outside the system.

At the programming-language level, rich IFC policies have been extensively explored, with many proposed designs for static [19, 40, etc.] and dynamic [3, 20, 39, 44, etc.] enforcement mechanisms and a huge literature on their formal properties [19, 40, etc.]. Similarly, operating systems with information-flow tracking have been a staple of the OS literature for over a decade [28, etc.]. But progress at the hardware level has been more limited, with most proposals concentrating on hardware acceleration for taint-tracking schemes [12, 15, 45, 47, etc.]. SAFE extends the state of the art in two significant ways. First, the SAFE machine offers hardware support for sound and efficient purely-dynamic tracking of both explicit and implicit flows (i.e., information leaks through both data and control flow) for *arbitrary* machine code programs—not just programs accepted by static analysis, or produced by translation or transformation. Moreover, rather than using just a few "taint bits," SAFE associates a word-sized tag to every word of data in the machine—both memory and registers. In particular, SAFE tags can be pointers to arbitrary data structures in memory. The interpretation of these tags is left entirely to software: the hardware just propagates tags from operands to results as each instruction is executed, following software-defined rules. Second, the SAFE design has been informed from the start by an intensive effort to formalize critical properties of its key mechanisms and produce machine-checked proofs, in parallel with the design and implementation of its hardware and system software. Though some prior work (surveyed in §12) shares some of these aims, to the best of our knowledge no project has attempted this combination of innovations.

Abstractly, the tag propagation rules in SAFE can be viewed as a partial function from argument tuples of the form (*opcode*, *pc tag*, $argument_1$ *tag*, $argument_2$ *tag*, ...) to result tuples of the form (*new pc tag*, *result tag*), meaning "if the next instruction to be executed is *opcode*, the current tag of the program counter (PC) is *pc tag*, and the arguments expected by this opcode are tagged $argument_1$ *tag*, etc., then executing the instruction is allowed and, in the new state of the machine, the PC should be tagged *new pc tag* and any new data created by the instruction should be tagged *result tag*." (The individual argument-result pairs in this function's graph are called *rule instances*, to distinguish them from the symbolic *rules* used at the software level.) In general, the graph of this function *in extenso* will be huge; so, concretely, the hardware maintains a *cache* of recently-used rule instances. On each instruction dispatch (in parallel with the logic implementing the usual behavior of the instruction—e.g., addition), the hardware forms an argument tuple as described above and looks it up in the rule cache. If the lookup is successful, the result tuple includes a new tag for the PC and a tag for the result of the instruction (if any); these are combined with the ordinary results of instruction execution to yield the next machine state. Otherwise, if the lookup is unsuccessful, the hardware invokes a *cache fault handler*—a trusted piece of system software with the job of checking whether the faulting combination of tags corresponds to a policy violation or whether it should be allowed. In the latter case, an appropriate rule instance specifying tags for the instruction's results is added to the cache, and the faulting instruction is restarted. Thus, the hardware is generic and the interpretation of policies (e.g., IFC, memory safety or control

POPL '14, January 22–24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01.
http://dx.doi.org/10.1145/2535838.2535839

flow integrity) is programmed in software, with the results cached in hardware for common-case efficiency.

The first contribution of this paper is to explain and formalize, in Coq, the key ideas in this design via a simplified model of the SAFE machine, embodying its tagging mechanisms in a distilled form and focusing on enforcing IFC using these general mechanisms. In §2, we outline the features of the full SAFE system and enumerate the most significant simplifications in our model. To streamline the exposition, most of the paper describes a further-simplified version of the system, deferring to §11 the discussion of the more sophisticated memory model and IFC label representation that we have actually formalized in Coq. We begin by defining a very simple *abstract IFC machine* with a built-in, purely dynamic IFC enforcement mechanism and an abstract lattice of IFC labels (§3). We then show, in three steps, how this abstract machine can be implemented using the low-level mechanisms we propose. The first step introduces a *symbolic IFC rule machine* that reorganizes the semantics of the abstract machine, splitting out the IFC enforcement mechanism into a separate judgment parameterized by a *symbolic IFC rule table* (§4). The second step defines a generic *concrete machine* (§5) that provides low-level support for efficiently implementing many different high-level policies (IFC and others) with a combination of a hardware *rule cache* and a software *fault handler*. The final step instantiates the concrete machine with a concrete fault handler enforcing IFC. We do this using an *IFC fault handler generator* (§6), which compiles the symbolic IFC rule table into a sequence of machine instructions implementing the IFC enforcement judgment.

Our second contribution is a machine-checked proof that this simplified SAFE system is *correct* and *secure*, in the sense that user code running on the concrete machine equipped with the IFC fault handler behaves the same way as on the abstract machine and enjoys the standard *noninterference* property that "high inputs do not influence low outputs." The interplay of the concrete machine and fault handler is complex, so some proof abstraction is essential. In our proof architecture, a first abstraction layer is based on *refinement*. This allows us to reason in terms of a high-level view of memory, ignoring the concrete implementation of IFC labels, while setting up the intricate indistinguishability relation used in the noninterference proof. A second layer of abstraction is required for reasoning about the correctness of the fault handler. Here, we rely on a verified custom Hoare logic that abstracts from low-level machine instructions into a reusable set of verified structured code generators.

In §7 we prove that the IFC fault handler generator correctly compiles a symbolic IFC rule table and a concrete representation of an abstract label lattice into an appropriate sequence of machine instructions. We then introduce a standard notion of refinement (§8) and show that the concrete machine running the generated IFC fault handler refines the abstract IFC machine and vice-versa, using the symbolic IFC rule machine as an intermediate refinement point in each direction of the proof (§9). In our deterministic setting, showing refinement in both directions guarantees that the concrete machine does not diverge or get stuck when handling a fault. We next introduce a standard *termination-insensitive noninterference (TINI)* property (§10) and show that it holds for the abstract machine. Since deterministic TINI is preserved by refinement, we conclude that the concrete machine running the generated IFC fault handler also satisfies TINI. Finally, we explain how to accommodate two important features that are handled by our Coq development but elided from the foregoing sections: dynamic memory allocation and tags representing sets of principals (§11). We close with a survey of related work (§12) and a discussion of future directions (§13). We omit proofs and some parts of longer definitions; a long version and a Coq script formalizing the entire development are available at `http://www.crash-safe.org`.

2. Overview of SAFE

To establish context, we begin with a brief overview of the full SAFE system, concentrating on its OS- and hardware-level features. More detailed descriptions can be found elsewhere [14, 16, 17, 21, 22, 29, 34].

SAFE's system software performs process scheduling, stream-based interprocess communication, storage allocation and garbage collection, and management of the low-level tagging hardware (the focus of this paper). The goal is to organize these services as a collection of mutually suspicious compartments following the principle of least privilege (a *zero-kernel OS* [43]), so that an attacker would need to compromise multiple compartments to gain complete control of the machine. It is programmed in a combination of assembly and *Tempest*, a new low-level programming language.

The SAFE hardware integrates a number of mechanisms for eliminating common vulnerabilities and supporting higher-level security primitives. To begin with, SAFE is (dynamically) typed at the hardware level: each data word is indelibly marked as a number, an instruction, a pointer, etc. Next, the hardware is memory safe: every pointer consists of a triple of base, bounds, and offset (compactly encoded into 64 bits [17, 29]), and every pointer operation includes a hardware bounds check [29]. Finally, the hardware associates each word in the registers and memory, as well as the PC, with a large (59-bit) tag. The hardware rule cache, enabling software-specified propagation of tags from operands to result on each machine step, is implemented using a combination of multiple hash functions to approximate a fully-associative cache [16].

An unusual feature of the SAFE design is that formal modeling and verification of its core mechanisms have played a central role in the design process since the beginning. The long-term goal—formally specifying and verifying the entire set of critical runtime services—is still some ways in the future, but key properties of simplified models have been verified both at the level of *Breeze* [21] (a mostly functional, security-oriented, dynamic language used for user-level programming on SAFE) and, in the present work, at the hardware and abstract machine level. Experiments are also underway to use random testing of properties like noninterference as a means to speed the design process [22].

Our goal in this paper is to develop a clear, precise, and mathematically tractable model of one of the main innovations in the SAFE design: its scheme for efficiently supporting high-level data use policies using a combination of hardware and low-level system software. To make the model easy to work with, we simplify away many important facets of the real SAFE system. In particular, (i) we focus only on IFC and noninterference, although the tagging facilities of the SAFE machine are generic and can be applied to other policies (we return to this point in §13); (ii) we ignore the Breeze and Tempest programming languages and concentrate on the hardware and runtime services; (iii) we use a stack instead of registers, and we distill the instruction set to just a handful of opcodes; (iv) we drop SAFE's fine-grained privilege separation in favor of a more conventional user-mode / kernel-mode dichotomy; (v) we shrink the rule cache to a single entry (avoiding issues of replacement and eviction) and maintain it in kernel memory, accessed by ordinary loads and stores, rather than in specialized cache hardware; (vi) we omit a large number of IFC-related concepts (dynamic principals, downgrading, public labels, integrity, clearance, etc.); (vii) we handle exceptional conditions, including potential security violations, by simply halting the whole machine; and (viii) most importantly, we ignore concurrency, process scheduling, and interprocess communication, assuming instead that the whole machine has a single, deterministic thread of control. The absence

instr	::=		Basic instruction set
	|	Add	addition
	|	Output	output top of stack
	|	Push n	push integer constant
	|	Load	indirect load from data memory
	|	Store	indirect store to data memory
	|	Jump	unconditional indirect jump
	|	Bnz n	conditional relative jump
	|	Call	indirect call
	|	Ret	return

Figure 1. Instruction set

of concurrency is a particularly significant simplification, given that we are talking about an operating system that offers IFC as a service. However, we conjecture that it may be possible to add concurrency to our formalization, while maintaining a high degree of determinism, by adapting the approach used in the proof of noninterference for the seL4 microkernel [35, 36]. We return to this point in §13.

3. Abstract IFC Machine

We begin the technical development by defining a very simple stack-and-pointer machine with "hard-wired" dynamic IFC. This machine concisely embodies the IFC mechanism we want to provide to higher-level software and serves as a specification for the symbolic IFC rule machine (§4) and for the concrete machine (§5) running our IFC fault handler (§6). The three machines share a tiny instruction set (Fig. 1) designed to be a convenient target for compiling the symbolic IFC rule table (the Coq development formalizes several other instructions). All three machines use a fixed *instruction memory* ι, a partial function from addresses to instructions.

The machine manipulates integers (ranged over by n, m, and p); unlike the real SAFE machine, we make no distinction between raw integers and pointers (we re-introduce this distinction in §11). Each integer is protected by an individual IFC *label* (ranged over by L). We assume an arbitrary set of labels $\mathcal{L}$ equipped with a partial order ($\leq$), a least upper bound operation ($\vee$), and a bottom element ($\bot$). For instance we might take $\mathcal{L}$ to be the set of levels $\{\bot, \top\}$ with $\bot \leq \top$ and $\bot \vee \top = \top$. We call a pair of an integer n and its protecting label L an *atom*, written $n@L$ and ranged over by a.

An *abstract machine state* $\langle \mu\ [\sigma]\ pc \rangle$ consists of a data memory μ, a stack σ, and a program counter pc. (We sometimes drop the outer brackets.) The *data memory* μ is a partial function from integer addresses to atoms. We write $\mu(p) \leftarrow a$ for the memory that coincides with μ everywhere except at p, where its value is a. The *stack* σ is essentially a list of atoms, but we distinguish stacks beginning with return addresses (written $pc; \sigma$) from ones beginning with regular atoms (written a, σ). The *program counter* (PC) pc is an atom whose label is used to track implicit flows, as explained below.

The step relation of the abstract machine, written $\iota \vdash \mu_1\ [\sigma_1]\ pc_1 \xrightarrow{\alpha} \mu_2\ [\sigma_2]\ pc_2$, is a partial function taking a machine state to a machine state plus an output action α, which can be either an atom or the silent action τ. We generally omit ι from transitions because it is fixed. Throughout the paper we study other, similar relations, and consistently refer to non-silent actions as *events* (ranged over by e).

The stepping rules in Fig. 2 adapt a standard purely dynamic IFC enforcement mechanism [3, 39] to a low-level machine, following recent work by Hriţcu et al. [22]. The rule for Add joins ($\vee$) the labels of the two operands to produce the label of the result, which ensures that the result is at least as classified as each of the operands. The rule for Push labels the integer constant added to the stack as public ($\bot$). The rule for Jump uses join to raise the label

$$\frac{\iota(n) = \mathsf{Add}}{\mu\ \ [n_1@L_1, n_2@L_2, \sigma]\ \ n@L_{pc} \xrightarrow{\tau} \mu\ \ [(n_1{+}n_2)@(L_1 \vee L_2), \sigma]\ \ (n{+}1)@L_{pc}}$$

$$\frac{\iota(n) = \mathsf{Output}}{\mu\ [m@L_1, \sigma]\ n@L_{pc} \xrightarrow{m@(L_1 \vee L_{pc})} \mu\ [\sigma]\ (n{+}1)@L_{pc}}$$

$$\frac{\iota(n) = \mathsf{Push}\ m}{\mu\ [\sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [m@\bot, \sigma]\ (n{+}1)@L_{pc}}$$

$$\frac{\iota(n) = \mathsf{Load} \qquad \mu(p) = m@L_2}{\mu\ [p@L_1, \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [m@(L_1 \vee L_2), \sigma]\ (n{+}1)@L_{pc}}$$

$$\frac{\iota(n) = \mathsf{Store} \qquad \mu(p) = k@L_3 \qquad L_1 \vee L_{pc} \leq L_3 \qquad \mu(p) \leftarrow (m@L_1 \vee L_2 \vee L_{pc}) = \mu'}{\mu\ [p@L_1, m@L_2, \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu'\ [\sigma]\ (n{+}1)@L_{pc}}$$

$$\frac{\iota(n) = \mathsf{Jump}}{\mu\ [n'@L_1, \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [\sigma]\ n'@(L_1 \vee L_{pc})}$$

$$\frac{\iota(n) = \mathsf{Bnz}\ k \qquad n' = n{+}((m = 0)?1 : k)}{\mu\ [m@L_1, \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [\sigma]\ n'@(L_1 \vee L_{pc})}$$

$$\frac{\iota(n) = \mathsf{Call}}{\mu\ [n'@L_1, a, \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [a, (n{+}1)@L_{pc}; \sigma]\ n'@(L_1 \vee L_{pc})}$$

$$\frac{\iota(n) = \mathsf{Ret}}{\mu\ [n'@L_1; \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [\sigma]\ n'@L_1}$$

Figure 2. Semantics of IFC abstract machine

of the PC by the label of the target address of the jump. Similarly, Bnz raises the label of the PC by the label of the tested integer. In both cases the value of the PC after the instruction depends on data that could be secret, and we use the label of the PC to track the label of data that has influenced control flow. In order to prevent *implicit flows* (leaks exploiting the control flow of the program), the Store rule joins the PC label with the original label of the written integer and with the label of the pointer through which the write happens. Additionally, since the labels of memory locations are allowed to vary during execution, we prevent leaking information via labels using a "no-sensitive-upgrade" check [3, 48] (the $\leq$ precondition in the rule for Store). This check prevents memory locations labeled public from being overwritten when either the PC or the pointer through which the store happens have been influenced by secrets. The Output rule labels the emitted integer with the join of its original label and the current PC label.[1] Finally, because of the structured control flow imposed by the stack discipline, the rule for Ret can soundly restore the PC label to whatever it was at the time of the Call. (Readers less familiar with the intricacies of dynamic IFC may find some of these side conditions a bit mysterious. A longer explanation can be found in [22], but the details are not critical for present purposes.)

All data in the machine's initial state are labelled (as in all machine states), and the simple machine manages labels to ensure noninterference as defined and proved in §10. There are no instructions that explicitly raise the label (classification) of an atom. Such an instruction, joinP, is added to the machine in §11.

[1] We assume the observer of the events generated by Output is constrained by the rules of information flow—i.e., cannot freely "look inside" bare events. In the real SAFE machine, atoms being sent to the outside world need to be protected cryptographically; we are abstracting this away.

opcode	*allow*	e_{rpc}	e_r
add	TRUE	$\texttt{LAB}_{pc}$	$\texttt{LAB}_1 \sqcup \texttt{LAB}_2$
output	TRUE	$\texttt{LAB}_{pc}$	$\texttt{LAB}_1 \sqcup \texttt{LAB}_{pc}$
push	TRUE	$\texttt{LAB}_{pc}$	BOT
load	TRUE	$\texttt{LAB}_{pc}$	$\texttt{LAB}_1 \sqcup \texttt{LAB}_2$
store	$\texttt{LAB}_1 \sqcup \texttt{LAB}_{pc} \sqsubseteq \texttt{LAB}_3$	$\texttt{LAB}_{pc}$	$\texttt{LAB}_1 \sqcup \texttt{LAB}_2 \sqcup \texttt{LAB}_{pc}$
jump	TRUE	$\texttt{LAB}_1 \sqcup \texttt{LAB}_{pc}$	--
bnz	TRUE	$\texttt{LAB}_1 \sqcup \texttt{LAB}_{pc}$	--
call	TRUE	$\texttt{LAB}_1 \sqcup \texttt{LAB}_{pc}$	$\texttt{LAB}_{pc}$
ret	TRUE	$\texttt{LAB}_1$	--

Figure 3. Rule table $\mathcal{R}^{\mathsf{abs}}$ corresponding to abstract IFC machine

4. Symbolic IFC Rule Machine

In the abstract machine described above, IFC is tightly integrated into the step relation in the form of side conditions on each instruction. In contrast, the concrete machine (i.e., the "hardware") described in §5 is generic, designed to support a wide range of software-defined policies (IFC and other). The machine introduced in this section serves as a bridge between these two models. It is closer to the abstract machine—indeed, its machine states and the behavior of the step relation are identical. The important difference lies in the *definition* of the step relation, where all the IFC-related aspects are factored out into a separate judgment. While factoring out IFC enforcement into a separate reference monitor is commonplace [2, 39, 41], our approach goes further. We define a small DSL for describing symbolic IFC rules and obtain actual monitors by interpreting this DSL (in this section) and by compiling it into machine instructions using verified structured code generators (in §6 and §7).

More formally, each stepping rule of the new machine includes a uniform call to an *IFC enforcement* relation, which itself is parameterized by a *symbolic IFC rule table* $\mathcal{R}$. Given the labels of the values relevant to an instruction, the IFC enforcement relation (i) checks whether the execution of that instruction is allowed in the current configuration, and (ii) if so, yields the labels to put on the resulting PC and on any resulting value. This judgment has the form $\vdash_{\mathcal{R}} (L_{pc}, \ell_1, \ell_2, \ell_3) \rightsquigarrow_{opcode} L_{rpc}, L_r$, where $\mathcal{R}$ is the rule table and *opcode* is the kind of instruction currently executing.

For example, the stepping rule for Add

$$\frac{\iota(n) = \mathsf{Add} \qquad \vdash_{\mathcal{R}} (L_{pc}, L_1, L_2, \_) \rightsquigarrow_{\mathsf{add}} L_{rpc}, L_r}{\begin{array}{l}\mu\ [n_1@L_1, n_2@L_2, \sigma]\ \ n@L_{pc} \\ \mu\ \ [(n_1{+}n_2)@L_r, \sigma]\ \ (n{+}1)@L_{rpc}\end{array}} \xrightarrow{\tau}$$

passes three inputs to the IFC enforcement judgment: L_{pc}, the label of the current PC, and L_1 and L_2, the labels of the two operands at the top of the stack. (The fourth element of the input tuple is written as _ because it is not needed for Add.) The IFC enforcement judgment produces two labels: L_{rpc} is used to label the next program counter $(n + 1)$ and L_r is used to label the result value. All the other stepping rules follow a similar scheme. (The one for Store uses all four input labels.)

A symbolic IFC rule table $\mathcal{R}$ describes a particular IFC enforcement mechanism. For instance, the rule table $\mathcal{R}^{\mathsf{abs}}$ corresponding to the IFC mechanism of the abstract machine is shown in Fig. 3. In general, a table $\mathcal{R}$ associates a *symbolic IFC rule* to each instruction opcode (formally, $\mathcal{R}$ is a total function). Each of these rules is formed of three symbolic expressions: (i) a boolean expression indicating whether the execution of the instruction is allowed or not (i.e., whether it violates the IFC enforcement mechanism); (ii) a label-valued expression for L_{rpc}, the label of the next PC; and (iii) a label-valued expression for L_r, the label of the result value, if there is one.

These symbolic expressions are written in a simple domain-specific language (DSL) of operations over an IFC lattice. The grammar of this DSL includes label variables $\texttt{LAB}_{pc}, \ldots, \texttt{LAB}_3$, which correspond to the input labels $L_{pc}, \ldots, L_3$; the constant BOT; and the lattice operators $\sqcup$ (join) and $\sqsubseteq$ (flows).

The IFC enforcement judgment looks up the corresponding symbolic IFC rule in the table and directly *evaluates* the symbolic expressions in terms of the corresponding lattice operations. The definition of this interpreter is completely straightforward; we omit it for brevity. In contrast, in §6 we *compile* this rule table into the IFC fault handler for the concrete machine.

5. Concrete Machine

The concrete machine provides low-level support for efficiently implementing many different high-level policies (IFC and others) with a combination of a hardware rule cache and a software cache fault handler. In this section we focus on the concrete machine's hardware, which is completely generic, while in §6 we describe a specific fault handler corresponding to the IFC rules of the symbolic rule machine.

The concrete machine has the same general structure as the more abstract ones, but differs in several important respects. One is that it annotates data values with integer *tags* T, rather than with *labels* L from an abstract lattice; thus, the *concrete atoms* a in the data memories and the stack have the form $n@\texttt{T}$. Similarly, a *concrete action* α is either a concrete atom or the silent action τ. Using plain integers as tags allows us to delegate their interpretation entirely to software. In this paper we focus solely on using tags to implement IFC labels, although they could also be used for enforcing other policies, such as type and memory safety or control-flow integrity. For instance, to implement the two-point abstract lattice with $\bot \leq \top$, we could use 0 to represent $\bot$ and 1 to represent $\top$, making the operations $\vee$ and $\leq$ easy to implement (see §6). For richer abstract lattices, a more complex concrete representation might be needed; for example, a label containing an arbitrary set of principals might be represented concretely by a pointer to an array data structure (see §11). In places where a tag is needed but its value is irrelevant, the concrete machine uses a specific but arbitrary *default tag* value (e.g., -1), which we write $\texttt{T}_\texttt{D}$.

A second important difference is that the concrete machine has two modes: *user mode* (u), for executing the ordinary user program, and *kernel mode* (k), for handling rule cache faults. To support these two modes, the concrete machine's state contains a *privilege bit* π, a separate *kernel instruction memory* ϕ, and a separate *kernel data memory* κ, in addition to the user instruction memory ι, the user data memory μ, the stack σ, and the PC. When the machine is operating in user mode ($\pi = \mathsf{u}$), instructions are looked up using the PC as an index into ι, and loads and stores use μ; when in kernel mode ($\pi = \mathsf{k}$), the PC is treated as an index into ϕ, and loads and stores use κ. As before, since ι and ϕ are fixed, we normally leave them implicit.

The concrete machine has the same instruction set as the previous ones, allowing user programs to be run on all three machines unchanged. But the tag-related semantics of instructions depends on the privilege mode, and in user mode the semantics further depends on the state of the *rule cache*. In the real SAFE machine, the rule cache may contain thousands of entries and is implemented as a separate near-associative memory [16] accessed by special instructions. Here, for simplicity, we use a cache with just one entry, located at the start of kernel memory, and use Load and Store instructions to manipulate it; indeed, until §11, it constitutes the entirety of κ.

The rule cache holds a single rule instance, represented graphically like this: | *opcode* | $\texttt{T}_{pc}$ | $\texttt{T}_1$ | $\texttt{T}_2$ | $\texttt{T}_3$ | $\texttt{T}_{rpc}$ | $\texttt{T}_r$ |. Location 0 holds an integer representing an opcode. Location 1 holds the PC tag. Locations 2 to 4 hold the tags of any other arguments needed by this particular opcode. Location 5 holds the tag that should go

on the PC after this instruction executes, and location 6 holds the tag for the instruction's result value, if needed. For example, suppose the cache contains $\boxed{\mathsf{add}\,|\,0\,|\,1\,|\,1\,|\,\text{-}1\,||\,0\,|\,1}$. (Note that we are showing just the "payload" part of these seven atoms; by convention, the tag part is always $\mathtt{T_D}$, and we do not display it.) If 0 is the tag representing the label $\bot$, 1 represents $\top$, and -1 is the default tag $\mathtt{T_D}$, this can be interpreted abstractly as follows: "If the next instruction is Add, the PC is labeled $\bot$, and the two relevant arguments are both labeled $\top$, then the instruction should be allowed, the label on the new PC should be $\bot$, and the label on the result of the operation is $\top$."

There are two sets of stepping rules for the concrete machine in user mode; which set applies depends on whether the current machine state matches the current contents of the rule cache. In the "cache hit" case the instruction executes normally, with the cache's output determining the new PC tag and result tag (if any). In the "cache miss" case, the relevant parts of the current state (opcode, PC tag, argument tags) are stored into the input part of the single cache line and the machine simulates a Call to the fault handler.

To see how this works in more detail, consider the two user-mode stepping rules for the Add instruction.

$$\frac{\iota(n) = \mathsf{Add} \qquad \kappa = \boxed{\mathsf{add}\,|\,\mathtt{T}_{pc}\,|\,\mathtt{T}_1\,|\,\mathtt{T}_2\,|\,\mathtt{T_D}\,||\,\mathtt{T}_{rpc}\,|\,\mathtt{T}_r}}{\begin{array}{l}\mathsf{u}\ \kappa\ \mu\ [n_1@\mathtt{T}_1, n_2@\mathtt{T}_2, \sigma]\ \ n@\mathtt{T}_{pc} \xrightarrow{\tau} \\ \mathsf{u}\ \kappa\ \mu\ \ [(n_1{+}n_2)@\mathtt{T}_r, \sigma]\ \ n{+}1@\mathtt{T}_{rpc}\end{array}}$$

$$\frac{\iota(n) = \mathsf{Add} \qquad \kappa_i \neq \boxed{\mathsf{add}\,|\,\mathtt{T}_{pc}\,|\,\mathtt{T}_1\,|\,\mathtt{T}_2\,|\,\mathtt{T_D}} = \kappa_j}{\begin{array}{l}\mathsf{u}\ [\kappa_i, \kappa_o]\ \mu\ \ [n_1@\mathtt{T}_1, n_2@\mathtt{T}_2, \sigma]\ \ n@\mathtt{T}_{pc} \xrightarrow{\tau} \\ \mathsf{k}\ [\kappa_j, \kappa_\mathtt{D}]\ \mu\ [(n@\mathtt{T}_{pc}, \mathsf{u}); n_1@\mathtt{T}_1, n_2@\mathtt{T}_2, \sigma]\ \ 0@\mathtt{T_D}\end{array}}$$

In the first rule (cache hit), the side condition demands that the input part of the current cache contents have form $\boxed{\mathsf{add}\,|\,\mathtt{T}_{pc}\,|\,\mathtt{T}_1\,|\,\mathtt{T}_2\,|\,\mathtt{T_D}}$, where $\mathtt{T}_{pc}$ is the tag on the current PC, $\mathtt{T}_1$ and $\mathtt{T}_2$ are the tags on the top two atoms on the stack, and the fourth element is the default tag. In this case, the output part of the rule, $\boxed{\mathtt{T}_{rpc}\,|\,\mathtt{T}_r}$, determines the tag $\mathtt{T}_{rpc}$ on the PC and the tag $\mathtt{T}_r$ on the new atom pushed onto the stack in the next machine state.

In the second rule (cache miss), the notation $[\kappa_i, \kappa_o]$ means "let κ_i be the input part of the current rule cache and κ_o be the output part." The side condition says that the current input part κ_i does not have the desired form $\boxed{\mathsf{add}\,|\,\mathtt{T}_{pc}\,|\,\mathtt{T}_1\,|\,\mathtt{T}_2\,|\,\mathtt{T_D}}$, so the machine needs to enter the fault handler. The next machine state is formed as follows: (i) the input part of the cache is set to the desired form κ_j and the output part is set to $\kappa_\mathtt{D} \triangleq \boxed{\mathtt{T_D}\,|\,\mathtt{T_D}}$; (ii) a new return frame is pushed on top of the stack to remember the current PC and privilege bit (u); (iii) the privilege bit is set to k (which will cause the next instruction to be read from the kernel instruction memory); and (iv) the PC is set to 0, the location in the kernel instruction memory where the fault handler routine begins.

What happens next is up to the fault handler code. Its job is to examine the contents of the first five kernel memory locations and either (i) write appropriate tags for the result and new PC into the sixth and seventh kernel memory locations and then perform a Ret to go back to user mode and restart the faulting instruction, or (ii) stop the machine by jumping to an invalid PC (-1) to signal that the attempted combination of opcode and argument tags is illegal. This mechanism is general and can be used to implement many different high-level policies (IFC and others).

In kernel mode, the treatment of tags is almost completely degenerate: to avoid infinite regress, the concrete machine does not consult the rule cache while in kernel mode. For most instructions, tags read from the current machine state are ignored (indicated by _) and tags written to the new state are set to $\mathtt{T_D}$. This can be seen for instance in the kernel-mode step rule for addition

$$\frac{\phi(n) = \mathsf{Add}}{\begin{array}{l}\mathsf{k}\ \kappa\ \mu\ \ [n_1@\_, n_2@\_, \sigma]\ \ n@\_ \xrightarrow{\tau} \\ \mathsf{k}\ \kappa\ \mu\ [(n_1{+}n_2)@\mathtt{T_D}, \sigma]\ \ n{+}1@\mathtt{T_D}\end{array}}$$

The only significant exception to this pattern is Ret, which takes both the privilege bit and the new PC (including its tag!) from the return frame at the top of the stack. This is critical, since a Ret instruction is used to return from kernel to user mode when the fault handler has finished executing.

$$\frac{\phi(n) = \mathsf{Ret}}{\mathsf{k}\ \kappa\ \mu\ [(n'@\mathtt{T}_1, \pi); \sigma]\ \ n@\_ \xrightarrow{\tau} \ \pi\ \kappa\ \mu\ [\sigma]\ \ n'@\mathtt{T}_1}$$

A final point is that Output is not permitted in kernel mode, which guarantees that kernel actions are always the silent action τ.

6. Fault Handler for IFC

Now we assemble the pieces. A concrete IFC machine implementing the symbolic rule machine defined in §4 can be obtained by installing appropriate fault handler code in the kernel instruction memory of the concrete machine presented in §5. In essence, this handler must emulate how the symbolic rule machine looks up and evaluates the DSL expressions in a given IFC rule table. We choose to generate the handler code by compiling the lookup and DSL evaluation relations directly into machine code. (An alternative would be to represent the rule table as abstract syntax in the kernel memory and write an interpreter in machine code for the DSL, but the compilation approach seems to lead to simpler code and proofs.)

The handler compilation scheme is given (in part) in Fig. 4. Each gen$*$ function generates a list of concrete machine instructions; the sequence generated by the top-level genFaultHandler is intended to be installed starting at location 0 in the concrete machine's kernel instruction memory. The implicit addr$*$ parameters are symbolic names for the locations of the opcode and various tags in the concrete machine's rule cache, as described in §5. The entire generator is parameterized by an arbitrary rule table $\mathcal{R}$. We make heavy use of the (obvious) encoding of booleans where false is represented by 0 and true by any non-zero value. We omit the straightforward definitions of some of the leaf generators.

The top-level handler works in three phases. The first phase, genComputeResults, does most of the work: it consists of a large nested if-then-else chain, built using genIndexedCases, that compares the opcode of the faulting instruction against each possible opcode and, on a match, executes the code generated for the corresponding symbolic IFC rule. The code generated for each symbolic IFC rule (by genApplyRule) pushes its results onto the stack: a flag indicating whether the instruction is allowed and, if so, the result-PC and result-value tags. This first phase never writes to memory or transfers control outside the handler; this makes it fairly easy to prove correct.

The second phase, genStoreResults, reads the computed results off the stack and updates the rule cache appropriately. If the result indicates that the instruction is allowed, the result PC and value tags are written to the cache, and true is pushed on the stack; otherwise, nothing is written to the cache, and false is pushed on the stack.

The third and final phase of the top-level handler tests the boolean just pushed onto the stack and either returns to user code (instruction is allowed) or jumps to address -1 (disallowed).

The code for symbolic rule compilation is built by straightforward recursive traversal of the rule DSL syntax for label-valued expressions (genELab) and boolean-valued expressions (genBool). These functions are (implicitly) parameterized by lattice-specific generators genBot, genJoin, and genFlows. To implement these

```
genFaultHandler R = genComputeResults R ++
                    genStoreResults ++
                    genIf [Ret] [Push (-1); Jump]

genComputeResults R =
  genIndexedCases [] genMatchOp (genApplyRule ∘ Rule_R) opcodes

genMatchOp op =
  [Push op] ++ genLoadFrom addrOpLabel ++ genEqual

genApplyRule ⟨allow, e_rpc, e_r⟩ = genBool allow ++
  genIf (genSome (genELab e_rpc ++ genELab e_r)) genNone

genELab BOT          = genBot
        LAB_i        = genLoadFrom addrTag_i
        LE_1 ⊔ LE_2  = genELab LE_2 ++ genELab LE_1 ++ genJoin

genBool TRUE         = genTrue
        LE_1 ⊑ LE_2  = genELab LE_2 ++ genELab LE_1 ++ genFlows

genStoreResults =
  genIf (genStoreAt addrTag_r ++ genStoreAt addrTag_rpc ++ genTrue)
        genFalse

genIndexedCases genDefault genGuard genBody = g
   where g nil       = genDefault
         g (n :: ns) = genGuard n ++ genIf (genBody n) (g ns)

genIf t f      =  genSkipIf (length f') ++ f' ++ t
                    where f' = f ++ genSkip(length t)
genSkip n      =  genTrue ++ genSkipIf n
genSkipIf n    =  [Bnz (n+1)]

opcodes = add :: output :: ... :: ret :: nil
```

Figure 4. Generation of fault handler from IFC rule table.

generators for a particular lattice, we first need to choose how to represent abstract labels as integer tags, and then determine a sequence of instructions that encodes each operation. We call such an encoding scheme a *concrete lattice*. For example, the abstract labels in the two-point lattice can be encoded like booleans, representing $\bot$ by 0, $\top$ by non-0, and instantiating genBot, genJoin, and genFlows with code for computing false, disjunction, and implication, respectively. A simple concrete lattice like this can be formalized as a tuple $CL = (\mathsf{Tag}, \mathsf{Lab}, \mathsf{genBot}, \mathsf{genJoin}, \mathsf{genFlows})$, where the encoding and decoding functions Lab and Tag satisfy $\mathsf{Lab} \circ \mathsf{Tag} = id$; to streamline the exposition, we assume this form of concrete lattice for most of the paper. The more realistic encoding in §11 will require a more complex treatment.

To raise the level of abstraction of the handler code, we make heavy use of structured code generators; this makes it easier both to understand the code and to prove it correct using a custom Hoare logic that follows the structure of the generators (see §7). For example, the genIf function takes two code sequences, representing the "then" and "else" branches of a conditional, and generates code to test the top of the stack and dispatch control appropriately. The higher-order generator genIndexedCases takes a list of integer indices (e.g., opcodes) and functions for generating guards and branch bodies from an index, and generates code that will run the guards in order until one of them computes true, at which point the corresponding branch body is run.

7. Correctness of the Fault Handler Generator

We now turn our attention to verification, beginning with the fault handler. We must show that the generated fault handler emulates the IFC enforcement judgment $\vdash_{\mathcal{R}} (L_{pc}, \ell_1, \ell_2, \ell_3) \rightsquigarrow_{opcode} L_{rpc}, L_r$ of the symbolic rule machine. The statement and proof of correctness are parametric over the symbolic IFC rule table $\mathcal{R}$ and concrete lattice, and hence over correctness lemmas for the lattice operations.

Correctness statement Let $\mathcal{R}$ be an arbitrary rule table and $\phi_{\mathcal{R}} \triangleq$ genFaultHandler $\mathcal{R}$ be the corresponding generated fault handler. We specify how $\phi_{\mathcal{R}}$ behaves as a whole—as a relation between initial state on entry and final state on completion—using the relation $\phi \vdash cs_1 \rightarrow_{\mathsf{k}}^{\star} cs_2$, defined as the reflexive transitive closure of the concrete step relation, with the constraints that the fault handler code is ϕ and all intermediate states (i.e., strictly preceding cs_2) have privilege bit k.

The correctness statement is captured by the following two lemmas. Intuitively, if the symbolic IFC enforcement judgment allows some given user instruction, then executing $\phi_{\mathcal{R}}$ (stored at kernel mode location 0) updates the cache to contain the tag encoding of the appropriate result labels and returns to user-mode; otherwise, $\phi_{\mathcal{R}}$ halts the machine (pc = -1).

Lemma 7.1 (Fault handler correctness, allowed case)**.** Suppose that $\vdash_{\mathcal{R}} (L_{pc}, \ell_1, \ell_2, \ell_3) \rightsquigarrow_{opcode} L_{rpc}, L_r$ and

$$\kappa_i = \boxed{opcode \mid \mathsf{Tag}(L_{pc}) \mid \mathsf{Tag}(\ell_1) \mid \mathsf{Tag}(\ell_2) \mid \mathsf{Tag}(\ell_3)}\,.$$

Then $\phi_{\mathcal{R}} \vdash \begin{array}{llll} \langle \mathsf{k} & [\kappa_i, \kappa_o] & \mu & [(pc, \mathsf{u}); \sigma] \quad 0@\mathtt{T_D} \rangle \\ \langle \mathsf{u} & [\kappa_i, \kappa_o'] & \mu & [\sigma] \qquad\quad pc \rangle \end{array} \rightarrow_{\mathsf{k}}^{\star}$
with output cache $\kappa_o' = (\mathsf{Tag}\,(L_{rpc}), \mathsf{Tag}\,(L_r))$.

Lemma 7.2 (Fault handler correctness, disallowed case)**.** Suppose that $\vdash_{\mathcal{R}} (L_{pc}, \ell_1, \ell_2, \ell_3) \not\rightsquigarrow_{opcode}$, and

$$\kappa_i = \boxed{opcode \mid \mathsf{Tag}(L_{pc}) \mid \mathsf{Tag}(\ell_1) \mid \mathsf{Tag}(\ell_2) \mid \mathsf{Tag}(\ell_3)}\,.$$

Then, for some final stack σ',

$$\phi_{\mathcal{R}} \vdash \begin{array}{llll} \langle \mathsf{k} & [\kappa_i, \kappa_o] & \mu & [(pc, \mathsf{u}); \sigma] \quad 0@\mathtt{T_D} \rangle \\ \langle \mathsf{k} & [\kappa_i, \kappa_o] & \mu & [\sigma'] \qquad\quad \text{-}1@\mathtt{T_D} \rangle. \end{array} \rightarrow_{\mathsf{k}}^{\star}$$

Proof methodology The fault handler is compiled by composing generators (Fig. 4); accordingly, the proofs of these two lemmas reduce to correctness proofs for the generators. We employ a custom Hoare logic for specifying the generators themselves, which makes the code generation proof simple, reusable, and scalable. This is where defining a DSL for IFC rules and a *structured* compiler proves to be very useful approach, e.g., compared to symbolic interpretation of hand-written code.

Our logic comprises two notions of Hoare triple. The generated code mostly consists of self-contained instruction sequences that terminate by "falling off the end"—i.e., that never return or jump outside themselves, although they may contain internal jumps (e.g., to implement conditionals). The only exception is the final step of the handler (third line of genFaultHandler in Fig. 4). We therefore define a standard Hoare triple $\{P\}\ c\ \{Q\}$, suitable for reasoning about self-contained code, and use it for the bulk of the proof. To specify the final handler step, we define a non-standard triple $\{P\}\ c\ \{Q\}^{O}_{pc}$ for reasoning about escaping code.

Self-contained-code Hoare triples The triple $\{P\}\ c\ \{Q\}$, where P and Q are predicates on $\kappa \times \sigma$, says that, if the kernel instruction memory ϕ contains the code sequence c starting at the current PC, and if the current memory and stack satisfy P, then the machine will run (in kernel mode) until the PC points to the instruction immediately following the sequence c, with a resulting memory and stack satisfying Q. Note that the instruction memory ϕ is unconstrained outside of c, so if c is not self-contained, no triple about it will be provable; thus, these triples obey the usual composition laws. Also, because the concrete machine is deterministic, these triples express total, rather than partial, correctness, which is essential for proving termination in lemmas 7.1 and 7.2. To aid automation of proofs about code sequences, we give triples in weakest-precondition style.

We build proofs by composing atomic specifications of individual instructions, such as

$$\frac{P(\kappa,\sigma) := \exists\, n_1\ \mathtt{T}_1\ n_2\ \mathtt{T}_2\ \sigma'.\quad \sigma = n_1@\mathtt{T}_1, n_2@\mathtt{T}_2, \sigma' \\ \qquad\qquad\qquad\qquad \wedge\ Q(\kappa, ((n_1{+}n_2)@\mathtt{T}_\mathtt{D}, \sigma'))}{\{P\}\ [\mathsf{Add}]\ \{Q\}},$$

with specifications for structured code generators, such as

$$\frac{\begin{array}{l}P(\kappa,\sigma) := \exists\, n\ \mathtt{T}\ \sigma'.\ \sigma = n@\mathtt{T}, \sigma' \wedge (n \neq 0 \implies P_1(\kappa,\sigma')) \\ \qquad\qquad\qquad\qquad \wedge (n = 0 \implies P_2(\kappa,\sigma')) \\ \{P_1\}\ c_1\ \{Q\} \qquad \{P_2\}\ c_2\ \{Q\}\end{array}}{\{P\}\ \mathsf{genIf}\ c_1\ c_2\ \{Q\}}.$$

(We emphasize that all such specifications are *verified*, not *axiomatized* as the inference rule notation might suggest.)

The concrete implementations of the lattice operations are also specified using triples in this style.

$$\frac{P(\kappa,\sigma) := Q(\kappa, (\mathsf{Tag}\ (\bot)@\mathtt{T}_\mathtt{D}, \sigma))}{\{P\}\ \mathsf{genBot}\ \{Q\}}$$

$$\frac{P(\kappa,\sigma) := \exists\, L\, L'\, \sigma'.\quad \sigma = \mathsf{Tag}\ (L)@\mathtt{T}_\mathtt{D}, \mathsf{Tag}\ (L')@\mathtt{T}_\mathtt{D}, \sigma' \\ \qquad\qquad\qquad\qquad \wedge\ Q(\kappa, \mathsf{Tag}\ (L{\vee}L')@\mathtt{T}_\mathtt{D}, \sigma')}{\{P\}\ \mathsf{genJoin}\ \{Q\}}$$

$$\frac{P(\kappa,\sigma) := \exists\, L\, L'\, \sigma.\quad \sigma = \mathsf{Tag}\ (L)@\mathtt{T}_\mathtt{D}, \mathsf{Tag}\ (L')@\mathtt{T}_\mathtt{D}, \sigma' \\ \qquad\qquad\qquad\qquad \wedge\ Q(\kappa, (\text{if } L \leq L' \text{ then } 1 \text{ else } 0)@\mathtt{T}_\mathtt{D}, \sigma')}{\{P\}\ \mathsf{genFlows}\ \{Q\}}$$

For the two-point lattice, it is easy to prove that the implemented operators satisfy these specifications; §11 describes an analogous result for a lattice of sets of principals.

Escaping-code Hoare triples To be able to specify the entire code of the generated fault handler, we also define a second form of triple, $\{P\}\ c\ \{Q\}^O_{pc}$, which specifies mostly self-contained, total code c that either makes *exactly one* jump outside of c or returns out of kernel mode. More precisely, if P and Q are predicates on $\kappa \times \sigma$ and O is a function from $\kappa \times \sigma$ to outcomes (the constants `Success` and `Failure`), then $\{P\}\ c\ \{Q\}^O_{pc}$ holds if, whenever the kernel instruction memory ϕ contains the sequence c starting at the current PC, the current cache and stack satisfy P, and

- if O computes `Success` then the machine runs (in kernel mode) until it returns to user code at pc, and Q is satisfied.
- if O computes `Failure` then the machine runs (in kernel mode) until it halts ($pc = -1$ in kernel mode), and Q is satisfied.

To compose self-contained code with escaping code, we prove two composition laws for these triples, one for pre-composing with specified self-contained code and another for post-composing with arbitrary (unreachable) code:

$$\frac{\{P_1\}\ c_1\ \{P_2\} \qquad \{P_2\}\ c_2\ \{P_3\}^O_{pc}}{\{P_1\}\ c_1{+}{+}c_2\ \{P_3\}^O_{pc}} \qquad \frac{\{P\}\ c_1\ \{Q\}^O_{pc}}{\{P\}\ c_1{+}{+}c_2\ \{Q\}^O_{pc}}$$

We use these new triples to specify the Ret and Jump instructions, which could not be given useful specifications using the self-contained-code triples, e.g.

$$\frac{\begin{array}{l}P(\kappa,\sigma) := \exists\, \sigma'.\ Q(\kappa,\sigma') \wedge \sigma = (pc, \mathsf{u}); \sigma' \\ O(\kappa,\sigma) := \mathtt{Success}\end{array}}{\{P\}\ [\mathsf{Ret}]\ \{Q\}^O_{pc}}$$

Everything comes together in verifying the fault handler. We use contained-code triples to specify everything except for [Ret], [Jump], and the final genIf, and then use the escaping-code triple composition laws to connect the non-returning part of the fault handler to the final genIf.

8. Refinement

We have two remaining verification goals. First, we want to show that the concrete machine of §5 (running the fault handler of §6 compiled from $\mathcal{R}^{\mathsf{abs}}$) enjoys TINI. Proving this directly for the concrete machine would be dauntingly complex, so instead we show that the concrete machine is an implementation of the abstract machine, for which noninterference will be much easier to prove (§10). Second, since a trivial always-diverging machine also has TINI, we want to show that the concrete machine is a *faithful* implementation of the abstract machine that emulates all its behaviors.

We phrase these two results using the notion of *machine refinement*, which we develop in this section, and which we prove in §10 to be TINI preserving. In §9, we prove a two-way refinement (one direction for each goal), between the abstract and concrete machines, via the symbolic rule machine in both directions.

From here on we sometimes mention different machines (abstract, symbolic rule, or concrete) in the same statement (e.g., when discussing refinement), and sometimes talk about machines generically (e.g., when defining TINI for all our machines); for these purposes, it is useful to define a generic notion of machine.

Definition 8.1. A *generic machine* (or just *machine*) is a 5-tuple $M = (S, E, I, \cdot \dot{\to} \cdot, \mathit{Init})$, where S is a set of *states* (ranged over by s), E is a set of *events* (ranged over by e), $\cdot \dot{\to} \cdot \subseteq S \times (E + \{\tau\}) \times S$ is a step relation, and I is a set of *input data* (ranged over by i) that can be used to build *initial states* of the machine with the function $\mathit{Init} \in I \to S$. We call $E + \{\tau\}$ the set of *actions* of M (ranged over by α).

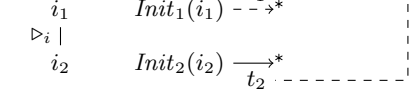
Conceptually, a machine's program is included in its input data and gets "loaded" by the function Init, which also initializes the machine memory, stack, and PC. The notion of generic machine abstracts all these details, allowing uniform definitions of refinement and TINI that apply to all three of our IFC machines. To avoid stating it several times below, we stipulate that when we instantiate Definition 8.1 to any of our IFC machines, Init must produce an initial stack with no return frames.

A generic step $s_1 \xrightarrow{e} s_2$ or $s_1 \xrightarrow{\tau} s_2$ produces event e or is silent. The reflexive-transitive closure of such steps, omitting silent steps (written $s_1 \xrightarrow{t}{}^\star\ s_2$) produces *traces*—i.e., lists, t, of events. When the end state of a step starting in state s is not relevant we write $s \xrightarrow{e}$, and similarly $s \xrightarrow{t}{}^\star$ for traces.

When relating executions of two different machines through a refinement, we establish a correspondence between their traces. This relation is usually derived from an elementary relation on events, $\triangleright \subseteq E_1 \times E_2$, which is lifted to actions and traces:

$$\begin{array}{rcl}\alpha_1\ [\triangleright]\ \alpha_2 & \triangleq & (\alpha_1 = \tau = \alpha_2\ \vee\ \alpha_1 = e_1 \triangleright e_2 = \alpha_2) \\ \vec{x}\ [\triangleright]\ \vec{y} & \triangleq & \mathsf{length}(\vec{x}) = \mathsf{length}(\vec{y}) \wedge \forall\, i.\ x_i \triangleright y_i.\end{array}$$

Definition 8.2 (Refinement)**.** Let $M_1 = (S_1, E_1, I_1, \cdot \dot{\to}_1 \cdot, \mathit{Init}_1)$ and $M_2 = (S_2, E_2, I_2, \cdot \dot{\to}_2 \cdot, \mathit{Init}_2)$ be two machines. A *refinement* of M_1 into M_2 is a pair of relations $(\triangleright_i, \triangleright_e)$, where $\triangleright_i \subseteq I_1 \times I_2$ and $\triangleright_e \subseteq E_1 \times E_2$, such that whenever $i_1 \triangleright_i i_2$ and $\mathit{Init}_2(i_2) \xrightarrow{t_2}{}^\star$, there exists a trace t_1 such that $\mathit{Init}_1(i_1) \xrightarrow{t_1}{}^\star$ and $t_1\ [\triangleright_e]\ t_2$. We also say that M_2 *refines* M_1. Graphically:



(Plain lines denote premises, dashed ones conclusions.)

In order to prove refinement, we need a variant that considers executions starting at arbitrary related states.

Definition 8.3 (Refinement via states)**.** Let M_1, M_2 be as above. A *state refinement* of M_1 into M_2 is a pair of relations $(\triangleright_s, \triangleright_e)$, where $\triangleright_s \subseteq S_1 \times S_2$ and $\triangleright_e \subseteq E_1 \times E_2$, such that, whenever $s_1 \triangleright_s s_2$ and $s_2 \xrightarrow{t_2}{}^{\star}$, there exists t_1 such that $s_1 \xrightarrow{t_1}{}^{\star}$ and $t_1\ [\triangleright_e]\ t_2$.

If the relation on inputs is compatible with the one on states, we can use state refinement to prove refinement.

Lemma 8.4. Suppose $i_1 \triangleright_i i_2 \Rightarrow \mathit{Init}_1(i_1) \triangleright_s \mathit{Init}_2(i_2)$, for all i_1 and i_2. If $(\triangleright_s, \triangleright_e)$ is a state refinement then $(\triangleright_i, \triangleright_e)$ is a refinement.

9. Refinements Between Concrete and Abstract

In this section, we show that (1) the concrete machine refines the symbolic rule machine, and (2) vice versa. Using (1) we will be able to show in §10 that the concrete machine is noninterfering. From (2) we know that the concrete machine faithfully implements the abstract one, exactly reflecting its execution traces.

Abstract and symbolic rule machines The symbolic rule machine (with the rule table $\mathcal{R}^{\mathsf{abs}}$) is a simple reformulation of the abstract machine. Their step relations are (extensionally) equal, and started from the same input data they emit the same traces.

Definition 9.1 (Abstract and symbolic rule machines as generic machines)**.** For both abstract and symbolic rule machines, input data is a 4-tuple (p, args, n, L) where p is a program, args is a list of atoms (the initial stack), and n is the size of the memory, initialized with n copies of $0@L$. The initial PC is $0@L$.

Lemma 9.2. The symbolic rule machine instantiated with the rule table $\mathcal{R}^{\mathsf{abs}}$ refines the abstract machine through $(=, =)$.

Concrete machine refines symbolic rule machine We prove this refinement using a fixed but arbitrary rule table, $\mathcal{R}$, an abstract lattice of labels, and a concrete lattice of tags. The proof uses the correctness of the fault handler (§7), so we assume that the fault handler of the concrete machine corresponds to the rule table of the symbolic rule machine ($\phi = \phi_{\mathcal{R}}$) and that the encoding of abstract labels as integer tags is correct.

Definition 9.3 (Concrete machine as generic machine)**.** The input data of the concrete machine is a 4-tuple $(p, \mathit{args}, n, \mathtt{T})$ where p is a program, args is a list of concrete atoms (the initial stack), and the initial memory is n copies of $0@\mathtt{T}$. The initial PC is $0@\mathtt{T}$. The machine starts in user mode, the cache is initialized with an illegal opcode so that the first instruction always faults, and the fault handler code parameterizing the machine is installed in the initial privileged instruction memory ϕ.

The input data and events of the symbolic rule and concrete machines are of different kinds; they are matched using relations ($\triangleright_i^c$ and $\triangleright_e^c$ respectively) stipulating that payload values should be equal and that labels should correspond to tags modulo the function Tag of the concrete lattice.

$$\frac{\mathit{args}' = \mathsf{map}\ (\lambda(n@L).\ n@\mathsf{Tag}(L))\ \mathit{args}}{(p, \mathit{args}, n, L) \triangleright_i^c (p, \mathit{args}', n, \mathsf{Tag}(L))} \qquad \frac{}{n@L \triangleright_e^c n@\mathsf{Tag}(L)}$$

Theorem 9.4. The concrete IFC machine refines the symbolic rule machine, through $(\triangleright_i^c, \triangleright_e^c)$.

We prove this theorem by a refinement via states (Lemma 9.7); this, in turn, relies on two technical lemmas (9.5 and 9.6).

The matching relation $\triangleright_s^c$ between the states of the concrete and symbolic rule machines is defined as

$$\frac{\mathcal{R} \vdash \kappa \quad \sigma_q \triangleright_\sigma \sigma_c \quad \mu_q \triangleright_m \mu_c}{\mu_q, [\sigma_q], n@L \ \triangleright_s^c\ \mathsf{u}, \kappa, \mu_c, [\sigma_c], n@\mathsf{Tag}(L)}$$

where the new notations are defined as follows. The relation $\triangleright_m$ demands that the memories be equal up to the conversion of labels to concrete tags. The relation $\triangleright_\sigma$ on stacks is similar, but additionally requires that return frames in the concrete stack have their privilege bit set to u. The basic idea is to match, in $\triangleright_s^c$, only concrete states that are in user mode. We also need to track an extra invariant, $\mathcal{R} \vdash \kappa$, which means that the cache κ is consistent with the table $\mathcal{R}$—i.e., κ never lies. More precisely, the output part of κ represents the result of applying the symbolic rule judgment of $\mathcal{R}$ to the opcode and labels represented in the input part of κ.

$$\begin{array}{l}\mathcal{R} \vdash [\kappa_i, \kappa_o] \triangleq\ \forall \mathit{opcode}\ L_1\ L_2\ L_3\ L_{pc},\\ \kappa_i = \boxed{\mathit{opcode} \mid \mathsf{Tag}(L_{pc}) \mid \mathsf{Tag}(L_1) \mid \mathsf{Tag}(L_2) \mid \mathsf{Tag}(L_3)} \Rightarrow\\ \exists L_{rpc}\ L_r, \quad \vdash_{\mathcal{R}} (L_{pc}, L_1, L_2, L_3) \rightsquigarrow_{\mathit{opcode}} L_{rpc}, L_r\\ \qquad\qquad \wedge\ \kappa_o = (\mathsf{Tag}\,(L_{rpc}), \mathsf{Tag}\,(L_r))\end{array}$$

To prove refinement via states, we must account for two situations. First, suppose the concrete machine can take a user step. In this case, we match that step with a single symbolic rule machine step. We write cs^{π} to denote a concrete state cs whose privilege bit is π.

Lemma 9.5 (Refinement, non-faulting concrete step)**.** Let cs_1^{u} be a concrete state and suppose that $cs_1^{\mathsf{u}} \xrightarrow{\alpha_c} cs_2^{\mathsf{u}}$. Let qs_1 be a symbolic rule machine state with $qs_1 \triangleright_s^c cs_1^{\mathsf{u}}$. Then there exist qs_2 and α_a such that $qs_1 \xrightarrow{\alpha_a} qs_2$, with $qs_2 \triangleright_s^c cs_2^{\mathsf{u}}$, and $\alpha_a\ [\triangleright_e^c]\ \alpha_c$.

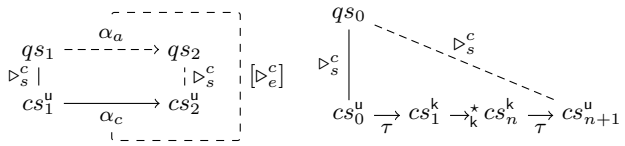
Since the concrete machine is able to make a user step, the input part of the cache must match the opcode and data of the current state. But the invariant $\mathcal{R} \vdash \kappa$ says that the corresponding symbolic rule judgment holds. Hence the symbolic rule machine can also make a step from qs_2, as required.

The second case is when the concrete machine faults into kernel mode and returns to user mode after some number of steps.

Lemma 9.6 (Refinement, faulting concrete step)**.** Let cs_0^{u} be a concrete state, and suppose that the concrete machine does a faulting step to cs_1^{k}, stays in kernel mode until cs_n^{k}, and then exits kernel mode by stepping to cs_{n+1}^{u}. Let qs_0 be a state of the symbolic rule machine that matches cs_0^{u}. Then $qs_0 \triangleright_s^c cs_{n+1}^{\mathsf{u}}$.

To prove this lemma, we must consider two cases. If the corresponding symbolic rule judgment holds, then we apply Lemma 7.1 to conclude directly—i.e., the machine exits kernel code into user mode. Otherwise, we apply Lemma 7.2 and derive a contradiction that the fault handler ends in a failing state in kernel mode.

Lemmas 9.5 and 9.6 can be summarized graphically by:



Given two matching states of the concrete and symbolic rule machines, and a concrete execution starting at that concrete state, these two lemmas can be applied repeatedly to build a matching execution of the symbolic rule machine. There is just one last case to consider, namely when the execution ends with a fault into kernel mode and never returns to user mode. However, no output is produced in this case, guaranteeing that the full trace is matched. We thus derive the following refinement via states, of which Theorem 9.4 is a corollary.

Lemma 9.7. The pair $(\triangleright_s^c, \triangleright_e^c)$ defines a refinement via states between the symbolic rule machine and the concrete machine.

Concrete machine refines abstract machine By composing the refinement of Lemma 9.2 and the refinement of Theorem 9.4 instantiated to the concrete machine running $\phi_{\mathcal{R}^{\mathsf{abs}}}$, we can conclude that the concrete machine refines the abstract one.

Abstract machine refines concrete machine The previous refinement, $(\triangleright_s^c, \triangleright_e^c)$, would also hold if the fault handler never returned when called. So, to ensure the concrete machine reflects the behaviors of the abstract machine, we next prove an inverse refinement:

Theorem 9.8. The abstract IFC machine refines the concrete IFC machine via $(\triangleright_i^{-c}, \triangleright_e^{-c})$, where $\triangleright_i^{-c}$ and $\triangleright_e^{-c}$ are the relational inverses of $\triangleright_i^c$ and $\triangleright_e^c$.

This guarantees that traces of the abstract machine are also emitted by the concrete machine. As above we use the symbolic rule machine as an intermediate step and show a state refinement of the concrete into the symbolic rule machine. We rely on the following lemma, where $\triangleright_s^{-c}$ is the inverse of $\triangleright_s^c$.

Lemma 9.9 (Forward refinement)**.** Let qs_0 and cs_0 be two states with $cs_0 \triangleright_s^{-c} qs_0$. Suppose that the symbolic rule machine takes a step $qs_0 \xrightarrow{\alpha_a} qs_1$. Then there exist concrete state cs_1 and action α_c such that $cs_0 \xrightarrow{\alpha_c}{}^{\star} cs_1$, with $cs_1 \triangleright_s^{-c} qs_1$ and $\alpha_c \; [\triangleright_e^{-c}] \; \alpha_a$.

To prove this lemma, we consider two cases. If the cache input of cs_0 matches the opcode and data of cs_0, then the concrete machine can take a step $cs_0 \xrightarrow{\alpha_c} cs_1$. Moreover, $\mathcal{R} \vdash \kappa$ in cs_0 says the cache output is consistent with the symbolic rule judgment, so the tags in α_c and cs_1 are properly related to the labels in α_a and qs_1. Otherwise, a cache fault occurs, loading the cache input and calling the fault handler. By Lemma 7.1 and the fact that $qs_0 \xrightarrow{\alpha_a} qs_1$, the cache output is computed to be consistent with $\mathcal{R}$, and this allows the concrete step as claimed.

Discussion The two top-level refinement properties (9.4 and 9.8) share the same notion of matching relations but they have been proved independently in our Coq development. In the context of compiler verification [30, 42], another proof methodology has been favored: a backward simulation proof can be obtained from a proof of forward simulation under the assumption that the lower-level machine is deterministic. (CompCertTSO [42] also requires a *receptiveness* hypothesis that trivially holds in our context.) Since our concrete machine is deterministic, we could apply a similar technique. However, unlike in compiler verification where it is common to assume that the source program has a well-defined semantics (i.e. it does not get stuck), we would have to consider the possibility that the high-level semantics (the symbolic rule machine) might block and prove that in this case either the IFC enforcement judgment is stuck (and Lemma 9.6 applies) or the current symbolic rule machine state and matching concrete state are both ill-formed.

10. Noninterference

In this section we define TINI [1, 19] for generic machines, show that the abstract machine of §3 satisfies TINI (Theorem 10.4), that TINI is preserved by refinement (Theorem 10.5), and finally, using the fact that the concrete IFC machine refines the abstract one (Theorem 9.4), that the concrete machine satisfies TINI (Theorem 10.7).

Termination-insensitive noninterference (TINI) To define noninterference, we need to talk about what can be observed about the output trace produced by a run of a machine.

Definition 10.1 (Observation)**.** A *notion of observation* for a generic machine is a 3-tuple $(\Omega, \lfloor\cdot\rfloor_\cdot, \cdot \approx_\cdot \cdot)$. Ω is a set of *observers* (i.e., different degrees of power to observe), ranged over by o. For each $o \in \Omega$, $\lfloor\cdot\rfloor_o \subseteq E$ is a predicate of *observability of events for observer o*, and $\cdot \approx_o \cdot \subseteq I \times I$ is a relation of *indistinguishability of input data for observer o*.

We write $\lfloor t \rfloor_o$ for the trace in which all unobservable events in t are filtered out using $\lfloor\cdot\rfloor_o$. We write $t_1 \approx t_2$ to say that traces t_1 and t_2 are *indistinguishable*; this truncates the longer trace to the same length as the shorter and then demands that the remaining elements be pairwise identical.

Definition 10.2 (TINI)**.** A machine $(S, E, I, \cdot \xrightarrow{\cdot} \cdot, Init)$ with a notion of observation $(\Omega, \lfloor\cdot\rfloor_\cdot, \cdot \approx_\cdot \cdot)$ satisfies TINI if, for any observer $o \in \Omega$, pair of indistinguishable initial data $i_1 \approx_o i_2$, and pair of executions $Init(i_1) \xrightarrow{t_1}{}^{\star}$ and $Init(i_2) \xrightarrow{t_2}{}^{\star}$, we have $\lfloor t_1 \rfloor_o \approx \lfloor t_2 \rfloor_o$.

Since a machine's program is part of its input data, this definition of TINI, quantified over all observers and input data, is conceptually quantified over all programs too. Because of the truncation of traces, the observer cannot detect the absence of output, i.e., it cannot distinguish between successful termination, failure with an error, or entering an infinite loop with no observable output. This TINI property is standard for a machine with output [1, 19].[2]

TINI for abstract machine

Definition 10.3 (Observation for abstract machine)**.** Let $\mathcal{L}$ be a lattice, with partial order $\leq$. Define indistinguishability of atoms, $a_1 \approx_o^a a_2$ by

$$\frac{}{a \approx_o^a a} \qquad \frac{\neg\lfloor a_1 \rfloor_o \quad \neg\lfloor a_2 \rfloor_o}{a_1 \approx_o^a a_2}. \tag{1}$$

The notion of observation is $(\mathcal{L}, \lfloor\cdot\rfloor^a, \cdot \approx_\cdot^a \cdot)$, where

$$\lfloor n@L \rfloor_o^a \triangleq L \leq o$$

$$(p, args_1, n, L) \approx_o^a (p, args_2, n, L) \triangleq args_1 \; [\approx_o^a] \; args_2.$$

(On the right-hand side of the second equation, $[\approx_o^a]$ is indistinguishability of atoms, lifted to lists.)

We prove TINI for the abstract machine using a set of standard *unwinding conditions* [18, 22]. For this we need to define indistinguishability on states, and thus also indistinguishability of stacks; this is where we encounter one subtlety. Indistinguishability of stacks is defined pointwise when the label of the PC is observable ($L_{pc} \leq o$). When the PC label is not observable, however, we only require that the stacks are pointwise related below the most recent Call from an observable state. This is necessary because the two machines run in lock step only when their PC labels are observable; they can execute completely different instructions otherwise.

Theorem 10.4. The abstract IFC machine enjoys TINI.

TINI preserved by refinement

Theorem 10.5 (TINI preservation)**.** Suppose that generic machine M_2 refines M_1 by refinement $(\triangleright_i, \triangleright_e)$ and that each machine is equipped with a notion of observation. Suppose that, for all observers o_2 of M_2, there exists an observer o_1 of M_1 such that the following compatibility conditions hold for all $e_1, e_1' \in E_1$, all $e_2, e_2' \in E_2$, and all $i_2, i_2' \in I_2$. (i) $e_1 \triangleright_e e_2 \Rightarrow (\lfloor e_1 \rfloor_{o_1} \Leftrightarrow \lfloor e_2 \rfloor_{o_2})$; (ii) $i_2 \approx_{o_2} i_2' \Rightarrow \exists\, i_1 \approx_{o_1} i_1'.\ (i_1 \triangleright_i i_2 \ \wedge \ i_1' \triangleright_i i_2')$; (iii) $(e_1 \approx_{o_1} e_1' \ \wedge \ e_1 \triangleright_e e_2 \ \wedge \ e_1' \triangleright_e e_2') \Rightarrow e_2 \approx_{o_2} e_2'$. Then, if M_1 has TINI, M_2 also has TINI.

Some formulations of noninterference are subject to the *refinement paradox* [23], in which refinements of a noninterferent system may violate noninterference. We avoid this issue by employing a strong notion of noninterference that restricts the amount of nondeterminism in the system and is thus preserved by *any* refinement (Theorem 10.5).[3] Since our abstract machine is deterministic, it is easy to show this strong notion of noninterference for it. In §13 we discuss a possible technique for generalizing to the concurrent setting while preserving a high degree of determinism.

[2] It is called "progress-insensitive noninterference" in a recent survey [19].

[3] The recent noninterference proof for the seL4 microkernel [35, 36] works similarly (see §12).

instr	::=		extensions to instruction set
	|	...	
	|	Alloc	allocate a new frame
	|	SizeOf	fetch frame size
	|	Eq	value equality
	|	SysCall *id*	system call
	|	GetOff	extract pointer offset
	|	Pack	atom from payload and tag
	|	Unpack	atom into payload and tag
	|	PushCachePtr	push cache address on stack
	|	Dup n	duplicate atom on stack
	|	Swap n	swap two data atoms on stack

Figure 5. Additional instructions for extensions

$$\frac{\iota(n) = \mathsf{Alloc} \qquad \mathsf{alloc}\ k\ (L \vee L_{pc})\ a\ \mu = (id, \mu')}{\begin{array}{l}\mu \quad [(\mathsf{Int}\ k)@L, a, \sigma]\ \ n@L_{pc} \\ \mu'\ [(\mathsf{Ptr}\ (id, 0))@L, \sigma]\ \ (n{+}1)@L_{pc}\end{array} \xrightarrow{\tau}}$$

$$\frac{\iota(n) = \mathsf{SizeOf} \qquad \mathsf{length}\ (\mu(id)) = k}{\mu\ [(\mathsf{Ptr}\ (id, o))@L, \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [(\mathsf{Int}\ k)@L, \sigma]\ (n{+}1)@L_{pc}}$$

$$\frac{\iota(n) = \mathsf{GetOff}}{\mu\ [(\mathsf{Ptr}\ (id, o))@L, \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [(\mathsf{Int}\ o)@L, \sigma]\ (n{+}1)@L_{pc}}$$

$$\frac{\iota(n) = \mathsf{Eq}}{\begin{array}{l}\mu \qquad [v_1@L_1, v_2@L_2, \sigma]\ \ n@L_{pc} \\ \mu\ [(\mathsf{Int}\ (v_1 == v_2))@(L_1 \vee L_2), \sigma]\ \ (n{+}1)@L_{pc}\end{array} \xrightarrow{\tau}}$$

$$\frac{\begin{array}{c}\iota(n) = \mathsf{SysCall}\ id \qquad T(id) = (k, f) \\ f(\sigma_1) = v@L \qquad \mathsf{length}\ (\sigma_1) = k\end{array}}{\mu\ [\sigma_1 {+\!+} \sigma_2]\ n@L_{pc} \xrightarrow{\tau} \mu\ [v@L, \sigma_2]\ (n{+}1)@L_{pc}}$$

Figure 6. Semantics of selected new abstract machine instructions

TINI for concrete machine with IFC fault handler It remains to define a notion of observation on the concrete machine, instantiating the definition of TINI for this machine. This definition refers to a concrete lattice CL, which must be a correct encoding of an abstract lattice $\mathcal{L}$: the lattice operators genBot, genJoin, and genFlows must satisfy the specifications in §7.

Definition 10.6 (Observation for the concrete machine)**.** Let $\mathcal{L}$ be an abstract lattice, and CL be correct with respect to $\mathcal{L}$. The observation for the concrete machine is $(\mathcal{L}, \lfloor\cdot\rfloor^c_\cdot, \cdot \approx^c_\cdot \cdot)$, where

$$\lfloor n@\mathtt{T} \rfloor^c_o \triangleq \mathsf{Lab}(\mathtt{T}) \leq o,$$

$$(p, args'_1, n, \mathtt{T}) \approx^c_o (p, args'_2, n, \mathtt{T}) \triangleq args_1\ [\approx^a_o]\ args_2,$$

and $args'_i = \mathsf{map}\ (\mathsf{fun}\ n@L \to n@\mathsf{Tag}(L))\ args_i$.

Finally, we prove that the backward refinement proved in §9 satisfies the compatibility constraints of Theorem 10.5, so we derive:

Theorem 10.7. The concrete IFC machine running the fault handler $\phi_{\mathcal{R}^{\mathsf{abs}}}$ satisfies TINI.

11. An Extended System

Thus far we have described our model and proof results only for a simple machine architecture and IFC discipline. Our Coq development actually works with a significantly more sophisticated model, extending the basic machine architecture with a *frame-based* memory model supporting *dynamic allocation* and a *system call* mechanism for adding special-purpose primitives. Building on these features, we define an abstract IFC machine that uses *sets of principals* as its labels and a corresponding concrete machine implementation

$$\frac{\begin{array}{c}\iota(n) = \mathsf{Alloc} \qquad \mathsf{alloc}\ k\ \mathsf{u}\ a\ \mu = (id, \mu') \\ \mu(cache) = \boxed{\mathsf{alloc}\ |\ \mathtt{T}_{pc}\ |\ \mathtt{T}_1\ |\ \mathtt{T}_\mathtt{D}\ |\ \mathtt{T}_\mathtt{D}\ ||\ \mathtt{T}_{rpc}\ |\ \mathtt{T}_r}\end{array}}{\begin{array}{l}\mathsf{u}\ \ \mu\ \ [(\mathsf{Int}\ k)@\mathtt{T}_1, a, \sigma] \qquad n@\mathtt{T}_{pc} \\ \mathsf{u}\ \ \mu'\ [(\mathsf{Ptr}\ (id, 0))@\mathtt{T}_r, \sigma]\ \ (n{+}1)@\mathtt{T}_{rpc}\end{array} \xrightarrow{\tau}}$$

$$\frac{\phi(n) = \mathsf{Alloc} \qquad \mathsf{alloc}\ k\ \mathsf{k}\ a\ \mu = (id, \mu')}{\begin{array}{l}\mathsf{k}\ \ \mu\ \ [(\mathsf{Int}\ k)@\_, a, \sigma] \qquad n@\_ \\ \mathsf{k}\ \ \mu'\ [(\mathsf{Ptr}\ (id, 0))@\mathtt{T}_\mathtt{D}, \sigma]\ \ (n{+}1)@\mathtt{T}_\mathtt{D}\end{array} \xrightarrow{\tau}}$$

$$\frac{\phi(n) = \mathsf{PushCachePtr}}{\mathsf{k}\ \mu\ [\sigma]\ n@\_ \xrightarrow{\tau} \mathsf{k}\ \mu\ [(\mathsf{Ptr}\ (cache, 0))@\mathtt{T}_\mathtt{D}, \sigma]\ (n{+}1)@\mathtt{T}_\mathtt{D}}$$

$$\frac{\phi(n) = \mathsf{Unpack}}{\mathsf{k}\ \mu\ [v_1@v_2, \sigma]\ n@\_ \xrightarrow{\tau} \mathsf{k}\ \mu\ [v_2@\mathtt{T}_\mathtt{D}, v_1@\mathtt{T}_\mathtt{D}, \sigma]\ (n{+}1)@\mathtt{T}_\mathtt{D}}$$

$$\frac{\phi(n) = \mathsf{Pack}}{\mathsf{k}\ \mu\ [v_2@\_, v_1@\_, \sigma]\ n@\_ \xrightarrow{\tau} \mathsf{k}\ \mu\ [v_1@v_2, \sigma]\ (n{+}1)@\mathtt{T}_\mathtt{D}}$$

$$\frac{\iota(n) = \mathsf{SysCall}\ id \qquad T(id) = (k, n') \qquad \mathsf{length}\ (\sigma_1) = k}{\mathsf{u}\ \mu\ [\sigma_1 {+\!+} \sigma_2]\ n@\mathtt{T} \xrightarrow{\tau} \mathsf{k}\ \mu\ [\sigma_1 {+\!+} (n{+}1@\mathtt{T}, \mathsf{u}); \sigma_2]\ n'@\mathtt{T}_\mathtt{D}}$$

Figure 7. Semantics of selected new concrete machine instructions

where tags are pointers to dynamically allocated representations of these sets. While still much less complex than the real SAFE system, this extended model shows how our basic approach can be incrementally scaled up to more realistic designs. Verifying these extensions requires no major changes to the proof architecture of the basic system, serving as evidence of its robustness.

Fig. 5 shows the new instructions supported by the extended model. Instruction PushCachePtr, Unpack, and Pack are used only by the concrete machine, for the compiled fault handler (hence they only have a kernel-mode stepping rule; they simply get stuck if executed outside kernel mode, or on an abstract machine). We also add two stack-manipulation instructions, Dup and Swap, to make programming the kernel routines more convenient. It remains true that any program for the abstract machine makes sense to run on the abstract rule machine and the concrete machine. For brevity, we detail stepping rules only for the extended abstract IFC machine (Fig. 6) and concrete machine (Fig. 7); corresponding extensions to the symbolic IFC rule machine are straightforward (we also omit rules for Dup and Swap). Individual rules are explained below.

Dynamic memory allocation High-level programming languages usually assume a structured memory model, in which independently allocated *frames* are disjoint by construction and programs cannot depend on the relative placement of frames in memory. The SAFE hardware enforces this abstraction by attaching explicit runtime types to all values, distinguishing pointers from other data. Only data marked as pointers can be used to access memory. To obtain a pointer, one must either call the (privileged) memory manager to allocate a fresh *frame* or else offset an existing pointer. In particular, it is not possible to "forge" a pointer from an integer. Each pointer also carries information about its base and bounds, and the hardware prevents it from being used to access memory outside of its frame.

Frame-based memory model In our extended system, we model the user-level view of SAFE's memory system by adding a frame-structured memory, distinguished pointers (so *values*, the payload field of atoms and the tag field of concrete atoms, can now either be an integer (Int n) or a pointer (Ptr p)), and an allocation instruction to our basic machines. We do this (nearly) uniformly at all levels of

abstraction.[4] A *pointer* is a pair $p = (id, o)$ of a frame identifier id and an offset o into that frame. In the machine state, the data memory μ is a partial function from pointers to individual storage cells that is undefined on out-of-frame pointers. By abuse of notation, μ is also a partial function from frame identifiers to frames, which are just lists of atoms.

The most important new rule of the extended abstract machine is Alloc (Fig. 6). In this machine there is a separate memory region (assumed infinite) corresponding to each label. The auxiliary function alloc in the rule for Alloc takes a size k, the label (region) at which to allocate, and a default atom a; it extends μ with a fresh frame of size k, initializing its contents to a. It returns the id of the new frame and the extended memory μ'.

IFC and memory allocation We require that the frame identifiers produced by allocation at one label not be affected by allocations at other labels; e.g., alloc might allocate sequentially in each region. Thus, indistinguishability of low atoms is just syntactic equality, preserving Definition 10.3 from the simple abstract machine, which is convenient for proving noninterference, as we explain below. We allow a program to observe frame sizes using a new SizeOf instruction, which requires tainting the result of Alloc with L, the label of the size argument. There are also new instructions Eq, for comparing two values (including pointers) for equality, and GetOff, for extracting the offset field of a pointer into an integer. However, frame ids are intuitively *abstract*: the concrete representation of frame ids is not accessible, and pointers cannot be forged or output. The extended concrete machine stepping rules for these new instructions are analogous to the abstract machine rules, with the important exception of Alloc, which is discussed below.

A few small modifications to existing instructions in the basic machine (Fig. 2) are needed to handle pointers properly. In particular: (i) Load and Store require pointer arguments and get stuck if the pointer's offset is out of range for its frame. (ii) Add takes either two integers or an integer and a pointer, where $\mathsf{Int}\, n + \mathsf{Int}\, m = \mathsf{Int}\,(n{+}m)$ and $\mathsf{Ptr}\,(id, o_1) + \mathsf{Int}\, o_2 = \mathsf{Ptr}\,(id, o_1{+}o_2)$. (iii) Output works only on integers, not pointers. Analogous modifications are needed in the concrete machine semantic rules.

Concrete allocator The extended concrete machine's semantics for Alloc differ from those of the abstract machine in one key respect. Using one region per tag would not be a realistic strategy for a concrete implementation; e.g., the number of different tags might be extremely large. Instead, we use a single region for all user-mode allocations at the concrete level. We also collapse the separate user and kernel memories from the basic concrete machine into a single memory. Since we still want to be able to distinguish user and kernel frames, we mark each frame with a privilege mode (i.e., we use two allocation regions). Fig. 7 shows the corresponding concrete stepping rule for Alloc for two cases: non-faulting user mode and kernel mode. The concrete Load and Store rules prevent dereferencing kernel pointers in user mode. The rule cache is now just a distinguished kernel frame *cache*; to access it, the fault handler uses the (privileged) PushCachePtr instruction.

Proof by refinement As before, we prove noninterference for the concrete machine by combining a proof of noninterference of the abstract machine with a two-stage proof that the concrete machine refines the abstract machine. By using this approach we avoid some well-known difficulties in proving noninterference directly for the concrete machine. In particular, when frames allocated in low and high contexts share the same region, allocations in high contexts can cause variations in the precise pointer values returned for allocations in low contexts, and these variations must be taken into account when defining the indistinguishability relation. For example, Banerjee and Naumann [4] prove noninterference by parameterizing their indistinguishability relation with a partial bijection that keeps track of indistinguishable memory addresses. Our approach, by contrast, defines pointer indistinguishability only at the abstract level, where indistinguishable low pointers are identical. This proof strategy still requires relating memory addresses when showing refinement, but this relation does not appear in the noninterference proof at the abstract level. The refinement proof itself uses a simplified form of *memory injections* [31]. The differences in the memory region structure of both machines are significant, but invisible to programs, since no information about frame ids is revealed to programs beyond what can be obtained by comparing pointers for equality. This restriction allows the refinement proof to go through straightforwardly.

System calls To support the implementation of policy-specific primitives on top of the concrete machine, we provide a new *system call* instruction. The SysCall id instruction is parameterized by a system call identifier. The step relation of each machine is now parameterized by a table T that maps system call identifiers to their implementations.

In the abstract and symbolic rule machines, a system call implementation is an arbitrary Coq function that removes a list of atoms from the top of the stack and either puts a result on top of the stack or fails, halting the machine. The system call implementation is responsible for computing the label of the result and performing any checks that are needed to ensure noninterference.

In the concrete machine, system calls are implemented by kernel routines and the call table contains the entry points of these routines in the kernel instruction memory. Executing a system call involves inserting the return address on the stack (underneath the call arguments) and jumping to the corresponding entry point. The kernel code terminates either by returning a result to the user program or by halting the machine.

This feature has no major impact on the proofs of noninterference and refinement. For noninterference, we must show that all the abstract system calls preserve indistinguishability of abstract machine states; for refinement, we show that each concrete system call correctly implements the abstract one using the machinery of §7.

Labeling with sets of principals The full SAFE machine supports dynamic creation of security principals. In the extended model, we make a first step toward dynamic principal creation by taking principals to be integers and instantiating the (parametric) lattice of labels with the lattice of finite sets of integers.[5] In this lattice, $\bot$ is $\emptyset$, $\vee$ is $\cup$, and $\leq$ is $\subseteq$. We enrich our IFC model by adding a new *classification* primitive joinP that adds a principal to an atom's label, encoded using the system call mechanism described above. The operation of joinP is given by the following derived rule, which is an instance of the SysCall rule from Fig. 6.

$$\frac{\iota(n) = \mathsf{SysCall\ joinP}}{\begin{array}{lll}\mu & [v@L_1, (\mathsf{Int}\, m)@L_2, \sigma] & n@L_{pc} \\ \mu & [v@(L_1 \vee L_2 \vee \{m\}), \sigma] & (n{+}1)@L_{pc}\end{array}} \xrightarrow{\tau}$$

At the concrete level, a tag is now a pointer to an array of principals (integers) stored in kernel memory. To keep the fault handler code simple, we do not maintain canonical representations of sets: one set may be represented by different arrays, and a given array may have duplicate elements. (As a consequence, the mapping from abstract labels to tags is no longer a function; we return

[4] It would be interesting to describe an *implementation* of the memory manager in a still-lower-level concrete machine with no built-in Alloc instruction, but we leave this as future work.

[5] This lattice is statically known, but models dynamic creation by supporting unbounded labels and having no top element.

to this point below.) Since the fault handler generator in the basic system is parametric in the underlying lattice, it doesn't require any modification. All we must do is provide concrete implementations for the appropriate lattice operations: genJoin just allocates a fresh array and concatenates both argument arrays into it; genFlows checks for array inclusion by iterating through one array and testing whether each element appears in the other; and genBot allocates a new empty array. Finally, we provide kernel code to implement joinP, which requires two new privileged instructions, Pack and Unpack (Fig. 7), to manipulate the payload and tag fields of atoms; otherwise, the implementation is similar to that of genJoin.

A more realistic system would keep canonical representations of sets and avoid unnecessary allocation in order to improve its memory footprint and tag cache usage. But even with the present simplistic approach, both the code for the lattice operations and their proofs of correctness are significantly more elaborate than for the trivial two-point lattice. In particular, we need an additional code generator to build counted loops, e.g., for computing the join of two tags.

$$\begin{array}{l}\mathsf{genFor}\ c\ = \\ \quad [\mathsf{Dup}] +\!\!+ \mathsf{genIf}\ (\mathsf{genLoop}(c +\!\!+ [\mathsf{Push}\ (-1), \mathsf{Add}]))\ [] \\ \quad \text{where } \mathsf{genLoop}\ c\ =\ c +\!\!+ [\mathsf{Dup}, \mathsf{Bnz}\ (-(\mathsf{length}\ c + 1))]\end{array}$$

Here, c is a code sequence representing the loop body, which is expected to preserve an index value on top of the stack; the generator builds code to execute that body repeatedly, decrementing the index each time until it reaches 0. The corresponding specification is

$$\frac{\begin{array}{l}P_n(\kappa,\sigma) := \exists\, \mathtt{T}\, \sigma'.\ \sigma = n@\mathtt{T}, \sigma' \ \wedge\ \mathit{Inv}(\kappa,\sigma) \\ Q_n(\kappa,\sigma) := \exists\, \mathtt{T}\, \sigma'.\quad \sigma = n@\mathtt{T}, \sigma' \\ \qquad\qquad\qquad\qquad \wedge\, \forall\, \mathtt{T}'.\ \mathit{Inv}(\kappa, ((n-1)@\mathtt{T}', \sigma')) \\ \forall\, n.\ 0 < n \Longrightarrow \{P_n\}\ c\ \{Q_n\} \\ P(\kappa,\sigma) := \exists\, n\, \mathtt{T}\, \sigma'.\ 0 \le n \ \wedge\ \sigma = n@\mathtt{T}, \sigma' \ \wedge\ \mathit{Inv}(\kappa,\sigma) \\ Q(\kappa,\sigma) := \exists\, \mathtt{T}\, \sigma'.\ \sigma = 0@\mathtt{T}, \sigma' \ \wedge\ \mathit{Inv}(\kappa,\sigma)\end{array}}{\{P\}\ \mathsf{genFor}\ c\ \{Q\}}$$

To avoid reasoning about memory updates as far as possible, we code in a style where all local context is stored on the stack and manipulated using Dup and Swap. Although the resulting code is lengthy, it is relatively easy to automate the corresponding proofs.

Stateful encoding of labels Changing the representation of tags from integers to pointers requires modifying one small part of the basic system proof. Recall that in §6 we described the encoding of labels into tags as a *pure* function Lab. To deal with the memory-dependent and non-canonical representation of sets described above, the extended system instead uses a *relation* between an abstract label, a concrete tag that encodes it, and a memory in which this tag should be interpreted.

If tags are pointers to data structures, it is crucial that these data structures remain intact as long as the tags appear in the machine state. We guarantee this by maintaining the very strong invariant that each execution of the fault handler only allocates new frames, and never modifies the contents of existing ones, except for the *cache* frame (which tags never point into). A more realistic implementation might use mutable kernel memory for other purposes and garbage collect unused tags; this would require a more complicated memory invariant.

The TINI formulation is similar in essence to the one in §10, but some subtleties arise for concrete output events, since tags in events cannot be interpreted on their own anymore. We wish to (i) keep the semantics of the concrete machine independent of high-level policies such as IFC and (ii) give a statement of noninterference that does not refer to pointers. To achieve these seemingly contradictory aims, we model an event of the concrete machine as a pair of a concrete atom plus the whole state of the kernel memory. The resulting trace of concrete events is abstracted (i.e., interpreted in terms of abstract labels) only when stating and proving TINI. This is an idealization of what happens in the real SAFE machine, where communication of labeled data with the outside world involves cryptography. Modeling this is left as future work.

12. Related Work

The SAFE design spans a number of research areas, and a comprehensive overview of related work would be huge. We focus here on a small set of especially relevant points of comparison. The long version discusses additional related work.

Language-based IFC Static approaches to IFC have generally dominated language-based security research [40, etc.]; however, statically enforcing IFC at the lowest level of a real system is challenging. Soundly analyzing native binaries with reasonable precision is hard, even more so without the compiler's cooperation (e.g., for stripped or obfuscated binaries). Proof-carrying code [5, etc.] and typed assembly language [33, etc.] have been used for enforcing IFC on low-level code without low-level analysis or adding the compiler to the TCB. In SAFE [14, 17] we follow a different approach, enforcing noninterference using purely dynamic checks, for arbitrary binaries in a custom-designed instruction set. The mechanisms we use for this are similar to those found in recent work on purely dynamic IFC for high-level languages [3, 20, 21, 39, 44, etc.]; however, as far as we know, we are the first to push these ideas to the lowest level.

seL4 Murray et al. [35] recently demonstrated a machine-checked noninterference proof for the implementation of the seL4 microkernel. This proof is carried out by refinement and reuses the specification and most of the existing functional correctness proof of seL4 [27]. Like the TINI property in this paper, the variant of intransitive noninterference used by Murray et al. is preserved by refinement because it implies a high degree of determinism [36]. This organization of their proof was responsible for a significant saving in effort, even when factoring in the additional work required to remove all observable non-determinism from the seL4 specification. Beyond these similarities, SAFE and seL4 rely on completely different mechanisms to achieve different notions of noninterference. Whereas, in SAFE, each word of data has an IFC label and labels are propagated on each instruction, the seL4 kernel maintains separation between several large partitions (e.g., one partition can run an unmodified version of Linux) and ensures that information is conveyed between such partitions only in accordance with a fixed access control policy.

PROSPER In parallel work, Dam et al. [13, 26, etc.] verified information flow security for a tiny proof-of-concept separation kernel running on ARMv7 and using a Memory Management Unit for physical protection of memory regions belonging to different partitions. The authors argue that noninterference is not well suited for systems in which components are supposed to communicate with each other. Instead, they use the bisimulation proof method to show trace equivalence between the real system and an ideal top-level specification that is secure by construction. As in seL4 [35], the proof methodology precludes an abstract treatment of scheduling, but the authors contend this is to be expected when information flow is to be taken into account.

TIARA and ARIES The SAFE architecture embodies a number of innovations from earlier paper designs. In particular, the TIARA design [43] first proposed the idea of a zero-kernel operating system and sketched a concrete architecture, while the ARIES project proposed using a hardware rule cache to speed up information-flow tracking [7]. In TIARA and ARIES, tags had a fixed set of fields and were of limited length, whereas, in SAFE, tags are pointers to

arbitrary data structures, allowing them to represent complex IFC labels encoding sophisticated security policies [34]. Moreover, unlike TIARA and ARIES, which made no formal soundness claims, SAFE proposes a set of IFC rules aimed at achieving noninterference; the proof we present in this paper, though for a simplified model, provides evidence that this goal is within reach.

RIFLE and other binary-rewriting-based IFC systems RIFLE [46] enforces user-specified information-flow policies for x86 binaries using binary rewriting, static analysis, and augmented hardware. Binary rewriting is used to make implicit flows explicit; it heavily relies on static analysis for reconstructing the program's control-flow graph and performing reaching-definitions and alias analysis. The augmented hardware architecture associates labels with registers and memory and updates these labels on each instruction to track explicit flows. Additional security registers are used by the binary translation mechanism to help track implicit flows. Beringer [6] recently proved (in Coq) that the main ideas in RIFLE can be used to achieve noninterference for a simple While language. Unlike RIFLE, SAFE achieves noninterference purely dynamically and does not rely on binary rewriting or heroic static analysis of binaries. Moreover, the SAFE hardware is generic, simply caching instances of software-managed rules.

While many other information flow tracking systems based on binary rewriting have been proposed, few are concerned with soundly handling implicit flows [11, 32], and even these do so only to the extent they can statically analyze binaries. Since, unlike RIFLE (and SAFE), these systems use unmodified hardware, the overhead for tracking implicit flows can be large. To reduce this overhead, recent systems track implicit flows selectively [25] or not at all—arguably a reasonable tradeoff in settings such as malware analysis or attack detection, where speed and precision are more important than soundness.

Hardware taint tracking The last decade has seen significant progress in specialized hardware for accelerating taint tracking [12, 15, 45, 47, etc.]. Most commonly, a single tag bit is associated with each word to specify if it is tainted or not. Initially aimed at mitigating low-level memory corruption attacks by preventing the use of tainted pointers and the execution of tainted instructions [45, etc.], hardware-based taint tracking has also been used to prevent high-level attacks such as SQL injection and cross-site scripting [12]. In contrast to SAFE, these systems prioritize efficiency and overall helpfulness over the soundness of the analysis, striking a heuristic balance between false positives and false negatives (missed attacks). As a consequence, these systems ignore implicit flows and often don't even track all explicit flows. While early systems supported a single hard-coded taint propagation policy, recent ones allow the policy to be defined in software [12, 15, 47] and support monitoring policies that go beyond taint tracking [8, 15, etc.]. Harmoni [15], for example, provides a pair of caches that are quite similar to the SAFE rule cache. Possibly these could even be adapted to enforcing noninterference, in which case we expect the proof methodology introduced here to apply.

Verification of low-level code The distinctive challenge in verifying machine code is coping with unstructured control flow. Our approach using structured generators to build the fault handler is similar to the mechanisms used in Chlipala's Bedrock system [9, 10] and by Jensen et al. [24], but there are several points of difference. These systems each build macros on top of a powerful low-level program logic for machine code (Ni and Shao's XCAP [38], in the case of Bedrock), whereas we take a simpler, ad-hoc approach, building directly on our stack machine's relatively high-level semantics. Both these systems are based on separation logic, which we can do without since (at least in the present simplified model) we have very few memory operations to reason about. We have instead focused on developing a simple Hoare logic specifically suited to verifying structured runtime-system code; e.g., we omit support for arbitrary code pointers, but add support for reasoning about termination. We use total-correctness Hoare triples (similar to Myreen and Gordon [37]) and weakest preconditions to guarantee progress, not just safety, for our handler code. Finally, our level of automation is much more modest than Bedrock's, though still adequate to discharge most verification conditions on straight-line stack manipulation code rapidly and often automatically.

13. Conclusions and Future Work

We have presented a formal model of the key IFC mechanisms of the SAFE system: propagating and checking tags to enforce security, using a hardware cache for common-case efficiency and a software fault handler for maximum flexibility. To formalize and prove properties at such a low level (including features such as dynamic memory allocation and labels represented by pointers to in-memory data structures), we first construct a high-level abstract specification of the system, then refine it in two steps into a realistic concrete machine. A bidirectional refinement methodology allows us to prove (i) that the concrete machine, loaded with the right fault handler (i.e. correctly implementing the IFC enforcement of the abstract specification) satisfies a traditional notion of termination-insensitive noninterference, and (ii) that the concrete machine reflects all the behaviours of the abstract specification. Our formalization reflects the programmability of the fault handling mechanism, in that the fault handler code is compiled from a rule table written in a small DSL. We set up a custom Hoare logic to specify and verify the corresponding machine code, following the structure of a simple compiler for this DSL.

The development in this paper concerns three *deterministic* machines and simplifies away concurrency. While the lack of concurrency is a significant current limitation that we would like to remove as soon as possible by moving to a multithreading single-core model, we still want to maintain the abstraction layers of a proof-by-refinement architecture. This requires some care so as not to run afoul of the refinement paradox [23] since some standard notions of noninterference (for example possibilistic noninterference) are not preserved by refinement in the presence of non-determinism. One promising path toward this objective is inspired by the recent noninterference proof for seL4 [35, 36]. If we manage to share a common thread scheduler between the abstract and concrete machines, we could still prove a strong double refinement property (concrete refines abstract and vice versa) and hence preserve a strong notion of noninterference (such as the TINI notion from this work) or a possibilistic variation.

Although this paper focuses on IFC and noninterference, the tagging facilities of the concrete machine are completely generic. In current follow-on work, we aim to show that the same hardware can be used to efficiently support completely different policies targeting memory safety and control-flow integrity. Moreover, although the rule cache / fault handler design arose in the context of SAFE, we believe that this mechanism can also be ported to more traditional architectures. In the future, we plan to reuse and extend the formal development in this paper both to a larger set of high-level properties and to more conventional architectures. For instance, we expect the infrastructure for compiling DSLs to fault handler software using verified structured code generators to extend to runtime-system components (e.g. garbage collectors, device drivers, etc.), beyond IFC and SAFE.

Acknowledgments We are grateful to Maxime Dénès, Deepak Garg, Greg Morrisett, Toby Murray, Jeremy Planul, Alejandro Russo, Howie Shrobe, Jonathan M. Smith, Deian Stefan, and Greg Sullivan for useful discussions and helpful feedback on early drafts.

We also thank the anonymous reviewers for their insightful comments. This material is based upon work supported by the DARPA CRASH program through the US Air Force Research Laboratory (AFRL) under Contract No. FA8650-10-C-7090. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

[1] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. *ESORICS*. 2008.

[2] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. *CSF*. 2009.

[3] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. *PLAS*. 2009.

[4] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *JFP*, 15(2):131–177, 2005.

[5] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. *ESOP*. 2007.

[6] L. Beringer. End-to-end multilevel hybrid information flow control. *APLAS*. 2012.

[7] J. Brown and T. F. Knight, Jr. A minimally trusted computing base for dynamically ensuring secure information flow. Technical Report 5, MIT CSAIL, 2001. Aries Memo No. 15.

[8] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. P. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. *ISCA*. 2008.

[9] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. *PLDI*, 2011.

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

[11] J. A. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. *ISSTA*. 2007.

[12] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. *ISCA*, 2007.

[13] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. *CCS*, 2013. To appear.

[14] A. DeHon, B. Karel, T. F. Knight, Jr., G. Malecha, B. Montagu, R. Morisset, G. Morrisett, B. C. Pierce, R. Pollack, S. Ray, O. Shivers, J. M. Smith, and G. Sullivan. Preliminary design of the SAFE platform. *PLOS*, 2011.

[15] D. Y. Deng and G. E. Suh. High-performance parallel accelerator for flexible and efficient run-time monitoring. *DSN*. 2012.

[16] U. Dhawan and A. DeHon. Area-efficient near-associative memories on FPGAs. In *International Symposium on Field-Programmable Gate Arrays, (FPGA2013)*, 2013.

[17] 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.

[18] J. A. Goguen and J. Meseguer. Unwinding and inference control. *IEEE S&P*. 1984.

[19] D. Hedin and A. Sabelfeld. A perspective on information-flow control. Marktoberdorf Summer School. IOS Press, 2011.

[20] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. *CSF*. 2012.

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

[22] C. Hriţcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. *ICFP*, 2013.

[23] J. Jacob. On the derivation of secure components. *IEEE S&P*. 1989.

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

[25] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. *NDSS*. 2011.

[26] N. Khakpour, O. Schwarz, and M. Dam. Machine assisted proof of ARMv7 instruction level isolation properties. *CPP*, 2013. To appear.

[27] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. *SOSP*. 2009.

[28] M. N. Krohn and E. Tromer. Noninterference for a practical DIFC-based operating system. *IEEE S&P*. 2009.

[29] 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.

[30] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[31] 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.

[32] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. *ISSRE*. 2004.

[33] R. Medel, A. B. Compagnoni, and E. Bonelli. A typed assembly language for non-interference. *ICTCS*. 2005.

[34] B. Montagu, B. C. Pierce, and R. Pollack. A theory of information-flow labels. *CSF*. 2013.

[35] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: from general purpose to a proof of information flow enforcement. *IEEE S&P*. 2013.

[36] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein. Noninterference for operating system kernels. *CPP*. 2012.

[37] M. O. Myreen and M. J. C. Gordon. Hoare logic for realistically modelled machine code. *TACAS*. 2007.

[38] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. *POPL*. 2006.

[39] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. *CSF*. 2010.

[40] A. Sabelfeld and A. Myers. Language-based information-flow security. *JSAC*, 21(1):5–19, 2003.

[41] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Ershov Memorial Conference*. 2009.

[42] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. *POPL*. 2011.

[43] H. Shrobe, A. DeHon, and T. F. Knight, Jr. Trust-management, intrusion-tolerance, accountability, and reconstitution architecture (TIARA), 2009.

[44] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. *Haskell*. 2011.

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

[46] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. *MICRO*, 2004.

[47] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. *HPCA*, 2008.

[48] S. A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.