Testing Noninterference, Quickly

Cătălin Hriţcu[1] John Hughes[2] Benjamin C. Pierce[1] Antal Spector-Zabusky[1] Dimitrios Vytiniotis[3]
Arthur Azevedo de Amorim[1] Leonidas Lampropoulos[1]

[1]University of Pennsylvania [2]Chalmers University [3]Microsoft Research

Abstract

Information-flow control mechanisms are difficult to design and labor intensive to prove correct. To reduce the time wasted on doomed proofs for 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 a simple information-flow abstract machine. We find that both sophisticated strategies for generating well-distributed random programs and readily falsifiable formulations of noninterference properties are critically important. 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 all these bugs.

Keywords random testing, security, design, dynamic information-flow control, noninterference, abstract machine, QuickCheck

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, while dynamic mechanisms must appropriately propagate taints and raise security exceptions when necessary. 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 take as a case study the task of extending a simple abstract stack-and-pointer machine to track dynamic information flow and enforce termination-insensitive noninterference [39]. Although our machine is simple, this exercise is both nontrivial and novel. While simpler notions of dynamic *taint tracking* are well studied for both high- and low-level languages, it has only recently been shown [1, 40] that dynamic checks are capable of soundly enforcing strong security properties. Moreover, sound dynamic IFC has been studied only in the context of lambda-calculi [1, 2, 27, 42] and While programs [40]; the unstructured control flow of a low-level machine poses additional challenges. (Testing of static IFC mechanisms is left as future work.)

We show how QuickCheck [17], 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 [25] of our noninterference property can be much more effective than directly testing the original property.

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 enforcement mechanisms; 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 a low-level information-flow machine; (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; and (4) an effective methodology for shrinking large counterexamples to smaller, more readable ones. Our information-flow abstract machine, while simple, is also novel, and may be a useful artifact for further research.

2. Basic IFC

We begin by introducing the core of our abstract machine. In §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 information-flow control.

Some notation: we write $[\,]$ for the empty list and $x : s$ for the list whose first element is x and whose tail is s; we also write $[x_0, x_1, \ldots, x_n]$ for the list $x_0 : x_1 : \cdots : x_n : [\,]$. If l is a list and $0 \leq j < |l|$, then $l(j)$ selects the j^{th} element of l and $l\{j \mapsto x\}$ produces the list that is like l except that the j^{th} element is replaced by x.

2.1 Bare machine

Our basic machine (without information-flow labels) has seven instructions:

$$\mathit{Instr} ::= \mathit{Push}\ x \mid \mathit{Pop} \mid \mathit{Load} \mid \mathit{Store} \mid \mathit{Add} \mid \mathit{Noop} \mid \mathit{Halt}$$

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

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 $\boxed{pc \mid s \mid m \mid i}$. Often, i will be fixed in some context and we will 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:

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

$$\frac{i(pc) = \mathit{Push}\ x}{\boxed{pc \mid s \mid m} \Rightarrow \boxed{pc{+}1 \mid x : s \mid m}} \qquad (\textsc{Bare-Push})$$

$$\frac{i(pc) = \mathit{Pop}}{\boxed{pc \mid x : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m}} \qquad (\textsc{Bare-Pop})$$

$$\frac{i(pc) = \mathit{Load}}{\boxed{pc \mid x : s \mid m} \Rightarrow \boxed{pc{+}1 \mid m(x) : s \mid m}} \qquad (\textsc{Bare-Load})$$

$$\frac{i(pc) = \mathit{Store}}{\boxed{pc \mid x : y : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m\{x \mapsto y\}}} \qquad (\textsc{Bare-Store})$$

$$\frac{i(pc) = \mathit{Add}}{\boxed{pc \mid x : y : s \mid m} \Rightarrow \boxed{pc{+}1 \mid (x{+}y) : s \mid m}} \qquad (\textsc{Bare-Add})$$

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 $i(pc) = \mathit{Halt}$ 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 halted state, we write $S \Downarrow S'$.

2.2 Machine with labeled data

In a (fine-grained) dynamic IFC system [1, 2, 27, 40, 42] 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 *values* of the form $x@L$, where x is an integer and L is a *label*:

$$L ::= \bot \mid \top$$

We read $\bot$ as "low" (public) and $\top$ as "high" (secret). We order labels by $\bot \sqsubseteq \top$and write $L_1 \vee L_2$ for the *join* (least upper bound) of L_1 and L_2. When v is a value, we write $\mathcal{L}_v$ for v's label part and $v@L$ for the value obtained by joining L to $\mathcal{L}_v$—i.e., $(x@L_1)@L_2 = x@(L_1 \vee L_2)$.

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

$$\mathit{Instr} ::= \mathit{Push}\ v \mid \mathit{Pop} \mid \mathit{Load} \mid \mathit{Store} \mid \mathit{Add} \mid \mathit{Noop} \mid \mathit{Halt}$$

Machine states have the same shape as the basic machine, with the stack and memory now being lists of values. 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 $0@\bot$.

2.3 Noninterference (EENI)

We define what it means for this basic IFC machine to be "secure" using a standard notion of noninterference [1, 27, 39]; we call it *end-to-end noninterference* (or *EENI*) to distinguish it from the stronger notions we will introduce in §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 values labeled $\top$ in the initial state; because of the form of our initial states, such values can appear only in instruction memories. By secret outputs we mean values labeled $\top$ 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 H_1 and H_2, the final states H_1 and H_2 are also indistinguishable. Intuitively, two states are indistinguishable if they differ only in values labeled $\top$. To make this formal, we define an equivalence relation on states compositionally from equivalence relations over their components.

2.3.1 Definition:

- Two values $x_1@L_1$ and $x_2@L_2$ are said to be *indistinguishable*, written $x_1@L_1 \approx x_2@L_2$, if either $L_1 = L_2 = \top$ or else $x_1 = x_2$ and $L_1 = L_2 = \bot$.
- Two instructions i_1 and i_2 are indistinguishable if they are the same instruction, or if $i_1 = \mathit{Push}\ v_1$, and $i_2 = \mathit{Push}\ v_2$, and $v_1 \approx v_2$.
- Two lists (memories, stacks, or instruction memories) l_1 and l_2 are indistinguishable if they have the same length and $l_1(x) \approx l_2(x)$ for all x such that $0 \leq x < |l_1|$.

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 and pc 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.)

2.3.2 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 $i_1 \approx i_2$.

2.3.3 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 $\mathrm{EENI}_{\mathit{Start},\mathit{End},\approx}$, if for any $S_1, S_2 \in \mathit{Start}$ and $H_1, H_2 \in \mathit{End}$ such that $S_1 \approx S_2$ and such that $S_1 \Downarrow H_1$ and $S_2 \Downarrow H_2$, we have $H_1 \approx H_2$.

We take $\mathrm{EENI}_{\mathit{Init},\mathit{Halted},\approx_{mem}}$ 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 rules

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 QuickCheck-generated counterexamples to identify and help repair mistakes until no more can be found.

$$\frac{i(pc) = \mathit{Noop}}{\boxed{pc \mid s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m}} \qquad (\textsc{Noop})$$

$$\frac{i(pc) = \mathit{Push}\ v}{\boxed{pc \mid s \mid m} \Rightarrow \boxed{pc{+}1 \mid v : s \mid m}} \qquad (\textsc{Push})$$

$$\frac{i(pc) = \mathit{Pop}}{\boxed{pc \mid v : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m}} \qquad (\textsc{Pop})$$

$$\frac{i(pc) = \mathit{Load}}{\boxed{pc \mid x@L_x : s \mid m} \Rightarrow \boxed{pc{+}1 \mid m(x) : s \mid m}} \qquad (\textsc{Load*})$$

$$\frac{i(pc) = \mathit{Store}}{\boxed{pc \mid x@L_x : v : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m\{x \mapsto v\}}} \quad (\text{STORE*AB})$$

$$\frac{i(pc) = \mathit{Add}}{\boxed{pc \mid x@L_x : y@L_y : s \mid m} \Rightarrow \boxed{pc{+}1 \mid (x{+}y)@\bot : s \mid m}} \quad (\text{ADD*})$$

The NOOP rule is the same as in the unlabeled machine. In the PUSH and POP rules, we simply change the relevant integers to be labeled values; 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 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 $0@\top$, while the second pushes $1@\top$, and these two values 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 $[0@\bot, 0@\bot]$; the stack starts out empty; and the next instruction to be executed ($i(pc)$) is *Push* $1@\bot$. On the next step, this value has been pushed on the stack and the next instruction is either *Push* $0@\top$ or *Push* $1@\top$; one or the other of these values is pushed on the stack. On the next, we *Store* the second stack element ($1@\bot$) into the location pointed to by the first (either $0@\top$ or $1@\top$), so that now the memory contains $1@\bot$ in either location 0 or location 1 (the other location remains unchanged, and contains $0@\bot$). 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 $\top$), but in the final states the memories contain different values at the same location, both of which are labeled $\bot$.

Thinking about this counterexample, it soon becomes apparent what went wrong with the *Store* instruction: since pointers labeled $\top$ are allowed to vary between the two runs, it is not safe to store a low value 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 $L_x = \bot$ 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:

$$\frac{i(pc) = \mathit{Store}}{\boxed{pc \mid x@L_x : y@L_y : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m\{x \mapsto y@(L_x \vee L_y)\}}} \quad (\text{STORE*B})$$

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 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 value with the label of the pointer, the fact that the *Store* changes different locations is visible in the fact that a label changes from $\bot$ to $\top$ on a different memory location in each run. To avoid this issue, we adopt the "no sensitive upgrades" rule [1, 46], 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 terminates the machine). Adding this side condition brings us to a correct version of the STORE rule.

$$\frac{i(pc) = \mathit{Store} \qquad L_x \sqsubseteq \mathcal{L}_{m(x)}}{\boxed{pc \mid x@L_x : y@L_y : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m\{x \mapsto y@(L_x \vee L_y)\}}} \quad (\text{STORE})$$

The next counterexample found by QuickCheck (Figure 3) points out a straightforward problem in the ADD* rule: adding

$i = [\ \mathit{Push}\ 1@\bot, \mathit{Push}\ \frac{0}{1}@\top, \mathit{Store}, \mathit{Halt}\ ]$

pc	*m*	*s*	*i*(*pc*)
0	$[0@\bot, 0@\bot]$	$[]$	*Push* $1@\bot$
1	$[0@\bot, 0@\bot]$	$[1@\bot]$	*Push* $\frac{0}{1}@\top$
2	$[0@\bot, 0@\bot]$	$[\frac{0}{1}@\top, 1@\bot]$	*Store*
3	$[\frac{1}{0}@\bot, \frac{0}{1}@\bot]$	$[]$	*Halt*

Figure 1. Counterexample to STORE*AB

$i = [\ \mathit{Push}\ 0@\bot, \mathit{Push}\ \frac{0}{1}@\top, \mathit{Store}, \mathit{Halt}\ ]$

pc	*m*	*s*	*i*(*pc*)
0	$[0@\bot, 0@\bot]$	$[]$	*Push* $0@\bot$
1	$[0@\bot, 0@\bot]$	$[0@\bot]$	*Push* $\frac{0}{1}@\top$
2	$[0@\bot, 0@\bot]$	$[\frac{0}{1}@\top, 0@\bot]$	*Store*
3	$[0@\frac{\top}{\bot}, 0@\frac{\bot}{\top}]$	$[]$	*Halt*

Figure 2. Counterexample to STORE*B

$i = [\ \mathit{Push}\ \frac{0}{1}@\top, \mathit{Push}\ 0@\bot, \mathit{Add}, \mathit{Push}\ 0@\bot, \mathit{Store}, \mathit{Halt}\ ]$

pc	*m*	*s*	*i*(*pc*)
0	$[0@\bot]$	$[]$	*Push* $\frac{0}{1}@\top$
1	$[0@\bot]$	$[\frac{0}{1}@\top]$	*Push* $0@\bot$
2	$[0@\bot]$	$[0@\bot, \frac{0}{1}@\top]$	*Add*
3	$[0@\bot]$	$[\frac{0}{1}@\bot]$	*Push* $0@\bot$
4	$[0@\bot]$	$[0@\bot, \frac{0}{1}@\bot]$	*Store*
5	$[\frac{0}{1}@\bot]$	$[]$	*Halt*

Figure 3. Counterexample to ADD*

[1] See the first clause of Definition 2.3.1. One might imagine that this could be fixed easily by changing the definition so that labels are not observable—i.e., $x@\bot \approx x@\top$ for any x. Sadly, this is known not to work [38]. (QuickCheck can also find a counterexample; see §B.1).

$$i = \left[\begin{array}{l} \mathit{Push}\ 0@\bot, \mathit{Push}\ 1@\bot, \mathit{Push}\ 0@\bot, \mathit{Store}, \mathit{Push}\ \tfrac{0}{1}@\top, \\ \mathit{Load}, \mathit{Store}, \mathit{Halt} \end{array}\right]$$

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

Figure 4. Counterexample to LOAD*

$0@\bot$ to $0@\top$ yields $0@\bot$. The problem is that the taints on the arguments to *Add* are not propagated to its result. The *Store* 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:

$$\frac{i(pc) = \mathit{Add}}{\boxed{pc \mid x@L_x : y@L_y : s \mid m} \Rightarrow \boxed{pc{+}1 \mid (x{+}y)@(L_x{\vee}L_y) : s \mid m}} \quad (\text{ADD})$$

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 $0@\bot$ 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 value 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*,

$$\frac{i(pc) = \mathit{Load}}{\boxed{pc \mid x@L_x : s \mid m} \Rightarrow \boxed{pc{+}1 \mid m(x)@L_x : s \mid m}} \quad (\text{LOAD})$$

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 they are not the only incorrect way to write these rules. Here are a couple of others:

$$\frac{i(pc) = \mathit{Push}\ x@L}{\boxed{pc \mid s \mid m} \Rightarrow \boxed{pc{+}1 \mid x@\bot : s \mid m}} \quad (\text{PUSH*})$$

$$\frac{i(pc) = \mathit{Store}}{\boxed{pc \mid x@L_x : y@L_y : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m\{x \mapsto y@\bot\}}} \quad (\text{STORE*C})$$

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:

$$\frac{i(pc) = \mathit{Store} \qquad L_x \sqsubseteq \mathcal{L}_{m(x)}}{\boxed{pc \mid x@L_x : v : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m\{x \mapsto v\}}} \quad (\text{STORE*A})$$

3. QuickCheck

We test noninterference using QuickCheck [17], 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 values, instructions, and machine states (each of which depends on the previous generator: machine states contain instructions, some of which contain values). 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 §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 slow 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 "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 which 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 §7.

4. Test 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: $\forall S_1, S_2 \in \mathit{Start}.\ S_1 \approx S_2 \implies \ldots$. This is a very strong precondition, which is unlikely to be satisfied for independently generated states. Instead, we generate *indistinguishable pairs* of states together. The first state is generated ran-

Generation strategy *Smart integers?*	NAIVE NO	WEIGHTED NO	SEQUENCE NO	SEQUENCE YES	BYEXEC 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.

domly 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 we have not compromised 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 bugs 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 16 and GHC 7.4.2, and using QuickCheck 2.5.1.1. We run each test for 5 minutes (300 seconds) or until 4000 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 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 §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 0.46 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 values).

Average number of execution steps: 0.47	
74%	stack underflow
21%	halt
4%	load or store out of range

Figure 6. Execution statistics for naive generation. Executions fail early, and the main reason for failure is stack underflow.

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

Figure 7. Weighted distribution – the main reason for failure is *Halt*, followed by stack underflows

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 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 shown in the second column of Figure 5. Although this strategy still performs badly for the LOAD* and STORE*A bugs, there is a significant improvement on both discard rates and the MTTF. Run length is also better, averaging 2.71 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 can generate *sequences* of instructions that make sense together. For instance, instead of generating single *Store* instructions, we can additionally generate sequences of the form [*Push* a, *Store*] (where a is a valid address), and similarly for other instructions that use stack elements. The results are shown in the third column of Figure 5. With sequence

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

Figure 8. Generating sequences of instructions – out-of-range addresses are now the biggest reason for termination

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

Figure 9. Smart integers – the percentage of address out of range errors has halved

generation we can now find all bugs, faster than before. Programs run for slightly longer (3.87). 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 can give preference to *valid* memory addresses, i.e., integers within memory bounds, when generating values. We do this not only when generating the state of the first machine, but also when *varying* it, since both machines need to halt 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 the MTTF and the average run length is now 4.23 steps, and the percentage of address-out-of-range errors is decreased.

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 plainly *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: (1) 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. (2) Having generated this instruction or sequence we simply execute it, compute a new state, and return to the previous step. We repeat this process 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 §5.

As we generate more instructions, we make sure to 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

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

Figure 10. Generation by execution, variation breakdown

$$i = \left[\begin{array}{l} \mathit{Push}\ \tfrac{2}{5}@\top, \mathit{Jump}, \mathit{Push}\ 1@\bot, \mathit{Push}\ 0@\bot, \mathit{Store}, \\ \mathit{Halt} \end{array}\right]$$

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

Figure 11. A textbook example of an implicit flow.

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 a very simple—almost trivial—information-flow 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:

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

(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* [39]. 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.

The standard way to prevent implicit flows is to label the pc—i.e., to make it a value, not a bare integer. Initial states have $pc =$

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

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

Figure 12. Jump should not lower the pc label

$0@\bot$, and after a jump to a secret address the label of the pc becomes $\top$:

$$\frac{i(pc) = \mathit{Jump}}{\boxed{pc \mid x@L_x : s \mid m} \Rightarrow \boxed{x@L_x \mid s \mid m}} \quad (\text{JUMP*B})$$

While the pc is 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 $\top$, and "low" otherwise.)

5.1.1 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 memories, written $S_1 \approx_{mem} S_2$, if either $\mathcal{L}_{pc_1} = \mathcal{L}_{pc_2} = \top$ or else $\mathcal{L}_{pc_1} = \mathcal{L}_{pc_2} = \bot$ 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.

$$\frac{i(pc) = \mathit{Jump}}{\boxed{pc \mid x@L_x : s \mid m} \Rightarrow \boxed{x@(L_x \vee \mathcal{L}_{pc}) \mid s \mid m}} \quad (\text{JUMP})$$

With this rule for jumps QuickCheck no longer finds any counterexamples. Some readers may find this odd: In order to fully address implicit flows, don't we also need to strengthen the store rule to handle the case where the pc is labeled high [1, 38]? The answer is no, but the reason is subtle: in the current machine, the pc can go from $\bot$ to $\top$ when we jump to a secret address, but it never goes from $\top$ to $\bot$! 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 $\bot$ after it has become $\top$. 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[2] than they are for jumps. 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 can now be either a value (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 $\mathsf{R}(x_1@L_1) \approx \mathsf{R}(x_2@L_2)$ if either $L_1 = L_2 = \top$ or else $x_1 = x_2$ and $L_1 = L_2 = \bot$ (this is the same as for values).[3]

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 an integer indicating how many stack values should be passed or returned (0 or 1 in the case of *Return*). Formally, *Call* n expects an address $x@L_x$ followed by n values on the stack. It sets the pc to x, labels this new pc by the join of L_x 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 n arguments.

$$\frac{i(pc) = \mathit{Call}\ n \qquad L = L_x \vee L_{pc}}{\begin{array}{c}\boxed{x_{pc}@L_{pc} \mid x@L_x : v_1 : \ldots : v_n : s \mid m} \Rightarrow \\ \boxed{x@L \mid v_1 : \ldots : v_n : \mathsf{R}(x_{pc}{+}1@L_{pc}) : s \mid m}\end{array}} \quad (\text{CALL*B})$$

Return n' 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 n' elements on the stack as return values, discarding all other elements in this stack frame.

$$\frac{i(pc) = \mathit{Return}\ n' \qquad n' \in \{0, 1\} \qquad k \geq n'}{\begin{array}{c}\boxed{pc \mid v_1 : \ldots : v_k : \mathsf{R}(x@L_x) : s \mid m} \Rightarrow \\ \boxed{x@L_x \mid v_1 : \ldots : v_{n'} : s \mid m}\end{array}} \quad (\text{RETURN*AB})$$

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 §B.2), we need to change the EENI instance we use to $\text{EENI}_{\mathit{Init},\mathit{Halted}\cap\mathit{Low},\approx_{mem}}$, i.e., 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 need to change the rule for *Store* so that the new memory contents are tainted with the current pc label. This eliminates the current counterexample; QuickCheck then finds a very similar one in which

[2] As long as we don't have exceptions! See [27] for a long discussion.

[3] High return addresses and high values need to be distinguishable to a low observer, as we discovered when QuickCheck generated an unexpected counterexample (which we list in §B.3).

$$i = \left[\begin{array}{l} \mathit{Push}\ \tfrac{3}{6}@\top, \mathit{Call}\ 0, \mathit{Halt}, \mathit{Push}\ 1@\bot, \mathit{Push}\ 0@\bot, \\ \mathit{Store}, \mathit{Return}\ 0 \end{array} \right]$$

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

Figure 13. Raising the 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.

$$i = \left[\begin{array}{l} \mathit{Push}\ \tfrac{3}{6}@\top, \mathit{Call}\ 0, \mathit{Halt}, \mathit{Push}\ 0@\bot, \mathit{Push}\ 0@\bot, \\ \mathit{Store}, \mathit{Return}\ 0 \end{array} \right]$$

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

Figure 14.

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 [1, 46]. This leads to the correct rule for stores:

$$\frac{i(pc) = \mathit{Store} \qquad \mathcal{L}_{pc} \vee L_x \sqsubseteq \mathcal{L}_{m(x)}}{\boxed{pc \mid x@L_x : y@L_y : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m\{x \mapsto y@(L_x \vee L_y \vee \mathcal{L}_{pc})\}}} \qquad (\textsc{Store})$$

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.

$$\frac{i(pc) = \mathit{Return}\ n' \qquad n' \in \{0,1\} \qquad k \geq n'}{\boxed{pc \mid v_1 : \ldots : v_k : \mathsf{R}(x@L_x) : s \mid m} \Rightarrow \boxed{x@L_x \mid v_1@\mathcal{L}_{pc} : \ldots : v_{n'}@\mathcal{L}_{pc} : s \mid m}} \qquad (\textsc{Return*b})$$

The next counterexample, listed in Figure 16, shows (maybe somewhat surprisingly) that it is unsound to specify the number of

$$i = \left[\begin{array}{l} \mathit{Push}\ 1@\bot, \mathit{Push}\ \tfrac{7}{6}@\top, \mathit{Call}\ 1, \mathit{Push}\ 0@\bot, \mathit{Store}, \\ \mathit{Halt}, \mathit{Push}\ 0@\bot, \mathit{Return}\ 1 \end{array} \right]$$

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

Figure 15. *Return* needs to taint the returned values.

$$i = \left[\begin{array}{l} \mathit{Push}\ 0@\bot, \mathit{Push}\ \tfrac{6}{7}@\top, \mathit{Call}\ 0, \mathit{Push}\ 0@\bot, \mathit{Store}, \\ \mathit{Halt}, \mathit{Return}\ 0, \mathit{Push}\ 0@\bot, \mathit{Return}\ 1 \end{array} \right]$$

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

Figure 16. It is unsound to choose how many results to return in the *Return* instruction.

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; accordingly, each stack frame should include not only a return address but also a number of return values. These changes lead us to the correct rules:

$$\frac{i(pc) = \mathit{Call}\ n\ n' \qquad n' \in \{0,1\} \qquad L = L_x \vee L_{pc}}{\boxed{x_{pc}@L_{pc} \mid x@L_x : v_1 : \ldots : v_n : s \mid m} \Rightarrow \boxed{x@L \mid v_1 : \ldots : v_n : \mathsf{R}(x_{pc}{+}1, n')@L_{pc} : s \mid m}} \qquad (\textsc{Call})$$

$$\frac{i(pc) = \textit{Return} \quad k \geq n'}{\boxed{pc \mid v_1 : \ldots : v_k : \mathsf{R}(x, n')@L_x : s \mid m} \Rightarrow \boxed{x@L_x \mid v_1@\mathcal{L}_{pc} : \ldots : v_{n'}@\mathcal{L}_{pc} : s \mid m}} \quad (\textsc{Return})$$

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 (we use e to denote an arbitrary stack entry):

$$\frac{i(pc) = \textit{Pop}}{\boxed{pc \mid e : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m}} \quad (\textsc{Pop*})$$

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

$$\frac{i(pc) = \textit{Pop}}{\boxed{pc \mid v : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m}} \quad (\textsc{Pop})$$

5.3 Generation by execution and control flow

Generation by execution is still applicable in this setting. However, interesting control flow necessitates small modifications to the original algorithm. We still generate a single instruction or sequence that does not crash, as before, and we execute to compute a new state. However, unlike before, while executing this newly generated sequence of instructions, we might "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 stops, or crashes (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.

In the presence of control flow, the state that we use to generate an instruction is only accurate the first time we execute this instruction. Subsequent executions of the instruction may cause the machine to crash and hence one might be worried about the discard ratio for EENI. However, the ever increasing probability of generating a *Halt* (discussed in §4) counterbalances this issue.

5.4 Generation by execution with lookahead

In generation by execution we never generate an instruction that causes the machine to crash in *just one* step. A further optimization is to avoid generating an instruction that causes the machine to crash in *a number of steps*. We refer to this number of steps as the *lookahead* parameter and in our experiments we use a lookahead of just 2 steps. If it is impossible to generate such an instruction, we retry with a smaller lookahead, until we succeed.

5.5 Finding the bugs

We experimentally evaluated the effectiveness of testing for this new version of the 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 slightly 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*. 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.[4] Second, as described previously, generation by execution in the presence of control flow is much less accurate.

[4] Detailed profiling revealed that 18% of the pairs of machines both loop, and loopy machines push the average number of execution steps to 22.

6. Strengthening the Tested Property

The last few counterexamples in §5 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 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 §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:

6.1.1 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 $\mathcal{L}_{pc_1} = \mathcal{L}_{pc_2} = \top$ or else $\mathcal{L}_{pc_1} = \mathcal{L}_{pc_2} = \bot$, $m_1 \approx m_2$, $i_1 \approx i_2$, $s_1 \approx s_2$, and $pc_1 \approx pc_2$.

We now strengthen $\text{EENI}_{\textit{Init},\textit{Halted}\cap\textit{Low},\approx_{mem}}$, the property we have been testing, to $\text{EENI}_{\textit{Init},\textit{Halted}\cap\textit{Low},\approx_{low}}$; 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 $\boxed{0@\bot \mid s \mid m \mid i}$, 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 two *Call* instructions. 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 counterexamples which cannot actually arise in a real execution. For example, a quasi-initial state may have a non-zero value in memory, even though the program contains no *Store* instruction that could have written it. 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, we have not encountered any spurious counterexamples for our machine, even with quasi-initial states.

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

$$i = \begin{bmatrix} \textit{Push}\ 5@\bot, \textit{Call}\ 0\ 1, \textit{Push}\ 0@\bot, \textit{Store}, \textit{Halt}, \\ \textit{Push}\ 0@\bot, \textit{Push}\ \frac{8}{9}@\top, \textit{Call}\ 0\ 0, \textit{Pop}, \textit{Push}\ 0@\bot, \\ \textit{Return} \end{bmatrix}$$

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

Figure 17. It is unsound not to protect the call stack.

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	TINYSSNI
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.

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*).

The function *trace* S computes the (possibly infinite) list of states obtained by executing our machine starting from state S. This is a function because our machine is deterministic.

6.3.1 Definition: $\mathit{trace}\ S = \begin{cases} [S] & \text{if } S \text{ is stuck} \\ S : \mathit{trace}\ S' & \text{if } S \Rightarrow S' \end{cases}$

While, in practice, we test LLNI over finite prefixes of traces, the definition below is also valid for potentially infinite traces.

6.3.2 Definition: A machine semantics is *low-lockstep noninterfering* with respect to the indistinguishability relation $\approx$ (written LLNI$_\approx$) if, for any quasi-initial states S_1 and S_2 with $S_1 \approx S_2$, we have $\mathit{trace}\ S_1 \approx^* \mathit{trace}\ S_2$, where $\approx^*$ is defined coinductively by the following rules:

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

$$\frac{S_1 \notin \mathit{Low} \quad t_1 \approx^* t_2}{(S_1 : t_1) \approx^* t_2} \quad (\text{HIGH FILTER})$$

$$\frac{}{[\,] \approx^* [\,]} \quad (\text{LOCKSTEP END})$$

$$\frac{S_1 \notin \mathit{Low}}{[S_1] \approx^* t_2} \quad (\text{HIGH END})$$

$$\frac{S_1 \in \mathit{Low} \quad S_1 \notin \mathit{Halted} \quad S_1 \approx S_2}{[S_1] \approx^* (S_2 : t_2)} \quad (\text{LOW ERROR END})$$

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

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 rules are 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 in a state that is not low (HIGH END) or not halted (LOW ERROR END). Additionally, we allow the two traces to terminate simultaneously (LOCKSTEP END). We implement these rules in Haskell as a recursive predicate over lazy lists.

In general, LLNI implies EENI, 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 wrt 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 or infinite executions. The generation strategies described in §4 also apply to LLNI without much change; also, as for EENI, generation by execution (with lookahead of 2 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 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" [25]; 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 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 when 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 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; we write S^L for a machine state whose *pc* label is L.

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

1. For all low states $S_1^\perp$ and $S_2^\perp$, if $S_1^\perp \approx S_2^\perp$, $S_1^\perp \Rightarrow S_1$, and $S_2^\perp \Rightarrow S_2$, then $S_1 \approx S_2$;
2. For all high states $S^\top$ with $S^\top \Rightarrow S_\star^\top$, we have $S^\top \approx S_\star^\top$;
3. For all high states $S_1^\top$ and $S_2^\top$, if $S_1^\top \approx S_2^\top$, $S_1^\top \Rightarrow S_1^\perp$, $S_2^\top \Rightarrow S_2^\perp$, and states $S_1^\perp$ and $S_2^\perp$ are low, then $S_1^\perp \approx S_2^\perp$;
4. For all low states $S_1^\perp$ and $S_2^\perp$, if $S_1^\perp \approx S_2^\perp$ and $S_1^\perp$ is halted, then $S_2^\perp$ is stuck.

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 (QuickCheck can find counterexamples to condition 3). 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.) These counterexamples (which are given in §B.4) show that indistinguishable high states can have different *pc*s 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.

6.4.2 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 whole machine states*, written $S_1 \approx_{full} S_2$, if $m_1 \approx m_2$, $i_1 \approx i_2$, $\mathcal{L}_{pc_1} = \mathcal{L}_{pc_2}$, and additionally

- if $\mathcal{L}_{pc_1} = \perp$ then $s_1 \approx s_2$ and $pc_1 \approx pc_2$, and
- if $\mathcal{L}_{pc_1} = \top$ then $\mathit{cropStack}\ s_1 \approx \mathit{cropStack}\ s_2$.

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).

The fifth column of Figure 18 shows that, even with arbitrary starting states generated completely naively, 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 milliseconds. (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 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. In the case of SSNI we used additional invariants of our machine (captured by $\approx_{full}$) and finding these invariants would probably constitute 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, we expect that for more realistic machines the process of interpreting the counterexamples and manually refining the invariants will be significantly harder than for our very simple machine.

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 our simple machine, Delphine Demange did in fact prove formally in Coq that SSNI holds, and did not find any bugs that had escaped our testing. Moreover, we proved in Coq that for any deterministic machine and for any indistinguishability relation that is an equivalence, SSNI implies LLNI and LLNI implies EENI.

7. Shrinking Strategies

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

7.1 Shrinking labeled values, instructions, and stack elements

By default, QuickCheck already implements a shrinking strategy for integers. For labels, we shrink $\top$ to $\bot$, because we prefer to see counterexamples in which labels are only $\top$ 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.)

Instructions are shrunk as follows: 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 the bug had already manifested by the time control flow reached that instruction. To avoid an infinite shrinking loop, we do not allow *Noop* to shrink at all, while *Halt* can shrink only to *Noop*. Instructions of the form *Push a* are also shrunk by shrinking a. Finally, instructions of the form *Call a r* are also shrunk by shrinking a, by shrinking r, or by replacing the whole instruction with *Jump*.

For quasi-initial or arbitrary states the stack contains a mixture of values and return addresses, which are shrunk pointwise.

7.2 Machine States

Machine states contain an instruction memory, a data memory, a stack, and the initial pc. The first three of these are candidates for shrinking. We allow any *element* of the memories or the stack to be shrunk by the methods above; additionally, shrinking may *remove* elements from any of these.

We allow shrinking to remove arbitrary elements of the data memory or the stack, but in the case of the data memory we first try to remove the last value from the memory. This is because removing other elements changes the addresses of all subsequent memory cells, which is quite likely to invalidate a counterexample, rendering the shrinking step unsuccessful.

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 thus risks invalidating a counterexample. Other instructions can still be removed in two stages, by first shrinking them to a *Noop*.

7.3 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 which are also indistinguishable.

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 behaviour 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 values 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 value on the initial stack, or any value 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 examples, from which no instruction, stack element, or memory element could be removed without invalidating the counterexample.

8. Related Work

Testing programs by generating random inputs is a large research area, but the particular sub-area of testing language implementations by generating random *programs* is less well studied. Redex [28–30] (*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 testing 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 generation 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 unfeasible.

Klein et al. [31] use PLT Redex's QuickCheck-inspired random testing framework to asses 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 [45] is a C compiler testing tool that generates random C programs, avoiding ones whose behaviour 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 work with smaller counterexamples, Regehr et al. present C-Reduce [36], a tool for reducing test-case C programs such as those produced by CSmith. They observe 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 which 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, beyond 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 algorithm *ddmin* [47], a generic (non-domain-specific) algorithm for simplifying and isolating failure-inducing program inputs using an extension of binary search. In our work, as in Regehr et al.'s, using domain-specific knowledge is crucial to the success of shrinking.

Another relevant example of testing programs by generating random input is Randoop [34], 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 powerful and widely used approach to testing is symbolic execution—in particular, *concolic testing and related dynamic symbolic execution techniques* [14, 33]. 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 [24] and PathCrawler [44], a variety of tools and methods have appeared; some of the state-of-the-art tools include CUTE [41], CREST [11], and KLEE [12] (which evolved from EXE [13]) . We wondered whether dynamic symbolic excecution 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 [15], so we don't cover enough of it to reach the particular IFC-violating configurations.

Balliu et al. [3] 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 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.

The Haskell library Feat [20] is designed to perform efficient arbitrary enumeration of all values of algebraic data types. Duregård et al. show how to perform this enumeration mechanically for most data types. One important concern when writing a QuickCheck generator (§4) is to ensure that the state space is covered; for this reason, Feat includes a function for picking bounded-size values uniformly at random from any enumerable type.

In the context of interactive theorem proving, automatically generating counterexamples for false conjectures can prevent wasting time and effort on doomed proof attempts. Dybjer [21] propose a QuickCheck-like random testing tool for the Agda/Alfa proof assistant. Berghofer and Nipkow [7] implemented a QuickCheck-like tool for Isabelle/HOL. This was recently extended by Bulwahn [9] to also support exhaustive and narrowing-based symbolic testing [16, 32, 37]. 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 [10]. Eastlund [22] implemented DoubleCheck, an adaption of QuickCheck for ACL2. Chamarthi et al. [35] later proposed a more advanced counterexample finding tool for ACL2s, which uses the full power of the theorem prover and associated 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 this would not work very well without incorporating domain knowledge about the machine and the property being tested (e.g., the naive program generator for EENI didn't perform well at finding bugs even for the simplest machine we investigate, as Figure 5 illustrates). We hope to compare our work against these tools in the future and provide experimental evidence for this intuition.

On the dynamic IFC side Birgisson et al. [8] have a good overview of related work. Our correct rule for *Store* is called the *no-sensitive-upgrades* policy in the literature and was first proposed by Zdancewic [46] and later adapted to the purely dynamic IFC setting by Austin and Flanagan [1]. To improve precision, Austin and Flanagan [2] 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 and Sabelfeld [26] 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 apply their technique to a core calculus of JavaScript that includes objects, higher-order functions, exceptions, and dynamic code evaluation. Birgisson et al. [8] 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.

Tachio and Aiken [43] and later Barthe et al. [5] 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 [4, 6]) 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 [39], relational program logics [4, 6], and self-composition based tools [5].

9. 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. Even if we ultimately care about full security proofs, 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.

What crucially remains to be seen is whether our results will scale up to more realistic settings. We are actively pursuing this question in the context of CRASH/SAFE, an ambitious hardware–software co-design effort underway at Penn, Harvard, Northeastern, and BAE Systems [18, 19], with IFC as a key feature at all levels, from hardware to application code. The design involves a number of abstract machines, all much more complex than the stack machine we have studied here. We hope to use random testing both for checking noninterference properties of individual abstract machines and for checking that the code running on lower-level abstract machines correctly implements the higher-level abstractions.

Just how well do our methods need to perform, to find bugs effectively in these machines? The true answer is anybody's guess, but for a very rough estimate we might guess there will be around $10\times$ as many instructions on a real SAFE machine, that each instruction might be on average $3\times$ more complex, and that several cross-cutting features will induce additional complexity factors—e.g., "public labels" [27] ($?2\times$), dynamic storage allocation ($?2\times$), a "fat pointer" memory model ($?2\times$), a more complex lattice of labels ($?2\times$), and dynamic principal generation ($?2\times$). Combining these, we could be finding bugs more than $1000\times$ more slowly.

If this calculation is in the right ballpark, then EENI is nowhere near fast enough: even for our stack machine, it can take several minutes to find some bugs. Between LLNI and SSNI, on the other hand, there is a tradeoff (discussed in §6) between the overhead of finding good invariants for SSNI and the increased bug-finding rate once this is done. Both approaches seem potentially useful (and potentially fast enough), perhaps at different points in the design process. In particular, checking SSNI may help find invariants that will also be needed for a formal proof.

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 [4, 6])—in particular, the many variants and generalizations of noninterference. 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 to the discussion at the IFIP WG 2.8 Storulvån meeting that originated this work: Ulf Norell, Rishiyur S. Nikhil, Michał Pałka, Nick Smallbone, and Meng Wang. We are grateful to Delphine Demange for proving SSNI in Coq for our simple abstract machine. We also thank 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, Andrew Tolmach, for providing feedback on a draft, and the members of the CRASH/SAFE team for fruitful discussions. This material is based upon work supported by the DARPA CRASH program through the US Air Force Research Laboratory (AFRL) under Contract No. FA8650-10-C-7090. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

[1] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Workshop on Programming Languages and Analysis for Security*, PLAS. 2009.

[2] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th Workshop on Programming Languages and Analysis for Security*, PLAS. 2010.

[3] M. Balliu, M. Dam, and G. L. Guernic. Encover: Symbolic exploration for information flow security. In *25th IEEE Computer Security Foundations Symposium (CSF 2012)*. 2012.

[4] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *17th International Symposium on Formal Methods (FM)*, volume 6664 of *Lecture Notes in Computer Science*. 2011.

[5] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[6] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2004.

[7] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In *2nd International Conference on Software Engineering and Formal Methods (SEFM)*. 2004.

[8] A. Birgisson, D. Hedin, and A. Sabelfeld. Boosting the permissiveness of dynamic information-flow tracking by testing. In *17th European Symposium on Research in Computer Security*, ESORICS. 2012.

[9] L. Bulwahn. The new Quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. In *2nd International Conference on Certified Programs and Proofs (CPP)*, volume 7679 of *Lecture Notes in Computer Science*. 2012.

[10] L. Bulwahn. Smart testing of functional programs in Isabelle. In *18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 7180 of *Lecture Notes in Computer Science*. 2012.

[11] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008.

[12] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX conference on Operating systems design and implementation*, OSDI. 2008.

[13] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *13th ACM conference on Computer and communications security*, CCS '06. 2006.

[14] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *33rd International Conference on Software Engineering*, ICSE '11. 2011.

[15] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[16] J. Christiansen and S. Fischer. Easycheck - test data for free. In *9th International Symposium on Functional and Logic Programming (FLOPS)*, volume 4989 of *Lecture Notes in Computer Science*. 2008.

[17] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming*, ICFP. 2000.

[18] A. DeHon, B. Karel, T. F. Knight, Jr., G. Malecha, B. Montagu, R. Morisset, G. Morrisett, B. C. Pierce, R. Pollack, S. Ray, O. Shivers, J. M. Smith, and G. Sullivan. Preliminary design of the SAFE platform. In *6th Workshop on Programming Languages and Operating Systems*, PLOS, October 2011.

[19] U. Dhawan, A. Kwon, E. Kadric, C. Hriţcu, B. C. Pierce, J. M. Smith, G. Malecha, G. Morrisett, T. F. Knight, Jr., A. Sutherland, T. Hawkins, A. Zyxnfryx, D. Wittenberg, P. Trei, S. Ray, G. Sullivan, and A. DeHon. Hardware support for safety interlocks and introspection. In *SASO Workshop on Adaptive Host and Network Security*, September 2012.

[20] J. Duregård, P. Jansson, and M. Wang. Feat: functional enumeration of algebraic types. In *Haskell Symposium*. 2012.

[21] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2758 of *Lecture Notes in Computer Science*. 2003.

[22] C. Eastlund. Doublecheck your theorems. In *ACL2*, 2009.

[23] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.

[24] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI. 2005.

[25] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, 1984.

[26] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *25th IEEE Computer Security Foundations Symposium*, CSF. 2012.

[27] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. In *34th IEEE Symposium on Security and Privacy (Oakland)*, May 2013. To appear.

[28] C. Klein. Experience with randomized testing in programming language metatheory. Master's thesis, Northwestern, August 2009. `http://plt.eecs.northwestern.edu/klein-masters.pdf`.

[29] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: On the effectiveness of lightweight mechanization. In *Principles of Programming Languages (POPL)*, 2012.

[30] C. Klein and R. B. Findler. Randomized testing in PLT Redex. In *Workshop on Scheme and Functional Programming (SFP)*, 2009.

[31] C. Klein, M. Flatt, and R. B. Findler. The Racket virtual machine and randomized testing. Submitted to HOSC, 2010.

[32] F. Lindblad. Property directed generation of first-order test data. In *8th Symposium on Trends in Functional Programming (TFP)*, 2007.

[33] R. Majumdar and K. Sen. Hybrid concolic testing. In *29th international conference on Software Engineering*, ICSE. 2007.

[34] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems And Applications*, OOPSLA. 2007.

[35] H. Raju Chamarthi, P. Dillinger, M. Kaufmann, and P. Manolios. Integrating testing and interactive theorem proving. In *ACL2*, 2011.

[36] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM, 2012.

[37] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *1st ACM SIGPLAN Symposium on Haskell*. 2008.

[38] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *23rd Computer Security Foundations Symposium*, CSF. 2010.

[39] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

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

[41] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13. 2005.

[42] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th Symposium on Haskell*. 2011.

[43] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *12th International Symposium on Static Analysis (SAS)*, volume 3672 of *Lecture Notes in Computer Science*. 2005.

[44] N. Williams, B. Marre, and P. Mouy. On-the-fly generation of K-path tests for C functions. In *19th IEEE International Conference on Automated Software Engineering*, ASE. 2004.

[45] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI. 2011.

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

[47] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

A. Variations in Haskell

At the beginning of §4 we explained how we generate pairs indistinguishable states. This appendix provides a few more details for readers familiar with Haskell and QuickCheck: We define a type of *variations* on a type a

$$\textbf{data}\ \mathit{Variation}\ a = \mathit{Variation}\ a\ a$$

with an invariant that the components should be indistinguishable, plus a class of *observable types* for which we can generate a variation on a previously generated value:

$$\begin{array}{l} \textbf{class}\ \mathit{Observable}\ a\ \textbf{where} \\ \quad (\approx) \quad :: \ a \to a \to \mathit{Bool} \\ \quad \mathit{vary} \quad :: \ a \to \mathit{Gen}\ a \end{array}$$

(where *Gen* a is a QuickCheck generator for values of type a). We ensure that *vary* a generates values a' for which $a' \approx a$. We define *vary* for values, instructions, and machine states, for example replacing values labeled $\top$ by random values. Then we generate *Variation*s by generating the first component randomly, and generating the second component from the first using *vary*. Our test cases for noninterference properties are *Variation*s on machine states.

B. Varying the Indistinguishability Relation

B.1 Counterexamples showing that labels need to be observable

As seen in §2.4, the counterexample in Figure 2 illustrates that our definition of indistinguishability of values (Definition 2.3.1) 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, both of which are, however, wrong. First, one could try defining indistinguishability of atoms so that $x@\bot \approx y@\top$ for any x and y. QuickCheck easily finds a counterexample to this (Figure 19). Second, one could try refining this so that only $x@\bot \approx x@\top$, i.e., a high atom is equivalent with a low one only when the payloads are equal. QuickCheck also disproves this alternative (Figure 20), and the counterexample produced by QuickCheck illustrates how, even with the correct rules, a difference in the labels of two atoms can be turned into a difference in the values of two atoms. This counterexample is reminiscent of a well-known "flow-sensitivity attack" (Figure 1 in [38]; attributed to [23]).

B.2 Counterexample motivating the instantiation of EENI with *Halted* ∩ *Low* when adding calls and returns

The counterexample in Figure 21 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 §5.2 is to weaken the EENI instance, by considering only ending states that are both halting and low (i.e., we change to $\mathrm{EENI}_{\mathit{Init},\mathit{Halted}\cap\mathit{Low},\approx_{mem}}$).

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

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

Figure 19. Counterexample showing that is wrong to make high atoms be equivalent to all other atoms.

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

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

Figure 20. Counterexample showing that making high atoms be equivalent only to low atoms with the same payload is also wrong.

B.3 Counterexample motivating the indistinguishability of stack elements for the machine with calls and returns

In §5.2 we defined the indistinguishability relation on stack elements so that return addresses are only equivalent to other return addresses and $\mathsf{R}(x_1@L_1) \approx \mathsf{R}(x_2@L_2)$ if either $L_1 = L_2 = \top$ or else $x_1 = x_2$ and $L_1 = L_2 = \bot$ (this is the same as for values). If instead we considered high return addresses to be indistinguishable from high values, QuickCheck would find a counterexample. This counterexample requires quasi-initial states (and $\approx_{low}$) and is listed in Figure 22. The first machine performs only one *Return* that throws away two elements from the stack and then halts. The second machine returns twice: first time to the same address in the code but unwinding the stack and raising the pc in the process, and the second time to the *Halt* instruction, labeling the return value high with the high pc in the process. At two final machine states are distinguishable because the elements on the stack have different labels. As we saw in §B.1, such a counterexample can be extended to one in which values also differ.

$$i = \left[\ \mathit{Push}\ \tfrac{2}{3}@\top, \mathit{Call}\ 0\ 0, \mathit{Halt}, \mathit{Return}\ \right]$$

pc	m	s	$i(pc)$
$0@\bot$	$[]$	$[]$	$\mathit{Push}\ \frac{2}{3}@\top$
$1@\bot$	$[]$	$\left[\frac{2}{3}@\top\right]$	$\mathit{Call}\ 0\ 0$
Machine 1 continues…			
$2@\top$	$[]$	$[\mathsf{R}(2,0)@\bot]$	Halt
Machine 2 continues…			
$3@\top$	$[]$	$[\mathsf{R}(2,0)@\bot]$	Return
$2@\bot$	$[]$	$[]$	Halt

Figure 21. Counterexample justifying the change to $\text{EENI}_{\mathit{Init},\mathit{Halted}\cap\mathit{Low},\approx_{mem}}$ in §5.2.

$$i = \left[\ \mathit{Return}, \mathit{Halt}\ \right]$$

pc	m	s	$i(pc)$
$0@\bot$	$[]$	$\left[0@\bot, \frac{0@\top}{\mathsf{R}(0,0)@\top}, 0@\bot, \mathsf{R}(1,1)@\bot\right]$	Return
Machine 1 continues…			
$1@\bot$	$[]$	$[0@\bot]$	Halt
Machine 2 continues…			
$0@\top$	$[]$	$[0@\bot, \mathsf{R}(1,1)@\bot]$	Return
$1@\bot$	$[]$	$[0@\top]$	Halt

Figure 22. Counterexample motivating the indistinguishability of stack elements for the machine with calls and returns.

$$i = \left[\ \mathit{Return}\ \right]$$

pc	m	s	$i(pc)$
$0@\top$	$[]$	$\left[\mathsf{R}(\frac{0}{1},0)@\bot\right]$	Return
Machine 1 continues…			
$0@\bot$	$[]$	$[]$	Return
Machine 2 continues…			
$1@\bot$	$[]$	$[]$	–

Figure 23. 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.

B.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 6.4.1 (see Figure 23).

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 6.4.1 (see Figure 24). 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 6.4.2). 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.

$$i = \left[\ \mathit{Pop}\ \right]$$

pc	m	s	$i(pc)$
$0@\top$	$[]$	$[0@\bot]$	Pop
$1@\top$	$[]$	$[]$	–

Figure 24. Counterexample showing that treating high states exactly like low states in the indistinguishability relation over machine states is too strong and breaks condition 2 in Definition 6.4.1. When a machine steps from a high state to another high state the contents of the stack can change.

$$i = \left[\ \mathit{Push}\ 1@\bot, \mathit{Push}\ \tfrac{0}{1}@\top, \mathit{Push}\ 0@\bot, \mathit{Add}, \mathit{Store}, \mathit{Halt}\ \right]$$

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

Figure 25. Counterexample to ADD*

C. Individual bugs

For completeness we list all the 14 individual bugs we introduce in the machine with control flow from §5, and show the counterexamples QuickCheck finds for each of the bugs. These 14 bugs correspond to the 14 rows of Figure 18. The property we test is the original variant of EENI, $\text{EENI}_{\mathit{Init},\mathit{Halted},\approx_{mem}}$, which corresponds to the first column in Figure 18.

C.1 ADD* (Counterexample in Figure 25)

$$\frac{i(pc) = \mathit{Add}}{\boxed{pc \mid x@L_x : y@L_y : s \mid m} \Rightarrow \boxed{pc{+}1 \mid (x{+}y)@\bot : s \mid m}} \qquad (\text{ADD*})$$

C.2 PUSH* (Counterexample in Figure 26)

$$\frac{i(pc) = \mathit{Push}\ x@L}{\boxed{pc \mid s \mid m} \Rightarrow \boxed{pc{+}1 \mid x@\bot : s \mid m}} \qquad (\text{PUSH*})$$

C.3 LOAD* (Counterexample in Figure 27)

$$\frac{i(pc) = \mathit{Load}}{\boxed{pc \mid x@L_x : s \mid m} \Rightarrow \boxed{pc{+}1 \mid m(x) : s \mid m}} \qquad (\text{LOAD*})$$

C.4 STORE*A (Counterexample in Figure 28)

$$\frac{i(pc) = \mathit{Store} \qquad \mathcal{L}_{pc} \vee L_x \sqsubseteq \mathcal{L}_{m(x)}}{\boxed{pc \mid x@L_x : y@L_y : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m\{x \mapsto y@(L_y \vee \mathcal{L}_{pc})\}}} \qquad (\text{STORE*A})$$

$$i = \left[\ \textit{Push}\ \tfrac{0}{1}@\top, \textit{Push}\ 0@\bot, \textit{Store}, \textit{Halt}\ \right]$$

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

Figure 26. Counterexample to PUSH*

$$i = \left[\begin{array}{l}\textit{Push}\ 1@\bot, \textit{Push}\ 0@\bot, \textit{Store}, \textit{Push}\ \tfrac{0}{1}@\top, \textit{Load},\\ \textit{Push}\ 0@\bot, \textit{Store}, \textit{Halt}\end{array}\right]$$

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

Figure 27. Counterexample to LOAD*

$$i = \left[\begin{array}{l}\textit{Push}\ 0@\top, \textit{Push}\ 0@\bot, \textit{Store}, \textit{Push}\ 0@\bot, \textit{Push}\ 0@\top,\\ \textit{Push}\ 1@\bot, \textit{Store}, \textit{Push}\ \tfrac{1}{0}@\top, \textit{Store}, \textit{Halt}\end{array}\right]$$

pc	m	s	$i(pc)$
$0@\bot$	$[0@\bot, 0@\bot]$	$[\,]$	$\textit{Push}\ 0@\top$
$1@\bot$	$[0@\bot, 0@\bot]$	$[0@\top]$	$\textit{Push}\ 0@\bot$
$2@\bot$	$[0@\bot, 0@\bot]$	$[0@\bot, 0@\top]$	$\textit{Store}$
$3@\bot$	$[0@\top, 0@\bot]$	$[\,]$	$\textit{Push}\ 0@\bot$
$4@\bot$	$[0@\top, 0@\bot]$	$[0@\bot]$	$\textit{Push}\ 0@\top$
$5@\bot$	$[0@\top, 0@\bot]$	$[0@\top, 0@\bot]$	$\textit{Push}\ 1@\bot$
$6@\bot$	$[0@\top, 0@\bot]$	$[1@\bot, 0@\top, 0@\bot]$	$\textit{Store}$
$7@\bot$	$[0@\top, 0@\top]$	$[0@\bot]$	$\textit{Push}\ \frac{1}{0}@\top$
$8@\bot$	$[0@\top, 0@\top]$	$\left[\frac{1}{0}@\top, 0@\bot\right]$	$\textit{Store}$
$9@\bot$	$\left[0@\frac{\top}{\bot}, 0@\frac{\bot}{\top}\right]$	$[\,]$	$\textit{Halt}$

Figure 28. Counterexample to STORE*A

C.5 STORE*B (Counterexample in Figure 29)

$$\frac{i(pc) = \textit{Store} \qquad \mathcal{L}_{pc} \sqsubseteq \mathcal{L}_{m(x)}}{\boxed{pc \mid x@L_x : y@L_y : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m\{x \mapsto y@(L_x \vee L_y \vee \mathcal{L}_{pc})\}}} \qquad (\text{STORE*B})$$

C.6 STORE*C (Counterexample in Figure 30)

$$\frac{i(pc) = \textit{Store} \qquad \mathcal{L}_{pc} \vee L_x \sqsubseteq \mathcal{L}_{m(x)}}{\boxed{pc \mid x@L_x : y@L_y : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m\{x \mapsto y@(L_x \vee \mathcal{L}_{pc})\}}} \qquad (\text{STORE*C})$$

$$i = \left[\ \textit{Push}\ 0@\bot, \textit{Push}\ \tfrac{1}{0}@\top, \textit{Store}, \textit{Halt}\ \right]$$

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

Figure 29. Counterexample to STORE*B

$$i = \left[\ \textit{Push}\ \tfrac{0}{1}@\top, \textit{Push}\ 0@\bot, \textit{Store}, \textit{Halt}\ \right]$$

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

Figure 30. Counterexample to STORE*C

$$i = \left[\begin{array}{l}\textit{Push}\ 1@\bot, \textit{Push}\ 0@\bot, \textit{Push}\ \tfrac{4}{5}@\top, \textit{Jump}, \textit{Store},\\ \textit{Halt}\end{array}\right]$$

pc	m	s	$i(pc)$
$0@\bot$	$[0@\bot]$	$[\,]$	$\textit{Push}\ 1@\bot$
$1@\bot$	$[0@\bot]$	$[1@\bot]$	$\textit{Push}\ 0@\bot$
$2@\bot$	$[0@\bot]$	$[0@\bot, 1@\bot]$	$\textit{Push}\ \frac{4}{5}@\top$
$3@\bot$	$[0@\bot]$	$\left[\frac{4}{5}@\top, 0@\bot, 1@\bot\right]$	$\textit{Jump}$
Machine 1 continues…			
$4@\bot$	$[0@\bot]$	$[0@\bot, 1@\bot]$	$\textit{Store}$
$5@\bot$	$[1@\bot]$	$[\,]$	$\textit{Halt}$
Machine 2 continues…			
$5@\bot$	$[0@\bot]$	$[0@\bot, 1@\bot]$	$\textit{Halt}$

Figure 31. Counterexample to JUMP*A

C.7 JUMP*A (Counterexample in Figure 31)

$$\frac{i(pc) = \textit{Jump}}{\boxed{pc \mid x@L_x : s \mid m} \Rightarrow \boxed{x@\mathcal{L}_{pc} \mid s \mid m}} \qquad (\text{JUMP*A})$$

C.8 JUMP*B (Counterexample in Figure 32)

$$\frac{i(pc) = \textit{Jump}}{\boxed{pc \mid x@L_x : s \mid m} \Rightarrow \boxed{x@L_x \mid s \mid m}} \qquad (\text{JUMP*B})$$

C.9 STORE*D (Counterexample in Figure 33)

$$\frac{i(pc) = \textit{Store} \qquad \mathcal{L}_{pc} \vee L_x \sqsubseteq \mathcal{L}_{m(x)}}{\boxed{pc \mid x@L_x : y@L_y : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m\{x \mapsto y@(L_x \vee L_y)\}}} \qquad (\text{STORE*D})$$

$$i = \left[\begin{array}{l} \mathit{Push}\ 1@\bot, \mathit{Push}\ \frac{5}{6}@\top, \mathit{Jump}, \mathit{Store}, \mathit{Halt}, \\ \mathit{Push}\ 0@\bot, \mathit{Push}\ 0@\bot, \mathit{Push}\ 3@\bot, \mathit{Jump} \end{array}\right]$$

pc	m	s	$i(pc)$
$0@\bot$	$[0@\bot]$	$[]$	$\mathit{Push}\ 1@\bot$
$1@\bot$	$[0@\bot]$	$[1@\bot]$	$\mathit{Push}\ \frac{5}{6}@\top$
$2@\bot$	$[0@\bot]$	$\left[\frac{5}{6}@\top, 1@\bot\right]$	Jump
Machine 1 continues…			
$5@\top$	$[0@\bot]$	$[1@\bot]$	$\mathit{Push}\ 0@\bot$
$6@\top$	$[0@\bot]$	$[0@\bot, 1@\bot]$	$\mathit{Push}\ 0@\bot$
$7@\top$	$[0@\bot]$	$[0@\bot, 0@\bot, 1@\bot]$	$\mathit{Push}\ 3@\bot$
$8@\top$	$[0@\bot]$	$[3@\bot, 0@\bot, 0@\bot, 1@\bot]$	Jump
$3@\bot$	$[0@\bot]$	$[0@\bot, 0@\bot, 1@\bot]$	Store
$4@\bot$	$[0@\bot]$	$[1@\bot]$	Halt
Machine 2 continues…			
$6@\top$	$[0@\bot]$	$[1@\bot]$	$\mathit{Push}\ 0@\bot$
$7@\top$	$[0@\bot]$	$[0@\bot, 1@\bot]$	$\mathit{Push}\ 3@\bot$
$8@\top$	$[0@\bot]$	$[3@\bot, 0@\bot, 1@\bot]$	Jump
$3@\bot$	$[0@\bot]$	$[0@\bot, 1@\bot]$	Store
$4@\bot$	$[1@\bot]$	$[]$	Halt

Figure 32. Counterexample to JUMP*B

$$i = \left[\begin{array}{l} \mathit{Push}\ \frac{6}{9}@\top, \mathit{Push}\ 0@\top, \mathit{Push}\ 0@\bot, \mathit{Store}, \mathit{Call}\ 0\ 0, \\ \mathit{Halt}, \mathit{Push}\ 0@\bot, \mathit{Push}\ 0@\bot, \mathit{Store}, \mathit{Return} \end{array}\right]$$

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

Figure 33. Counterexample to STORE*D

C.10 STORE*E (Counterexample in Figure 34)

$$\frac{i(pc) = \mathit{Store} \qquad L_x \sqsubseteq \mathcal{L}_{m(x)}}{\boxed{pc \mid x@L_x : y@L_y : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m\{x \mapsto y@(L_x \vee L_y \vee \mathcal{L}_{pc})\}}} \quad (\text{STORE*E})$$

C.11 CALL*A (Counterexample in Figure 35)

$$\frac{i(pc) = \mathit{Call}\ n\ n' \qquad n' \in \{0,1\}}{\boxed{x_{pc}@L_{pc} \mid x@L_x : v_1 : \ldots : v_n : s \mid m} \Rightarrow \boxed{x@L_{pc} \mid v_1 : \ldots : v_n : \mathsf{R}(x_{pc}{+}1, n')@L_{pc} : s \mid m}} \quad (\text{CALL*A})$$

$$i = \left[\begin{array}{l} \mathit{Push}\ 5@\bot, \mathit{Jump}, \mathit{Push}\ 0@\bot, \mathit{Store}, \mathit{Return}, \\ \mathit{Push}\ 0@\bot, \mathit{Push}\ \frac{2}{4}@\top, \mathit{Call}\ 1\ 0, \mathit{Halt} \end{array}\right]$$

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

Figure 34. Counterexample to STORE*E

$$i = \left[\begin{array}{l} \mathit{Push}\ 1@\bot, \mathit{Push}\ 0@\bot, \mathit{Store}, \mathit{Push}\ \frac{8}{6}@\top, \mathit{Jump}, \\ \mathit{Halt}, \mathit{Push}\ 10@\bot, \mathit{Call}\ 0\ 0, \mathit{Push}\ 5@\bot, \mathit{Call}\ 0\ 0, \\ \mathit{Push}\ 0@\bot, \mathit{Push}\ 0@\bot, \mathit{Store}, \mathit{Return} \end{array}\right]$$

pc	m	s	$i(pc)$
$0@\bot$	$[0@\bot]$	$[]$	$\mathit{Push}\ 1@\bot$
$1@\bot$	$[0@\bot]$	$[1@\bot]$	$\mathit{Push}\ 0@\bot$
$2@\bot$	$[0@\bot]$	$[0@\bot, 1@\bot]$	Store
$3@\bot$	$[1@\bot]$	$[]$	$\mathit{Push}\ \frac{8}{6}@\top$
$4@\bot$	$[1@\bot]$	$\left[\frac{8}{6}@\top\right]$	Jump
Machine 1 continues…			
$8@\top$	$[1@\bot]$	$[]$	$\mathit{Push}\ 5@\bot$
$9@\top$	$[1@\bot]$	$[5@\bot]$	$\mathit{Call}\ 0\ 0$
$5@\bot$	$[1@\bot]$	$[\mathsf{R}(10,0)@\top]$	Halt
Machine 2 continues…			
$6@\top$	$[1@\bot]$	$[]$	$\mathit{Push}\ 10@\bot$
$7@\top$	$[1@\bot]$	$[10@\bot]$	$\mathit{Call}\ 0\ 0$
$10@\bot$	$[1@\bot]$	$[\mathsf{R}(8,0)@\top]$	$\mathit{Push}\ 0@\bot$
$11@\bot$	$[1@\bot]$	$[0@\bot, \mathsf{R}(8,0)@\top]$	$\mathit{Push}\ 0@\bot$
$12@\bot$	$[1@\bot]$	$[0@\bot, 0@\bot, \mathsf{R}(8,0)@\top]$	Store
$13@\bot$	$[0@\bot]$	$[\mathsf{R}(8,0)@\top]$	Return
$8@\top$	$[0@\bot]$	$[]$	$\mathit{Push}\ 5@\bot$
$9@\top$	$[0@\bot]$	$[5@\bot]$	$\mathit{Call}\ 0\ 0$
$5@\bot$	$[0@\bot]$	$[\mathsf{R}(10,0)@\top]$	Halt

Figure 35. Counterexample to CALL*A

C.12 RETURN*A (Counterexample in Figure 36)

$$\frac{i(pc) = \mathit{Return} \qquad k \geq n'}{\boxed{pc \mid v_1 : \ldots : v_k : \mathsf{R}(x, n')@L_x : s \mid m} \Rightarrow \boxed{x@L_x \mid v_1 : \ldots : v_{n'} : s \mid m}} \quad (\text{RETURN*A})$$

C.13 CALL*B and RETURN*B (Counterexample in Figure 37)

$$\frac{i(pc) = \mathit{Call}\ n \qquad L = L_x \vee L_{pc}}{\boxed{x_{pc}@L_{pc} \mid x@L_x : v_1 : \ldots : v_n : s \mid m} \Rightarrow \boxed{x@L \mid v_1 : \ldots : v_n : \mathsf{R}(x_{pc}{+}1@L_{pc}) : s \mid m}} \quad (\text{CALL*B})$$

$$i = \left[\begin{array}{l} \textit{Push}\ 1@\bot, \textit{Push}\ \frac{6}{7}@\top, \textit{Call}\ 1\ 1, \textit{Push}\ 0@\bot, \textit{Store}, \\ \textit{Halt}, \textit{Push}\ 0@\bot, \textit{Return} \end{array}\right]$$

pc	m	s	$i(pc)$
$0@\bot$	$[0@\bot]$	$[]$	*Push* $1@\bot$
$1@\bot$	$[0@\bot]$	$[1@\bot]$	*Push* $\frac{6}{7}@\top$
$2@\bot$	$[0@\bot]$	$\left[\frac{6}{7}@\top, 1@\bot\right]$	*Call* $1\ 1$
Machine 1 continues…			
$6@\top$	$[0@\bot]$	$[1@\bot, \mathsf{R}(3,1)@\bot]$	*Push* $0@\bot$
$7@\top$	$[0@\bot]$	$[0@\bot, 1@\bot, \mathsf{R}(3,1)@\bot]$	*Return*
$3@\bot$	$[0@\bot]$	$[0@\bot]$	*Push* $0@\bot$
$4@\bot$	$[0@\bot]$	$[0@\bot, 0@\bot]$	*Store*
$5@\bot$	$[0@\bot]$	$[]$	*Halt*
Machine 2 continues…			
$7@\top$	$[0@\bot]$	$[1@\bot, \mathsf{R}(3,1)@\bot]$	*Return*
$3@\bot$	$[0@\bot]$	$[1@\bot]$	*Push* $0@\bot$
$4@\bot$	$[0@\bot]$	$[0@\bot, 1@\bot]$	*Store*
$5@\bot$	$[1@\bot]$	$[]$	*Halt*

Figure 36. Counterexample to RETURN*A

$$\frac{i(pc) = \textit{Return}\ n' \quad n' \in \{0,1\} \quad k \geq n'}{\boxed{pc \mid v_1 : \ldots : v_k : \mathsf{R}(x@L_x) : s \mid m} \Rightarrow \boxed{x@L_x \mid v_1@\mathcal{L}_{pc} : \ldots : v_{n'}@\mathcal{L}_{pc} : s \mid m}} \quad (\text{RETURN*B})$$

C.14 POP* (Counterexample in Figure 38)

$$\frac{i(pc) = \textit{Pop}}{\boxed{pc \mid e : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m}} \quad (\text{POP*})$$

$$i = \left[\begin{array}{l} \textit{Push } 0@\bot, \textit{Push } \frac{6}{10}@\top, \textit{Call } 0, \textit{Push } 8@\bot, \textit{Call } 1, \\ \textit{Halt}, \textit{Push } 0@\bot, \textit{Return } 1, \textit{Push } 0@\bot, \textit{Store}, \\ \textit{Return } 0 \end{array}\right]$$

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

Figure 37. Counterexample to CALL*B and RETURN*B

$$i = \begin{bmatrix} \textit{Push } \frac{5}{13}@\top, \textit{Push } 1@\bot, \textit{Store}, \textit{Push } 15@\bot, \textit{Jump}, \\ \textit{Push } 0@\bot, \textit{Return}, \textit{Push } 0@\bot, \textit{Push } 1@\bot, \textit{Load}, \\ \textit{Call } 0\ 1, \textit{Push } 0@\bot, \textit{Store}, \textit{Pop}, \textit{Return}, \\ \textit{Push } 7@\bot, \textit{Call } 0\ 0, \textit{Halt} \end{bmatrix}$$

pc	m	s	$i(pc)$
$0@\bot$	$[0@\bot, 0@\bot]$	$[]$	*Push* $\frac{5}{13}@\top$
$1@\bot$	$[0@\bot, 0@\bot]$	$\left[\frac{5}{13}@\top\right]$	*Push* $1@\bot$
$2@\bot$	$[0@\bot, 0@\bot]$	$\left[1@\bot, \frac{5}{13}@\top\right]$	*Store*
$3@\bot$	$\left[0@\bot, \frac{5}{13}@\top\right]$	$[]$	*Push* $15@\bot$
$4@\bot$	$\left[0@\bot, \frac{5}{13}@\top\right]$	$[15@\bot]$	*Jump*
$15@\bot$	$\left[0@\bot, \frac{5}{13}@\top\right]$	$[]$	*Push* $7@\bot$
$16@\bot$	$\left[0@\bot, \frac{5}{13}@\top\right]$	$[7@\bot]$	*Call* $0\ 0$
$7@\bot$	$\left[0@\bot, \frac{5}{13}@\top\right]$	$[\mathsf{R}(17,0)@\bot]$	*Push* $0@\bot$
$8@\bot$	$\left[0@\bot, \frac{5}{13}@\top\right]$	$[0@\bot, \mathsf{R}(17,0)@\bot]$	*Push* $1@\bot$
$9@\bot$	$\left[0@\bot, \frac{5}{13}@\top\right]$	$[1@\bot, 0@\bot, \mathsf{R}(17,0)@\bot]$	*Load*
$10@\bot$	$\left[0@\bot, \frac{5}{13}@\top\right]$	$\left[\frac{5}{13}@\top, 0@\bot, \mathsf{R}(17,0)@\bot\right]$	*Call* $0\ 1$
Machine 1 continues…			
$5@\top$	$[0@\bot, 5@\top]$	$[\mathsf{R}(11,1)@\bot, 0@\bot, \mathsf{R}(17,0)@\bot]$	*Push* $0@\bot$
$6@\top$	$[0@\bot, 5@\top]$	$[0@\bot, \mathsf{R}(11,1)@\bot, 0@\bot, \mathsf{R}(17,0)@\bot]$	*Return*
$11@\bot$	$[0@\bot, 5@\top]$	$[0@\top, 0@\bot, \mathsf{R}(17,0)@\bot]$	*Push* $0@\bot$
$12@\bot$	$[0@\bot, 5@\top]$	$[0@\bot, 0@\top, 0@\bot, \mathsf{R}(17,0)@\bot]$	*Store*
$13@\bot$	$[0@\top, 5@\top]$	$[0@\bot, \mathsf{R}(17,0)@\bot]$	*Pop*
$14@\bot$	$[0@\top, 5@\top]$	$[\mathsf{R}(17,0)@\bot]$	*Return*
$17@\bot$	$[0@\top, 5@\top]$	$[]$	*Halt*
Machine 2 continues…			
$13@\top$	$[0@\bot, 13@\top]$	$[\mathsf{R}(11,1)@\bot, 0@\bot, \mathsf{R}(17,0)@\bot]$	*Pop*
$14@\top$	$[0@\bot, 13@\top]$	$[0@\bot, \mathsf{R}(17,0)@\bot]$	*Return*
$17@\bot$	$[0@\bot, 13@\top]$	$[]$	*Halt*

Figure 38. Counterexample to POP*