JFP **26**, e4, 62 pages, 2016. © Cambridge University Press 2016
doi:10.1017/S0956796816000058

Testing noninterference, quickly

CĂTĂLIN HRIŢCU
Inria Paris, Prosecco team, Paris, France
(e-mail: catalin.hritcu@inria.fr)

LEONIDAS LAMPROPOULOS, ANTAL SPECTOR-ZABUSKY
and ARTHUR AZEVEDO DE AMORIM
Department of Computer and Information Science, University of Pennsylvania, Philadelphia, USA

MAXIME DÉNÈS
Inria Paris, Gallium team, Paris, France

JOHN HUGHES
Computer Science and Engineering, Chalmers University, Gothenburg, Sweden

BENJAMIN C. PIERCE
Department of Computer and Information Science, University of Pennsylvania, Philadelphia, USA

DIMITRIOS VYTINIOTIS
Programming Principles and Tools group, Microsoft Research, Cambridge, UK

Abstract

Information-flow control mechanisms are difficult both to design and to prove correct. To reduce the time wasted on doomed proof attempts due to broken definitions, we advocate modern random-testing techniques for finding counterexamples during the design process. We show how to use QuickCheck, a property-based random-testing tool, to guide the design of increasingly complex information-flow abstract machines, leading up to a sophisticated register machine with a novel and highly permissive flow-sensitive dynamic enforcement mechanism that is sound in the presence of first-class public labels. We find that both sophisticated strategies for generating well-distributed random programs and readily falsifiable formulations of noninterference properties are critically important for efficient testing. We propose several approaches and evaluate their effectiveness on a collection of injected bugs of varying subtlety. We also present an effective technique for shrinking large counterexamples to minimal, easily comprehensible ones. Taken together, our best methods enable us to quickly and automatically generate simple counterexamples for more than 45 bugs. Moreover, we show how testing guides the discovery of the sophisticated invariants needed for the noninterference proof of our most complex machine.

1 Introduction

Secure information-flow control (IFC) is nearly impossible to achieve by careful design alone. The mechanisms involved are intricate and easy to get wrong: static type systems must impose numerous constraints that interact with other typing

rules in subtle ways (Sabelfeld & Myers, 2003), while dynamic mechanisms must appropriately propagate taints and raise security exceptions when necessary (Fenton, 1974; Austin & Flanagan, 2009; Sabelfeld & Russo, 2009; Austin & Flanagan, 2010). In a dynamic setting, allowing IFC labels to vary dynamically (i.e., performing flow-sensitive analysis) can lead to subtle information leaks through the labels themselves (Zheng & Myers, 2007; Russo & Sabelfeld, 2010); these leaks are particularly hard to avoid if labels are observable inside the language (Stefan *et al*., 2011; Hriţcu *et al*. 2013a). This intricacy makes it hard to be confident in the correctness of such mechanisms without detailed proofs; however, carrying out these proofs while designing the mechanisms can be an exercise in frustration, with a great deal of time spent attempting to verify broken definitions! The question we address in this paper is: Can we use modern *testing* techniques to discover bugs in IFC enforcement mechanisms quickly and effectively? If so, then we can use testing to catch most errors during the design phase, postponing proof attempts until we are reasonably confident that the design is correct.

To answer this question, we undertake two case studies. The first is aimed at extending a simple abstract stack-and-pointer machine to track dynamic information flow and enforce termination-insensitive noninterference (Sabelfeld & Myers, 2003). Although this machine is simple, the exercise is nontrivial. While even simpler notions of dynamic *taint tracking* are well studied for both high- and low-level languages, it has only recently been shown (Austin & Flanagan, 2009; Sabelfeld & Russo, 2009) that dynamic checks are capable of soundly enforcing strong security properties. Moreover, until recently (Hriţcu *et al*. 2013b; Azevedo de Amorim *et al*., 2014; Bichhawat *et al*. 2014a), sound dynamic IFC has been studied only in the context of high-level languages (Austin & Flanagan, 2009; Sabelfeld & Russo, 2009; Stefan *et al*., 2011; Hedin & Sabelfeld, 2012; Hriţcu *et al*. 2013a; Bichhawat *et al*., 2014b); the unstructured control flow of a low-level machine poses additional challenges.

We show how QuickCheck (Claessen & Hughes, 2000), a popular property-based testing tool, can be used to formulate and test noninterference properties of our abstract machine, quickly find a variety of missing-taint and missing-exception bugs, and incrementally guide the design of a correct version of the machine. One significant challenge is that both the strategy for generating random programs and the precise formulation of the noninterference property have a dramatic impact on the time required to discover bugs; we benchmark several variations of each to identify the most effective choices. In particular, we observe that checking the unwinding conditions (Goguen & Meseguer, 1984) of our noninterference property can be much more effective than directly testing the original property.

The second case study demonstrates the scalability of our techniques by targeting the design of a novel and highly permissive flow-sensitive dynamic IFC mechanism. This experiment targets a more sophisticated register machine that is significantly more realistic than the first and that includes advanced features such as first-class public labels and dynamically allocated memory with mutable labels. We still quickly find all introduced flaws. Moreover, we can use testing to discover the sophisticated invariants required by a complex noninterference proof.

Our results should be of interest both to researchers in language-based security, who can now add random testing to their tools for debugging subtle IFC enforcement mechanisms and their noninterference proofs; and to the random-testing community, where our techniques for generating and shrinking random programs may be useful for checking other properties of abstract machines. Our primary contributions are: (1) a demonstration of the effectiveness of random testing for discovering counterexamples to noninterference in low-level information-flow machines; (2) a range of program generation strategies for finding such counterexamples; (3) an empirical comparison of how effective combinations of these strategies and formulations of noninterference are in finding counterexamples; (4) an effective methodology for shrinking large counterexamples to smaller, more readable ones; (5) a demonstration that these techniques can speed the design of a state-of-the-art flow-sensitive dynamic IFC mechanism that is highly permissive and sound even though labels are observable; (6) a demonstration that our techniques can aid in discovering the complex invariants involved in the noninterference proofs for this novel IFC mechanism; (7) a mechanized noninterference proof for this mechanism.

Sections 2 to 7 gradually introduce our testing methodology using the simple stack machine as the running example. Section 8 shows that our methodology scales up to the more realistic register machine with advanced IFC features. Section 9 presents related work and Section 10 concludes and discusses future work. Accompanying Haskell code associated to this paper and the Coq proofs mentioned in Section 8 and Appendix B are available online at `https://github.com/QuickChick`.

A preliminary version of this work appeared in the proceedings of the ICFP 2013 conference (Hriţcu *et al.* 2013b). Section 8 of this paper and the contributions therein (points 5 to 7 above) are new. The other sections have also been improved and extended with additional counterexamples.

2 Basic IFC

We begin by introducing the core of our abstract stack machine. In Section 5, we will extend this simple core with control flow (jumps and procedure calls), but the presence of pointers already raises opportunities for some subtle mistakes in IFC.

Some notation: we write [] for the empty list and $x : xs$ for the list whose first element is x and whose tail is xs; we also write $[x_0, x_1, \ldots, x_n]$ for the list $x_0 : x_1 : \cdots : x_n : []$. If xs is a list and $0 \leqslant j < |xs|$, then $xs(j)$ selects the jth element of xs and $xs[j := x]$ produces the list that is like xs except that the jth element is replaced by x.

2.1 Bare stack machine

The most basic variant of our stack machine (without information-flow labels) has seven instructions:

$$Instr ::= Push\ n \mid Pop \mid Load \mid Store \mid Add \mid Noop \mid Halt.$$

The n argument to *Push* is an integer (an immediate constant).

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

A machine state S is a 4-tuple consisting of a program counter pc (an integer), a stack s (a list of integers), a memory m (another list of integers), and an instruction memory i (a list of instructions), written $\boxed{pc \mid s \mid m \mid i}$. Since i is cannot change during execution, we will often write just $\boxed{pc \mid s \mid m}$ for the varying parts of the machine state.

The single-step reduction relation on machine states, written $S \Rightarrow S'$, is defined by the following rules:

$$\frac{i(pc) = Noop}{\boxed{pc \mid s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m}} \quad \text{(Bare-Noop)}$$

$$\frac{i(pc) = Push\ n}{\boxed{pc \mid s \mid m} \Rightarrow \boxed{pc{+}1 \mid n : s \mid m}} \quad \text{(Bare-Push)}$$

$$\frac{i(pc) = Pop}{\boxed{pc \mid n : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m}} \quad \text{(Bare-Pop)}$$

$$\frac{i(pc) = Load \qquad m(p) = n}{\boxed{pc \mid p : s \mid m} \Rightarrow \boxed{pc{+}1 \mid n : s \mid m}} \quad \text{(Bare-Load)}$$

$$\frac{i(pc) = Store \qquad m' = m[p := n]}{\boxed{pc \mid p : n : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m'}} \quad \text{(Bare-Store)}$$

$$\frac{i(pc) = Add}{\boxed{pc \mid n_1 : n_2 : s \mid m} \Rightarrow \boxed{pc{+}1 \mid (n_1{+}n_2) : s \mid m}} \quad \text{(Bare-Add)}$$

This relation is a partial function: it is deterministic, but some machine states don't step to anything. Such a stuck machine state is said to be *halted* if $i(pc) = Halt$ and *failed* in all other cases (e.g., if the machine is trying to execute an *Add* with an empty stack, or if the pc points outside the bounds of the instruction memory). We write $\Rightarrow^*$ for the reflexive, transitive closure of $\Rightarrow$. When $S \Rightarrow^* S'$ and S' is a stuck state, we write $S \Downarrow S'$.

2.2 Stack machine with labeled data

In a (fine-grained) dynamic IFC system (Austin & Flanagan, 2009; Sabelfeld & Russo, 2009; Stefan *et al.*, 2011; Hedin & Sabelfeld, 2012; Hriţcu *et al.* 2013a; Azevedo de Amorim *et al.*, 2014; Bichhawat *et al.*, 2014b) security levels (called labels) are attached to runtime values and propagated during execution, enforcing the constraint that information derived from secret data does not leak to untrusted processes or to the public network. Each value is protected by an individual IFC label representing a security level (e.g., secret or public). We now add labeled data to our simple stack machine. Instead of bare integers, the basic data items in the instruction and data memories and the stack are now *labeled integers* of the form $n@\ell$, where n is an integer and ℓ is a *label*:

$$\ell ::= L \mid H.$$

We read L as "low" (public) and H as "high" (secret). We order labels by $L \sqsubseteq H$ and write $\ell_1 \vee \ell_2$ for the *join* (least upper bound) of ℓ_1 and ℓ_2.

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

The instructions are exactly the same except that the immediate argument to *Push* becomes a labeled integer:

$$Instr ::= Push\ n@\ell \mid Pop \mid Load \mid Store \mid Add \mid Noop \mid Halt.$$

Machine states have the same shape as the basic machine, with the stack and memory now being lists of labeled integers. The set of *initial* states of this machine, *Init*, contains states of the form $\boxed{0 \mid [\,] \mid m_0 \mid i}$, where m_0 can be of any length and contains only $0@L$. We use *Halted* to denote the set of halted states of the machine, i.e., $i(pc) = Halt$.

2.3 Noninterference (EENI)

We define what it means for this basic IFC machine to be "secure" using a standard notion of termination-insensitive noninterference (Sabelfeld & Myers, 2003; Austin & Flanagan, 2009; Hriţcu *et al.* 2013a; Azevedo de Amorim *et al.*, 2014); we call it *end-to-end noninterference* (or *EENI*) to distinguish it from the stronger notions we will introduce in Section 6. The main idea of EENI is to directly encode the intuition that secret inputs should not influence public outputs. By secret inputs, we mean integers labeled H in the initial state; because of the form of our initial states, such labeled integers can appear only in instruction memories. By secret outputs, we mean integers labeled H in a halted state. More precisely, EENI states that for any two executions starting from initial states that are *indistinguishable to a low observer* (or just *indistinguishable*) and ending in halted states S_1 and S_2, the final states S_1 and S_2 are also indistinguishable. Intuitively, two states are indistinguishable if they differ only in integers labeled H. To make this formal, we define an equivalence relation on states compositionally from equivalence relations over their components.

2.1 Definition:

- Two labeled integers $n_1@\ell_1$ and $n_2@\ell_2$ are said to be *indistinguishable*, written $n_1@\ell_1 \approx n_2@\ell_2$, if either $\ell_1 = \ell_2 = H$ or else $n_1 = n_2$ and $\ell_1 = \ell_2 = L$.
- Two instructions i_1 and i_2 are indistinguishable if they are the same, or if $i_1 = Push\ n_1@\ell_1$, and $i_2 = Push\ n_2@\ell_2$, and $n_1@\ell_1 \approx n_2@\ell_2$.
- Two lists (memories, stacks, or instruction memories) xs and ys are indistinguishable if they have the same length and $xs(i) \approx ys(i)$ for all i such that $0 \leqslant i < |xs|$.

For machine states, we have a choice as to how much of the state we want to consider observable; we choose (somewhat arbitrarily) that the observer can only see the data and instruction memories, but not the stack or the *pc*. (Other choices would give the observer either somewhat more power—e.g., we could make the stack observable—or somewhat less—e.g., we could restrict the observer to some designated region of "I/O memory," or extend the architecture with I/O instructions and only observe the traces of inputs and outputs (Azevedo de Amorim *et al.*, 2014).)

2.2 Definition: Machine states $S_1 = \boxed{pc_1 \mid s_1 \mid m_1 \mid i_1}$ and $S_2 = \boxed{pc_2 \mid s_2 \mid m_2 \mid i_2}$ are *indistinguishable with respect to memories*, written $S_1 \approx_{mem} S_2$, if $m_1 \approx m_2$ and $i_1 \approx i_2$.

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

2.3 Definition: A machine semantics is *end-to-end noninterfering* with respect to some sets of states *Start* and *End* and an indistinguishability relation $\approx$, written $\text{EENI}_{Start,End,\approx}$, if for any $S_1, S_2 \in Start$ such that $S_1 \approx S_2$ and such that $S_1 \Downarrow S_1'$, $S_2 \Downarrow S_2'$, and $S_1', S_2' \in End$, we have $S_1' \approx S_2'$.

We take $\text{EENI}_{Init,Halted,\approx_{mem}}$ as our baseline security property; i.e., we only consider executions starting in initial states and ending in halted states, and we use indistinguishability with respect to memories. The EENI definition above is, however, more general, and we will consider other instantiations of it later.

2.4 Information-flow rule design with QuickCheck

Our next task is to enrich the rules for the step function to take information-flow labels into account. For most of the rules, there are multiple plausible ways to do this, and some opportunities for subtle mistakes even with these few instructions. To illustrate the design methodology we hope to support, we first propose a naive set of rules and then use counterexamples generated using QuickCheck and our custom generation and shrinking techniques (described in detail in the following sections) to identify and help repair mistakes until no more can be found.

$$\frac{i(pc) = Noop}{\boxed{pc}\,\boxed{s}\,\boxed{m} \Rightarrow \boxed{pc{+}1}\,\boxed{s}\,\boxed{m}} \quad \text{(NOOP)}$$

$$\frac{i(pc) = Push\ n@\ell}{\boxed{pc}\,\boxed{s}\,\boxed{m} \Rightarrow \boxed{pc{+}1}\,\boxed{n@\ell : s}\,\boxed{m}} \quad \text{(PUSH)}$$

$$\frac{i(pc) = Pop}{\boxed{pc}\,\boxed{n@\ell : s}\,\boxed{m} \Rightarrow \boxed{pc{+}1}\,\boxed{s}\,\boxed{m}} \quad \text{(POP)}$$

$$\frac{i(pc) = Load \qquad m(p) = n@\ell_n}{\boxed{pc}\,\boxed{p@\ell_p : s}\,\boxed{m} \Rightarrow \boxed{pc{+}1}\,\boxed{n@\ell_n : s}\,\boxed{m}} \quad \text{(LOAD*)}$$

$$\frac{i(pc) = Store \qquad m' = m[p := n@\ell_n]}{\boxed{pc}\,\boxed{p@\ell_p : n@\ell_n : s}\,\boxed{m} \Rightarrow \boxed{pc{+}1}\,\boxed{s}\,\boxed{m'}} \quad \text{(STORE*AB)}$$

$$\frac{i(pc) = Add}{\boxed{pc}\,\boxed{n_1@\ell_1 : n_2@\ell_2 : s}\,\boxed{m} \Rightarrow \boxed{pc{+}1}\,\boxed{(n_1{+}n_2)@L : s}\,\boxed{m}} \quad \text{(ADD*)}$$

The NOOP rule is the same as in the unlabeled machine. In the PUSH and POP rules, we simply change from bare to labeled integers; luckily, this obvious adaptation happens to be correct. But now our luck runs out: the simple changes that we've made in the other rules will all turn out to be wrong. (We include a star in the names of incorrect rules to indicate this. The rule STORE*AB actually contains *two* bugs, which we refer to as A and B; we will discuss them separately later.) Fortunately, QuickCheck can rapidly pinpoint the problems, as we will see.

Figure 1 shows the first counterexample that QuickCheck gives us when we present it with the step function defined by the six rules above and ask it to try to

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$$i = \left[\mathit{Push}\ 1@L, \mathit{Push}\ \frac{0}{1}@H, \mathit{Store}, \mathit{Halt} \right]$$

pc	*m*	*s*	*i*(*pc*)
0	$[0@L, 0@L]$	[]	*Push* $1@L$
1	$[0@L, 0@L]$	$[1@L]$	*Push* $\frac{0}{1}@H$
2	$[0@L, 0@L]$	$\left[\frac{0}{1}@H, 1@L\right]$	*Store*
3	$\left[\frac{1}{0}@L, \frac{0}{1}@L\right]$	[]	*Halt*

Figure 1. Counterexample to STORE*AB.

invalidate the EENI property. (The LaTeX source for all the figures was generated automatically by our QuickCheck testing infrastructure.) The first line of the figure is the counterexample itself: a pair of four-instruction programs, differing only in the constant argument of the second *Push*. The first program pushes $0@H$, while the second pushes $1@H$, and these two labeled integers are indistinguishable. We display the two programs, and the other parts of the two machine states, in a "merged" format. Pieces of data that are the same between the two machines are written just once; at any place where the two machines differ, the value of the first machine is written above the value of the second machine, separated by a horizontal line. The rest of the figure shows what happens when we run this program. On the first step, the *pc* starts out at 0; the memory, which has two locations, starts out as $[0@L, 0@L]$; the stack starts out empty; and the next instruction to be executed ($i(pc)$) is *Push* $1@L$. On the next step, this labeled integer has been pushed on the stack and the next instruction is either *Push* $0@H$ or *Push* $1@H$; one or the other of these labeled integers is pushed on the stack. On the next, we *Store* the second stack element ($1@L$) into the location pointed to by the first (either $0@H$ or $1@H$), so that now the memory contains $1@L$ in either location 0 or location 1 (the other location remains unchanged, and contains $0@L$). At this point, both machines halt. This pair of execution sequences shows that EENI fails: in the initial state, the two programs are indistinguishable to a low observer (their only difference is labeled H), but in the final states the memories contain different integers at the same location, both of which are labeled L.

Thinking about this counterexample, it soon becomes apparent what went wrong with the *Store* instruction: since pointers labeled H are allowed to vary between the two runs, it is not safe to store a low integer through a high pointer. One simple but draconian fix is simply to stop the machine if it tries to perform such a store (i.e., we could add the side-condition $\ell_p = L$ to the rule). A more permissive option is to allow the store to take place, but require it to taint the stored value with the label on the pointer:

$$\frac{i(pc) = \mathit{Store} \qquad m' = m[p := n@(\ell_n \vee \ell_p)]}{\boxed{pc} \; \boxed{p@\ell_p : n@\ell_n : s} \; \boxed{m} \Rightarrow \boxed{pc{+}1} \; \boxed{s} \; \boxed{m'}} \qquad \text{(STORE*B)}$$

Unfortunately, QuickCheck's next counterexample (Figure 2) shows that this rule is still not quite good enough. This counterexample is quite similar to the first one, but it illustrates a more subtle point: our definition of noninterference allows the

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$$i = \left[Push\ 0@L, Push\ \frac{0}{1}@H, Store, Halt \right]$$

pc	m	s	$i(pc)$
0	$[0@L, 0@L]$	$[]$	$Push\ 0@L$
1	$[0@L, 0@L]$	$[0@L]$	$Push\ \frac{0}{1}@H$
2	$[0@L, 0@L]$	$\left[\frac{0}{1}@H, 0@L\right]$	$Store$
3	$\left[0@\frac{H}{L}, 0@\frac{L}{H}\right]$	$[]$	$Halt$

Figure 2. Counterexample to STORE*B.

$$i = \left[\begin{array}{l} Push\ \frac{0}{1}@H, Push\ 0@L, Add, Push\ 0@L, Store, \\ Halt \end{array} \right]$$

pc	m	s	$i(pc)$
0	$[0@L]$	$[]$	$Push\ \frac{0}{1}@H$
1	$[0@L]$	$\left[\frac{0}{1}@H\right]$	$Push\ 0@L$
2	$[0@L]$	$\left[0@L, \frac{0}{1}@H\right]$	Add
3	$[0@L]$	$\left[\frac{0}{1}@L\right]$	$Push\ 0@L$
4	$[0@L]$	$\left[0@L, \frac{0}{1}@L\right]$	$Store$
5	$\left[\frac{0}{1}@L\right]$	$[]$	$Halt$

Figure 3. Counterexample to ADD*.

observer to distinguish between final memory states that differ only in their *labels*.[1] Since the STORE*B rule taints the label of the stored integer with the label of the pointer, the fact that the *Store* changes different locations is visible in the fact that a label changes from L to H on a different memory location in each run. To avoid this issue, we adopt the "no sensitive upgrades" rule (Zdancewic, 2002; Austin & Flanagan, 2009), which demands that the label on the current contents of a memory location being stored into are above the label of the pointer used for the store —i.e., it is illegal to overwrite a low value via a high pointer (and trying to do so results in a fatal failure). Adding this side condition brings us to a correct version of the STORE rule.

$$\frac{i(pc) = Store \qquad m(p) = n'@\ell'_n \qquad \ell_p \sqsubseteq \ell'_n \qquad m' = m[p := n@(\ell_n \vee \ell_p)]}{\boxed{pc}\ \boxed{p@\ell_p : n@\ell_n : s}\ \boxed{m} \Rightarrow \boxed{pc+1}\ \boxed{s}\ \boxed{m'}} \quad \text{(STORE)}$$

The next counterexample found by QuickCheck (Figure 3) points out a straightforward problem in the ADD* rule: adding $0@L$ to $0@H$ yields $0@L$. The problem is that the taints on the arguments to *Add* are not propagated to its result. The *Store*

[1] See the first clause of Definition 2.1. One might imagine that this could be fixed easily by changing the definition so that whether a label is high or low is not observable—i.e., $n@L \approx n@H$ for any n. Sadly, this is known not to work (Fenton, 1974; Russo & Sabelfeld, 2010). (QuickCheck can also find a counterexample, which we present in Section A.1. The counterexample relies on control flow, which is only introduced in Section 5.)

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$$i = \left[\begin{array}{l} \mathit{Push}\ 0@L, \mathit{Push}\ 1@L, \mathit{Push}\ 0@L, \mathit{Store}, \mathit{Push}\ \frac{0}{1}@H, \\ \mathit{Load}, \mathit{Store}, \mathit{Halt} \end{array}\right]$$

pc	*m*	*s*	*i*(*pc*)
0	$[0@L, 0@L]$	$[\,]$	$\mathit{Push}\ 0@L$
1	$[0@L, 0@L]$	$[0@L]$	$\mathit{Push}\ 1@L$
2	$[0@L, 0@L]$	$[1@L, 0@L]$	$\mathit{Push}\ 0@L$
3	$[0@L, 0@L]$	$[0@L, 1@L, 0@L]$	*Store*
4	$[1@L, 0@L]$	$[0@L]$	$\mathit{Push}\ \frac{0}{1}@H$
5	$[1@L, 0@L]$	$\left[\frac{0}{1}@H, 0@L\right]$	*Load*
6	$[1@L, 0@L]$	$\left[\frac{1}{0}@L, 0@L\right]$	*Store*
7	$\left[\frac{1}{0}@L, 0@L\right]$	$[\,]$	*Halt*

Figure 4. Counterexample to LOAD*.

is needed in order to make the difference observable. The easy (and standard) fix is to use the join of the argument labels as the label of the result:

$$\frac{i(pc) = \mathit{Add}}{\boxed{pc}\,\boxed{n_1@\ell_1 : n_2@\ell_2 : s}\,\boxed{m} \Rightarrow \boxed{pc{+}1}\,\boxed{(n_1{+}n_2)@(\ell_1 \vee \ell_2) : s}\,\boxed{m}} \quad (\textsc{Add})$$

The final counterexample found by QuickCheck (Figure 4) alerts us to the fact that the LOAD* rule contains a similar mistake to the original STORE*AB rule: loading a low value through a high pointer should taint the loaded value. The program in Figure 4 is a little longer than the one in Figure 1 because it needs to do a little work at the beginning to set up a memory state containing two different low values. It then pushes a high address pointing to one or the other of those cells onto the stack; loads (different, low addresses) through that pointer; and finally stores $0@L$ to the resulting address in memory and halts. In this case, we can make the same change to LOAD* as we did to STORE*AB: we taint the loaded integer with the join of its label and the address's label. This time (unlike the case of *Store*, where the fact that we were changing the memory gave us additional opportunities for bugs), this change gives us the correct rule for *Load*,

$$\frac{i(pc) = \mathit{Load} \qquad m(p) = n@\ell_n}{\boxed{pc}\,\boxed{p@\ell_p : s}\,\boxed{m} \Rightarrow \boxed{pc{+}1}\,\boxed{n@(\ell_n \vee \ell_p) : s}\,\boxed{m}} \quad (\textsc{Load})$$

and QuickCheck is unable to find any further counterexamples.

2.5 More bugs

The original IFC version of the step rules illustrate one set of mistakes that we might plausibly have made, but there are more possible ones:

$$\frac{i(pc) = \mathit{Push}\ n@\ell}{\boxed{pc}\,\boxed{s}\,\boxed{m} \Rightarrow \boxed{pc{+}1}\,\boxed{n@L : s}\,\boxed{m}} \quad (\textsc{Push}^*)$$

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$$\frac{i(pc) = Store \qquad m' = m[p := y@L]}{\boxed{pc \mid p@\ell_p : n@\ell_n : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m'}} \quad (\textsc{Store*c})$$

Although it is unlikely that we'd write these rather silly rules by accident, it is worth including them in our experiments because they can be invalidated by short counterexamples and thus provide useful data points for less effective testing strategies.

We will also gather statistics for a partially fixed but still wrong rule for *Store*, in which the no-sensitive-upgrades check is performed but the result is not properly tainted:

$$\frac{i(pc) = Store \qquad m(p) = n'@\ell'_n \qquad \ell_p \sqsubseteq \ell'_n \qquad m' = m[p := n@\ell_n]}{\boxed{pc \mid p@\ell_p : n@\ell_n : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m'}} \quad (\textsc{Store*a})$$

3 QuickCheck

We test noninterference using QuickCheck (Claessen & Hughes, 2000), a tool that tests properties expressed in Haskell. Often, QuickCheck is used to test properties that should hold for all inhabitants of a certain type. QuickCheck repeatedly generates random values of the desired type, instantiates the property with them, and checks it directly by evaluating it to a Boolean. This process continues until either a counterexample is found or a specified timeout is reached. QuickCheck supplies default test data generators for many standard types. Additionally, the user can supply custom generators for their own types. In order to test EENI, for example, we needed to define custom generators for labeled integers, instructions, and machine states (each of which depends on the previous generator: machine states contain instructions, some of which contain labeled integers). The effectiveness of testing (i.e., mean time to discover bugs) depends on the sophistication of these generators, a topic we explore in detail in Section 4.

QuickCheck properties may also be guarded by *preconditions*; EENI is an example of why this is necessary, as it only applies to pairs of indistinguishable initial machine states that both successfully execute to halted states. Testing a property with a precondition proceeds similarly: a sequence of random values are generated and tested, up to a user-specified maximum. The difference is that if there is a precondition, it is instantiated with the random value first. If the precondition does not hold, this random value is summarily discarded. If the precondition does hold, then the rest of the property is checked just as before. Although preconditions are very useful, too high a proportion of discards can lead to very ineffective testing or a badly skewed distribution of test cases (since some kinds of test case may be discarded much more often than others). To help diagnose such problems, QuickCheck can collect statistics about the tests it tried.

When a test fails, the failing test case is often large, containing many irrelevant details. QuickCheck then tries to *shrink* the test case, by searching for a similar but smaller test case that also fails. To do this, it greedily explores a number of "shrinking candidates": modifications of the original failing test case that are

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

"smaller" in some sense. The property is tested for each of these, and as soon as a failure is found, that candidate becomes the starting point for a new shrinking search (and the other candidates are discarded). Eventually, this process terminates in a failing test case that is *locally minimal*: none of its shrinking candidates fails. This failing case is then reported to the user. It is often very much smaller than the original randomly generated test case, and it is thus easy to use it to diagnose the failure because it (hopefully) contains no irrelevant details. Just like generation strategies, shrinking strategies are type dependent; they are defined by QuickCheck for standard types, and by the user for other types. We discuss the custom shrinking strategies we use for machine states in Section 7.

4 State generation strategies

We are ready now to begin exploring ways to generate potential counterexamples. At the outset, we need to address one fundamental issue. Noninterference properties quantify over a *pair* of indistinguishable starting states: $\forall S_1, S_2 \in \mathit{Start}.\ S_1 \approx S_2 \Longrightarrow \ldots$. This is a very strong precondition, which is extremely unlikely to be satisfied for independently generated states. Instead, we generate *indistinguishable pairs* of states together. The first state is generated randomly using one of the techniques described later in this section. The second is obtained by randomly varying the "high parts" of the first. We refer to the second state as the *variation* of the first. The resulting pair thus satisfies indistinguishability by construction. Note that when implemented correctly this does not compromise completeness: by generating a random state and randomly varying we still guarantee that it is possible to generate all pairs of indistinguishable states. Naturally, the resulting distributions will depend on the specifics of the generation and variation methods used, as we shall see.

Since EENI considers only executions that start at initial states, we only need to randomly generate the contents of the instruction memory (the program that the machine executes) together with the *size* of the data memory (in initial states, the contents of the memory are fixed and the stack is guaranteed to be empty).

Figure 5 offers an empirical comparison of all the generation strategies described in this section. For a given test-generation strategy, we inject the bugs from Section 2 one at a time into the machine definition and measure the time spent on average until that bug is found (*mean time to failure*, or MTTF). Tests were run one at a time on seven identical machines, each with 4× 2.4 GHz Intel processors and 11.7 GB of RAM; they were running Fedora 18 and GHC 7.4.1, and using QuickCheck 2.7.6. We run each test for 5 minutes (300 seconds) or until 4,000 counterexamples are found, whichever comes first.

4.1 Naive instruction generation

The simplest way to generate programs is by choosing a sequence of instructions *independently* and *uniformly*. We generate individual instructions by selecting an instruction type uniformly (i.e., *Noop*, *Push*, etc.) and then filling in its fields using QuickCheck's built-in generators. Labels are also chosen uniformly. We then build

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

Generation strategy	*Naive*	*Weighted*	*Sequence*	*Sequence*	*ByExec*
Smart integers?	NO	NO	NO	YES	YES
ADD*	83247.01	5344.26	561.58	30.05	0.87
PUSH*	3552.54	309.20	0.21	0.07	0.01
LOAD*	—	—	73115.63	2258.93	4.03
STORE*A	—	—	38036.22	32227.10	1233.51
STORE*B	47365.97	1713.72	0.85	0.12	0.25
STORE*C	7660.07	426.11	0.41	0.31	0.02
MTTF arithmetic mean	**—**	**—**	**18619.15**	**5752.76**	**206.45**
MTTF geometric mean	**—**	**—**	**69.73**	**13.33**	**0.77**
Average tests / second	24129	11466	8541	7915	3284
Average discard rate	79%	62%	65%	59%	4%

Figure 5. Comparison of generation strategies for the basic machine. The first part of the table shows the mean time to find a failing test case (MTTF) in milliseconds for each bug. The second part lists the arithmetic and geometric mean for the MTTF over all bugs. The third part shows the number of tests per second and the proportion of test cases that were discarded because they did not satisfy some precondition.

Average number of execution steps: 0.47

74%	stack underflow
21%	halt
4%	load or store out of range

Figure 6. Initial-Naive-False: Execution statistics for naive instruction generation. Executions fail early, and the main reason for failure is stack underflow.

the instruction memory by sampling a number (currently a random number between 20 and 50) of instructions from this generator.

The first column of Figure 5 shows how this strategy performs on the bugs from Section 2. Disappointingly, but perhaps not too surprisingly, naive instruction generation can only find four of the six bugs within 5 minutes. How can we do better?

One obvious weakness is that the discard rate is quite high, indicating that one or both machines often fail to reach a halted state. By asking QuickCheck to collect statistics on the execution traces of test cases (Figure 6), we can also see a second problem: the average execution length is only 0.47 steps! Such short runs are not useful for finding counterexamples to EENI (at a minimum, any counterexample must include a *Store* instruction to put bad values into the memory and a *Halt* so that the run terminates, plus whatever other instructions are needed to produce the bad states). So our next step is to vary the distribution of instructions so as to generate programs that run for longer and thus have a chance to get into more interesting states.

4.2 Weighted distribution on instructions

Figure 6 shows that by far the most common reason for early termination is a stack underflow. After a bit of thought, this makes perfect sense: the stack is initially

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

Average number of execution steps: 2.69	
38%	halt
35%	stack underflow
25%	load or store out of range

Figure 7. Initial-Weighted-False: Execution statistics when generating instructions with a weighted distribution. The main reason for failure is now *Halt*, followed by stack underflow.

Average number of execution steps: 3.86	
37%	halt
28%	load or store out of range
20%	stack underflow
13%	sensitive upgrade

Figure 8. Initial-Sequence-False: Execution statistics when generating sequences of instructions. Out-of-range addresses are now the biggest reason for termination.

empty, so if the first instruction that we generate is anything but a *Push*, *Halt*, or *Noop*, we will fail immediately. Instead of a uniform distribution on instructions, we can do better by increasing the weights of *Push* and *Halt*—*Push* to reduce the number of stack underflows, and *Halt* because each execution must reach a halted state to satisfy EENI's precondition. The results after this change are shown in the second column of Figure 5. Although this strategy still fails to find the LOAD* and STORE*A bugs in the allocated time, there is a significant improvement on both discard rates and the MTTF for the other bugs. Run length is also better, averaging 2.69 steps. As Figure 7 shows, executing *Halt* is now the main reason for termination, with stack underflows and out-of-range accesses close behind.

4.3 Generating useful instruction sequences more often

To further reduce stack underflows, we additionally generate *sequences* of instructions that make sense together. For instance, in addition to generating single *Store* instructions, we also generate sequences of the form $[Push\ n@\ell, Push\ ma@\ell', Store]$, where *ma* is a valid memory address. We also generate such sequences for the other two instructions that use stack elements: $[Push\ ma@\ell, Load]$ where *ma* is a valid memory address, and $[Push\ n_1@\ell_1, Push\ n_2@\ell_2, Add]$. The results are shown in the third column of Figure 5. With sequence generation, we can now find all bugs, faster than before. Programs run for slightly longer (3.86 steps on average). As expected, stack underflows are less common than before (Figure 8) and out-of-range addresses are now the second biggest reason for termination.

4.4 Smart integers: generating addresses more often

To reduce the number of errors caused by out-of-range addresses, we additionally give preference to *valid* memory addresses, i.e., integers within (data and instruction) memory bounds, when generating integers. We do this not only when generating the state of the first machine, but also when *varying* it, since both machines need to halt

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

Average number of execution steps: 4.22	
41%	halt
21%	stack underflow
21%	load or store out of range
15%	sensitive upgrade

Figure 9. Initial-Sequence-True: Execution statistics when using smart integers with sequences of instructions. The percentage of address out of range errors has halved.

successfully in order to satisfy the precondition of EENI. Column four of Figure 5 shows the results after making this improvement to the previous generator. We see an improvement on the MTTF. The average run length is now 4.22 steps, and the percentage of address-out-of-range errors is decreased (Figure 9).

4.5 Generation by execution

We can go even further. In addition to weighted distributions, sequences, and smart integers, we try to generate instructions that *do not cause a crash*. In general (for more interesting machines), deciding whether an arbitrary instruction sequence causes a crash is undecidable. In particular, we cannot know in advance all possible states in which an instruction will be executed. We can only make a guess—a very accurate one for this simple machine. This leads us to the following *generation by execution* strategy: We generate a single instruction or a small sequence of instructions, as before, except that now we restrict generation to instructions that do not cause the machine to crash in the current state. When we find one, we execute it to reach a new state and then repeat the process to generate further instructions. We continue until we have generated a reasonably sized instruction stream (currently, randomly chosen between 20–50 instructions). We discuss how this idea generalizes to machines with nontrivial control flow in Section 5.3.

As we generate more instructions, we increase the probability of generating a *Halt* instruction, to reduce the chances of the machine running off the end of the instruction stream. As a result, (i) we maintain low discard ratios for EENI since we increase the probability that executions finish with a *Halt* instruction, and (ii) we avoid extremely long executions whose long time to generate and run could be more fruitfully used for other test cases.

The MTTF (last column of Figure 5) is now significantly lower than in any previous generation method, although this strategy runs fewer tests per second than the previous ones (because both test case generation and execution take longer). Figure 10 shows that 94% of the pairs both successfully halt, which is in line with the very low discard rate of Figure 5, and that programs run for much longer. Happily, varying a machine that successfully halts has a high probability of generating a machine that also halts.

5 Control flow

Up to this point, we've seen how varying the program generation strategy can make orders-of-magnitude difference in the speed at which counterexamples are found for

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

	Generated	Variation
Steps	11.60	11.26
95%	halt	halt
3%	halt	load or store out of range
1%	halt	sensitive upgrade

Figure 10. Initial-ByExec-True: Execution statistics for generation by execution, broken down for the variations.

$$i = \begin{bmatrix} Push\ \frac{2}{5}@H, Jump, Push\ 1@L, Push\ 0@L, Store, \\ Halt \end{bmatrix}$$

pc	*m*	*s*	*i*(*pc*)
0	$[0@L]$	$[\,]$	$Push\ \frac{2}{5}@H$
1	$[0@L]$	$\left[\frac{2}{5}@H\right]$	*Jump*
Machine 1 continues...			
2	$[0@L]$	$[\,]$	$Push\ 1@L$
3	$[0@L]$	$[1@L]$	$Push\ 0@L$
4	$[0@L]$	$[0@L, 1@L]$	*Store*
5	$[1@L]$	$[\,]$	*Halt*
Machine 2 continues...			
5	$[0@L]$	$[\,]$	*Halt*

Figure 11. Counterexample to JUMP*AB: A textbook example of an implicit flow.

a very simple—almost trivial—information-flow stack machine. Now we are ready to make the machine more interesting and see how these techniques perform on the new bugs that arise, as well as how their performance changes on the bugs we've already seen. In this section, we add *Jump*, *Call*, and *Return* instructions—and, with them, the possibility that information can leak via the program's control flow.

5.1 Jumps, implicit flows, and the pc label

We first add a new *Jump* instruction that takes the first element from the stack and sets the *pc* to that address:

$$\frac{i(pc) = Jump}{\boxed{pc}\boxed{n@\ell_n : s}\boxed{m} \Rightarrow \boxed{n}\boxed{s}\boxed{m}} \quad (\text{JUMP*AB})$$

(The jump target may be an invalid address. In this case, the machine will be stuck on the next instruction.)

Note that this rule simply ignores the label on the jump target on the stack. This is unsound, and QuickCheck easily finds the counterexample in Figure 11—a textbook case of an *implicit flow* (Sabelfeld & Myers, 2003). A secret is used as the target of a jump, which causes the instructions that are executed afterwards to differ between the two machines; one of the machines halts immediately, whereas the other one does a *Store* to a low location and only then halts, causing the final memories to be distinguishable.

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$$i = \left[\begin{array}{l} \mathit{Push}\ 1@L, \mathit{Push}\ \frac{4}{6}@H, \mathit{Jump}, \mathit{Halt}, \mathit{Push}\ 0@L, \\ \mathit{Store}, \mathit{Push}\ 3@L, \mathit{Jump} \end{array}\right]$$

pc	m	s	$i(pc)$
$0@L$	$[0@L]$	$[\,]$	$\mathit{Push}\ 1@L$
$1@L$	$[0@L]$	$[1@L]$	$\mathit{Push}\ \frac{4}{6}@H$
$2@L$	$[0@L]$	$\left[\frac{4}{6}@H, 1@L\right]$	Jump
Machine 1 continues...			
$4@H$	$[0@L]$	$[1@L]$	$\mathit{Push}\ 0@L$
$5@H$	$[0@L]$	$[0@L, 1@L]$	Store
$6@H$	$[1@L]$	$[\,]$	$\mathit{Push}\ 3@L$
$7@H$	$[1@L]$	$[3@L]$	Jump
$3@L$	$[1@L]$	$[\,]$	Halt
Machine 2 continues...			
$6@H$	$[0@L]$	$[1@L]$	$\mathit{Push}\ 3@L$
$7@H$	$[0@L]$	$[3@L, 1@L]$	Jump
$3@L$	$[0@L]$	$[1@L]$	Halt

Figure 12. Counterexample to JUMP*B: *Jump* should not lower the *pc* label.

The standard way to prevent implicit flows is to label the *pc*—i.e., to make it a labeled integer, not a bare integer. Initial states have $pc@\ell_{pc} = 0@L$, and after a jump to a secret address the label of the *pc* becomes H:

$$\frac{i(pc) = \mathit{Jump}}{\boxed{pc@\ell_{pc}}\,\boxed{n@\ell_n : s}\,\boxed{m} \Rightarrow \boxed{n@\ell_n}\,\boxed{s}\,\boxed{m}} \qquad \text{(JUMP*B)}$$

While the *pc* is (labeled) high, the two machines may be executing different instructions, and so we cannot expect the machine states to correspond. We therefore extend the definition of $\approx_{mem}$ so that *all* high machine states are deemed equivalent. (We call a state "high" if the *pc* is labeled H, and "low" otherwise.)

5.1 Definition: Machine states $S_1 = \boxed{pc_1@\ell_{pc_1}}\,\boxed{s_1}\,\boxed{m_1}\,\boxed{i_1}$ and $S_2 = \boxed{pc_2@\ell_{pc_2}}\,\boxed{s_2}\,\boxed{m_2}\,\boxed{i_2}$ are indistinguishable with respect to memories, written $S_1 \approx_{mem} S_2$, if either $\ell_{pc_1} = \ell_{pc_2} = H$ or else $\ell_{pc_1} = \ell_{pc_2} = L$ and $m_1 \approx m_2$ and $i_1 \approx i_2$.

The JUMP*B rule is still wrong, however, since it not only raises the *pc* label when jumping to a high address but also lowers it when jumping to a low address. The counterexample in Figure 12 illustrates that the latter behavior is problematic. The fix is to label the *pc* after a jump with the *join* of the current *pc* label and the label of the target address.

$$\frac{i(pc) = \mathit{Jump}}{\boxed{pc@\ell_{pc}}\,\boxed{n@\ell_n : s}\,\boxed{m} \Rightarrow \boxed{n@(\ell_n \vee \ell_{pc})}\,\boxed{s}\,\boxed{m}} \qquad \text{(JUMP)}$$

With this rule for jumps QuickCheck no longer finds any counterexamples. Some readers may find this odd: In order to fully address implicit flows, it is usually necessary to modify the rules for memory stores to handle the case where the *pc* is labeled high (Austin & Flanagan, 2009; Russo & Sabelfeld, 2010). The current

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

machine doesn't require this, but the reason is subtle: here, the *pc* can go from L to H when we jump to a secret address, but it never goes from H to L! It doesn't matter what the machine does when the *pc* is high, because none of its actions will ever be observable—all high machine states are indistinguishable.

To make things more interesting, we need to enrich the machine with some mechanism that allows the *pc* to safely return to L after it has become H. One way to achieve this is to add *Call* and *Return* instructions, a task we turn to next.

5.2 Restoring the *pc* label with calls and returns

IFC systems (both static and dynamic) generally rely on control flow *merge points* (i.e., post-dominators of the branch point in the control flow graph where the control was tainted by a secret) to detect when the influence of secrets on control flow is no longer relevant and the *pc* label can safely be restored. Control flow merge points are, however, much more evident for structured control features such as conditionals than they are for jumps (as long as we don't have exceptions (Hriţcu *et al.* 2013a; Bichhawat *et al.* 2014a)). Moreover, since we are doing purely dynamic IFC we cannot distinguish between safe uses of jumps and unsafe ones (e.g., the one in Figure 12). So we keep jumps as they are (only raising the *pc* label) and add support for structured programming and restoring the *pc* label in the form of *Call* and *Return* instructions, which are of course useful in their own right.

To support these instructions, we need some way of representing stack frames. We choose a straightforward representation, in which each stack element e can now be either a labeled integer $n@\ell$ (as before) or a *return address*, marked R, recording the *pc* (including its label!) from which the corresponding *Call* was made. We also extend the indistinguishability relation on stack elements so that return addresses are only equivalent to other return addresses and $R(n_1@\ell_1) \approx R(n_2@\ell_2)$ if either $\ell_1 = \ell_2 = H$ or else $n_1 = n_2$ and $\ell_1 = \ell_2 = L$ (this is the same as for labeled integers). (High return addresses and high integers need to be distinguishable to a low observer, as we discovered when QuickCheck generated an unexpected counterexample, which we list in Section A.3—understanding it requires reading the rest of this section.)

We also need a way to pass arguments to and return results from a called procedure. For this, we annotate the *Call* and *Return* instructions with a positive integer indicating how many integers should be passed or returned (0 or 1 in the case of *Return*). Formally, *Call* k expects an address $n@\ell_n$ followed by k integers ns on the stack. It sets the *pc* to n, labels this new *pc* by the join of ℓ_n and the current *pc* label (as we did for *Jump*—we're eliding the step of getting this bit wrong at first and letting QuickCheck find a counterexample), and adds the return address frame to the stack *under* the k arguments.

$$\frac{i(pc) = \mathit{Call}\ k \qquad ns = n_1@\ell_1 : \ldots : n_k@\ell_k}{\boxed{pc@\ell_{pc}}\ \boxed{n@\ell_n : ns : s}\ \boxed{m} \Rightarrow \boxed{n@(\ell_n \vee \ell_{pc})}\ \boxed{ns : R((pc{+}1)@\ell_{pc}) : s}\ \boxed{m}} \quad (\textsc{Call*b})$$

Return k' traverses the stack until it finds the first return address and jumps to it. Moreover, it restores the *pc* label to the label stored in that R entry, and preserves the first k' elements on the stack as return values, discarding all other elements in

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$$i = \left[\begin{array}{l} Push\ \frac{3}{6}@H, Call\ 0, Halt, Push\ 1@L, Push\ 0@L, \\ Store, Return\ 0 \end{array}\right]$$

pc	m	s	$i(pc)$
$0@L$	$[0@L]$	$[\,]$	$Push\ \frac{3}{6}@H$
$1@L$	$[0@L]$	$\left[\frac{3}{6}@H\right]$	$Call\ 0$
Machine 1 continues...			
$3@H$	$[0@L]$	$[R(2@L)]$	$Push\ 1@L$
$4@H$	$[0@L]$	$[1@L, R(2@L)]$	$Push\ 0@L$
$5@H$	$[0@L]$	$[0@L, 1@L, R(2@L)]$	$Store$
$6@H$	$[1@L]$	$[R(2@L)]$	$Return\ 0$
$2@L$	$[1@L]$	$[\,]$	$Halt$
Machine 2 continues...			
$6@H$	$[0@L]$	$[R(2@L)]$	$Return\ 0$
$2@L$	$[0@L]$	$[\,]$	$Halt$

Figure 13. Counterexample to STORE*DE rule: Raising pc label is not enough to prevent implicit flows. Once we have a mechanism (like *Return*) for restoring the pc label, we need to be more careful about stores in high contexts.

this stack frame (like ns, ns' stands for a list of labeled integers; in particular it cannot contain return addresses).

$$\frac{i(pc) = Return\ k' \quad k' \in \{0,1\} \quad ns = n_1@\ell_1 : \ldots : n_{k'}@\ell_{k'}}{\boxed{pc@\ell_{pc}}\,\boxed{ns : ns' : R(n@\ell_n) : s}\,\boxed{m} \Rightarrow \boxed{n@\ell_n}\,\boxed{ns : s}\,\boxed{m}} \quad (\text{RETURN*AB})$$

Finally, we observe that we cannot expect the current EENI instantiation to hold for this changed machine, since now one machine can halt in a high state while the other can continue, return to a low state, and only then halt. Since we cannot equate high and low states (see Section A.2 for a counterexample; again understanding it requires reading the rest of this section), we need to change the EENI instance we use to $\text{EENI}_{Init, Halted \cap Low, \approx_{mem}}$, where Low denotes the set of states with $\ell_{pc} = L$. Thus, we only consider executions that end in a low halting state.

After these changes, we can turn QuickCheck loose and start finding more bugs. The first one, listed in Figure 13, is essentially another instance of the implicit flow bug, which is not surprising given the discussion at the end of the previous subsection. We adapted the Store rule trivially to the new setting, but that is clearly not enough:

$$\frac{i(pc) = Store \quad m(p) = n'@\ell'_n \quad \ell_p \sqsubseteq \ell'_n \quad m' = m[p := n@(\ell_n \vee \ell_p)]}{\boxed{pc@\ell_{pc}}\,\boxed{p@\ell_p : n@\ell_n : s}\,\boxed{m} \Rightarrow \boxed{(pc+1)@\ell_{pc}}\,\boxed{s}\,\boxed{m'}} \quad (\text{STORE*DE})$$

We need to change this rule so that the value written in memory is tainted with the current pc label:

$$\frac{i(pc) = Store \quad m(p) = n'@\ell'_n \quad \ell_p \sqsubseteq \ell'_n \quad m' = m[p := n@(\ell_n \vee \ell_p \vee \ell_{pc})]}{\boxed{pc@\ell_{pc}}\,\boxed{p@\ell_p : n@\ell_n : s}\,\boxed{m} \Rightarrow \boxed{(pc+1)@\ell_{pc}}\,\boxed{s}\,\boxed{m'}} \quad (\text{STORE*E})$$

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$$i = \left[\begin{array}{l} Push\ \frac{3}{6}@H, Call\ 0, Halt, Push\ 0@L, Push\ 0@L, \\ Store, Return\ 0 \end{array}\right]$$

pc	m	s	$i(pc)$
$0@L$	$[0@L]$	$[\,]$	$Push\ \frac{3}{6}@H$
$1@L$	$[0@L]$	$\left[\frac{3}{6}@H\right]$	$Call\ 0$
Machine 1 continues...			
$3@H$	$[0@L]$	$[R(2@L)]$	$Push\ 0@L$
$4@H$	$[0@L]$	$[0@L, R(2@L)]$	$Push\ 0@L$
$5@H$	$[0@L]$	$[0@L, 0@L, R(2@L)]$	$Store$
$6@H$	$[0@H]$	$[R(2@L)]$	$Return\ 0$
$2@L$	$[0@H]$	$[\,]$	$Halt$
Machine 2 continues...			
$6@H$	$[0@L]$	$[R(2@L)]$	$Return\ 0$
$2@L$	$[0@L]$	$[\,]$	$Halt$

Figure 14. Counterexample to STORE*D: Implicit flow via labels.

This eliminates the current counterexample; QuickCheck then finds a very similar one in which the *labels* of values in the memories differ between the two machines (Figure 14). The usual way to prevent this problem is to extend the no-sensitive-upgrades check so that low-labeled data cannot be overwritten in a high context (Zdancewic, 2002; Austin & Flanagan, 2009). This leads to the correct rule for stores:

$$\frac{i(pc) = Store \quad m(p) = n'@\ell'_n \quad \ell_p \vee \ell_{pc} \sqsubseteq \ell'_n \quad m' = m[p := n@(\ell_n \vee \ell_p \vee \ell_{pc})]}{\boxed{pc@\ell_{pc}}\ \boxed{p@\ell_p : n@\ell_n : s}\ \boxed{m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc}}\ \boxed{s}\ \boxed{m'}} \ (\textsc{Store})$$

The next counterexample found by QuickCheck (Figure 15) shows that returning values from a high context to a low one is unsound if we do not label those values as secrets. To fix this, we taint all the returned values with the pre-return pc label.

$$\frac{\begin{array}{c} i(pc) = Return\ k' \quad k' \in \{0,1\} \quad ns = n_1@\ell_1 : \ldots : n_{k'}@\ell_{k'} \\ ns_{pc} = n_1@(\ell_1 \vee \ell_{pc}) : \ldots : n_{k'}@(\ell_{k'} \vee \ell_{pc}) \end{array}}{\boxed{pc@\ell_{pc}}\ \boxed{ns : ns' : R(n@\ell_n) : s}\ \boxed{m} \Rightarrow \boxed{n@\ell_n}\ \boxed{ns_{pc} : s}\ \boxed{m}} \ (\textsc{Return*b})$$

The next counterexample, listed in Figure 16, shows (maybe somewhat surprisingly) that it is unsound to specify the number of results to return in the *Return* instruction, because then the number of results returned may depend on secret flows of control. To restore soundness, we need to pre-declare at each *Call* whether the corresponding *Return* will return a value—i.e., the *Call* instruction should be annotated with *two* integers, one for parameters and the other for results. Stack elements e are accordingly either labeled values $n@\ell$ or (labelled) pairs of a return address n and the number of return values k.

$$e ::= n@\ell \mid R(n,k)@\ell$$

These changes lead us to the correct rules:

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$$i = \begin{bmatrix} \mathit{Push}\ 1@L, \mathit{Push}\ \frac{7}{6}@H, \mathit{Call}\ 1, \mathit{Push}\ 0@L, \mathit{Store}, \\ \mathit{Halt}, \mathit{Push}\ 0@L, \mathit{Return}\ 1 \end{bmatrix}$$

pc	m	s	$i(pc)$
$0@L$	$[0@L]$	$[\,]$	$\mathit{Push}\ 1@L$
$1@L$	$[0@L]$	$[1@L]$	$\mathit{Push}\ \frac{7}{6}@H$
$2@L$	$[0@L]$	$\left[\frac{7}{6}@H, 1@L\right]$	$\mathit{Call}\ 1$
Machine 1 continues...			
$7@H$	$[0@L]$	$[1@L, R(3@L)]$	$\mathit{Return}\ 1$
$3@L$	$[0@L]$	$[1@L]$	$\mathit{Push}\ 0@L$
$4@L$	$[0@L]$	$[0@L, 1@L]$	Store
$5@L$	$[1@L]$	$[\,]$	Halt
Machine 2 continues...			
$6@H$	$[0@L]$	$[1@L, R(3@L)]$	$\mathit{Push}\ 0@L$
$7@H$	$[0@L]$	$[0@L, 1@L, R(3@L)]$	$\mathit{Return}\ 1$
$3@L$	$[0@L]$	$[0@L]$	$\mathit{Push}\ 0@L$
$4@L$	$[0@L]$	$[0@L, 0@L]$	Store
$5@L$	$[0@L]$	$[\,]$	Halt

Figure 15. Counterexample to RETURN*AB: *Return* needs to taint the returned values.

$$i = \begin{bmatrix} \mathit{Push}\ 0@L, \mathit{Push}\ \frac{6}{7}@H, \mathit{Call}\ 0, \mathit{Push}\ 0@L, \mathit{Store}, \\ \mathit{Halt}, \mathit{Return}\ 0, \mathit{Push}\ 0@L, \mathit{Return}\ 1 \end{bmatrix}$$

pc	m	s	$i(pc)$
$0@L$	$[0@L]$	$[\,]$	$\mathit{Push}\ 0@L$
$1@L$	$[0@L]$	$[0@L]$	$\mathit{Push}\ \frac{6}{7}@H$
$2@L$	$[0@L]$	$\left[\frac{6}{7}@H, 0@L\right]$	$\mathit{Call}\ 0$
Machine 1 continues...			
$6@H$	$[0@L]$	$[R(3@L), 0@L]$	$\mathit{Return}\ 0$
$3@L$	$[0@L]$	$[0@L]$	$\mathit{Push}\ 0@L$
$4@L$	$[0@L]$	$[0@L, 0@L]$	Store
$5@L$	$[0@L]$	$[\,]$	Halt
Machine 2 continues...			
$7@H$	$[0@L]$	$[R(3@L), 0@L]$	$\mathit{Push}\ 0@L$
$8@H$	$[0@L]$	$[0@L, R(3@L), 0@L]$	$\mathit{Return}\ 1$
$3@L$	$[0@L]$	$[0@H, 0@L]$	$\mathit{Push}\ 0@L$
$4@L$	$[0@L]$	$[0@L, 0@H, 0@L]$	Store
$5@L$	$[0@H]$	$[0@L]$	Halt

Figure 16. Counterexample to CALL*B and RETURN*B: It is unsound to choose how many results to return on *Return*.

$$\frac{i(pc) = \mathit{Call}\ k\ k' \qquad k' \in \{0,1\} \qquad ns = n_1@\ell_1 : \ldots : n_k@\ell_k}{\boxed{pc@\ell_{pc}}\ \boxed{n@\ell_n : ns : s}\ \boxed{m} \Rightarrow \boxed{n@(\ell_n \vee \ell_{pc})}\ \boxed{ns : R(pc{+}1, k')@\ell_{pc} : s}\ \boxed{m}} \quad \text{(CALL)}$$

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$$i = \left[\begin{array}{l} \mathit{Push}\ 5@L, \mathit{Call}\ 0\ 1, \mathit{Push}\ 0@L, \mathit{Store}, \mathit{Halt}, \\ \mathit{Push}\ 0@L, \mathit{Push}\ \frac{8}{9}@H, \mathit{Call}\ 0\ 0, \mathit{Pop}, \mathit{Push}\ 0@L, \\ \mathit{Return} \end{array}\right]$$

pc	*m*	*s*	*i*(*pc*)
$0@L$	$[0@L]$	$[]$	*Push* $5@L$
$1@L$	$[0@L]$	$[5@L]$	*Call* 0 1
$5@L$	$[0@L]$	$[R(2,1)@L]$	*Push* $0@L$
$6@L$	$[0@L]$	$[0@L, R(2,1)@L]$	*Push* $\frac{8}{9}@H$
$7@L$	$[0@L]$	$\left[\frac{8}{9}@H, 0@L, R(2,1)@L\right]$	*Call* 0 0
Machine 1 continues...			
$8@H$	$[0@L]$	$[R(8,0)@L, 0@L, R(2,1)@L]$	*Pop*
$9@H$	$[0@L]$	$[0@L, R(2,1)@L]$	*Push* $0@L$
$10@H$	$[0@L]$	$[0@L, 0@L, R(2,1)@L]$	*Return*
$2@L$	$[0@L]$	$[0@H]$	*Push* $0@L$
$3@L$	$[0@L]$	$[0@L, 0@H]$	*Store*
$4@L$	$[0@H]$	$[]$	*Halt*
Machine 2 continues...			
$9@H$	$[0@L]$	$[R(8,0)@L, 0@L, R(2,1)@L]$	*Push* $0@L$
$10@H$	$[0@L]$	$[0@L, R(8,0)@L, 0@L, R(2,1)@L]$	*Return*
$8@L$	$[0@L]$	$[0@L, R(2,1)@L]$	*Pop*
$9@L$	$[0@L]$	$[R(2,1)@L]$	*Push* $0@L$
$10@L$	$[0@L]$	$[0@L, R(2,1)@L]$	*Return*
$2@L$	$[0@L]$	$[0@L]$	*Push* $0@L$
$3@L$	$[0@L]$	$[0@L, 0@L]$	*Store*
$4@L$	$[0@L]$	$[]$	*Halt*

Figure 17. Counterexample to Pop*: It is unsound not to protect the call stack.

$$\frac{\begin{array}{c} i(pc) = \mathit{Return} \qquad ns = n_1@\ell_1 : \ldots : n_{k'}@\ell_{k'} \\ ns_{pc} = n_1@(\ell_1 \vee \ell_{pc}) : \ldots : n_{k'}@(\ell_{k'} \vee \ell_{pc}) \end{array}}{\boxed{pc@\ell_{pc}}\boxed{ns : ns' : R(n,k')@\ell : s}\boxed{m} \Rightarrow \boxed{n@\ell}\boxed{ns_{pc} : s}\boxed{m}} \quad (\textsc{Return})$$

The final counterexample found by QuickCheck is quite a bit longer (see Figure 17). It shows that we cannot allow instructions like *Pop* to remove return addresses from the stack, as does the following broken rule (recall that e denotes an arbitrary stack entry):

$$\frac{i(pc) = \mathit{Pop}}{\boxed{pc@\ell_{pc}}\boxed{e : s}\boxed{m} \Rightarrow \boxed{(pc+1)@\ell_{pc}}\boxed{s}\boxed{m}} \quad (\textsc{Pop}^*)$$

To protect the call frames on the stack, we change this rule to only pop integers (all the other rules can already only operate on integers).

$$\frac{i(pc) = \mathit{Pop}}{\boxed{pc@\ell_{pc}}\boxed{n@\ell_n : s}\boxed{m} \Rightarrow \boxed{(pc+1)@\ell_{pc}}\boxed{s}\boxed{m}} \quad (\textsc{Pop})$$

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

5.3 Generation by execution and control flow

Generation by execution is still applicable in the presence of interesting control flow but we have to make small modifications to the original algorithm. We still generate a single instruction or sequence[2] that does not crash, as before, and we execute it to compute a new state. However, unlike before, while executing this newly generated sequence of instructions, it is possible to "land" in a position in the instruction stream where we have already generated an instruction (e.g., via a backward jump). If this happens, then we keep executing the already generated instructions. If the machine halts (or we reach a loop-avoiding cutoff), then we stop the process and return the so-far generated instruction stream. If there are no more instructions to execute, then we go on to generate more instructions. There is one more possibility though: the machine may *crash* while executing an already generated instruction. To address this issue, we make sure that we never generate an instruction (e.g., a jump) that causes the machine to crash in a certain number of steps. We refer to this number of steps as the *lookahead* parameter and in our experiments we use a lookahead of 2 steps. If we cannot generate any instruction satisfying this constraint, we retry with a smaller lookahead, until we succeed.

Since it now becomes possible to generate instruction streams that cause the machine to crash in some number of steps, one might be worried about the discard ratio for EENI. However, the ever increasing probability of generating a *Halt* (discussed in Section 4.5) counterbalances this issue.

5.4 Finding the bugs

We experimentally evaluated the effectiveness of testing for this new version of the stack machine, by adding the bugs discussed in this section to the ones applicable for the previous machine. The results of generation by execution with lookahead for this machine are shown in the first column of Figure 18. As we can see, all old bugs are still found relatively fast. It takes longer to find them when compared to the previous machine, but this is to be expected: when we extend the machine, we are also increasing the state space to be explored. The new control-flow-specific bugs are all found, with the exception of Pop*, which requires a larger timeout. Discard rates are much higher compared to generation by execution in Figure 5, for two reasons. First, control flow can cause loops, so we discard machines that run for more than 50 steps without halting. Detailed profiling revealed that 18% of the pairs of machines both loop, and loopy machines push the average number of execution steps to 22. Second, as described previously, generation by execution in the presence of control flow is much less accurate.

[2] In addition to the instruction sequences from Section 4.3, we use two new sequences for *Jump* and *Call*: [*Push ia*@ℓ, *Jump*] and [*Push* n_k@ℓ_k, ..., *Push* n_1@ℓ_1, *Push ia*@ℓ, *Call k k'*], where *ia* is a valid memory address.

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

Tested property	EENI	EENI	EENI	LLNI	SSNI	SSNI
Starting states	*Init*	*Init*	*QInit*	*QInit*	*All*	*All*
Equivalence relation	$\approx_{mem}$	$\approx_{low}$	$\approx_{low}$	$\approx_{low}$	$\approx_{full}$	$\approx_{full}$
Generation strategy	*ByExec2*	*ByExec2*	*ByExec2*	*ByExec2*	*Naive*	*Tiny*
ADD*	37.07	2.38	1.38	0.36	0.24	0.11
PUSH*	0.22	0.02	0.02	0.01	1.06	0.06
LOAD*	155.07	37.50	5.73	1.14	3.25	0.61
STORE*A	20018.67	18658.56	124.78	84.08	289.63	16.32
STORE*B	13.02	12.87	16.10	5.25	31.11	0.33
STORE*C	0.35	0.34	0.33	0.08	0.73	0.03
JUMP*A	48.84	7.58	5.26	0.08	1.45	0.09
JUMP*B	2421.99	158.36	104.62	2.80	16.88	0.49
STORE*D	13289.39	12295.65	873.79	232.19	8.77	1.13
STORE*E	1047.56	1129.48	717.72	177.75	2.26	0.29
CALL*A	3919.08	174.66	115.15	5.97	31.71	0.62
RETURN*A	12804.51	4698.17	1490.80	337.74	1110.09	3.10
CALL*B+RETURN*B	69081.50	6940.67	1811.66	396.37	1194.30	4.56
POP*	—	51753.13	16107.22	1828.56	30.68	0.42
MTTF arithmetic mean	**—**	**6847.81**	**1526.75**	**219.46**	**194.44**	**2.01**
MTTF geometric mean	**—**	**135.76**	**46.48**	**7.69**	**12.87**	**0.47**
Average tests / second	2795	2797	2391	1224	8490	18407
Average discard rate	65%	65%	69%	0%	40%	9%

Figure 18. Experiments for control flow machine. MTTF given in milliseconds.

5.5 Alternative generation strategies

Generation by execution has proved effective in finding bugs. Even this method, however, required some tuning, driven by experimental evaluation. For instance, our first implementations did not involve gradually increasing the probability of *Halt* instructions. We also experimented with different lookahead values. Larger lookaheads introduce significant overheads in generation as every generated instruction costs many steps of execution, and the payoff of lower discard rates was not worth the increased generation cost.

We have also explored (and dismissed) several other generation strategies, and we outline two of these below:

- *Generation by forward execution.* Generation by execution fills in the instruction stream in patches, due to generated jumps. It is hence possible for the instruction stream to contain "holes" filled with *Noop* instructions. An alternative strategy is to generate instructions in a forward style only: if we generate a branch, then we *save* the current state along with the branch target, but keep generating instructions as if the branch was not taken. If we ever reach the target of the branch, we may use the saved state as a potentially more accurate state that we can use to generate more instructions. This strategy delivered similar results as generation by execution, but due to its more complicated nature we dismissed it and used the basic design instead.

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

- *Variational generation by execution.* In this strategy, we first generate a machine with generation by execution. We then *vary* the machine and run generation by execution for the resulting machine, in the hope that we can fill in the holes in the originally generated instruction stream with instructions from a variational execution. As before, we did not find that the results justified the generation overheads and complexity of this strategy.

6 Strengthening the tested property

The last few counterexamples in Section 5.2 are fairly long and quite difficult for QuickCheck to find, even with the best test-generation strategy. In this section, we explore a different approach: strengthening the *property* we are testing so that counterexamples become shorter and easier to find. Figure 18 in Section 5.4 summarizes the variants of noninterference that we consider and how they affect test performance.

6.1 Making entire low states observable

Every counterexample that we've seen involves pushing an address, executing a *Store* instruction, and halting. These steps are all necessary because of the choice we made in Section 2.3 to ignore the stack when defining indistinguishability on machine states. A counterexample that leaks a secret onto the stack must continue by storing it into memory; similarly, a counterexample that leaks a secret into the *pc* must execute *Store* at least twice. This suggests that we can get shorter counterexamples by redefining indistinguishability as follows:

6.1 Definition: Machine states $S_1 = \boxed{pc_1 \mid s_1 \mid m_1 \mid i_1}$ and $S_2 = \boxed{pc_2 \mid s_2 \mid m_2 \mid i_2}$ are indistinguishable with respect to entire low states, written $S_1 \approx_{low} S_2$, if either $\ell_{pc_1} = \ell_{pc_2} = H$ or else $\ell_{pc_1} = \ell_{pc_2} = L$, $m_1 \approx m_2$, $i_1 \approx i_2$, $s_1 \approx s_2$, and $pc_1 \approx pc_2$.

We now strengthen $\text{EENI}_{Init,Halted \cap Low,\approx_{mem}}$, the property we have been testing so far, to $\text{EENI}_{Init,Halted \cap Low,\approx_{low}}$; this is stronger because $\approx_{mem}$ and $\approx_{low}$ agree on initial states, while for halted states $\approx_{low} \subset \approx_{mem}$. Indeed, for this stronger property, QuickCheck finds bugs faster (compare the first two columns of Figure 18).

6.2 Quasi-initial states

Many counterexamples begin by pushing values onto the stack and storing values into memory. This is necessary because each test starts with an empty stack and low, zeroed memory. We can make counterexamples easier to find by allowing the two machines to start with arbitrary (indistinguishable) stacks and memories; we call such states *quasi-initial*. Formally, the set *QInit* of quasi-initial states contains all states of the form $\boxed{0@L \mid s \mid m \mid i}$, for arbitrary *s*, *m*, and *i*.

The advantage of generating more varied start states is that parts of the state space may be difficult to reach by running generated code from an initial state; for example, to get two return addresses on the stack, we must successfully execute

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

two *Call* instructions (see e.g., Figure 17). Thus, bugs that are only manifested in these hard-to-reach states may be discovered very slowly or not at all. Generating "intermediate" states directly gives us better control over their distribution, which can help eliminate such blind spots in testing. The disadvantage of this approach is that a quasi-initial state may not be *reachable* from any initial state, so in principle QuickCheck may report spurious problems that cannot actually arise in any real execution. In general, we could address such problems by carefully formulating the important invariants of reachable states and ensuring that we generate quasi-initial states satisfying them. In practice, though, for this extremely simple machine, we have not encountered any spurious counterexamples, even with quasi-initial states. (This is different for the more complex register machine from Section 8; in that setting a generator for non-initial states needs to produce only states satisfying strong invariants associated with reachable states.)

Instantiating EENI with *QInit*, we obtain a stronger property $\text{EENI}_{QInit,Halted \cap Low,\approx_{low}}$ (stronger because $Init \subset QInit$) that finds bugs faster, as column 3 of Figure 18 shows.

6.3 LLNI: Low-lockstep noninterference

While making the full state observable and starting from quasi-initial states significantly improves EENI, we can get even better results by moving to a yet stronger noninterference property. The intuition is that EENI generates machines and runs them for a long time, but it only compares the final states, and only when both machines successfully halt; these preconditions lead to rather large discard rates. Why not compare *intermediate* states as well, and report a bug as soon as intermediate states are distinguishable? While the *pc* is high, the two machines may be executing different instructions, so their states will naturally differ; we therefore ignore these states and require only that low execution states are pointwise indistinguishable. We call this new property *low-lockstep noninterference* (or *LLNI*). We write $S \Rightarrow_t^*$ when an execution from state S produces trace t (a list of states). Since this is just a state-collecting variant of the reflexive transitive closure of $\Rightarrow$, we allow partial executions and in particular do not require that the last state in the trace is stuck or halting.

6.2 Definition: A machine semantics is *low-lockstep noninterfering* with respect to the indistinguishability relation $\approx$ (written $\text{LLNI}_{\approx}$) if, for any quasi-initial states S_1 and S_2 if with $S_1 \approx S_2$, $S_1 \Rightarrow_{t_1}^*$, and $S_2 \Rightarrow_{t_2}^*$, we have $t_1 \approx^* t_2$, where indistinguishability on traces $\approx^*$ is defined inductively by the following rules:

$$\frac{S_1, S_2 \in Low \quad S_1 \approx S_2 \quad t_1 \approx^* t_2}{(S_1 : t_1) \approx^* (S_2 : t_2)} \qquad \text{(Low Lockstep)}$$

$$\frac{S_1 \notin Low \quad t_1 \approx^* t_2}{(S_1 : t_1) \approx^* t_2} \qquad \text{(High Filter)}$$

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$$\frac{}{t \approx^* []} \quad \text{(End)}$$

$$\frac{t_1 \approx^* t_2}{t_2 \approx^* t_1} \quad \text{(Symmetry)}$$

The rule Low Lockstep requires low states in the two traces to be pointwise indistinguishable, while High Filter (together with Symmetry) simply filters out high states from either trace. The remaining rule is about termination: because we are working with termination-insensitive noninterference, we allow one of the traces to continue (maybe forever) even if the other has terminated. We implement these rules in Haskell as a recursive predicate over finite traces.

In general, LLNI implies EENI (see Appendix Appendix B.), but not vice versa. However, the correct version of our machine does satisfy LLNI, and we have not observed any cases where QuickChecking a buggy machine with LLNI finds a bug that is not also a bug with regard to EENI. Testing LLNI instead of EENI leads to significant improvement in the bug detection rate for all bugs, as the results in the fourth column Figure 18 show. In these experiments, no generated machine states are discarded, since LLNI applies to both successful (halting) executions and failing executions. The generation strategies described in Section 4 apply to LLNI without much change; also, as for EENI, generation by execution (with lookahead of two steps) performs better than the more basic strategies, so we don't consider those for LLNI.

6.4 SSNI: Single-step noninterference

Until now, we have focused on using sophisticated (and potentially slow) techniques for generating long-running initial (or quasi-initial) machine states, and then checking equivalence for low halting states (EENI) or at every low step (LLNI). An alternative is to define a stronger property that talks about *all* possible single steps of execution starting from two indistinguishable states.

Proofs of noninterference usually go by induction on a pair of execution traces; to preserve the corresponding invariant, the proof needs to consider how each execution step affects the indistinguishability relation. This gives rise to properties known as "unwinding conditions" (Goguen & Meseguer, 1984); the corresponding conditions for our machine form a property we call *single-step noninterference* (*SSNI*).

We start by observing that LLNI implies that, if two low states are indistinguishable and each takes a step to another low state, then the resulting states are also indistinguishable. However, this alone is not a strong enough inductive invariant to guarantee the indistinguishability of whole traces. In particular, if the two machines perform a *Return* from a high state to a low state, we would need to conclude that the two low states are equivalent without knowing anything about the original high states. This indicates that, for SSNI, we can no longer consider all high states indistinguishable. The indistinguishability relation on high states needs to be strong enough to ensure that if both machines return to low states, those low states are also indistinguishable. Moreover, we need to ensure that if one of the machines takes a

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

step from a high state to another high state, then the old and new high states are equivalent. The following definition captures all these constraints formally.

6.3 Definition: A machine semantics is *single-step noninterfering* with respect to the indistinguishability relation $\approx$ (written $\text{SSNI}_{\approx}$) if the following conditions are satisfied:

1. For all $S_1, S_2 \in Low$, if $S_1 \approx S_2$, $S_1 \Rightarrow S_1'$, and $S_2 \Rightarrow S_2'$, then $S_1' \approx S_2'$;
2. For all $S \notin Low$ if $S \Rightarrow S'$ and $S' \notin Low$, then $S \approx S'$;
3. For all $S_1, S_2 \notin Low$, if $S_1 \approx S_2$, $S_1 \Rightarrow S_1'$, $S_2 \Rightarrow S_2'$, and $S_1', S_2' \in Low$, then $S_1' \approx S_2'$.

Note that SSNI talks about completely arbitrary states, not just (quasi-)initial ones.

The definition of SSNI is parametric in the indistinguishability relation used, and it can take some work to find the right relation. As discussed above, $\approx_{low}$ is too weak and QuickCheck can easily find counterexamples to condition 3, e.g., by choosing two indistinguishable machine states with $i = [Return]$, $pc = 0@H$, and $s = \left[R(\frac{0}{1}, 0)@L\right]$; after a single step, the two machines have distinguishable pcs $0@L$ and $1@L$, respectively. On the other hand, treating high states exactly like low states in the indistinguishability relation is too strong. In this case QuickCheck finds counterexamples to condition 2, e.g., a single machine state with $i = \left[Pop\right]$, $pc = 0@H$, and $s = [0@L]$ steps to a state with $s = []$, which would not be considered indistinguishable. These counterexamples show that indistinguishable high states can have different pcs and can have completely different stack frames at the top of the stack. So all we can require for two high states to be equivalent is that their memories and instruction memories agree and that the parts of the stacks below the topmost low return address are equivalent. This is strong enough to ensure condition 3.

6.4 Definition: Machine states $S_1 = \boxed{pc_1 \mid s_1 \mid m_1 \mid i_1}$ and $S_2 = \boxed{pc_2 \mid s_2 \mid m_2 \mid i_2}$ are *indistinguishable with respect to whole machine states*, written $S_1 \approx_{full} S_2$, if $m_1 \approx m_2$, $i_1 \approx i_2$, $\ell_{pc_1} = \ell_{pc_2}$, and additionally

- if $\ell_{pc_1} = L$ then $s_1 \approx s_2$ and $pc_1 \approx pc_2$, and
- if $\ell_{pc_1} = H$ then $cropStack\ s_1 \approx cropStack\ s_2$.

The *cropStack* helper function takes a stack and removes elements from the top until it reaches the first low return address (or until all elements are removed). (As most of our definitions so far, this definition is tailored to a 2-element lattice; we will generalize it to an arbitrary lattice in Section 8.4.)

The fifth column of Figure 18 shows that, even with arbitrary starting states generated completely naively, $\text{SSNI}_{\approx_{full}}$ performs very well. If we tweak the weights a bit and additionally observe that since we only execute the generated machine for only one step, we can begin with very small states (e.g., the instruction memory can be of size 2), then we can find all bugs very quickly. As the last column of Figure 18 illustrates, each bug is found in under 20 ms. (This last optimization is a bit risky, since we need to make sure that these very small states are still large enough to exercise all bugs we might have—e.g., an instruction memory of size 1 is not enough

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

to exhibit the CALL*B+RETURN*B bug using SSNI.) Compared to other properties, QuickCheck executes many more tests per second with SSNI for both generation strategies.

6.5 Discussion

In this section, we have seen that strengthening the noninterference property is a very effective way of improving the effectiveness of random testing our IFC machine. It is not without costs, though. Changing the security property required some expertise and, in the case of LLNI and SSNI, manual proofs showing that the new property implies EENI, the baseline security property (see Appendix Appendix B.). In the case of LLNI and SSNI, we used additional invariants of our machine (e.g., captured by $\approx_{full}$) and finding these invariants is the most creative part of doing a full security proof. While we could use the counterexamples provided by QuickCheck to guide our search for the right invariants, for more realistic machines the process of interpreting the counterexamples and manually refining the invariants is significantly harder than for our very simple machine (see Section 8).

The potential users of our techniques will have to choose a point in the continuum between testing and proving that best matches the characteristics of their practical application. At one end, we present ways of testing the original EENI property without changing it in any way, by putting all the smarts in clever generation strategies. At the other end, one can imagine using random testing just as the first step towards a full proof of a stronger property such as SSNI. For a variant of our simple stack machine, Azevedo de Amorim *et al.* (2014) did in fact prove recently in Coq that SSNI holds, and did not find any bugs that had escaped our testing. Moreover, we proved in Coq that under reasonable assumptions SSNI implies LLNI and LLNI implies EENI (see Appendix Appendix B.).

7 Shrinking strategies

The counterexamples presented in this paper are not the initial randomly generated machine states; they are the result of QuickCheck shrinking these to minimal counterexamples. For example, randomly generated counterexamples to EENI for the PUSH* bug usually consist of 20–40 instructions; the minimal counterexample uses just 4 (see Figure 2). In this section, we describe the shrinking strategies we used.

7.1 Shrinking labeled values and instructions

By default, QuickCheck already implements a shrinking strategy for integers. For labels, we shrink H to L, because we prefer to see counterexamples in which labels are only H if this is essential to the failure. Values are shrunk by shrinking either the label or the contents. If we need to shrink *both* the label and the contents, then this is achieved in two separate shrinking steps.

We allow any instruction to shrink to *Noop*, which preserves a counterexample if the instruction was unnecessary; or to *Halt*, which preserves a counterexample if

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

the bug had already manifested by the time that instruction was reached. To avoid an infinite shrinking loop, we do not shrink *Noop* at all, while *Halt* can shrink only to *Noop*. Instructions of the form *Push* $n@\ell$ are also shrunk by shrinking $n@\ell$. Finally, instructions of the form *Call* n n' are shrunk by shrinking n or n', or by being replaced with *Jump*.

7.2 Shrinking machine states

Machine states contain a data memory, a stack, an instruction memory, and the *pc*. For data memories, we can simply shrink the elements using the techniques of the previous subsection. In addition, we allow shrinking to remove arbitrary elements of the data memory completely. However, the first element that we try to remove is the last one: removing other elements changes all subsequent memory addresses, potentially invalidating the counterexample. Stacks can be shrunk similarly: we can shrink their data elements or remove them completely. We need to be extra clever in shrinking return addresses—otherwise, it is very easy to obtain crashing states. This is elaborated in the next subsection.

In the case of the instruction memory, we only try to remove *Noop* instructions, since removing other instructions is likely to change the stack or the control flow fairly drastically, and is thus likely to invalidate any counterexample. Other instructions can still be removed in two stages, by first shrinking them to a *Noop*.

Finally, we choose not to shrink the *pc*. Generation by execution works by generating valid instructions starting from the initial *pc*. Shrinking its address will most likely lead to immediate failure. One strategy we considered is *shrinking by execution*, where we shrink by taking a step in the machine. However, we didn't get a lot of benefit from such an approach. Even worse, if the indistinguishability relation is too coarse grained, then shrinking by execution can lead our states past the point where they become distinct, but are still considered equivalent; such a counterexample is not useful for debugging!

7.3 Shrinking variations

One difficulty that arises when shrinking noninterference counterexamples is that the test cases must be pairs of *indistinguishable* machines. Shrinking each machine state independently will most likely yield distinguishable pairs, which are invalid test cases, since they fail to satisfy the precondition of the property we are testing. In order to shrink effectively, we need to shrink both states of a variation *simultaneously*, and in the same way.

For instance, if we shrink one machine state by deleting a *Noop* in the middle of its instruction memory, then we must delete the same instruction in the corresponding variation. Similarly, if a particular element gets shrunk in a memory location, then the same location should be shrunk in the other state of the variation, and only in ways that produce indistinguishable states. We have implemented all of the shrinking strategies described above as operations on *pairs* of indistinguishable states, and ensured that they generate only shrinking candidates that are also indistinguishable.

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

When we use the full state equivalence $\approx_{full}$, we can shrink stacks slightly differently: we only need to synchronize shrinking steps on the *low* parts of the stacks. Since the equivalence relation ignores the high half of the stacks, we are free to shrink those parts of the two states independently, provided that high return addresses don't get transformed into low ones.

7.4 Optimizing shrinking

We applied a number of optimizations to make the shrinking process faster and more effective. One way we sped up shrinking was by turning on QuickCheck's "smart shrinking," which optimizes the order in which shrinking candidates are tried. If a counterexample a can be shrunk to any b_i, but the first k of these are not counterexamples, then it is likely that the first k shrinking candidates for b_{k+1} will not be counterexamples either, because a and b_{k+1} are likely to be similar in structure and so to have similar lists of shrinking candidates. Smart shrinking just changes the order in which these candidates are tried: it defers the first k shrinking candidates for b_{k+1} until after more likely ones have been tried. This sped up shrinking dramatically in our tests.

We also observed that many reported counterexamples contained *Noop* instructions—in some cases many of them—even though we implemented *Noop* removal as a shrinking step. On examining these counterexamples, we discovered that they could not be shrunk because removing a *Noop* changes the addresses of subsequent instructions, at least one of which was the target of a *Jump* or *Call* instruction. So to preserve the behavior of the counterexample, we needed to remove the *Noop* instruction *and adjust the target of a control transfer* in the same shrinking step. Since control transfer targets are taken off the stack, and such addresses can be generated during the test in many different ways, we simply allowed *Noop* removal to be combined with any other shrinking step—which might, for example, decrement any integer on the initial stack, or any integer stored in the initial memory, or any constant in a *Push* instruction. This combined shrinking step was much more effective in removing unnecessary *Noop*s.

Occasionally, we observed shrunk counterexamples containing two or more unnecessary *Noop*s, but where removing just one *Noop* led to a non-counterexample. We therefore used QuickCheck's *double shrinking*, which allows a counterexample to shrink in two steps to another counterexample, even if the intermediate value is not a counterexample. With this technique, QuickCheck could remove all unnecessary *Noop*s, albeit at a cost in shrinking time.

We also observed that some reported test cases contained unnecessary *sequences* of instructions, which could be elided together, but not one by one. We added a shrinking step that can replace any two instructions by *Noop*s simultaneously (and thus, thanks to double shrinking, up to four), which solved this problem.

With this combination of methods, almost all counterexamples we found shrink to minimal ones, from which no instruction, stack element, or memory element could be removed without invalidating the counterexample.

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

8 Information-flow register machine

We demonstrate the scalability of the techniques above by studying a more realistic information-flow register machine. Beyond registers (Section 8.1), this machine includes advanced features such as first-class public labels (Section 8.2) and dynamically allocated memory with mutable labels (Section 8.3). The combination of these features makes the design of sound IFC rules highly nontrivial, and thus discovering flaws early by testing even more crucial. The counterexamples produced by testing have guided us in designing a novel and highly permissive flow-sensitive dynamic enforcement mechanism and in the discovery of the sophisticated invariants needed for the final noninterference proof (Section 8.4). Most importantly for the purpose of this paper, we experimentally evaluate the scalability of our best generation strategy (generation by execution) and readily falsifiable formulations of noninterference by testing this more complex machine (Section 8.5).

8.1 The core of the register machine

The core instructions of the new machine are very similar to those of the simple stack machine from the previous sections:

$$\begin{aligned} \mathit{Instr} ::= {} & \mathit{Put}\ n\ r_d \mid \mathit{Mov}\ r_s\ r_d \mid \mathit{Load}\ r_p\ r_d \mid \mathit{Store}\ r_p\ r_s \mid \mathit{Add}\ r_1\ r_2\ r_d \mid \mathit{Mult}\ r_1\ r_2\ r_d \mid \\ & \mathit{Noop} \mid \mathit{Halt} \mid \mathit{Jump}\ r \mid \mathit{BranchNZ}\ n\ r \mid \mathit{Call}\ r_1\ r_2\ r_3 \mid \mathit{Return}. \end{aligned}$$

The main difference is that the instructions now take their arguments from and store their result into registers. We use the meta-variable r to range over a finite set of register identifiers. The register machine has no *Pop* instruction and the $\mathit{Push}\ n@\ell$ instruction of the stack machine is replaced by $\mathit{Put}\ n\ r_d$, which stores the integer constant n into the destination register r_d. $\mathit{Mov}\ r_s\ r_d$ copies the contents of register r_s into register r_d. $\mathit{Load}\ r_p\ r_d$ and $\mathit{Store}\ r_p\ r_s$ take the pointer from register r_p, and load the result into r_d or store the value of r_s into memory. On top of the familiar *Add* we also add a multiplication instruction *Mult*. *Noop*, *Halt*, and *Jump* work as before, and we additionally add a $\mathit{BranchNZ}\ n\ r$ (branch not zero) instruction that performs a conditional relative jump by adding the integer n to the *pc* if the register r contains a non-zero integer value; otherwise the *pc* is simply incremented.

While most instructions of the new machine only work with registers, the *Call* and *Return* instructions also use a (protected) call stack. We hard-code a simple calling convention in which the values of all registers are passed from the callee to the caller, but only one register is used for passing back a result value, while all the other registers are automatically restored to the values before the call. Similarly to the simple stack machine, where to enforce noninterference it was necessary to specify the number of returned values on calls, on the register machine the register in which the result is returned and the label of the result are both specified on *Call*, saved on the stack, and used on the corresponding *Return*.

The states of the register machine have the form $\boxed{pc@\ell_{pc}}\boxed{rf}\boxed{cs}\boxed{m}\boxed{i}$ and include the register file (*rf*, mapping register identifiers and to their values) and the call stack (*cs*, a list of stack frames). The labeled *pc*, the data memory *m*, and the instruction

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

memory i (omitted by convention below) are familiar from the stack machine. The stepping rules for most instructions above are simple; for instance:

$$\frac{i(pc) = \mathit{Mult}\ r_1\ r_2\ r_d \qquad rf[r_1] = n_1@\ell_1 \qquad rf[r_2] = n_2@\ell_2 \qquad rf' = rf[r_d := (n_1 \times n_2)@(\ell_1 \vee \ell_2)]}{\boxed{pc@\ell_{pc} \mid rf \mid cs \mid m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc} \mid rf' \mid cs \mid m}} \quad (\textsc{Mult})$$

The interaction between public labels and *Call* and *Return*, however, is complex; we discuss it in detail in Section 8.2. The precise structure of the data memory and the rules for *Load* and *Store* are discussed in Section 8.3.

8.2 First-class public labels

First-class public labels are an important feature of several recent IFC systems for functional programming languages (Stefan *et al.*, 2011; Hriţcu *et al.* 2013a). They support the development of realistic applications in which new principals and labels are created dynamically (Giffin *et al.*, 2012), and they are a key ingredient in recently proposed mechanisms for soundly recovering from IFC violations (Stefan *et al.*, 2012; Hriţcu *et al.* 2013a). While for simplicity, we consider neither dynamic labels nor recoverable exceptions, our register machine does have first-class public labels. Enforcing noninterference for public labels is highly nontrivial, especially in the presence of varying memory labels (Section 8.3).

Even before that, adding a *LabelOf* $r_s\ r_d$ instruction that puts in r_d the label of the value in r_s as a public first-class value is unsound for a label lattice with at least three elements and a standard mechanism for restoring the *pc* on control flow merge point (e.g., the *Call*s and *Return*s of the stack machine in Section 5.2). In a functional language, we would write the counterexample as follows:

$$\mathit{LabelOf}\ (\mathsf{if}\ m\ \mathsf{then}\ 0@M\ \mathsf{else}\ 0@H).$$

This encodes the secret bit m protected by label M (where $L \sqsubseteq M \sqsubseteq H$) by varying the *label* of the conditional's result (M on the if branch and H on the else one), and then uses *LabelOf* to expose that label. Note that the counterexample involves two different labels (in our case M and H) that are higher or equal in lattice order ($\sqsubseteq$) than the label of a secret (M); a two-point lattice is not rich enough.

To express the same counterexample on our low-level machine, consider two indistinguishable states $\boxed{0@L \mid rf_1 \mid [\,] \mid [\,]}$ and $\boxed{0@L \mid rf_2 \mid [\,] \mid [\,]}$ differing only in the value of the r_0 register, which contains a secret bit labeled M: $rf_1[r_0] = 0@M$ while $rf_2[r_0] = 1@M$. We use three more registers with the same value in both states: $rf_1[r_1] = rf_2[r_1] = 3@L$, $rf_1[r_2] = rf_2[r_2] = 0@M$, and $rf_1[r_3] = rf_2[r_3] = 0@H$. The instruction memory of both machines contains the following program:

$$[\mathit{Call}\ r_1\ r_3,\ \mathit{LabelOf}\ r_3\ r_0,\ \mathit{Halt},\ \mathit{BranchNZ}\ 2\ r_0,\ \mathit{Mov}\ r_2\ r_3,\ \mathit{Return}].$$

The *Call* transfers control to the *BranchNZ* instruction and specifies r_3 as the return register. Together with the *Return*, this ensures that the *pc* label is restored to L after branching on the secret bit via *BranchNZ*. The *BranchNZ* ensures that the *Mov* is executed when r_0 is $0@M$ (the first execution) and skipped when r_0 is $1@M$ (the

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

second one). The *pc* is raised to M on the *BranchNZ*, but that cannot have any effect on the *Mov* since the labels of the values in r_2 and r_3 are anyway higher or equal than M. Similarly, the *Return* could potentially join the *pc* label M to the label of the returned register r_3, but that would again have no effect. After the *Return*, the *pc* is again labeled L and the register r_3 stores $0@M$ in the first execution and $0@H$ in the second. Executing *LabelOf* exposes this label difference to the value level. In the end, the first machine halts with $M@L$ in register r_3, while the second one halts with $H@L$ in r_3, a distinguishable difference.

Following Hriţcu *et al.* (2013a) and Stefan *et al.* (2011), we can solve this problem by separating the choice of label (which needs to be done in a low context) from the computation of the labeled data (which happens in a high context). Concretely, we require the programmer to specify the label of the result of each procedure as part of the *Call* (in our simple instruction language the first-class label is first put in a register with *PutLabel*, and only then passed to the *Call*). In the example above, the *Return* succeeds on both branches only if the *Call* is annotated with H, i.e., a label that is more secure than the label of the result register on either branch:

$$[\mathit{PutLabel}\ H\ r_4,\ \mathit{Call}\ r_1\ r_3\ r_4,\ \mathit{LabelOf}\ r_3\ r_0,\ \mathit{Halt},\ \mathit{BranchNZ}\ 2\ r_0,\ \mathit{Mov}\ r_2\ r_3,\ \mathit{Return}].$$

Regardless of which branch is chosen, the procedure will return $0@H$ in register r_3, thus preventing m from being leaked via the labels. Concretely, the new *PutLabel* instruction loads the label H into register r_4, and then this register is passed as a third argument to the *Call*. The *Call* saves the H label in the new stack frame, and the *Return* checks that the *pc* label and the label of the result in r_3 are both below or equal to H (which is trivially true on both branches) and then *raises* the label of the result to H. The check performed on *Return* would fail at least on one of the branches if the *Call* were wrongly annotated with L or M.

The rest of this subsection explains the part of our machine definition concerning first-class public labels, culminating with the stepping rules for *Call* and *Return*. For the sake of brevity, in the rest of this and the following subsections, as opposed to Section 2 and Section 5, we do not list wrong rules and the QuickCheck-found counterexamples that have guided our search for the right formulations. For testing, we consider labels drawn from a four-element diamond lattice:

$$\ell ::= L \mid M_1 \mid M_2 \mid H,$$

where $L \sqsubseteq M_1$, $L \sqsubseteq M_2$, $M_1 \sqsubseteq H$, and $M_2 \sqsubseteq H$. The labels M_1 and M_2 are incomparable. This four-element lattice was rich enough for finding all bugs introduced in the experiments from Section 8.5. With this richer lattice our definition of "low" and "high" becomes relative to an arbitrary observer label ℓ: we call ℓ_1 low with respect to ℓ if $\ell_1 \sqsubseteq \ell$ and high otherwise. The noninterference proof from Section 8.4 is parameterized over an arbitrary finite lattice.

The register machine works with labeled values $v@\ell$, where v is an integer n or a first-class label ℓ or a pointer (more on pointers in Section 8.3). Besides *PutLabel* and *LabelOf*, which we have seen above, we add three more new instructions that

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

work with first-class labels:

$$Instr ::= \ldots \mid PutLabel\ \ell\ r_d \mid LabelOf\ r_s\ r_d \mid PcLabel\ r_d \mid Join\ r_1\ r_2\ r_d \mid FlowsTo\ r_1\ r_2\ r_d.$$

PcLabel returns the label of the *pc*, while *Join* and *FlowsTo* compute $\vee$ and $\sqsubseteq$ on first-class labels. The stepping rules for these new instructions are all very simple:

$$\frac{i(pc) = PutLabel\ \ell\ r_d \qquad rf' = rf[r_d := \ell @ L]}{\boxed{pc@\ell_{pc} \mid rf \mid cs \mid m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc} \mid rf' \mid cs \mid m}} \quad (\textsc{PutLabel})$$

$$\frac{i(pc) = LabelOf\ r_s\ r_d \qquad rf[r_s] = n@\ell \qquad rf' = rf[r_d := \ell @ L]}{\boxed{pc@\ell_{pc} \mid rf \mid cs \mid m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc} \mid rf' \mid cs \mid m}} \quad (\textsc{LabelOf})$$

$$\frac{i(pc) = PcLabel\ r_d \qquad rf' = rf[r_d := \ell_{pc} @ L]}{\boxed{pc@\ell_{pc} \mid rf \mid cs \mid m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc} \mid rf' \mid cs \mid m}} \quad (\textsc{PcLabel})$$

$$\frac{\begin{array}{c} i(pc) = Join\ r_1\ r_2\ r_d \qquad rf[r_1] = \ell_1 @ \ell'_1 \qquad rf[r_2] = \ell_2 @ \ell'_2 \\ rf' = rf[r_d := (\ell_1 \vee \ell_2) @ (\ell'_1 \vee \ell'_2)] \end{array}}{\boxed{pc@\ell_{pc} \mid rf \mid cs \mid m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc} \mid rf' \mid cs \mid m}} \quad (\textsc{Join})$$

$$\frac{\begin{array}{c} i(pc) = FlowsTo\ r_1\ r_2\ r_d \qquad rf[r_1] = \ell_1 @ \ell'_1 \qquad rf[r_2] = \ell_2 @ \ell'_2 \\ n = \text{if } \ell_1 \sqsubseteq \ell_2 \text{ then 1 else 0} \qquad rf' = rf[r_d := n @ (\ell'_1 \vee \ell'_2)] \end{array}}{\boxed{pc@\ell_{pc} \mid rf \mid cs \mid m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc} \mid rf' \mid cs \mid m}} \quad (\textsc{FlowsTo})$$

Note that result of the *LabelOf* instruction is a "label value" that is itself labeled *L*. So in a low context the labels of values in registers—even labels on secret data—are public information.

The stepping rules for *Call* and *Return* are more complex. The *Call* $r_1\ r_2\ r_3$ instruction has three register arguments: r_1 stores the address of a procedure, r_2 is marked as a result register and is not restored on return, and r_3 stores a first-class label that is used to label the result value on return. On executing *Call* $r_1\ r_2\ r_3$, the return address ($pc{+}1$), the contents of the whole register file (rf), the return register identifier r_2, and the label in r_3 (ℓ), are all saved in a new stack frame, and control is passed to the address in r_1 (n):

$$\frac{i(pc) = Call\ r_1\ r_2\ r_3 \qquad rf[r_1] = n@\ell_n \qquad rf[r_3] = \ell @ \ell'}{\boxed{pc@\ell_{pc} \mid rf \mid cs \mid m} \Rightarrow \boxed{n@(\ell_{pc} \vee \ell_n) \mid rf \mid ((pc{+}1)@(\ell_{pc} \vee \ell'), rf, r_2, \ell) : cs \mid m}} \quad (\textsc{Call})$$

As was the case in Section 5.2, the address of the called procedure (n) can be influenced by secrets, so we join its label (ℓ_n) to the *pc* label (ℓ_{pc}). Finally, since the label used to annotate the call (ℓ) is first class, it has itself a protecting label (ℓ'), which we join to the label of the return address ($pc{+}1$).

On a *Return* the top frame on the call stack is popped, the saved *pc* and register file are restored to their previous values, with the exception of the return register whose content is preserved, so that a value is passed from the callee to the caller in this register.

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$$\frac{i(pc) = \mathit{Return} \qquad rf[r] = v@\ell \qquad \ell \vee \ell_{pc} \sqsubseteq \ell' \vee \ell'_{pc} \qquad rf'' = rf'[r := v@\ell']}{\boxed{pc@\ell_{pc}}\boxed{rf}\boxed{(n@\ell'_{pc}, rf', r, \ell') : cs}\boxed{m} \Rightarrow \boxed{n@\ell'_{pc}}\boxed{rf''}\boxed{cs}\boxed{m}} \quad \text{(Return)}$$

The label check considers the total protection of a value as the join of its explicit label and the *pc* label Hriţcu *et al.* (2013a). This check ensures that the total protection of the returned value (v) after the *Return* (label $\ell' \vee \ell'_{pc}$) is at least as strong as its protection before the return ($\ell \vee \ell_{pc}$); in other words, it prevents *Return* from declassifying the result, which would break noninterference.

8.3 Permissive flow-sensitive memory updates

Testing has helped us more easily explore the intricate space of IFC mechanisms and design a new one that addresses a current research challenge in an interesting way. The challenge we address is allowing the labels of the values in memory to vary at runtime yet still be observable. IFC systems that allow labels to change during execution are usually called *flow-sensitive*, and are generally more permissive than flow-insensitive systems that require labels to be fixed once and for all. Devising flow-sensitive dynamic IFC systems is, however, challenging. The first solutions proposed in the literature used static analysis to soundly approximate the effects of branches not taken on IFC labels (Guernic *et al.*, 2006; Guernic, 2007; Russo & Sabelfeld, 2010). Later, sound purely dynamic flow-sensitive monitors were proposed, based on dynamic checks called no-sensitive-upgrades (Zdancewic, 2002; Austin & Flanagan, 2009) (which we used for the stack machine in Section 5.2) and permissive upgrades (Austin & Flanagan, 2010; Bichhawat *et al.*, 2014b). These checks are, however, not sound when labels are observable (e.g., via the *LabelOf* instruction we introduced in Section 8.2); intuitively they allow secret information to leak into the labels of the values in memory and ensure soundness by preventing these labels from being observed. We are aware of only one flow-sensitive IFC system featuring public (i.e., observable) labels: the one recently proposed by Buiras *et al.* (2014). Our solution is similar but in many ways more permissive than the one by Buiras *et al.* (2014); on the other hand, their technique extends well to concurrency, while here we only study a sequential setting. The precise connection to Buiras *et al.* (2014) is discussed throughout this subsection.

The main idea is simple: we associate a label with each memory block and this label protects not only the values inside, but also their individual labels ((Buiras *et al.*, 2014) don't store values in blocks, but use a similar concept called "the label on the reference label"). This block label is chosen by the programmer at allocation time and is fixed throughout execution, while the label on the values in the block can vary more or less arbitrarily. The only restriction we impose is that label updates can only happen in contexts that are less classified than the label of the memory block containing the updated label. The rest of this subsection presents the technical details of our solution.

The register machine features a block-based memory model (Leroy & Blazy, 2008; Leroy *et al.*, 2012; Azevedo de Amorim *et al.*, 2014). As mentioned above, values

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

include integers, first-class labels, and pointers:

$$v ::= n \mid \ell \mid (b, o).$$

A pointer is a pair (b, o) of a block identifier b and an integer offset o. The memory m is a partial function from a block identifier to a labeled list of labeled values $vs@\ell_b$; we call ℓ_b the block label. The stepping rule for *Load* r_p r_d looks up the value of r_p in the register file, and proceeds only if it is a pointer (b, o) labeled ℓ_p:

$$\frac{i(pc) = \mathit{Load}\ r_p\ r_d \qquad rf[r_p] = (b, o)@\ell_p \qquad m[b] = vs@\ell_b \qquad vs[o] = v@\ell_v \qquad rf' = rf[r_d := v@\ell_v]}{\boxed{pc@\ell_{pc} \mid rf \mid cs \mid m} \Rightarrow \boxed{(pc{+}1)@(\ell_{pc} \vee \ell_p \vee \ell_b) \mid rf' \mid cs \mid m}} \quad (\textsc{Load})$$

It looks up the block identifier b in the memory m and if the block is allocated it obtains a list of values vs labeled by the block label ℓ_b. The result of the *Load* is a labeled value $v@\ell_v$ obtained by looking up at offset o in vs. The most interesting part is that the resulting pc label is the join of the previous pc label ℓ_{pc}, the pointer label ℓ_p, and the block label ℓ_b. Intuitively, before the load, the labels ℓ_b and ℓ_p protect the value v as well as its label ℓ_v. After the load, we could have protected the value v by joining ℓ_b and ℓ_p to ℓ_v instead of the pc label and that would have been more permissive. However, this would have left the label ℓ_v unprotected, and directly accessible via *LabelOf*, breaking noninterference.

The stepping rule for *Store* takes a labeled value $v@\ell_v$ and writes it to memory, overwriting the previous value at that location as well as its label.

$$\frac{i(pc) = \mathit{Store}\ r_p\ r_s \qquad rf[r_p] = (b, o)@\ell_p \qquad rf[r_s] = v@\ell_v \qquad m[b] = vs@\ell_b \qquad (\ell_{pc} \vee \ell_p) \sqsubseteq \ell_b \qquad vs' = vs[o := v@\ell_v] \qquad m' = m[b := vs'@\ell_b]}{\boxed{pc@\ell_{pc} \mid rf \mid cs \mid m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc} \mid rf \mid cs \mid m'}} \quad (\textsc{Store})$$

Because the previous value is overwritten its label doesn't need to be related in any way with the label of the new value. This allows for *arbitrary* label changes in memory and is thus more permissive than previous work based on upgrade operations that can only raise the label of a value in memory (Hedin & Sabelfeld, 2012; Buiras *et al.*, 2014). The label check $(\ell_{pc} \vee \ell_p) \sqsubseteq \ell_b$ ensures that the label ℓ_b that will protect $v@\ell_v$ after the store is high enough to prevent revealing information about the context in which or the pointer through which this store happened. This ensures that no program can branch on a secret and based on this change a labeled value in a memory block with a public block label, since this would be observable via *Load* as soon as the branching ends and the pc label is restored. Similarly, this ensures that no program can vary a pointer based on secrets and then use that pointer to do a store to a block with a public block label, since that block can potentially also be accessible via public pointers that can observe the stored value or its label. This is analogous to one of the checks performed for the upgrade operation of Buiras *et al.* (2014) (upgrade is further discussed below); perhaps, surprisingly this is the *only* check we need for our *Store* instruction.

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

Beyond *Load* and *Store*, we have eight other instructions that deal with pointers and memory:

$$\begin{aligned} \mathit{Instr} ::= \ldots \;&|\; \mathit{Alloc}\ r_n\ r_l\ r_d \mid \mathit{Write}\ r_p\ r_s \mid \mathit{Upgrade}\ r_p\ r_l \mid \mathit{Eq}\ r_1\ r_2\ r_d \mid \mathit{GetOffset}\ r_p\ r_d \\ &|\; \mathit{SetOffset}\ r_p\ r_o\ r_d \mid \mathit{GetBlockSize}\ r_p\ r_d \mid \mathit{GetBlockLabel}\ r_p\ r_d \end{aligned}$$

The *Alloc* $r_n\ r_l\ r_d$ instruction allocates a fresh block of size r_n with block label r_l and stores in r_d a pointer to the first position in this block. The block is initially filled with $0@L$:

$$\frac{\begin{array}{c} i(pc) = \mathit{Alloc}\ r_n\ r_l\ r_d \qquad rf\,[r_n] = n@\ell_n \qquad n > 0 \qquad rf\,[r_l] = \ell@\ell' \\ \mathit{fresh}\ m\ (\ell_{pc} \vee \ell_n \vee \ell') = b \qquad m' = m[b := [0@L, 0@L, \ldots, 0@L]@\ell] \\ rf' = rf\,[r_d := (b,0)@(\ell_n \vee \ell')] \end{array}}{\boxed{pc@\ell_{pc} \mid rf \mid cs \mid m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc} \mid rf' \mid cs \mid m'}} \quad (\textsc{Alloc})$$

The returned pointer is protected by both ℓ_n, the label of the requested block size, and by ℓ', the label of the requested block label. If the requested block size is positive and there are still blocks left, our *Alloc* rule succeeds; in particular, the block label ℓ can be chosen *arbitrarily*. This allows us to allocate a low block in high context, knowing that at the end of the high context access to these blocks will only be possible through high pointers; this invariant is a cornerstone of our noninterference proof (Section 8.4). This is more permissive than the reference allocation rule of Buiras *et al.* (2014), which can only use "the current label" (roughly analogous to our *pc* label) as "the label on the reference label" (analogous to our block label). We return to the *fresh* $m\ \ldots = b$ condition in Section 8.4.

Our *Store* instruction arbitrarily changes the label of the overwritten value. Inspired by Buiras *et al.* (2014), we additionally provide a *Write* instruction that behaves the same as *Store*, just that it keeps the label of the overwritten value unchanged:

$$\frac{\begin{array}{c} i(pc) = \mathit{Write}\ r_p\ r_s \quad rf\,[r_p] = (b,o)@\ell_p \quad rf\,[r_s] = v@\ell_v \quad m[b] = vs@\ell_b \quad vs[o]{=}v'@\ell'_v \\ (\ell_{pc} \vee \ell_p \vee \ell_v) \sqsubseteq (\ell_b \vee \ell'_v) \quad vs'{=}vs[o := v@\ell'_v] \quad m'{=}m[b := vs'@\ell_b] \end{array}}{\boxed{pc@\ell_{pc} \mid rf \mid cs \mid m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc} \mid rf \mid cs \mid m'}} \quad (\textsc{Write})$$

The label check is analogous to the one of Buiras *et al.* (2014); it can be broken into two parts: The first part, $(\ell_{pc} \vee \ell_p) \sqsubseteq (\ell_b \vee \ell'_v)$, is more permissive than the $(\ell_{pc} \vee \ell_p) \sqsubseteq \ell_b$ check of *Store*. Because the write keeps the label ℓ'_v unchanged, we do not need to additionally protect this label; we only need to protect the new value v and for this ℓ'_v can help. This allows for instance writing in a high context to a block with a low label as long as we overwrite a value previously labeled high; a *Store* would be disallowed in this setting, because it could potentially leak information via a label change. The second part of the check, $\ell_v \sqsubseteq (\ell_b \vee \ell'_v)$, ensures that the written value v is at least as protected after the write (by the block label ℓ_b and the preserved value label ℓ'_v) as it was before the write (by ℓ_v). This check was unnecessary for *Store* because the label of the stored value does not change.

Buiras *et al.* (2014) also have an upgrade operation that can raise the label of a value in memory before entering a high context. This upgrade operation can in fact

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

be faithfully encoded using our *Load* and *Store* instructions (as well as judicious use of *Call* and *Return*). For the purpose of stressing our testing methodology, we chose to include this as a primitive instruction, with the following (otherwise derivable) operational semantics rule:

$$\frac{\begin{array}{c} i(pc) = \mathit{Upgrade}\ r_p\ r_l \quad rf[r_p] = (b,o)@\ell_p \quad rf[r_l] = \ell@\ell' \quad \ell'_{pc} = \ell_{pc} \vee \ell' \\ m[b] = vs@\ell_b \qquad vs[o] = v'@\ell'_v \qquad \ell'_v \sqsubseteq (\ell \vee \ell_b) \\ (\ell'_{pc} \vee \ell_p) \sqsubseteq \ell_b \qquad vs' = vs[o := v'@\ell] \qquad m' = m[b := vs'@\ell_b] \end{array}}{\boxed{pc@\ell_{pc}}\boxed{rf}\boxed{cs}\boxed{m} \Rightarrow \boxed{(pc{+}1)@\ell'_{pc}}\boxed{rf'}\boxed{cs}\boxed{m'}} \text{ (UPGRADE)}$$

Perhaps surprisingly, this rule is more complex and more restrictive than our *Store* rule. *Store* does not have to deal with the label ℓ' protecting the first-class label ℓ, or with the label ℓ'_v of the value v'. In particular, our *Store* rule does not have the $\ell'_v \sqsubseteq (\ell \vee \ell_b)$ check, because for a *Store* the value v' is overwritten, and thus does not need to be protected in any way. An important consequence of this is that *Store* can change labels arbitrarily, while *Upgrade* can only change labels in a way that does not diminish the total protection of the existing value in memory. Using *Store* to overwrite a memory location with $0@\ell$ is thus a better way to change the label of a location whose value is no longer relevant to ℓ.

The remaining instructions are much simpler. *Eq* simply illustrates that all values, including pointers, can be compared for equality. From an IFC perspective, the rule for *Eq* is the same as the ones for for *Add* and *Mult*:

$$\frac{\begin{array}{c} i(pc) = \mathit{Eq}\ r_1\ r_2\ r_d \qquad rf[r_1] = v_1@\ell_1 \qquad rf[r_2] = v_2@\ell_2 \\ \text{if } v_1 == v_2 \text{ then } n = 1 \text{ else } n = 0 \qquad rf' = rf[r_d := n@(\ell_1 \vee \ell_2)] \end{array}}{\boxed{pc@\ell_{pc}}\boxed{rf}\boxed{cs}\boxed{m} \Rightarrow \boxed{pc{+}1@\ell_{pc}}\boxed{rf'}\boxed{cs}\boxed{m}} \text{ (EQ)}$$

GetOffset and *SetOffset* allow direct access to the offset of any pointer:

$$\frac{i(pc) = \mathit{GetOffset}\ r_p\ r_d \qquad rf[r_p] = (b,o)@\ell_p \qquad rf' = rf[r_d := o@\ell_p]}{\boxed{pc@\ell_{pc}}\boxed{rf}\boxed{cs}\boxed{m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc}}\boxed{rf'}\boxed{cs}\boxed{m}} \text{ (GETOFFSET)}$$

$$\frac{\begin{array}{c} i(pc) = \mathit{SetOffset}\ r_p\ r_o\ r_d \qquad rf[r_p] = (b,o')@\ell_p \qquad rf[r_o] = o@\ell_o \\ rf' = rf[r_d := (b,o)@(\ell_p \vee \ell_o)] \end{array}}{\boxed{pc@\ell_{pc}}\boxed{rf}\boxed{cs}\boxed{m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc}}\boxed{rf'}\boxed{cs}\boxed{m}} \text{ (SETOFFSET)}$$

GetBlockSize $r_p\ r_d$ returns the size of the block referenced by the pointer in r_p:

$$\frac{\begin{array}{c} i(pc) = \mathit{GetBlockSize}\ r_p\ r_d \qquad rf[r_p] = (b,o)@\ell_p \\ m[b] = vs@\ell_b \qquad rf' = rf[r_d := (\mathit{length}\ vs)@\ell_b] \end{array}}{\boxed{pc@\ell_{pc}}\boxed{rf}\boxed{cs}\boxed{m} \Rightarrow \boxed{(pc{+}1)@(\ell_{pc} \vee \ell_p)}\boxed{rf'}\boxed{cs}\boxed{m}} \text{ (GETBLOCKSIZE)}$$

The result has to be protected by the block label ℓ_b, which in turn has to be protected by the pointer label ℓ_p. The latter is can only achieved by raising the *pc* by ℓ_p. Finally, *GetBlockLabel* $r_p\ r_d$ returns the label of the block referenced by the pointer in r_p:

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$$\frac{i(pc) = GetBlockLabel\ r_p\ r_d \qquad rf[r_p] = (b, o)@\ell_p \qquad m[b] = vs@\ell_b \qquad rf' = rf[r_d := \ell_b@\ell_p]}{\boxed{pc@\ell_{pc}}\boxed{rf}\boxed{cs}\boxed{m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc}}\boxed{rf'}\boxed{cs}\boxed{m}} \text{ (GetBlockLabel)}$$

8.4 Per-level allocation, stamps, reachability, and noninterference

Dynamic allocation in high contexts can cause the values of the pointers to differ between the two executions considered by noninterference (Banerjee & Naumann, 2005). We ensure soundness by breaking up each pointer into a memory block identifier and an offset into the memory block. While offsets are fully observable to the program, block identifiers are opaque and can only be tested for equality. To further simplify the technical development, we allocate block identifiers "per level", i.e., we assume that we have a separate allocator for each allocation context label. This assumption ensures that, at the level of abstraction we consider here, allocations in high contexts cannot influence the values of pointers allocated in low contexts, and we can thus use syntactic equality to check indistinguishability of pointers. While this assumption on allocation might seem unrealistic, previous work has shown formally that because block identifiers are opaque, this machine can be realized by a lower-level machine with a single standard allocator (Azevedo de Amorim *et al.*, 2014).

To understand per-level allocation, one needs to understand the structure of the block identifiers we have already used in the previous subsection. Block identifiers are not opaque; they are pairs of a label ℓ_σ, which we will call a *stamp*, and an integer index i:

$$b ::= (\ell_\sigma, i).$$

As mentioned in Section 8.3, a memory m is a partial map between block identifiers and labeled lists of values. One can also see the memory as a three-dimensional array indexed first by stamps, then by indices, and finally by offsets. Stamps record the level of the allocation, i.e., the label of the context in which the allocation occurred, and ensure that allocation at one level cannot influence allocation at other levels. The side-condition

$$fresh\ m\ (\ell_{pc} \vee \ell_n \vee \ell') = b$$

in the Alloc rule from Section 8.3 implements this per-level allocation idea. The function *fresh* takes a memory m and a stamp $\ell_\sigma = \ell_{pc} \vee \ell_n \vee \ell'$, then uses ℓ_σ to index into the memory m, then uses a deterministic strategy to find the first unallocated index i in $m[\ell_\sigma]$, and finally returns the block identifier $b = (\ell_\sigma, i)$. Formally, we have

$$fresh\ m\ \ell_\sigma = (\ell_\sigma,\ find\_undefined\_index\ m[\ell_\sigma]).$$

It turns out that stamps are not only a convenient mechanism for implementing per-level allocation (thus simplifying the definition of indistinguishability for pointers to just syntactic equality), but are also a crucial ingredient in another complex invariant of our noninterference proof. The Alloc rule from Section 8.3 allows choosing an arbitrary block label, even in a high context. This is only sound because

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

at the end of the high context access to the newly allocated blocks is only possible through high pointers. The stamp in each block identifier captures precisely the label of the context in which the allocation of that block occurred. A key invariant used in our noninterference proof is that intuitively *an allocated block with identifier* (ℓ_σ, i) *can be reached from registers only via pointer paths that are protected by labels that are, when taken together, at least as secure as* ℓ_σ. In the following, we will formalize this reachability invariant and use it to define indistinguishability.

We start by defining the "root set" of our reachability invariant, the memory blocks that are directly accessible at a certain label ℓ. Given a machine state $\boxed{n@\ell_{pc} \mid rf \mid cs \mid m}$, the root set includes the blocks that can be directly accessed by pointers $(b,o)@\ell_p$ in the register file rf for which $(\ell_p \vee \ell_{pc}) \sqsubseteq \ell$. Because pointers are protected both by their explicit label ℓ_p and the *pc* label ℓ_{pc}, if the machine is in a high state (one for which $\ell_{pc} \not\sqsubseteq \ell$), then the current register file does not contribute at all to the root set. The saved register files on the call stack cs are added to the root set or not depending on whether the label of the return address in the same call frame is below ℓ or not. The label of the return address becomes the new *pc* on the corresponding *Return*, so even if the current *pc* is high the root set has to include all the low pointers in all register files saved in low-saved-*pc* call frames.

More formally, we define *root-set*, a function from a label and a machine state to a set of blocks, as follows:

$$\mathit{root\text{-}set}\ \ell\ \boxed{n@\ell_{pc} \mid rf \mid cs \mid m} = (\mathit{root\text{-}set'}\ \ell\ cs) \cup \begin{cases} \mathit{blocks}\ \ell\ rf & \text{if } \ell_{pc} \sqsubseteq \ell \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{root\text{-}set'}\ \ell\ ((n@\ell_{pc}, rf, r, \ell_{res}) : cs) = (\mathit{root\text{-}set'}\ \ell\ cs) \cup \begin{cases} \mathit{blocks}\ \ell\ rf & \text{if } \ell_{pc} \sqsubseteq \ell \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{root\text{-}set'}\ \ell\ [\,] = \emptyset$$

$$\mathit{blocks}\ \ell\ vs = \{b \mid (b,i)@\ell_v \in vs \wedge \ell_v \sqsubseteq \ell\}$$

Reachability with respect to a label and a state is a relation on block identifiers defined as the reflexive transitive closure of a direct $\mathit{link}\ \ell\ m$ relation on block identifiers:

$$\mathit{reachable}\ \ell\ \boxed{pc \mid rf \mid cs \mid m} = (\mathit{link}\ \ell\ m)^*$$

$$\mathit{link}\ \ell\ m = \{(b,b') \mid m[b] = vs@\ell_b \wedge \ell_b \sqsubseteq \ell \wedge b' \in \mathit{blocks}\ \ell\ vs\}$$

We formally state the reachability invariant as a well-formedness property of stamps:

8.1 Definition: We call a machine state S *well-stamped* if for all labels ℓ and for all block identifiers b and b', if $b \in \mathit{root\text{-}set}\ \ell\ S$ and $(b,b') \in \mathit{reachable}\ \ell\ S$, then $b' = (\sigma, i)$ for some $\sigma \sqsubseteq \ell$.

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

We have discovered and refined the form of the well-stamped property by testing. Subsequently, we have also proved in Coq that it is indeed an invariant of the execution of our register machine.[3]

8.2 Lemma: If S is well-stamped and $S \Rightarrow S'$ then S' is well-stamped.

The effort of proving this lemma was reduced by considering the correct definitions and statement from the start.

The indistinguishability relation for register machine states requires that both the considered states are well-stamped. Like reachability, indistinguishability is defined with respect to an observation level ℓ.

8.3 Definition: Machine states S_1 and S_2 are *indistinguishable* given observer level ℓ, written $S_1 \approx^{\ell}_{full\text{-}ws} S_2$, if S_1 and S_2 are both well-stamped and $S_1 \approx^{\ell}_{full} S_2$.

The $S_1 \approx^{\ell}_{full} S_2$ relation is defined similarly to the relation of the same name in Section 6.4 (Definition 6.4):

8.4 Definition: Machine states $S_1 = \boxed{pc_1 \mid rf_1 \mid cs_1 \mid m_1 \mid i_1}$ and $S_2 = \boxed{pc_2 \mid rf_2 \mid cs_2 \mid m_2 \mid i_2}$ are *indistinguishable at level ℓ with respect to whole machine states*, written $S_1 \approx^{\ell}_{full} S_2$, if $m_1 \approx^{\ell} m_2$, $i_1 \approx^{\ell} i_2$, and additionally

- if $\ell_{pc_1} \sqsubseteq \ell$ or $\ell_{pc_2} \sqsubseteq \ell$ then $pc_1 = pc_2$ and $rf_1 \approx^{\ell} rf_2$ and $cs_1 \approx^{\ell} cs_2$.
- otherwise *dropWhile* (*stack-frame-high* ℓ) $cs_1 \approx^{\ell}$ *dropWhile* (*stack-frame-high* ℓ) cs_2, where *stack-frame-high* ℓ $(n@\ell_{pc}, rf, r, \ell_{res}) = \ell_{pc} \not\sqsubseteq \ell$.

The differences with respect to Definition 6.4 are caused by moving from a 2-label lattice to a more general one. In case one of the *pc*s is high we still compare the stacks after cropping all high elements, just that "being high" is now defined as being protected by a label that does not flow to the observation label ℓ. Moreover, if both *pc*s are high then the two *pc* labels are not required to be equal, while for the 2-label lattice from Section 6.4 any two high labels have to be equal.

The definition above relies on several auxiliary relations, most interestingly on an indistinguishability relation for memories defined as follows:

$$m_1 \approx^{\ell} m_2 = \forall(\ell_\sigma, i).\ \ell_\sigma \sqsubseteq \ell \Rightarrow (m_1[(\ell_\sigma, i)] \Uparrow \wedge\ m_2[(\ell_\sigma, i)] \Uparrow) \vee\ m_1[(\ell_\sigma, i)] \approx^{\ell} m_2[(\ell_\sigma, i)].$$

We require each observable block identifier (ℓ_σ, i) either to be undefined in both memories or to point to respectively indistinguishable blocks. Indistinguishability for labeled things (used both for labeled blocks and the labeled values inside) is defined as follows:

$$y_1@\ell_1 \approx^{\ell} y_2@\ell_2 = (\ell_1 = \ell_2 \wedge (\ell_1 \sqsubseteq \ell \implies y_1 \approx^{\ell} y_2)).$$

Indistinguishability for lists (of values or stack frames) is defined pointwise:

$$(y_1 : ys_1) \approx^{\ell} (y_2 : ys_2) = y_1 \approx^{\ell} y_2 \wedge ys_1 \approx^{\ell} ys_2 \quad \text{and} \quad [] \approx^{\ell} [].$$

[3] Lemma `well_stamped_preservation` at `https://github.com/QuickChick/IFC/blob/master/NIProof.v`

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

Because of per-level allocation, indistinguishability for values (including for pointers) is defined simply as syntactic equality:

$$v_1 \approx^{\ell} v_2 = (v_1 = v_2).$$

Indistinguishability for (potentially cropped, see Definition 8.4) stacks is defined using list indistinguishability and the following indistinguishability relation on stack frames:

$$(pc_1, rf_1, r_1, \ell_{res_1}) \approx^{\ell} (pc_2, rf_2, r_2, \ell_{res_2}) = \\ (\ell_{pc_1} \sqsubseteq \ell \vee \ell_{pc_2} \sqsubseteq \ell) \implies (pc_1 = pc_2 \wedge rf_1 \approx^{\ell} rf_2 \wedge r_1 = r_2 \wedge \ell_{res_1} = \ell_{res_2}).$$

If one of the stored *pc*s is low, then the other has to be low as well, and all elements of the stack frame have to be pointwise related. If both stored *pc*s are high, we do not impose any additional constraints on the stack frame, in particular, the two high *pc*s can have different labels. This mirrors the handing of *pc*s and register files in Definition 8.4. While this definition seems natural in retrospect, it took us a while to reach it;Section A.5 presents the wrong alternatives with which we started.

While we discovered the rules and the well-stamped invariant by testing, we finally proved in Coq that this IFC mechanism has noninterference with respect to $\approx_{full\text{-}ws}$.[4]

8.5 Theorem: The register information-flow machine satisfies $\mathrm{SSNI}_{\approx_{full\text{-}ws}}$.

While the proof of this theorem discovered no errors in the rules or the well-stamped invariant, it did discover a serious flaw in the indistinguishability relation for stacks, which was previously hidden by an error in our stack generator (the wrong definition is described in Section A.5). This illustrates that keeping generators and checkers in sync is challenging and brings further motivation to recent work on domain-specific languages for generators (Claessen *et al.*, 2014; Fetscher *et al.*, 2015; Lampropoulos *et al.*, 2015).

8.5 Testing results

In order to evaluate how well our testing techniques scale, we apply the best strategies from Sections 4 and 6 to the register machine and devise an even stronger property that is even better at finding bugs. In this subsection, we explain and discuss in detail the experimental results summarized in Figure 19.

For these experiments, we introduced bugs by dropping taints and checks and by moving taints from the *pc* to the result. A missing taint bug is formed by dropping some label in the result of the correct IFC rule. For example, we can insert a bug in the MULT rule by only tainting the result with the label of one of its arguments (we taint $n_1 \times n_2$ with ℓ_1 instead or $\ell_1 \vee \ell_2$):

$$\frac{i(pc) = Mult\ r_1\ r_2\ r_d \qquad rf[r_1] = n_1@\ell_1 \qquad rf[r_2] = n_2@\ell_2 \qquad rf' = rf[r_d := (n_1 \times n_2)@\ell_1]}{\boxed{pc@\ell_{pc}}\boxed{rf}\boxed{cs}\boxed{m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc}}\boxed{rf'}\boxed{cs}\boxed{m}} \quad (\text{MULT}^*)$$

[4] https://github.com/QuickChick/IFC/blob/master/NIProof.v

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

Tested property	EENI	EENI	LLNI	LLNI	SSNI	SSNI	MSNI	MSNI
Starting states	*Init*	*Any*	*Any*	*Any*	*Tiny*	*Tiny*	*Any*	*Any*
Indistinguishability	$\approx_{ints\text{-}in\text{-}regs}$	$\approx_{full\text{-}ws}$	$\approx_{full\text{-}ws}$	$\approx_{full\text{-}ws}$	$\approx_{full\text{-}ws}$	$\approx_{full\text{-}ws}$	$\approx_{full\text{-}ws}$	$\approx_{full\text{-}ws}$
Generation strategy	*ByExec*	*ByExec*	*ByExec*	*ByExec*	*Tiny*	*Tiny*	*ByExec*	*ByExec*
Variant			*basic*	*optimized*	*basic*	*optimized*	*basic*	*optimized*
Mov	17.29	224.26	13.34	13.11	16.39	22.25	12.66	16.10
Load	5349.00	1423.70	70.57	57.30	116.54	121.63	67.73	71.49
Load	—	—	144.64	119.82	77.62	83.60	34.02	36.49
Load	—	80.00	111.28	95.12	127.10	131.50	104.07	110.92
Store	—	1896.42	206.73	15.94	102.48	50.05	43.71	19.23
Store	—	—	983.00	1077.07	47.56	22.04	22.47	12.17
Store	5365.00	1027.42	123.57	41.57	187.58	87.10	121.79	50.74
$+,*,\sqsubseteq,=$	55.15	2395.66	103.83	101.91	151.27	209.19	98.97	127.95
$+,*,\sqsubseteq,=$	55.79	2683.88	101.93	102.05	154.52	210.68	98.08	123.61
Noop	286.07	123.00	1.57	9.91	8.02	11.07	8.51	11.85
Jump	1003.05	118.99	32.86	44.18	73.02	99.31	31.13	50.61
Jump	161.16	2731.78	11.90	11.82	20.93	29.19	11.31	14.29
BranchNZ	2079.02	—	79.02	78.56	27.97	37.74	72.87	95.59
BranchNZ	755.22	118.71	29.51	40.31	72.66	101.32	27.78	46.84
Call	858.23	911.36	36.08	35.21	14.26	19.53	15.79	19.30
Call	2128.50	117.68	93.99	148.48	10.83	14.71	4.18	6.67
Call	249.53	4497.32	12.45	12.22	21.38	29.10	11.65	14.73
Return	—	452.99	1311.57	19.52	47.38	32.55	36.95	23.66
Return	—	1322.03	1346.00	119.56	178.14	123.99	215.00	152.14
Return	695.30	239.04	19.48	10.43	21.01	14.40	18.78	12.59
Return	3015.38	712.16	21.56	11.06	13.83	9.50	19.83	13.21
Alloc	344.85	20.32	36.93	28.42	45.87	47.95	35.23	34.25
Alloc	324.16	22.60	39.60	30.77	46.84	47.23	37.20	36.85
Write	—	—	1340.00	773.40	183.35	85.13	76.82	40.87
Write	—	4110.29	404.04	58.61	348.65	202.48	153.03	73.19
Write	—	—	1104.73	69.98	288.42	152.07	197.57	83.56
Write	—	757.69	69.92	25.22	28.96	13.24	14.31	7.55
Upgrade	—	—	790.67	144.17	399.17	338.26	352.94	184.00
Upgrade	—	—	1138.00	922.00	238.66	112.21	97.13	58.80
Upgrade	—	—	1843.50	84.70	267.86	132.53	223.79	101.73
Upgrade	—	—	860.49	90.22	407.16	295.01	242.51	109.57
Upgrade	—	2706.80	460.02	174.89	422.01	357.12	481.55	222.27
GetOffset	1236.10	513.52	17.49	17.89	27.55	38.86	16.46	21.25
SetOffset	3180.00	371.24	38.71	42.00	50.58	68.97	38.34	51.00
SetOffset	2188.00	744.37	30.16	29.25	49.54	69.20	28.91	35.26
GetBlockSize	—	975.58	50.82	50.84	47.62	67.06	49.03	57.85
GetBlockSize	—	1960.54	64.69	70.67	78.47	108.20	59.54	82.94
GetBlockLabel	—	812.58	24.47	25.97	28.09	38.55	23.33	31.16
Arithmetic mean	**—**	**—**	**346.55**	**126.42**	**117.09**	**95.65**	**84.34**	**59.53**
Geometric mean	**—**	**—**	**100.46**	**53.44**	**67.33**	**62.29**	**46.53**	**40.65**

Figure 19. Experiments for the register machine. MTTF given in milliseconds.

A missing check bug is formed by dropping some part of the requirements of the IFC rule. For example, to insert a bug in the STORE rule we turn the $\ell_{pc} \vee \ell_p \sqsubseteq \ell_b$ check into just $\ell_p \sqsubseteq \ell_b$:

$$\frac{\begin{array}{c} i(pc) = Store\ r_p\ r_s \qquad rf[r_p] = (b,o)@\ell_p \qquad rf[r_s] = v@\ell_v \\ m[b] = vs@\ell_b \quad \ell_p \sqsubseteq \ell_b \quad vs' = vs[o := v@\ell_v] \quad m' = m[b := vs'] \end{array}}{\boxed{pc@\ell_{pc}}\boxed{rf}\boxed{cs}\boxed{m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc}}\boxed{rf}\boxed{cs}\boxed{m'}} \quad (\text{STORE}^*)$$

A final and more subtle class of bugs is moving the taint from the *pc* to the result. For some rules, like the one for *Load*, it is imperative that the *pc* is tainted

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

instead of the result so that the labels involved are protected. The following incorrect rule, in which we taint the value with the block label ℓ_b instead of the pc, yields a counterexample:

$$\frac{\begin{array}{c} i(pc) = \mathit{Load}\ r_p\ r_d \qquad rf[r_p] = (b,o)@\ell_p \\ m[b] = vs@\ell_b \qquad vs[o] = v@\ell_v \qquad rf' = rf[r_d := v@(\ell_v \vee \ell_b)] \end{array}}{\boxed{pc@\ell_{pc}}\boxed{rf}\boxed{cs}\boxed{m} \Rightarrow \boxed{(pc+1)@(\ell_{pc} \vee \ell_p)}\boxed{rf'}\boxed{cs}\boxed{m}} \quad (\textsc{Load}^*)$$

The baseline for our comparison is generation by execution and $\mathrm{EENI}_{\mathit{Init},\mathit{Halted}\cap\mathit{Low},\approx_{\mathit{ints\text{-}in\text{-}regs}}}$, a basic instantiation of EENI (as defined in Section 2.3 and Appendix Appendix B.), stating that starting from empty initial states and executing the same program, if both machines reach a low halting state then their register files need to contain low integers at the same positions and these integers need to be pairwise equal. Formally, we define indistinguishability as follows:

8.6 Definition: $S_1 = \boxed{pc_1}\boxed{rf_1}\boxed{cs_1}\boxed{m_1}$ and $S_2 = \boxed{pc_2}\boxed{rf_2}\boxed{cs_2}\boxed{m_2}$ are indistinguishable with respect to integers stored in registers, written $S_1 \approx_{\mathit{ints\text{-}in\text{-}regs}} S_2$, if $rf_1 \approx_{\mathit{ints}} rf_2$, which is the pointwise extension of the following indistinguishability relation on values:

$$v_1@\ell_1 \approx^{\ell}_{\mathit{ints}} v_2@\ell_2 = (\ell_1 = \ell_2 \wedge (\ell_1 \sqsubseteq \ell \implies (v_1 = n \Leftrightarrow v_2 = n))).$$

We choose this property as the baseline because it is simple; in particular, it does not compare pointers or memories or stacks, which as we saw in the previous subsection is very involved. The results for this property appear in the first column of Figure 19 and as expected are not satisfactory: most of the bugs are not found at all even after 5 minutes of testing. For the rest of the experiments, we use the indistinguishability relation $\approx_{\mathit{full\text{-}ws}}$ described in the previous subsection. Moreover, we start execution from arbitrary states, which also significantly improves testing.

The next two columns show the result of using the $\mathrm{EENI}_{\mathit{Any},\mathit{Halted}\cap\mathit{Low},\approx_{\mathit{full\text{-}ws}}}$ and $\mathrm{LLNI}_{\mathit{Any},\approx_{\mathit{full\text{-}ws}}}$ properties (LLNI is defined generically in Section 6.3 and Appendix Appendix B.). These properties do not actually use the full invariants shown in Section 8.4, just the parts of it that pertain to low states, since both EENI and LLNI will only compare such states. The generation strategy is again generation by execution—simpler strategies result in very poor performance. The results show that the extended machine is too complex for EENI to discover all injected bugs, while LLNI does find all of them.

Two simple observations allow us to improve LLNI even further. The first one is that our implementation of generation by execution is "naive" about generating each next instruction. To be precise, we continuously update the uninitialized instruction memory with new random instructions by repeatedly indexing into the list. This is clearly the source of some overhead, which can be alleviated by using a random-access data structure like a map or—as we chose to use—a *zipper*; a zipper provides even faster average-case performance since most of the time the program counter is only incremented by one. This improvement yields a small performance boost.

The second and more important observation is that some instructions have very restrictive IFC checks (*Store*, *Return*, *Write*, and *Upgrade*), which often lead to the

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

machine failing as soon as they are encountered. This causes these instructions to be underrepresented in the machine states produced by generation by execution, which only chooses an instruction if this instruction can execute for at least a step. Adjusting the frequency of instruction generation experimentally, so that each instruction ends up being equally frequent among the ones that can successfully take a step, leads to the optimized LLNI column of Figure 19. This strategy successfully discovers all bugs relatively quickly. There is, however, a trade-off: the bugs that were easier to find before become slightly harder to find.

We also consider an instance of the SSNI property (Section 6.4), $\text{SSNI}_{\approx_{full\text{-}ws}}$, which we expect to take advantage of all our invariants. Similarly to LLNI, IFC-check-heavy instructions cause a lot of failures; in this case, the failures lead to many discarded tests, since only one instruction is run. In Figure 19, we show the performance of $\text{SSNI}_{\approx_{full\text{-}ws}}$ with uniform and weighted instruction generation so that instructions empirically appear uniform in the non-discarded tests. The required weights turn out to be very similar to the ones required for LLNI. Moreover, the same trade-off appears here: we can find the hard-to-find bugs faster by sacrificing a bit of speed for the easy-to-find ones.

As was the case for the basic machine, when optimizing the generation for SSNI, we must be extremely cautious to avoid ruling out useful parts of the state space. Since SSNI operates by executing a machine state for a *single* step to check the invariant, being able to generate the entire state space of pairs of indistinguishable machines becomes very important. For example, a reasonable assumption might seem to be that the stacks are "monotonic", as described in the end of the previous section. However, if we use the incorrect indistinguishability relation in Equation (A 1) and generate only "monotonic" stacks for the starting states, SSNI does not uncover the bug in Figure A 7, whereas LLNI does.

Comparing LLNI and SSNI with respect to their efficiency in testing, we can spot an interesting tradeoff. On the one hand, a significant limitation of LLNI is that bugs that appear when the *pc* is high are not detected immediately, but only after the *pc* goes back low, if ever. One example is the STORE* buggy rule above, where we do not check whether the *pc* label flows to the label of the memory cell. On such bugs LLNI has orders of magnitude worse results. On the other hand, SSNI is significantly less robust with respect to starting state generation. If we do not generate every valid starting state, then SSNI will not test executions starting in the missing states, since it only executes one instruction. LLNI avoids this problem as long as all valid states are eventually reachable from the generated starting states.

These observations lead us to formulate a new property: *multi-step noninterference (MSNI)*, that combines the advantages of both LLNI and SSNI. The formal definition of MSNI is given in Appendix Appendix B.. Informally, we start from an arbitrary pair of indistinguishable machine states and we check the SSNI *unwinding conditions* along a whole execution trace. Using generation by execution with uniform and adjusted instruction frequencies for this property yields the last two columns of Figure 19. MSNI performs on most bugs on par with the better of SSNI or LLNI by uncovering IFC violations as soon as they appear; at the same time, unlike SSNI, MSNI is robust against faulty generation.

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

9 Related work

Generating random inputs for testing is a large research area, but the particular sub-area of testing language implementations by generating random *programs* is less well studied. Redex (Klein, 2009; Klein & Findler, 2009; Klein *et al.*, 2012; Fetscher *et al.*, 2015) (*né* PLT Redex) is a domain-specific language for defining operational semantics within Racket (*né* PLT Scheme), which includes a property-based random testing framework inspired by QuickCheck. This framework uses a formalized language definition to automatically generate simple test-cases. To generate better test cases, however, Klein *et al.* find that the generation strategy needs to be tuned for the particular language; this agrees with our observation that fine-tuned strategies are required to obtain the best results. They argue that the effort required to find bugs using Redex is less than the effort required for a formal proof of correctness, and that random testing is sometimes viable in cases where full proof seems infeasible.

Klein *et al.* (2013) use PLT Redex's QuickCheck-inspired random testing framework to assess the safety of the bytecode verification algorithm for the Racket virtual machine. They observe that naively generated programs only rarely pass bytecode verification (88% discard rate), and that many programs fail verification because of a few common violations that can be easily remedied in a post-generation pass that for instance replaces out-of-bounds indices with random in-bounds ones. These simple changes to the generator are enough for reducing the discard rate (to 42%) and for finding more than two dozen bugs in the virtual machine model, as well as a few in the Racket machine implementation, but three known bugs were missed by this naive generator. The authors conjecture that a more sophisticated test generation technique could probably find these bugs.

CSmith (Yang *et al.*, 2011) is a C compiler testing tool that generates random C programs, avoiding ones whose behavior is undefined by the C99 standard. When generating programs, CSmith does not attempt to model the current state of the machine; instead, it chooses program fragments that are correct with respect to some static safety analysis (including type-, pointer-, array-, and initializer-safety, etc.). We found that modeling the actual state of our (much simpler) machine to check that generated programs were hopefully well-formed, as in our generation by execution strategy, made our test-case generation far more effective at finding noninterference bugs. In order to get smaller counterexamples, Regehr *et al.* present C-Reduce (Regehr *et al.*, 2012), a tool for reducing test-case C programs such as those produced by CSmith. They note that conventional shrinking methods usually introduce test cases with undefined behavior; thus, they put a great deal of effort and domain specific knowledge into shrinking well-defined programs only to programs that remain well-defined. To do this, they use a variety of search techniques to find better reduction steps and to couple smaller ones together. Our use of QuickCheck's *double shrinking* is similar to their simultaneous reductions, although we observed no need in our setting for more sophisticated searching methods than the greedy one that is guaranteed to produce a local minimum. Regehr *et al.*'s work on reduction is partly based on Zeller and Hildebrandt's formalization of the delta debugging

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

algorithm *ddmin* (Zeller & Hildebrandt, 2002), a non-domain-specific algorithm for simplifying and isolating failure-inducing program inputs with an extension of binary search. In our work, as in Regehr *et al.*'s, domain-specific knowledge is crucial for successful shrinking. In recent work, Koopman *et al.* (2014) propose a technique for model-based shrinking.

Another relevant example of testing programs by generating random input is Randoop (Pacheco & Ernst, 2007), which generates random sequences of calls to Java APIs. Noting that many generated sequences crash after only a few calls, before any interesting bugs are discovered, Randoop performs *feedback directed* random testing, in which previously found sequences that did not crash are randomly extended. This enables Randoop to generate tests that run much longer before crashing, which are much more effective at revealing bugs. Our generation by execution strategy is similar in spirit, and likewise results in a substantial improvement in bug detection rates.

A state-machine modeling library for (an Erlang version of) QuickCheck has been developed by Quviq (Hughes, 2007). It generates sequences of API calls to a stateful system satisfying preconditions formulated in terms of a model of the system state, associating a (model) state transition function with each API call. API call generators also use the model state to avoid generating calls whose preconditions cannot be satisfied. Our generation-by-execution strategy works in a similar way for straightline code.

A powerful and widely used approach to testing is symbolic execution—in particular, *concolic testing* and related dynamic symbolic execution techniques (Majumdar & Sen 2007; Cadar *et al.*, 2011). The idea is to mix symbolic and concrete execution in order to achieve higher code coverage. The choice of which concrete executions to generate is guided by a constraint solver and path conditions obtained from the symbolic executions. Originating with DART (Godefroid *et al.*, 2005) and PathCrawler (Williams *et al.*, 2004), a variety of tools and methods have appeared; some of the state-of-the-art tools include CUTE (Sen *et al.*, 2005), CREST (Burnim & Sen, 2008), and KLEE (Cadar *et al.*, 2008) (which evolved from EXE (Cadar *et al.*, 2006)) . We wondered whether dynamic symbolic execution could be used instead of random testing for finding noninterference bugs. As a first step, we implemented a simulator for a version of our abstract machine in C and tested it with KLEE. Using KLEE out of the box and without any expert knowledge in the area, we attempted to invalidate various assertions of noninterference. Unfortunately, we were only able to find a counterexample for PUSH*, the simplest possible bug, in addition to a few implementation errors (e.g., out-of-bound pointers for invalid machine configurations). The main problem seems to be that the state space we need to explore is too large(Cadar & Sen, 2013), so we don't cover enough of it to reach the particular IFC-violating configurations. More recently, Torlak & Bodík (2014) have used our information-flow stack machine and its bugs with respect to EENI as a case study for their symbolic virtual machine, and report better results.

Balliu *et al.* (2012) created ENCOVER, an extension of Java PathFinder, to verify information-flow properties of Java programs by means of concolic testing. In their work, concolic testing is used to extract an abstract model of a program so

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

that security properties can be verified by an SMT solver. While ENCoVer tests the security of individual programs, we use testing to check the soundness of an entire enforcement mechanism. Similarly, Milushev *et al.* (2012) have used KLEE for testing the noninterference of individual programs, as opposed to our focus on testing dynamic IFC mechanisms that are meant to provide noninterference for all programs.

In recent work, (Lampropoulos *et al.*, 2015) introduce a domain-specific language for random generators that puts together random instantiation (Antoy, 2000; Claessen *et al.*, 2014; Fetscher *et al.*, 2015) and constraint solving (Mohr & Henderson, 1986). As a case study, they tests noninterference for the register machine from Section 8 using our generators for indistinguishable machine states (including generation by execution) and the SSNI and LLNI properties.

In interactive theorem provers, automatically generating counterexamples for false conjectures can prevent wasting time and effort on proof attempts doomed to fail (Groce *et al.*, 2007). Dybjer *et al.* (2003) propose a QuickCheck-like tool for the Agda/Alfa proof assistant. Berghofer & Nipkow (2004) proposed a QuickCheck-like tool for Isabelle/HOL. This was recently extended by Bulwahn (2012a) to also support exhaustive and narrowing-based symbolic testing(Lindblad, 2007; Christiansen & Fischer, 2008; Runciman *et al.*, 2008). Moreover, Bulwahn's tool uses Horn clause data flow analysis to automatically devise generators that only produce data that satisfies the precondition of the tested conjecture (Bulwahn 2012b). Eastlund (2009) implemented DoubleCheck, an adaption of QuickCheck for ACL2. Chamarthi *et al.* (2011) later proposed a more advanced counterexample finding tool for ACL2s, which uses the full power of the theorem prover and libraries to simplify conjectures so that they are easier to falsify. While all these tools are general and only require the statement of the conjecture to be in a special form (e.g., executable specification), so they could in principle be applied to test noninterference, our experience with QuickCheck suggests that for the best results one has to incorporate domain knowledge about the machine and the property being tested. We hope to compare our work against these tools in the future and provide experimental evidence for this intuition. Recently, Paraskevopoulou *et al.* (2015) introduced a port of Haskell QuickCheck to Coq together with a foundational verification framework for testing code and use our testing noninterference techniques as their main case study, proving our generator for "Tiny" indistinguishable states used to test SSNI for the register machine (Section 8.5) sound and complete with respect to indistinguishability.

On the dynamic IFC side, Birgisson *et al.* (2012) have a good overview of related work. Our correct rule for *Store* for the stack machine is called the *no-sensitive-upgrades* policy in the literature and was first proposed by Zdancewic (2002) and later adapted to the dynamic IFC setting by Austin & Flanagan (2009). To improve precision, Austin & Flanagan (2010) later introduced a different *permissive-upgrade* policy, where public locations can be written in a high context as long as branching on these locations is later prohibited, and they discuss adding *privatization operations* that would even permit this kind of branching safely. Hedin & Sabelfeld (2012) improve the precision of the no-sensitive-upgrades policy by explicit *upgrade annotations*, which raise the level of a location before branching on secrets. They

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

apply their technique to a core calculus of JavaScript that includes objects, higher order functions, exceptions, and dynamic code evaluation. Birgisson *et al.* (2012) show that random testing with QuickCheck can be used to infer upgrade instructions in this setting. The main idea is that whenever a random test causes the program to be stopped by the IFC monitor because it attempts a sensitive upgrade, the program can be rewritten by introducing an upgrade annotation that prevents the upgrade from being deemed sensitive on the next run of the program. In recent work, Bichhawat *et al.* (2014b) generalize the permissive-upgrade check to arbitrary IFC lattices. They present involved counterexamples, apparently discovered manually while doing proofs. We believe that our testing techniques are well-suited at automatically discovering such counterexamples.

Terauchi & Aiken (2005) and later Barthe *et al.* (2011b) propose a technique for statically verifying the noninterference of individual programs using the idea of self-composition. This reduces the problem of verifying secure information flow for a program P to a safety property for a program $\hat{P}$ derived from P, by composing P with a renaming of itself. Self-composition enables the use of standard (i.e., not relational (Benton, 2004; Barthe *et al.* 2011a)) program logics and model checking for showing noninterference. The problem we address in this paper is different: we test the soundness of dynamic IFC mechanisms by randomly generating (a large number of) pairs of related programs. One could imagine extending our technique in the future to testing the soundness of static IFC mechanisms such as type systems (Sabelfeld & Myers, 2003), relational program logics (Benton, 2004; Barthe *et al.* 2011a), and self-composition based tools (Barthe *et al.* 2011b).

In recent work, Ochoa *et al.* (2015) discuss a preliminary model-checking based technique for discovering unwanted information flows in specifications expressed as extended finite state machines. They also discuss about testing systems for unwanted flows using unwinding-based coverage criteria and mutation testing. In a recent position paper, Kinder (2015) discusses testing of hyperproperties (Clarkson & Schneider, 2010).

10 Conclusions and outlook

We have shown how random testing can be used to discover counterexamples to noninterference in a simple information-flow machine and how to shrink counterexamples discovered in this way to simpler, more comprehensible ones. The techniques we present bring many orders of magnitude improvement in the rate at which bugs are found, and for the hardest-to-find bugs (to EENI) the minimal counterexamples are 10–15 instructions long—well beyond the scope of naive exhaustive testing. Even if we ultimately care about full security proofs (Azevedo de Amorim *et al.*, 2014), using random testing should greatly speed the initial design process and allow us to concentrate more of our energy on proving things that are correct or nearly correct.

We are hopeful that we can scale the methodology introduced in this paper to test noninterference and other properties (Azevedo de Amorim *et al.*, 2015) for abstract machines built on top of real-life instruction set architectures. The results in Section 8 are particularly encouraging in this respect. For a real-life architecture, even if

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

were to find bugs 100× slower than for the machine in Section 8, that would still only be a matter of seconds.

We expect that our techniques are flexible enough to be applied to checking other relational properties of programs (i.e., properties of pairs of related runs(Benton, 2004; Clarkson & Schneider, 2010; Barthe *et al.* 2011a))—in particular, the many variants and generalizations of noninterference, for instance, taking into account declassification (Sabelfeld & Sands, 2005). Beyond noninterference properties, preliminary experiments with checking correspondence between concrete and abstract versions of our current stack machine suggest that many of our techniques can also be adapted for this purpose. For example, the generate-by-execution strategy and many of the shrinking tricks apply just as well to single programs as to pairs of related programs. This gives us hope that they may be useful for checking yet further properties of abstract machines.

Acknowledgments

We thank the participants in the discussion at the IFIP WG 2.8 meeting in Storulvån that originated this work: Ulf Norell, Rishiyur S. Nikhil, Michał Pałka, Nick Smallbone, and Meng Wang. We are grateful to Johannes Borgström, Cristian Cadar, Delphine Demange, Matthias Felleisen, Robby Findler, Alex Groce, Casey Klein, Ben Karel, Scott Moore, Michał Pałka, John Regehr, Howard Reubenstein, Alejandro Russo, Deian Stefan, Greg Sullivan, and Andrew Tolmach for providing feedback on a draft, and to the members of the CRASH/SAFE team and to Manolis Papadakis for fruitful discussions. Finally, we thank the anonymous reviewers for their suggestions and Andreas Haeberlen for kindly providing us computing time on his cluster. 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, and NSF award 1421243, *Random Testing for Language Design*. 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. The work is also partially funded under the Swedish Foundation for Strategic Research grant RAWFP.

References

Antoy, S. (2000) A needed narrowing strategy. *J. ACM* **47**(4), 776–822.

Austin, T. H. & Flanagan, C. (2009) Efficient purely-dynamic information flow analysis. In Proceedings of 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS'09. ACM, pp. 113–124.

Austin, T. H. & Flanagan, C. (2010) Permissive dynamic information flow analysis. In Proceedings of 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS'10. ACM, pp. 3:1–3:12.

Azevedo de Amorim, A., Collins, N., DeHon, A., Demange, D., Hriţcu, C., Pichardie, D., Pierce, B. C., Pollack, R. & Tolmach, A. (January 2014) A verified information-flow architecture. In Proceedings of 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'14. ACM, pp. 165–178.

Azevedo de Amorim, A., Dénès, M., Giannarakis, N., Hriţcu, C., Pierce, B. C., Spector-Zabusky, A. & Tolmach, A. (2015) Micro-policies: Formally verified, tag-based security monitors. In Proceedings of 36th IEEE Symposium on Security and Privacy, SP'15. IEEE, pp. 813–830.

Balliu, M., Dam, M. & Guernic, G. L. (2012) Encover: Symbolic exploration for information flow security. In Proceedings of 25th IEEE Computer Security Foundations Symposium, CSF'12. IEEE, pp. 30–44.

Banerjee, A. & Naumann, D. A. (2005) Stack-based access control and secure information flow. *J. Funct. Program.* **15**(2), 131–177.

Barthe, G., Crespo, J. M. & Kunz, C. (2011a) Relational verification using product programs. In *Proceedings of 17th International Symposium on Formal Methods*, FM'11, Lecture Notes in Computer Science, vol. 6664. Springer, pp. 200–214.

Barthe, G., D'Argenio, P. R. & Rezk, T. (2011b) Secure information flow by self-composition. *Math. Struct. Comput. Sci.* **21**(6), 1207–1252.

Benton, N. (2004) Simple relational correctness proofs for static analyses and program transformations. In Proceedings of 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'04. ACM, pp. 14–25.

Berghofer, S. & Nipkow, T. (2004) Random testing in Isabelle/HOL. In Proceedings of 2nd International Conference on Software Engineering and Formal Methods, SEFM'04. IEEE CS, pp. 230–239.

Bichhawat, A., Rajani, V., Garg, D. & Hammer, C. (2014a) Information flow control in WebKit's JavaScript bytecode. In Proceedings of 3rd International Conference on Principles of Security and Trust, POST'14, Lecture Notes in Computer Science, vol. 8414. Springer, pp. 159–178.

Bichhawat, A., Rajani, V., Garg, D. & Hammer, C. (2014b) Generalizing permissive-upgrade in dynamic information flow analysis. In Proceedings of 9th Workshop on Programming Languages and Analysis for Security, PLAS'14. ACM, pp. 15–24.

Birgisson, A., Hedin, D. & Sabelfeld, A. (2012) Boosting the permissiveness of dynamic information-flow tracking by testing. In Proceedings of 17th European Symposium on Research in Computer Security, ESORICS'12, Lecture Notes in Computer Science, vol. 7459. Springer, pp. 55–72.

Buiras, P., Stefan, D., & Russo, A. (2014) On dynamic flow-sensitive floating-label systems. In Proceedings of 27th IEEE Computer Security Foundations Symposium, CSF'14. IEEE, pp. 65–79.

Bulwahn, L. (2012a) The new Quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. In Proceedings of 2nd International Conference on Certified Programs and Proofs, CPP'12, Lecture Notes in Computer Science, vol. 7679. Springer, pp. 92–108.

Bulwahn, L. (2012b) Smart testing of functional programs in Isabelle. In Proceedings of 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'12, Lecture Notes in Computer Science, vol. 7180, Springer, pp. 153–167.

Burnim, J. & Sen, K. (2008) Heuristics for scalable dynamic test generation. In Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE'08, IEEE Computer Society, pp. 443–446.

Cadar, C., Dunbar, D. & Engler, D. (2008) KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08. USENIX Association, pp. 209–224.

Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L. & Engler, D. R. (2006) EXE: Automatically generating inputs of death. In Proceedings of 13th ACM Conference on Computer and Communications Security, CCS'06. ACM, pp. 322–335.

Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C. S., Sen, K., Tillmann, N. & Visser, W. (2011) Symbolic execution for software testing in practice: preliminary assessment. In Proceedings of 33rd International Conference on Software Engineering, ICSE'11. ACM, pp. 1066–1071.

Cadar, C. & Sen, K. (2013) Symbolic execution for software testing: Three decades later. *Commun. ACM* **56**(2), 82–90.

Chamarthi, H. R., Dillinger, P. C., Kaufmann, M. & Manolios, P. (2011) Integrating testing and interactive theorem proving. In Proceedings of 10th International Workshop on the ACL2 Theorem Prover and its Applications, Electronic Proceedings in Theoretical Computer Science, vol. 70, pp. 4–19. `http://www.eptcs.org/`

Christiansen, J. & Fischer, S. (2008) EasyCheck – test data for free. In Proceedings of *9th International Symposium on Functional and Logic Programming*, FLOPS'08, Lecture Notes in Computer Science, vol. 4989. Springer, pp. 322–336.

Claessen, K., Duregård, J. & Pałka, M. H. (2014) Generating constrained random data with uniform distribution. In Proceedings of 12th International Symposium on Functional and Logic Programming, Lecture Notes in Computer Science, vol. 8475. Springer, pp. 18–34.

Claessen, K. & Hughes, J. (2000) QuickCheck: A lightweight tool for random testing of Haskell programs. In Proceedings of 5th ACM SIGPLAN International Conference on Functional Programming, ICFP'00. ACM, pp. 268–279.

Clarkson, M. R. & Schneider, F. B. (2010) Hyperproperties. *J. Comput. Secur.* **18**(6), 1157–1210.

Dybjer, P., Haiyan, Q. & Takeyama, M. (2003) Combining testing and proving in dependent type theory. In Proceedings of 16th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'03, Lecture Notes in Computer Science, vol. 2758. Springer, pp. 188–203.

Eastlund, C. (2009) DoubleCheck your theorems. In Proceedings of 8th International Workshop on the ACL2 Theorem Prover and its Applications, ACL2'09. ACM, pp. 42–46.

Fenton, J. S. (1974) Memoryless subsystems. *Comput. J.* **17**(2), 143–147.

Fetscher, B., Claessen, K., Palka, M. H., Hughes, J. & Findler, R. B. (2015) Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In Proceedings of 24th European Symposium on Programming, ESOP'15, Lecture Notes in Computer Science, vol. 9032. Springer, pp. 383–405.

Giffin, D. B., Levy, A., Stefan, D., Terei, D., Mazières, D., Mitchell, J. & Russo, A. (2012) Hails: Protecting data privacy in untrusted web applications. In Proceedings of 10th Symposium on Operating Systems Design and Implementation, OSDI'12. USENIX Association, pp. 47–60.

Godefroid, P., Klarlund, N., & Sen, K. (2005) DART: Directed automated random testing. In Proceedings of 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'05. ACM, pp. 213–223.

Goguen, J. A. & Meseguer, J. (1984) Unwinding and inference control. In Proceedings of IEEE 1984 Symposium on Security and Privacy. IEEE CS, pp. 75–87.

Groce, A., Holzmann, G. J. & Joshi, R. (2007) Randomized differential testing as a prelude to formal verification. In Proceedings of The 29th International Conference on Software Engineering, ICSE'07. IEEE CS, pp. 621–631.

Guernic, G. L. (2007) Automaton-based confidentiality monitoring of concurrent programs. In Proceedings of 20th Computer Security Foundations Symposium, CSF'07. IEEE CS, pp. 218–232.

Guernic, G. L., Banerjee, A., Jensen, T. P. & Schmidt, D. A. (2006) Automata-based confidentiality monitoring. In Proceedings of 11th Asian Computing Science Conference, ASIAN 2006. Springer, pp. 75–89.

Hedin, D. & Sabelfeld, A. (2012) Information-flow security for a core of JavaScript. In Proceedings of 25th IEEE Computer Security Foundations Symposium (CSF), CSF'12. IEEE CS, pp. 3–18.

Hriţcu, C., Greenberg, M., Karel, B., Pierce, B. C. & Morrisett, G. (2013a) All your IFCException are belong to us. In Proceedings of 34th IEEE Symposium on Security and Privacy, SP'13, IEEE CS, pp. 3–17.

Hriţcu, C., Hughes, J., Pierce, B. C., Spector-Zabusky, A., Vytiniotis, D., Azevedo de Amorim, A. & Lampropoulos, L. (2013b) Testing noninterference, quickly. In Proceedings of 18th ACM SIGPLAN International Conference on Functional Programming, ICFP'13. ACM, pp. 455–468.

Hughes, J. (2007) QuickCheck testing for fun and profit. In Proceedings of 9th International Symposium on Practical Aspects of Declarative Languages, PADL'07, Lecture Notes in Computer Science, vol. 4354. Springer, pp. 1–32.

Kinder, J. (2015) Hypertesting: The case for automated testing of hyperproperties. In Proceedings of 3rd Workshop on Hot Issues in Security Principles and Trust, HotSpot.

Klein, C. (August 2009) Experience with randomized testing in programming language metatheory. Master's Thesis, Northwestern. Available at: `http://plt.eecs.northwestern.edu/klein-masters.pdf`. Accessed Feb 26, 2016.

Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J. A., Rafkind, J., Tobin-Hochstadt, S. & Findler, R. B. (2012) Run your research: On the effectiveness of lightweight mechanization. In Proceedings of 39th ACM SIGPLAN-SIGACT Principles of Programming Languages, POPL'12, ACM, pp. 285–296.

Klein, C. & Findler, R. B. (2009) Randomized testing in PLT Redex. In Proceedings of Workshop on Scheme and Functional Programming, SFP, ACM, pp. 26–36.

Klein, C., Flatt, M. & Findler, R. (2013) The Racket virtual machine and randomized testing. In *Higher-Order and Symbolic Computation*, Springer, pp. 1–45. `http://dx.doi.org/10.1007/s10990-013-9091-1`

Koopman, P. W. M., Achten, P. & Plasmeijer, R. (2014) Model-based shrinking for state-based testing. In Proceedings of 14th International Symposium on Trends in Functional Programming, TFP 2013, Lecture Notes in Computer Science, vol. 8322, Springer, pp. 107–124.

Lampropoulos, L., Pierce, B. C., Hriţcu, C., Hughes, J., Paraskevopoulou, Z. & Xia, L. (July 2015) Making our own Luck: A language for random generators. Draft. `https://www.cis.upenn.edu/~llamp/pdf/Luck.pdf`

Leroy, X., Appel, A. W., Blazy, S. & Stewart, G. (June 2012) The CompCert memory model, version 2. Research report RR-7987, INRIA.

Leroy, X. & Blazy, S. (2008) Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Autom. Reason.* **41**(1), 1–31.

Lindblad, F. (2007) Property directed generation of first-order test data. In Proceedings of 8th Symposium on Trends in Functional Programming, TFP'07, Trends in Functional Programming, vol. 8. Intellect, pp. 105–123.

Majumdar, R. & Sen, K. (2007) Hybrid concolic testing. In Proceedings of 29th International Conference on Software Engineering, ICSE'07. IEEE CS, pp. 416–426.

Milushev, D., Beck, W. & Clarke, D. (2012) Noninterference via symbolic execution. In Proceedings of Joint 14th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems and 32nd IFIP WG 6.1 International Conference, FMOODS 2012 and FORTE 2012, Lecture Notes in Computer Science, vol. 7273. Springer, pp. 152–168.

Mohr, R. & Henderson, T. C. (1986) Arc and path consistency revisited. *Artif. Intell.* **28**(2), 225–233.

Ochoa, M., Cuéllar, J., Pretschner, A. & Hallgren, P. (2015) Idea: Unwinding based model-checking and testing for non-interference on EFSMs. In Proceedings of 7th International Symposium on Engineering Secure Software and Systems, ESSoS'15, Lecture Notes in Computer Science, vol. 8978. Springer, pp. 34–42.

Pacheco, C. & Ernst, M. D. (2007) Randoop: Feedback-directed random testing for Java. In Proceedings of 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems And Applications, OOPSLA'07. ACM, pp. 815–816.

Paraskevopoulou, Z., Hriţcu, C., Dénès, M., Lampropoulos, L. & Pierce, B. C. (2015) Foundational property-based testing. In *Proceedings of 6th International Conference on Interactive Theorem Proving*, Urban, C. & Zhang, X. (eds), ITP'15, Lecture Notes in Computer Science, vol. 9236. Springer, pp. 325–343.

Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C. & Yang, X. (2012) Test-case reduction for C compiler bugs. In Proceedings of 33rd ACM SIGPLAN conference on Programming Language Design and Implementation, PLDI'12. ACM, pp. 335–346.

Runciman, C., Naylor, M. & Lindblad, F. (2008) SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In Proceedings of 1st ACM SIGPLAN Symposium on Haskell. ACM, pp. 37–48.

Russo, A. & Sabelfeld, A. (2010) Dynamic versus static flow-sensitive security analysis. In Proceedings of 23rd Computer Security Foundations Symposium, CSF'10. IEEE CS, pp. 186–199.

Sabelfeld, A. & Myers, A. (January 2003) Language-based information-flow security. *IEEE J. Sel. Areas Commu.* **21**(1), 5–19.

Sabelfeld, A. & Russo, A. (2010) From dynamic to static and back: Riding the roller coaster of information-flow control research. In Proceedings of 7th International Andrei Ershov Memorial Conference, PSI 2009, Lecture Notes in Computer Science, vol. 5947. Springer, pp. 352–365.

Sabelfeld, A. & Sands, D. (2005) Dimensions and principles of declassification. In Proceedings of 18th IEEE Workshop on Computer Security Foundations, CSF'05. IEEE CS, pp. 255–269.

Sen, K., Marinov, D. & Agha, G. (2005) CUTE: A concolic unit testing engine for C. In Proceedings of 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13. ACM, pp. 263–272.

Stefan, D., Russo, A., Mitchell, J. C. & Mazières, D. (2011) Flexible dynamic information flow control in Haskell. In Proceedings of 4th Symposium on Haskell. ACM, pp. 95–106.

Stefan, D., Russo, A., Mitchell, J. C. & Mazières, D. (July 2012) Flexible dynamic information flow control in the presence of exceptions. *ArXiv e-print 1207.1457*.

Terauchi, T. & Aiken, A. (2005) Secure information flow as a safety problem. In Proceedings of 12th International Symposium on Static Analysis, SAS 2005, Lecture Notes in Computer Science, vol. 3672. Springer, pp. 352–367.

Torlak, E. & Bodík, R. (2014) A lightweight symbolic virtual machine for solver-aided host languages. In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014. ACM, 2014, pp. 530–541.

Williams, N., Marre, B. & Mouy, P. (2004) On-the-fly generation of K-path tests for C functions. In Proceedings of 19th IEEE International Conference on Automated Software Engineering, ASE. IEEE CS, pp. 290–293.

Yang, X., Chen, Y., Eide, E. & Regehr, J. (2011) Finding and understanding bugs in C compilers. In Proceedings of 32nd SIGPLAN Conference on Programming Language Design and Implementation, PLDI'11, ACM, pp. 283–294.

Zdancewic, S. A. (2002) *Programming Languages for Information Security*. PhD Thesis, Cornell University.

Zeller, A. & Hildebrandt, R. (2002) Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.* **28**(2), 183–200.

Zheng, L. & Myers, A. C. (2007) Dynamic security labels and static information flow control. *Int. J. Inform. Secur.* **6**(2–3), 67–84.

Appendix A. Varying the indistinguishability relation

A.1 Labels being high or low needs to be observable

As seen in Section 2.4, our definition of indistinguishability of values (Definition 2.1) allows the observer to distinguish between final memory states that differ only in their *labels*. One might imagine changing the definition of indistinguishability so that labels are not observable. There are at least two ways one can imagine doing this; however, both are wrong. First, one could try defining indistinguishability of values so that $x@L \approx y@H$ for any x and y. QuickCheck easily finds a counterexample to this (Figure A 1). Second, one could try refining this so that only $x@L \approx x@H$, i.e., a high value is equivalent with a low one only when the payloads are equal. QuickCheck also disproves this alternative (Figure A 2), and the counterexample produced by QuickCheck illustrates how, even with the correct rules, a difference in the labels of two values can be turned into a difference in the values of two values. This counterexample is reminiscent of a well-known "flow-sensitivity attack" (Figure 1 in Russo & Sabelfeld (2010); attributed to Fenton (1974)). This counterexample relies on *Call* and *Return* as introduced in Section 5.

A.2 Weakening EENI when adding calls and returns

The counterexample in Figure A 3 shows that once we have a way to restore the *pc* label, we can no longer expect all pairs of halting states to be indistinguishable in EENI. In particular, as the counterexample shows, one machine can halt in a high state, while the other can return to low, and only then halt. Since our indistinguishability relation only equates states with the same *pc* label, these two halting states are distinguishable. The solution we use in Section 5.2 is to weaken the EENI instance, by considering only ending states that are both halting and low (i.e., we change to $\mathrm{EENI}_{Init,Halted \cap Low,\approx_{mem}}$).

A.3 Indistinguishability for stack elements when adding calls and returns

In Section 5.2, we defined the indistinguishability relation on stack elements so that return addresses are only equivalent to other return addresses and (as for values) $R(x_1@\ell_1) \approx R(x_2@\ell_2)$ if either $\ell_1 = \ell_2 = H$ or $x_1 = x_2$ and $\ell_1 = \ell_2 = L$. If instead we considered high return addresses and high values to be indistinguishable,

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$$i = \left[\begin{array}{l} \mathit{Push}\ 1@L, \mathit{Push}\ 0@H, \mathit{Push}\ 1@L, \mathit{Store}, \mathit{Push}\ \frac{1@H}{0@L}, \\ \mathit{Store}, \mathit{Halt} \end{array}\right]$$

pc	m	s	$i(pc)$
$0@L$	$[0@L, 0@L]$	$[]$	$\mathit{Push}\ 1@L$
$1@L$	$[0@L, 0@L]$	$[1@L]$	$\mathit{Push}\ 0@H$
$2@L$	$[0@L, 0@L]$	$[0@H, 1@L]$	$\mathit{Push}\ 1@L$
$3@L$	$[0@L, 0@L]$	$[1@L, 0@H, 1@L]$	Store
$4@L$	$[0@L, 0@H]$	$[1@L]$	$\mathit{Push}\ \frac{1@H}{0@L}$
$5@L$	$[0@L, 0@H]$	$\left[\frac{1@H}{0@L}, 1@L\right]$	Store
$6@L$	$\left[\frac{0}{1}@L, \frac{1}{0}@H\right]$	$[]$	Halt

Figure A 1. A counterexample showing that it is wrong to make high values be equivalent to all other values.

$$i = \left[\begin{array}{l} \mathit{Push}\ 1@L, \mathit{Push}\ 0@\frac{H}{L}, \mathit{Push}\ 0@L, \mathit{Store}, \mathit{Push}\ \frac{7}{9}@H, \\ \mathit{Call}\ 1\ 0, \mathit{Halt}, \mathit{Push}\ 0@L, \mathit{Store}, \mathit{Return} \end{array}\right]$$

pc	m	s	$i(pc)$
$0@L$	$[0@L]$	$[]$	$\mathit{Push}\ 1@L$
$1@L$	$[0@L]$	$[1@L]$	$\mathit{Push}\ 0@\frac{H}{L}$
$2@L$	$[0@L]$	$\left[0@\frac{H}{L}, 1@L\right]$	$\mathit{Push}\ 0@L$
$3@L$	$[0@L]$	$\left[0@L, 0@\frac{H}{L}, 1@L\right]$	Store
$4@L$	$\left[0@\frac{H}{L}\right]$	$[1@L]$	$\mathit{Push}\ \frac{7}{9}@H$
$5@L$	$\left[0@\frac{H}{L}\right]$	$\left[\frac{7}{9}@H, 1@L\right]$	$\mathit{Call}\ 1\ 0$
Machine 1 continues...			
$7@H$	$[0@H]$	$[1@L, R(6,0)@L]$	$\mathit{Push}\ 0@L$
$8@H$	$[0@H]$	$[0@L, 1@L, R(6,0)@L]$	Store
$9@H$	$[1@H]$	$[R(6,0)@L]$	Return
$6@L$	$[1@H]$	$[]$	Halt
Machine 2 continues...			
$9@H$	$[0@L]$	$[1@L, R(6,0)@L]$	Return
$6@L$	$[0@L]$	$[]$	Halt

Figure A 2. A counterexample showing that it is also wrong to make high values equivalent to low values with the same payload.

QuickCheck would find a counterexample. This counterexample requires quasi-initial states (and $\approx_{low}$) and is listed in Figure A 4. The first machine performs only one *Return* that throws away two elements from the stack and then halts. The second machine returns twice: the first time to the same *Return*, unwinding the stack and raising the *pc*; and the second time to the *Halt* instruction, labeling the return value high in the process. The final states are distinguishable because the elements on the stack have different labels. As we saw earlier, such a counterexample can be extended to one in which values also differ.

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$i = \left[Push\ \frac{2}{3}@H, Call\ 0\ 0, Halt, Return \right]$

pc	*m*	*s*	*i*(*pc*)
$0@L$	[]	[]	$Push\ \frac{2}{3}@H$
$1@L$	[]	$\left[\frac{2}{3}@H\right]$	$Call\ 0\ 0$
Machine 1 continues...			
$2@H$	[]	$[R(2,0)@L]$	*Halt*
Machine 2 continues...			
$3@H$	[]	$[R(2,0)@L]$	*Return*
$2@L$	[]	[]	*Halt*

Figure A 3. A counterexample justifying the change to $\text{EENI}_{Init,Halted \cap Low,\approx_{mem}}$ in Section 5.2.

$i = \left[Return, Halt \right]$

pc	*m*	*s*	*i*(*pc*)
$0@L$	[]	$\left[0@L, \frac{0@H}{R(0,0)@H}, 0@L, R(1,1)@L\right]$	*Return*
Machine 1 continues...			
$1@L$	[]	$[0@L]$	*Halt*
Machine 2 continues...			
$0@H$	[]	$[0@L, R(1,1)@L]$	*Return*
$1@L$	[]	$[0@H]$	*Halt*

Figure A 4. A counterexample that motivates the indistinguishability of stack elements for the machine with calls and returns.

A.4 Counterexamples justifying indistinguishability for SSNI

The indistinguishability relation high states used for SSNI needs to be strong enough to ensure that when both machines return to low states, those low states are also indistinguishable. Since $\approx_{low}$ is too weak, QuickCheck can find counterexamples to condition 3 in Definition 6.3 (see Figure A 5).

On the other hand, treating high states exactly like low states in the indistinguishability relation is too strong, since that would prevent the stacks to change between successive high states. In this case, QuickCheck finds counterexamples to condition 2 in Definition 6.3 (see Figure A 6). This motivates comparing stacks for high state only below the first low return, while allowing the tops of the stacks to vary arbitrarily, as done in the definition of $\approx_{full}$ (Definition 6.4). These two counterexamples guide our search for the correct indistinguishability relation—i.e., one that correctly captures the invariant that the machine can only alter stack frames below the current one by using the *Return* instruction.

A.5 Wrong alternatives for indistinguishability of register machine states

The definition of indistinguishability for the register machine (Definition 8.4) might seem natural in retrospect, but it took us a while to reach it. Here, we presents two

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$i = [\mathit{Return}]$

pc	m	s	$i(pc)$
$0@H$	[]	$\left[R(\frac{0}{1},0)@L\right]$	*Return*
Machine 1 continues...			
$0@L$	[]	[]	*Return*
Machine 2 continues...			
$1@L$	[]	[]	–

Figure A 5. A counterexample showing that $\approx_{low}$ is too weak for SSNI. Since the pc is initially high, $\approx_{low}$ does not require the initial stacks to be related in any way, which means the two machines can jump to two different addresses while still both lowering the pc. The two resulting states are, however, distinguishable, since they have different pcs.

$i = [\mathit{Pop}]$

pc	m	s	$i(pc)$
$0@H$	[]	$[0@L]$	*Pop*
$1@H$	[]	[]	–

Figure A 6. A counterexample that shows that treating high states exactly like low states in the indistinguishability relation over machine states is too strong and breaks condition 2 in Definition 6.3. When a machine steps from a high state to another high state the contents of the stack can change.

wrong alternatives with which we started. The handing of call stacks differs from Definition 8.4.

In the first wrong alternative, we required a very strong matching between stack frames:

$$(n_1@\ell_{pc_1}, rf_1, r_1, \ell_{res_1}) \approx^{\ell} (n_2@\ell_{pc_2}, rf_2, r_2, \ell_{res_2}) = \\ n_1 = n_2 \wedge \ell_{pc_1} = \ell_{pc_2} \wedge rf_1 \approx^{\ell} rf_2 \wedge r_1 = r_2 \wedge \ell_{res_1} = \ell_{res_2}. \quad \text{(A 1)}$$

However, the above indistinguishability relation assumes that the stacks are "monotonic", in the sense that the program counters stored in the stack are only decreasing with respect to the label flows-to relation. While this was true for the stack machines with the two-label lattice, this is the case for the register machine with the more complex diamond lattice, as can be seen in Figure A 7. In this counterexample even after cropping the top high part of the stack, a high stack element frame remains on the stack, which varies in the return label, causing our indistinguishability relation to fail when it shouldn't. Since the return pcs are high, this difference in labels is not observable, and therefore does not break noninterference.

In the second wrong alternative, we tried to deal with this observation by filtering out *all* high elements of the stack, leaving only low stack elements to compare pairwise. This required us to change Definition 8.4 as follows:

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$i = [Call\ 0\ 1\ 0, Call\ 0\ 2\ 0]$

pc	*m*	*r*	*s*	*i*(*pc*)
$0@M_1$	[]	$\left(1@L, \frac{M_1}{M_2}@M_2, L@L\right)$	[]	*Call* 0 1 0
$1@M_1$	[]	$\left(1@L, \frac{M_1}{M_2}@M_2, L@L\right)$	$\left[\{R\ 1@H, \frac{M_1}{M_2}, \ldots\}\right]$	*Call* 0 2 0
$1@M_1$	[]	$\left(1@L, \frac{M_1}{M_2}@M_2, L@L\right)$	$\left[\{R\ 2@M_1, L, \ldots\}, \{R\ 1@H, \frac{M_1}{M_2}, \ldots\}\right]$	*Call* 0 2 0

Figure A 7. An example of an execution trace that produces a non-monotonic stack.

$i = [Return]$

pc	*m*	*r*	*s*	*i*(*pc*)
Machine 1's trace...				
$0@L$	[]	()	$[\{R\ 0@H, \ldots\}, \{R\ 0@L, \ldots\}]$	*Return*
$0@H$	[]	()	$[\{R\ 0@L, \ldots\}]$	*Return*
Machine 2's trace...				
$0@L$	[]	()	$[\{R\ 0@L, \ldots\}]$	*Return*
$0@L$	[]	()	[]	*Return*

Figure A 8. Counterexample for the filtering stack-indistinguishability relation.

A.2 Definition: Machine states $S_1 = \boxed{pc_1 \mid rf_1 \mid cs_1 \mid m_1 \mid i_1}$ and $S_2 = \boxed{pc_2 \mid rf_2 \mid cs_2 \mid m_2 \mid i_2}$ are *indistinguishable at level* ℓ *with respect to whole machine states*, written $S_1 \approx^{\ell}_{full} S_2$, if $m_1 \approx^{\ell} m_2$, $i_1 \approx^{\ell} i_2$, $cs_1 \approx^{\ell} cs_2$, and additionally

- if $\ell_{pc_1} \sqsubseteq \ell$ or $\ell_{pc_2} \sqsubseteq \ell$ then $pc_1 = pc_2$ and $rf_1 \approx^{\ell} rf_2$.

We then defined indistinguishability of stacks as follows:

$$cs_1 \approx^{\ell} cs_2 = filter\ (stack\text{-}frame\text{-}below\ \ell)\ cs_1 \approx^{\ell} filter\ (stack\text{-}frame\text{-}below\ \ell)\ cs_2$$
$$stack\text{-}frame\text{-}below\ \ell\ (n@\ell_{pc}, rf, r, \ell_{res}) = \ell_{pc} \sqsubseteq \ell.$$

Unfortunately, this is now too weak and leads to an execution trace that breaks noninterference (Figure A 8). After taking a step in the first machine, we get an distinguishable pair of machines where one has an observable *pc* while the other one does not.

Appendix B. Theorems for strengthening IFC properties

We have proved in Coq[5] that, under some reasonable assumptions, MSNI implies SSNI, SSNI implies LLNI, and LLNI implies EENI. All these are generic properties of information-flow abstract machines; a machine M is composed of:

- an arbitrary type of machine states,
- a partial step function on states written $\Rightarrow$ (reduction is deterministic), and
- a set of observation levels o (e.g., labels).

[5] `https://github.com/QuickChick/IFC/blob/master/NotionsOfNI.v`

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

As in Section 2, we write $\Rightarrow^*$ for the reflexive, transitive closure of $\Rightarrow$. When $S \Rightarrow^* S'$ and S' is stuck ($\nexists S''.\ S' \Rightarrow S''$), we write $S \Downarrow S'$. We write $S \Rightarrow^*_t$ when an execution ($\Rightarrow^*$) from S produces trace t (a list of states).

B.1 Definition: A machine M has *EENI* with respect to

- a predicate on states *Start* (initial states), and
- a predicate on states *End* (successful ending states),
- an observation-level-indexed indistinguishability relation on states $\approx$,

written $\text{EENI}_{Start,End,\approx}\ M$, when

- for all states $S_1, S_2 \in Init$, if $S_1 \approx_o S_2$, $S_1 \Downarrow S_1'$, $S_2 \Downarrow S_2'$, and $S_1', S_2' \in End$ then $S_1' \approx_o S_2'$.

B.2 Definition: A machine M has *low-lockstep noninterference (LLNI)* with respect to

- a predicate on states *Start*,
- an indistinguishability relation $\approx$, and
- an observation-level-indexed predicate on states *Low* (e.g., in which only data labeled below a certain label has influenced control flow),

written $\text{LLNI}_{Start,Low,\approx}\ M$, when for all $S_1, S_2 \in Start$ with $S_1 \approx_o S_2$, $S_1 \Rightarrow^*_{t_1}$, and $S_2 \Rightarrow^*_{t_2}$, we have $t_1 \approx^*_o t_2$, where indistinguishability on traces $\approx^*_o$ is defined inductively by the following rules:

$$\frac{S_1, S_2 \in Low_o \quad S_1 \approx_o S_2 \quad t_1 \approx^*_o t_2}{(S_1 : t_1) \approx^*_o (S_2 : t_2)} \qquad \text{(Low Lockstep)}$$

$$\frac{S_1 \notin Low_o \quad t_1 \approx^*_o t_2}{(S_1 : t_1) \approx^*_o t_2} \qquad \text{(High Filter)}$$

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

$$\frac{}{t \approx_o^* []} \quad \text{(End)}$$

$$\frac{t_1 \approx_o^* t_2}{t_2 \approx_o^* t_1} \quad \text{(Symmetry)}$$

B.3 Theorem: $\text{LLNI}_{Start,Low,\approx} M$ implies $\text{EENI}_{Start,End,\approx} M$ provided that

- $\approx$ is symmetric,
- $End \subset Low$,
- $S \in End$ implies that S is stuck ($\nexists S'.\ S \Rightarrow S'$),
- $S_1 \approx S_2$ implies $S_1 \in End \Leftrightarrow S_2 \in End$.

B.4 Definition: A machine M has *SSNI* with respect to

- an indistinguishability relation $\approx$, and
- an observation-level-indexed predicate on states *Low*,

written $\text{SSNI}_{Low,\approx} M$, when the following conditions are satisfied:

1. For all o and $S_1, S_2 \in Low_o$, if $S_1 \approx_o S_2$, $S_1 \Rightarrow S_1'$, and $S_2 \Rightarrow S_2'$, then $S_1' \approx_o S_2'$;
2. For all o and $S \notin Low_o$ if $S \Rightarrow S'$ and $S' \notin Low_o$, then $S \approx_o S'$;
3. For all o and $S_1, S_2 \notin Low_o$, if $S_1 \approx_o S_2$, $S_1 \Rightarrow S_1'$, $S_2 \Rightarrow S_2'$, and $S_1', S_2' \in Low_o$, then $S_1' \approx_o S_2'$.

B.5 Theorem: $\text{SSNI}_{Low,\approx} M$ implies $\text{LLNI}_{Start,Low,\approx} M$ under the following assumptions:

- $\approx$ is a partial equivalence relation (symmetric and transitive),
- $S_1 \approx S_2$ implies $S_1 \in Low \Leftrightarrow S_2 \in Low$.

B.6 Definition: A machine M has *MSNI* with respect to

- an indistinguishability relation $\approx$, and
- an observation-level-indexed predicate on states *Low*,

written $\text{MSNI}_{Low,\approx} M$, when S_1 and S_2 with $S_1 \approx_o S_2$, $S_1 \Rightarrow_{t_1}^*$, $S_2 \Rightarrow_{t_2}^*$ we have $t_1 \approx_o^* t_2$, where indistinguishability on traces $\approx_o^*$ is defined inductively by the following rules:

$$\frac{S_1, S_2 \in Low_o \quad S_1' \approx_o S_2' \quad (S_1' : t_1) \approx_o^* (S_2' : t_2)}{(S_1 : S_1' : t_1) \approx_o^* (S_2 : S_2' : t_2)} \quad \text{(Low Steps)}$$

$$\frac{S_1, S_2 \notin Low_o \quad S_1', S_2' \in Low_o \quad S_1' \approx_o S_2' \quad (S_1' : t_1) \approx_o^* (S_2' : t_2)}{(S_1 : S_1' : t_1) \approx_o^* (S_2 : S_2' : t_2)} \quad \text{(High to Low Steps)}$$

$$\frac{S_1, S_1' \notin Low_o \quad S_1 \approx_o S_1' \quad (S_1' : t_1) \approx_o^* (S_2 : t_2)}{(S_1 : S_1' : t_1) \approx_o^* (S_2 : t_2)} \quad \text{(High to High Step)}$$

$$\frac{(S_1 \in Low_o \vee S_1' \in Low_o) \quad (S_1' : t_1) \approx_o^* (S_2 : [])}{(S_1 : S_1' : t_1) \approx_o^* (S_2 : [])} \quad \text{(Low Step End)}$$

$$\frac{}{(S_1 : []) \approx_o^* (S_2 : [])} \quad \text{(Both End)}$$

$$\frac{t_1 \approx_o^* t_2}{t_2 \approx_o^* t_1} \quad \text{(Symmetry)}$$

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press

B.7 Theorem: $\mathrm{MSNI}_{Low,\approx} M$ implies $\mathrm{SSNI}_{Low,\approx} M$ under the following assumptions:

- $\approx$ is reflexive and symmetric,
- $S_1 \approx S_2$ implies $S_1 \in Low \Leftrightarrow S_2 \in Low$.

https://doi.org/10.1017/S0956796816000058 Published online by Cambridge University Press