Testing Noninterference, Quickly

Cătălin Hritcu\textsuperscript{1}  John Hughes\textsuperscript{2}  Benjamin C. Pierce\textsuperscript{1}  Antal Spector-Zabusky\textsuperscript{1}  Dimitrios Vytiniotis\textsuperscript{3}  Arthur Azevedo de Amorim\textsuperscript{1}  Leonidas Lampropoulos\textsuperscript{1}

\textsuperscript{1}University of Pennsylvania  \textsuperscript{2}Chalmers University  \textsuperscript{3}Microsoft Research

Abstract

Information-flow control mechanisms are difficult to design and labor intensive to prove correct. To reduce the time wasted on proof attempts doomed to fail due to broken definitions, we advocate modern random testing techniques for finding counterexamples during the design process. We show how to use QuickCheck, a property-based random-testing tool, to guide the design of 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.

Categories and Subject Descriptors D.2.5 [Testing and Debugging]: Testing tools (e.g., data generators, coverage testing); D.4.6 [Security and Protection]: Information flow controls

General Terms Security, Languages, Design

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 noninterference 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\textsuperscript{[26]} 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 $j$\textsuperscript{th} element of $l$ and $l[j \mapsto x]$ produces the list that is like $l$ except that the $j$\textsuperscript{th} element is replaced by $x$.

2.1 Bare machine

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

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

The $x$ argument to \text{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 \(pc \vdash s \vdash m \vdash i\). Often, \(i\) will be fixed in some context and we will write just \(pc \vdash s \vdash m\) 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:

\[
\begin{align*}
i(pc) &= \text{Noop} \quad (\text{BARE-NOOP}) \\
i(pc) &= \text{Push } x \quad (\text{BARE-PUSH}) \\
i(pc) &= \text{Pop} \quad (\text{BARE-POP}) \\
i(pc) &= \text{Load} \quad (\text{BARE-LOAD}) \\
i(pc) &= \text{Store} \quad (\text{BARE-STORE}) \\
i(pc) &= \text{Add} \quad (\text{BARE-ADD})
\end{align*}
\]

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 \(i(pc) = \text{Halt}\) and failed in all other cases (e.g., if the machine is trying to execute an \text{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 \([1, 28, 42, 44]\) 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 \(x \ll L\), where \(x\) is an integer and \(L\) is a label:

\[
L ::= \bot \mid \top \mid x
\]

We read \(\bot\) as “low” (public) and \(\top\) as “high” (secret). We order labels by \(\bot \subseteq \top\) 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 \(L_v\) for \(v\)’s label part and \(v@L\) for the value obtained by joining \(L\) to \(L_v\) 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 \text{Push} becomes a value:

\[
\text{Instr ::= Push } v \mid \text{Pop} \mid \text{Load} \mid \text{Store} \mid \text{Add} \mid \text{Noop} \mid \text{Halt}
\]

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, \(\text{Init}\), contains states of the form \(0 \ll \begin{bmatrix} 1 & m_0 \vdash i \end{bmatrix}\), where \(m_0\) can be of any length and contains only \(0\)\ll\bot. We use \(\text{Halted}\) to denote the set of halted states of the machine.

### 2.3 Noninterference (EENI)

We define what it means for this basic IFC machine to be “secure” using a standard notion of termination-insensitive noninterference \([1, 28, 41]\); we call it \textit{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 \(\bot\) in a halted state. More precisely, EENI states that for any two executions starting from initial states that are \textit{indistinguishable to a low observer} (or just \textit{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 \(\bot\).

To make this formal, we define an equivalence relation on states compositionally from equivalence relations over their components.

#### 2.3.1 Definition:

- Two values \(x_1 \ll L_1\) and \(x_2 \ll L_2\) are said to be \textit{indistinguishable}, written \(x_1 \approx L_1 \approx x_2 \ll L_2\), if either \(L_1 \equiv L_2 \equiv \top\) 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 \(i_1 = \text{Push } v_1\) and \(i_2 = \text{Push } v_2\) and \(v_1 \approx v_2\).
- Two lists (memories, stacks, or instruction memories) \(I_1\) and \(I_2\) 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 \(S_1 = \begin{bmatrix} pc_1 \vdash s_1 \vdash m_1 \vdash i_1 \end{bmatrix}\) and \(S_2 = \begin{bmatrix} pc_2 \vdash s_2 \vdash m_2 \vdash i_2 \end{bmatrix}\) are \textit{indistinguishable with respect to memories}, written \(S_1 \approx \text{mem} \equiv S_2\), if \(m_1 \approx m_2\) and \(i_1 \approx i_2\).

#### 2.3.3 Definition: A machine semantics is \textit{end-to-end noninterfering} with respect to some sets of states \(\text{Start}\) and \(\text{End}\) and an indistinguishability relation \(\approx\), written \(\text{EENI}_{\text{Start}, \text{End}, \approx}\), if for any \(S_1, S_2 \in \text{Start}\) and \(H_1, H_2 \in \text{End}\) 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 \(\text{EENI}_{\text{Init}, \text{Halted}, \approx_{\text{mem}}}\) 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.

\[
i(pc) = \text{Noop} \quad (\text{NOOP})
\]

\[
i(pc) = \text{Push } v \quad (\text{PUSH})
\]
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 arguments to EENI fails: in the initial state, the two programs are differing only in the constant argument of the second Push. The first program pushes 0 ⊤ , while the second pushes 1 ⊥ , 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 ⊥ , 0 ⊥ ]; the stack starts out empty; and the next instruction to be executed (i(pc)) is Push 1 ⊥ . On the next step, this value has been pushed on the stack and the next instruction is either Push 0 ⊤ or Push 1 ⊤ ; one or the other of these values is pushed on the stack. On the next, we Store the second stack element (1 ⊥ ) into the location pointed to by the first (either 0 ⊤ or 1 ⊤ ), so that now the memory contains 1 ⊥ in either location 0 or location 1 (the other location remains unchanged, and contains 0 ⊥ ). 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 ⊤ ), but in the final states the memories contain different values at the same location, both of which are labeled ⊥ .

Thinking about this counterexample, it soon becomes apparent what went wrong with the Store instruction: since pointers labeled ⊤ 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 Lx = ⊥ 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:

\[
i(pc) = \text{Store} \quad (\text{STORE*})
\]
\[
\begin{align*}
\text{pc} &: x @ L_x : y @ L_y : s : m \\
\Rightarrow &: pc + 1 : m(x \mapsto y @ (L_x \lor L_y))
\end{align*}
\]

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.3 Since the STORE*AB 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 ⊥ to ⊤ on a different memory location in each run. To avoid this issue, we adopt the “no sensitive upgrades” rule [1, 48], 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.

\[
i(pc) = \text{Store} \quad \text{L}_x \subseteq \text{L}_{m(x)} \quad (\text{STORE})
\]
\[
\begin{align*}
\text{pc} &: x @ L_x : y @ L_y : s : m \\
\Rightarrow &: pc + 1 : m(x \mapsto y @ (L_x \lor L_y))
\end{align*}
\]

The next counterexample found by QuickCheck (Figure 3) points out a straightforward problem in the ADD* rule: adding 0 ⊥ to 0 ⊤ yields 0 ⊥ . 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:

\[
i(pc) = \text{Add} \quad (\text{ADD})
\]
\[
\begin{align*}
\text{pc} &: x @ L_x : y @ L_y : s : m \\
\Rightarrow &: pc + 1 : m(x \mapsto y @ (L_x \lor L_y))
\end{align*}
\]

The final counterexample found by QuickCheck (Figure 4) alerts us to the fact that the ADD* rule contains a similar mistake to the original STORE*AB rule: loading a low value through a high

\[\text{i} = \text{Push 1 @ ⊥, Push 0 @ ⊤, Store, Halt}\]

\[\begin{array}{ccc}
\text{pc} & \text{m} & \text{s} \\
0 & [0 @ ⊥, 0 @ ⊥] & [ ] \\
1 & [0 @ ⊥, 0 @ ⊥] & [1 ⊥] \\
2 & [0 @ ⊥, 0 @ ⊥] & [0 ⊤ @ ⊤, 1 ⊥] \\
3 & [1 ⊤ @ ⊤, 0 @ ⊥] & [ ] \\
\end{array}\]

\[\text{Halt}\]

\[\text{i} = \text{Push 0 @ ⊥, Push 0 @ ⊤, Store, Halt}\]

\[\begin{array}{ccc}
\text{pc} & \text{m} & \text{s} \\
0 & [0 @ ⊥, 0 @ ⊥] & [ ] \\
1 & [0 @ ⊥, 0 @ ⊥] & [0 @ ⊥] \\
2 & [0 @ ⊥, 0 @ ⊥] & [0 ⊤ @ ⊤, 0 @ ⊥] \\
3 & [0 ⊤ @ ⊤, 0 @ ⊥] & [ ] \\
\end{array}\]

\[\text{Halt}\]
the use of QuickCheck is unable to find any further counterexamples. (As
2.5 More bugs
the same change to LOAD* through that pointer; and finally stores 0@⊥ to one or the other of those cells onto the stack; loads (different, two different low values. It then pushes a high address pointing little work at the beginning to set up a memory state containing pointer should taint the loaded value. The program in Figure 4 is the memory gave us additional opportunities for bugs), this change time (unlike the case of STORE*, for example, we needed to define cus-
Figure 3. Counterexample to ADD*
Figure 4. Counterexample to LOAD*
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@⊥ 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,
i(pc) = Load
\[
\frac{pc \ x@Lx \ : \ s \ m}{pc+1 \ m(x)@Lx \ : \ s \ m}
\] (LOAD)
and QuickCheck is unable to find any further counterexamples. (As defined above, the use of ⊥ on a value that already has a label joins the new label to the existing one.)

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

\[
i(pc) = Push \ x@L
\]
\[
\frac{pc \ s \ m}{pc+1 \ x@\bot \ : \ s \ m}
\] (PUSH*)

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:

\[
i(pc) = Store \ Lx \subseteq L_{m(x)}
\]
\[
\frac{pc \ x@Lx \ : \ v \ : \ s \ m}{pc+1 \ s \ m(x \to v)}
\] (STORE*A)

3. QuickCheck
We test noninterference using QuickCheck [18], 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 cus-
martly 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 that is locally minimal: none of its shrinking candidates fails. This failing case is then reported to the user. It is often very much smaller than the original randomly generated test case, and it is thus easy to use it to diagnose the failure because it (hopefully) contains no irrelevant details. Just like generation strategies, shrinking strategies are type dependent; they are defined by QuickCheck for standard types, and by the user for other types. We discuss the custom shrinking strategies we use for machine states in §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: \( \forall S_1, S_2 \in \text{Start}, S_1 \approx S_2 \implies \ldots \). 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 randomly using one of the techniques described later in this section. The second is obtained by randomly varying the “high parts” of the first. We refer to the second state as the variation of the first. The resulting pair thus satisfies indistinguishability by construction. Note that 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 \( \times \) 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 and measure the time spent on average until that bug is found (mean). The first column of Figure 5 shows how this strategy performs on the bugs from \( \S \) 2, or MTTF. Tests were run one at a time on seven identical machines, each with 4 \( \times \) 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 and measure the time spent on average until that bug is found (mean).

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., \texttt{Noop}, \texttt{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 this strategy performs on the bugs from \( \S \) 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?

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.

<table>
<thead>
<tr>
<th>Generation strategy</th>
<th>Naive</th>
<th>Weighted</th>
<th>Sequence</th>
<th>Sequence</th>
<th>ByExec</th>
<th>Average number of execution steps: 0.47</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smart integers?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>74% stack underflow</td>
</tr>
<tr>
<td>ADD#</td>
<td>83247.01</td>
<td>5344.26</td>
<td>561.58</td>
<td>30.05</td>
<td>0.87</td>
<td>21% halt</td>
</tr>
<tr>
<td>PUSH#</td>
<td>3552.54</td>
<td>309.20</td>
<td>0.21</td>
<td>0.07</td>
<td>0.01</td>
<td>4% load or store out of range</td>
</tr>
<tr>
<td>LOAD#</td>
<td>—</td>
<td>—</td>
<td>73115.63</td>
<td>2258.93</td>
<td>4.03</td>
<td></td>
</tr>
<tr>
<td>STORE#A</td>
<td>—</td>
<td>—</td>
<td>38036.22</td>
<td>32227.10</td>
<td>1233.51</td>
<td></td>
</tr>
<tr>
<td>STORE#B</td>
<td>47365.97</td>
<td>1713.72</td>
<td>0.85</td>
<td>0.12</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>STORE#C</td>
<td>7660.07</td>
<td>426.11</td>
<td>0.41</td>
<td>0.31</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td><strong>MTTF arithmetic mean</strong></td>
<td>—</td>
<td>—</td>
<td>18619.15</td>
<td>5752.76</td>
<td>206.45</td>
<td></td>
</tr>
<tr>
<td><strong>MTTF geometric mean</strong></td>
<td>—</td>
<td>—</td>
<td>69.73</td>
<td>13.33</td>
<td>0.77</td>
<td></td>
</tr>
<tr>
<td>Average tests / second</td>
<td>24129</td>
<td>11466</td>
<td>8541</td>
<td>7915</td>
<td>3284</td>
<td></td>
</tr>
<tr>
<td>Average discard rate</td>
<td>79%</td>
<td>62%</td>
<td>65%</td>
<td>59%</td>
<td>4%</td>
<td></td>
</tr>
</tbody>
</table>

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 \texttt{Store} instruction to put bad values into the memory and a \texttt{Halt} so that the run terminates, plus whatever other instructions are needed to produce the bad values). 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 \texttt{Push}, \texttt{Halt}, or \texttt{Noop}, we will fail immediately. Instead of a uniform distribution on instructions, we can do better by increasing the weights of \texttt{Push} and \texttt{Halt}\textemdash \texttt{Push} to reduce the number of stack underflows, and \texttt{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 \texttt{LOAD}\# and \texttt{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 \texttt{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 \textit{sequences} of instructions that make sense together. For instance, instead of generating single \texttt{Store} instructions, we can additionally generate sequences of the form \texttt{[Push a, Store a]} (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 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 on the MTTF. The average run length is now 4.23 steps, and the percentage of address-out-of-range errors has halved.

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

\[
\text{Jump} \quad \text{pc} \leftarrow \text{pc} + \text{instruction length} \quad (\text{Jump}^\text{AB})
\]

(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 [41]. 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 \( pc = 0 \) and, after a jump to a secret address the label of the pc becomes \( \top \).

\[
\text{Jump} \quad \text{pc} \leftarrow \text{pc} + \text{instruction length} \quad (\text{Jump}^\text{AB})
\]

While the pc is high, the two machines may be executing different instructions, and so we cannot expect the machine states to corre-
5.1.1 Definition: Machine states $S_1 = \{pc, s, m, i\}$ and $S_2 = \{pc, s, m, i\}$ are indistinguishable with respect to memories, written $S_1 \approx_{mem} S_2$, if either $L_{pc_1} = L_{pc_2} = \top$ or else $L_{pc_1} = L_{pc_2} = \bot$ 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.

\[
i(pc) = \begin{cases} \text{Call} & n = L \lor L_{pc} \\ x \in L & v_1 : \ldots : v_n : s \rightarrow m \Rightarrow x \in L_{pc} \lor L \\ x \in L_{pc} & v_1 : \ldots : v_n : R(x_{pc} + 1 @ L_{pc}) : s \rightarrow m \Rightarrow (\text{RETURN}^*) \end{cases}
\]

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.

With this rule for jumps QuickCheck no longer finds any counterexamples. Some readers may find this odd: In order to fully address implicit flow, it is usually necessary to modify the rules for memory stores to handle the case where the pc is labeled high [1, 40]. The current machine doesn’t require this, but the reason is subtle: here, 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 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).

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 $R(x_{pc} @ L_1) \approx R(x_{pc} @ L_2)$ if either $L_1 = L_2 = \top$ or else $x_{pc} = x_{pc}$ and $L_1 = L_2 = \bot$ (this is the same as for values).

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 $x @ L_{pc}$ followed by $n$ values on the stack. It sets the pc to $x$, labels this new pc by the join of $L_{pc}$ 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.

\[
i(pc) = \begin{cases} \text{Call} & n = L \lor L_{pc} \\ x \in L & v_1 : \ldots : v_n : s \rightarrow m \Rightarrow x \in L_{pc} \lor L \\ x \in L_{pc} & v_1 : \ldots : v_n : R(x_{pc} + 1 @ L_{pc}) : s \rightarrow m \Rightarrow (\text{RETURN}^*) \end{cases}
\]

\[
i(pc) = \begin{cases} \text{Call} & n = L \lor L_{pc} \\ x \in L & v_1 : \ldots : v_n : s \rightarrow m \Rightarrow x \in L_{pc} \lor L \\ x \in L_{pc} & v_1 : \ldots : v_n : R(x_{pc} + 1 @ L_{pc}) : s \rightarrow m \Rightarrow (\text{RETURN}^*) \end{cases}
\]

$2$ As long as we don’t have exceptions! See [28] 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 §A.3).
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 §A.2), we need to change the EENI instance we use to EENIInit.Halted/Low, × mem, where Low denotes the set of states with pc = ⊥. Thus, we only consider end states 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 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, 48]. This leads to the correct rule for stores:

\[ i = \begin{cases} \text{Push } 3 @ T, \text{Call } 0, \text{Halt, Push } 0 @ \perp, \text{Store, Return } 0 & \text{Machine 1 continues...} \\ \text{Machine 2 continues...} & \text{Machine 2 continues...} \\ & \text{Call } 0 \end{cases} \]

\[ i = \begin{cases} \text{Push } 0 @ \perp, \text{Halt, Push } 0 @ \perp, \text{Return } 1 & \text{Machine 1 continues...} \\ \text{Machine 2 continues...} & \text{Machine 2 continues...} \\ \text{Halt} & \text{Halt} \end{cases} \]

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

\[ i = \begin{cases} \text{Push } 0 @ \perp, \text{Halt, Push } 0 @ \perp, \text{Store, Return } 0 & \text{Machine 1 continues...} \\ \text{Machine 2 continues...} & \text{Machine 2 continues...} \\ \text{Return } 0 & \text{Return } 0 \end{cases} \]

\[ i = \begin{cases} \text{Push } 0 @ \perp, \text{Halt, Push } 0 @ \perp, \text{Store, Return } 0 & \text{Machine 1 continues...} \\ \text{Machine 2 continues...} & \text{Machine 2 continues...} \\ \text{Halt} & \text{Halt} \end{cases} \]

Figure 14.

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 treat all the returned values with the pre-return pc label.

\[ i(pc) = \text{Return } n' \text{ } n' \in \{0, 1\} \text{ } k \geq n' \text{ (RETURN*B)} \]

\[ \begin{align*} \text{pc} & = x @ L_x : y @ L_y : x & \Rightarrow y @ (L_x \cup L_y \cup \mathcal{L}_{pc}) \\ x & = L_x : y @ L_y : x & \Rightarrow y @ (L_x \cup L_y \cup \mathcal{L}_{pc}) \end{align*} \]

The next counterexample, listed in Figure 16, shows (maybe somewhat surprisingly) that it is unsound to specify the number of results to return in the Return instruction, because 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:

\[ i(pc) = \text{Call } n \text{ } n' \in \{0, 1\} \text{ } L = L_x \cup L_{pc} \text{ (CALL)} \]

\[ \begin{align*} x @ L & = x @ L_x : v_1 : \ldots : v_k : x & \Rightarrow \mathcal{L}_{pc} : v_1 \times \ldots \times v_k @ L_{pc} : x & \Rightarrow \mathcal{L}_{pc} : v_1 \times \ldots \times v_k @ L_{pc} : x & \Rightarrow \mathcal{L}_{pc} \end{align*} \]

\[ i(pc) = \text{Return } k \geq n' \text{ (RETURN)} \]

\[ \begin{align*} \text{pc} & = x @ L : v_1 : \ldots : v_k : x & \Rightarrow \mathcal{L}_{pc} : x & \Rightarrow \mathcal{L}_{pc} \end{align*} \]

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):

\[ i(pc) = \text{Pop } e @ s : m & \Rightarrow pc + 1 @ s : m \text{ (Pop*)} \]
\[ i = \begin{cases} \text{Push } @\bot, \text{Push } \frac{6}{7} @\top, \text{Call } 0, \text{Push } 0@\bot, \text{Store,} \\ \text{Halt, Return } 0, \text{Push } 0@\bot, \text{Return } 1 \end{cases} \]

<table>
<thead>
<tr>
<th>pc</th>
<th>m</th>
<th>s</th>
<th>(i(pc))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0@⊥</td>
<td>0@⊥</td>
<td>{}</td>
<td>Push 0@⊥</td>
</tr>
<tr>
<td>1@⊥</td>
<td>0@⊥</td>
<td>0@⊥</td>
<td>Push \frac{6}{7} @\top</td>
</tr>
<tr>
<td>2@⊥</td>
<td>0@⊥</td>
<td>{}</td>
<td>Call 0</td>
</tr>
</tbody>
</table>

Machine 1 continues...

\(6@\top \{0@\bot\} \{R(3@\bot), 0@\bot\} \text{ Return } 0\)
\(3@\bot \{0@\bot\} \{0@\bot\} \text{ Push } 0@\bot\)
\(4@\bot \{0@\bot\} \{0@\bot, 0@\bot\} \text{ Store} \)
\(5@\bot \{0@\bot\} \{\} \text{ Halt} \)

Machine 2 continues...

\(7@\top \{0@\bot\} \{R(3@\bot), 0@\bot\} \text{ Push } 0@\bot\)
\(8@\top \{0@\bot\} \{0@\bot, R(3@\bot), 0@\bot\} \text{ Return } 1\)
\(3@\bot \{0@\bot\} \{0@\bot, 0@\bot\} \text{ Push } 0@\bot\)
\(4@\bot \{0@\bot\} \{0@\bot, 0@\bot, 0@\bot\} \text{ Store} \)
\(5@\bot \{0@\bot\} \{0@\bot\} \text{ Halt} \)

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

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).

\[ i(pc) = \text{Pop} \]

\(pc \leftarrow v : s \leftarrow m \Rightarrow pc + 1 \leftarrow s \leftarrow m\) (POP)

5.3 Generation by execution and control flow

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

Since it now becomes possible to generate instruction streams that cause the machine to crash in some number of steps, one might be worried about the discard ratio for EENI. However, the ever increasing probability of generating a Halt (discussed in §4) counterbalances this issue.

5.4 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.

Alternative generation strategies

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

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

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

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

6. Strengthening the Tested Property

The last few counterexamples in §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 non-interference 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:

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.
\[
S_1 = \left[ \begin{array}{ccc}
p & m & s \\
0@\bot & 0@\bot & [ ] \\
1@\bot & 0@\bot & 5@\bot \\
5@\bot & 0@\bot & \{R(2,1)@\bot\} \\
6@\bot & 0@\bot & [0@\bot, R(2,1)@\bot] \\
7@\bot & 0@\bot & \{\frac{5}{9} \oplus T, 0@\bot, R(2,1)@\bot\}
\end{array} \right]
\]

\[
S_2 = \left[ \begin{array}{ccc}
p & m & s \\
0@\bot & 0@\bot & [ ] \\
1@\bot & 0@\bot & 5@\bot \\
5@\bot & 0@\bot & \{R(2,1)@\bot\} \\
6@\bot & 0@\bot & [0@\bot, R(2,1)@\bot] \\
7@\bot & 0@\bot & \{\frac{5}{9} \oplus T, 0@\bot, R(2,1)@\bot\}
\end{array} \right]
\]

Machine 1 continues...

8@T [0@\bot] \{R(8,0)@\bot, 0@\bot, R(2,1)@\bot\] Pop 0@\bot

9@T [0@\bot] \{0@\bot, R(2,1)@\bot\] Push 0@\bot

10@T [0@\bot] \{0@\bot, 0@\bot, R(2,1)@\bot\] Return

2@\bot [0@\bot] \{0@\bot\] Push 0@\bot

3@\bot [0@\bot] \{0@\bot, R(2,1)@\bot\] Store

4@\bot [0@\bot] \{\} Halt

Machine 2 continues...

9@T [0@\bot] \{R(8,0)@\bot, 0@\bot, R(2,1)@\bot\] Push 0@\bot

10@T [0@\bot] \{0@\bot, R(8,0)@\bot, 0@\bot, R(2,1)@\bot\] Return

8@\bot [0@\bot] \{0@\bot, R(2,1)@\bot\} Pop

9@\bot [0@\bot] \{R(2,1)@\bot\] Push 0@\bot

10@\bot [0@\bot] \{0@\bot, R(2,1)@\bot\] Return

2@\bot [0@\bot] \{0@\bot\] Push 0@\bot

3@\bot [0@\bot] \{0@\bot, R(2,1)@\bot\] Store

4@\bot [0@\bot] \{\} Halt

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

<table>
<thead>
<tr>
<th>Tested property</th>
<th>EENI</th>
<th>EENI</th>
<th>EENI</th>
<th>LLNI</th>
<th>SSNI</th>
<th>SSNI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starting states</td>
<td>Init</td>
<td>Init</td>
<td>QInit</td>
<td>QInit</td>
<td>All</td>
<td>All</td>
</tr>
<tr>
<td>Equivalence relation</td>
<td>$\approx_{\text{mem}}$</td>
<td>$\approx_{\text{low}}$</td>
<td>$\approx_{\text{low}}$</td>
<td>$\approx_{\text{low}}$</td>
<td>$\approx_{\text{full}}$</td>
<td>$\approx_{\text{full}}$</td>
</tr>
<tr>
<td>Generation strategy</td>
<td>BYEXEC2</td>
<td>BYEXEC2</td>
<td>BYEXEC2</td>
<td>BYEXEC2</td>
<td>NAIVE</td>
<td>TINYSSNI</td>
</tr>
<tr>
<td>ADD*</td>
<td>37.07</td>
<td>2.38</td>
<td>1.38</td>
<td>0.36</td>
<td>0.24</td>
<td>0.11</td>
</tr>
<tr>
<td>PUSH*</td>
<td>0.22</td>
<td>0.02</td>
<td>0.02</td>
<td>0.01</td>
<td>1.06</td>
<td>0.06</td>
</tr>
<tr>
<td>LOAD*</td>
<td>155.07</td>
<td>37.50</td>
<td>5.73</td>
<td>1.14</td>
<td>3.25</td>
<td>0.61</td>
</tr>
<tr>
<td>STORE*A</td>
<td>20018.67</td>
<td>18658.56</td>
<td>124.78</td>
<td>84.08</td>
<td>289.63</td>
<td>16.32</td>
</tr>
<tr>
<td>STORE*B</td>
<td>13.02</td>
<td>12.87</td>
<td>16.10</td>
<td>5.25</td>
<td>31.11</td>
<td>0.33</td>
</tr>
<tr>
<td>STORE*C</td>
<td>0.35</td>
<td>0.34</td>
<td>0.33</td>
<td>0.08</td>
<td>0.73</td>
<td>0.03</td>
</tr>
<tr>
<td>JUMP*A</td>
<td>48.84</td>
<td>7.58</td>
<td>5.26</td>
<td>0.08</td>
<td>1.45</td>
<td>0.09</td>
</tr>
<tr>
<td>JUMP*B</td>
<td>2421.99</td>
<td>158.36</td>
<td>104.62</td>
<td>2.80</td>
<td>16.88</td>
<td>0.49</td>
</tr>
<tr>
<td>STORE*D</td>
<td>13289.39</td>
<td>12295.65</td>
<td>873.79</td>
<td>232.19</td>
<td>87.77</td>
<td>1.13</td>
</tr>
<tr>
<td>STORE*E</td>
<td>1047.56</td>
<td>1129.48</td>
<td>717.72</td>
<td>177.75</td>
<td>22.26</td>
<td>0.29</td>
</tr>
<tr>
<td>CALL*A</td>
<td>3919.08</td>
<td>174.66</td>
<td>115.15</td>
<td>5.97</td>
<td>31.71</td>
<td>0.62</td>
</tr>
<tr>
<td>RETURN*A</td>
<td>12804.51</td>
<td>4698.17</td>
<td>1490.80</td>
<td>337.4</td>
<td>1110.09</td>
<td>3.10</td>
</tr>
<tr>
<td>CALL<em>B+RETURN</em>B</td>
<td>69081.50</td>
<td>6940.67</td>
<td>1811.66</td>
<td>396.37</td>
<td>1194.30</td>
<td>4.56</td>
</tr>
<tr>
<td>POP*</td>
<td>—</td>
<td>51753.13</td>
<td>16107.22</td>
<td>1828.56</td>
<td>30.68</td>
<td>0.42</td>
</tr>
<tr>
<td><strong>MTTF arithmetic mean</strong></td>
<td>—</td>
<td>6847.81</td>
<td>1526.75</td>
<td>219.46</td>
<td>194.44</td>
<td>2.01</td>
</tr>
<tr>
<td><strong>MTTF geometric mean</strong></td>
<td>—</td>
<td>135.76</td>
<td>46.48</td>
<td>7.69</td>
<td>12.87</td>
<td>0.47</td>
</tr>
<tr>
<td>Average tests / second</td>
<td>2795</td>
<td>2797</td>
<td>2391</td>
<td>1224</td>
<td>8490</td>
<td>18407</td>
</tr>
<tr>
<td>Average discard rate</td>
<td>65%</td>
<td>65%</td>
<td>69%</td>
<td>0%</td>
<td>40%</td>
<td>9%</td>
</tr>
</tbody>
</table>

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

6.1.1 Definition: Machine states $S_1 = [pc_1, s_1, m_1, t_1]$ and $S_2 = [pc_2, s_2, m_2, t_2]$ are indistinguishable with respect to entire low states, written $S_1 \approx_{\text{low}} S_2$, if either $L_{pc_1} = L_{pc_2} = T$ or else $L_{pc_1} = L_{pc_2} = \bot$, $m_1 \approx m_2$, $t_1 \approx t_2$, $s_1 \approx s_2$, and $pc_1 \approx pc_2$.

We now strengthen EENI.Init.Halted\LLNI,approx_{\text{low}},approx_{\text{mem}}, the property we have been testing, to EENI.Init.Halted\LLNI,approx_{\text{low}},approx_{\text{mem}}, this is stronger because $approx_{\text{low}}$ agree on initial states, while for halted states $approx_{\text{low}} \subset approx_{\text{mem}}$. 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 \( \text{QInit} \) of quasi-initial states contains all states of the form \( \{0 \wedge \perp \mid s, m, i \} \), 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 \( \text{Call} \) 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 that 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 \( \text{Store} \) 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 \( \text{EENI} \) with \( \text{QInit} \), we obtain a stronger property \( \text{EENI}_{\text{QInit, Halting}, \text{Low, \!≈_{low}}} \) (stronger because \( \text{Init} \subset \text{QInit} \)) that does indeed find bugs faster, as column 3 of Figure 18 shows.

6.3 LLNI: Low-lockstep noninterference

While making the full state observable and starting from quasi-initial states significantly improves \( \text{EENI} \), we can get even better results by moving to a yet stronger noninterference property. The intuition is that \( \text{EENI} \) generates machines and runs them for a long time, but it only compares the final states, and only when both machines return to low states. This means that while the program is running, the two machines may be executing different instructions. Thus, bugs that are only manifested in these hard-to-reach states may not be found. 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 that 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 \( \text{Store} \) 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 \( \text{EENI} \) with \( \text{QInit} \), we obtain a stronger property \( \text{EENI}_{\text{QInit, Halting}, \text{Low, \!≈_{low}}} \) (stronger because \( \text{Init} \subset \text{QInit} \)) that does indeed find bugs faster, as column 3 of Figure 18 shows.

The rule \( \text{LOW LOCKSTEP} \) requires low states in the two traces to be pointwise indistinguishable, while \( \text{HIGH FILTER} \) (together with \( \text{SYMMETRY} \)) 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 (\( \text{HIGH END} \)) or not halted (\( \text{LOW ERROR END} \)). Additionally, we allow the two traces to terminate simultaneously (\( \text{LOCKSTEP END} \)). We implement these rules in Haskell as a recursive predicate over lazy lists.

In general, \( \text{LLNI} \) implies \( \text{EENI} \), but not vice versa. However, the correct version of our machine does satisfy \( \text{LLNI} \), and we have not observed any cases where QuickChecking a buggy machine with \( \text{LLNI} \) finds a bug that is not also a bug with regard to \( \text{EENI} \). Testing \( \text{LLNI} \) instead of \( \text{EENI} \) leads to significant improvements in the bug detection rate for all bugs, as the results in the fourth column of Figure 18 show. In these experiments no generated machine states are discarded, since \( \text{LLNI} \) applies to both successful (halting) executions and failing or infinite executions. The generation strategies described in \( \S 4 \) also apply to \( \text{LLNI} \) without much change; also, as for \( \text{EENI} \), generation by execution (with lookahead of 2 steps) performs better than the more basic strategies, so we don’t consider those for \( \text{LLNI} \).

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 (\( \text{EENI} \)) or at every low step (\( \text{LLNI} \)). 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” \( [26] \); the corresponding conditions for our machine form a property we call \( \text{single-step noninterference (SSNI)} \).

We start by observing that \( \text{LLNI} \) 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 \( \text{Return} \) 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 \( \text{SSNI} \), 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) if the following four conditions are all satisfied:

1. For all low states \( S_1 \) and \( S_2 \), if \( S_1 \approx S_2 \), then \( S_1 \approx S_2 \);
2. For all high states \( S' \) with \( S' \Rightarrow S \), we have \( S' \approx S \);
3. For all high states \( S_1' \) and \( S_2' \), if \( S_1' \approx S_2' \), then \( S_1' \approx S_2' \);
4. For all low states \( S_1 \) and \( S_2 \), if \( S_1 \approx S_2 \) and \( S_1 \) is halted, then \( S_2 \) 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, \( \approx_{\text{full}} \) is too weak and QuickCheck can easily find counterexamples to condition 3, e.g., by choosing two indistinguishable machine states with \( i = [\text{return}] \), \( pc = 0 \oplus \top \), and \( s = R(0,0) \oplus \bot \); after a single step the two machines have distinguishable \( pc 0 \oplus \bot \) and \( 1 \oplus \bot \), respectively. On the other hand, treating high states exactly like low states in the indistinguishability relation is too strong. In this case QuickCheck finds counterexamples to condition 2, e.g., a single machine state with \( i = \text{Pop} \), \( pc = 0 \oplus \top \), and \( s = [0] \oplus \bot \) steps to a state with \( s = \bot \), which would not be considered indistinguishable. These counterexamples show that indistinguishable high states can have different \( pcs \) and can have completely different stack frames at the top of the stack.

So all we can require for two high states to be equivalent is that their memories and instruction memories agree and that the parts of the stacks below the topmost low return address are equivalent. This is strong enough to ensure condition 3.

6.4.2 Definition: Machine states \( S_1 = [pc_1, s_1, m_1, i_1] \) and \( S_2 = [pc_2, s_2, m_2, i_2] \) are indistinguishable with respect to whole machine states, written \( S_1 \approx_{\text{fsall}} S_2 \), if \( m_1 \approx m_2 \), \( i_1 \approx i_2 \), \( L_{pc_1} \approx L_{pc_2} \), and additionally

- if \( L_{pc_1} = \bot \) then \( s_1 \approx s_2 \) and \( pc_1 \approx pc_2 \), and
- if \( L_{pc_1} = \top \) then \( \text{cropStack} \ s_1 \approx \text{cropStack} \ s_2 \).

The \text{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\(_{\approx_{\text{fsall}}} \) 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 \( \approx_{\text{full}} \)) 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, Azevedo de Amorim et al. [3] did in fact prove recently in Coq that SSNI holds, and did not find any bugs that had escaped our testing. Moreover, we proved in Coq that 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*\B 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.)

We allow any instruction to shrink to \textit{Noop}, which preserves a counterexample if the instruction was unnecessary; or to \textit{Halt}, which preserves a counterexample if the bug had already manifested by the time that instruction was reached. To avoid an infinite shrinking loop, we do not shrink \textit{Noop} at all, while \textit{Halt} can shrink only to \textit{Noop}. Instructions of the form \textit{Push} \( a \) are also shrunk by shrinking \( a \). Finally, instructions of the form \textit{Call} \( a \) or \( r \) are also shrunk by shrinking \( a \) or \( r \), or by being replaced with \textit{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 which can be shrunk. 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 the three.

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. This is because removing other elements changes all subsequent memory addresses, likely invalidating the counterexample and thus rendering the shrinking step unsuccessful.

In the case of the instruction memory, we only try to remove \textit{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 \textit{Noop}. 

12

2013/7/25
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 that are also indistinguishable.

When we use the full state equivalence \( \approx_{\text{full}} \), 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 have similar lists of shrinking candidates. Smart shrinking just changes the order in which these candidates are tried: it defers the first \( k \) shrinking candidates for \( b_{k+1} \) until after more likely ones have been tried. This sped up shrinking dramatically in our tests.

We also observed that many reported counterexamples contained Noop instructions—in some cases many of them—even though we implemented Noop removal as a shrinking step. On examining these counterexamples, we discovered that they could not be shrunk because removing a Noop changes the addresses of subsequent instructions, at least one of which was the target of a Jump or Call instruction. So to preserve the behavior of the counterexample, we needed to remove the Noop instruction and adjust the target of a control transfer in the same shrinking step. Since control transfer targets are taken off the stack, and such 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 Noops.

Occasionally, we observed that shrinking counterexamples containing two or more unnecessary Noops, 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 Noops, 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 Noops 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

Generating random inputs for testing is a large research area, but the particular sub-area of testing language implementations by generating random programs is less well studied. Redex \([30–32]\) (\textit{ne} PLT Redex) is a domain-specific language for defining operational semantics within Racket (\textit{ne} PLT Scheme), which includes a property-based random testing framework inspired by QuickCheck. This framework uses a formalized language definition to automatically generate simple test-cases. To generate better test cases, however, Klein et al. find that the generation strategy needs to be tuned for the particular language; this agrees with our observation that fine-tuned strategies are required to obtain the best results. They argue that the effort required to find bugs using Redex is less than the effort required for a formal proof of correctness, and that random testing is sometimes viable in cases where full proof seems infeasible.

Klein et al. \([33]\) use PLT Redex’s QuickCheck-inspired random testing framework to assess the safety of the bytecode verification algorithm for the Racket virtual machine. They observe that naively generated programs only rarely pass bytecode verification (88% discard rate), and that many programs fail verification because of a few common violations that can be easily remedied in a post-generation pass that for instance replaces out-of-bounds indices with random in-bounds ones. These simple changes to the generator are enough for reducing the discard rate (to 42%) and for finding more than two dozen bugs in the virtual machine model, as well as a few in the Racket machine implementation, but three known bugs were missed by this naive generator. The authors conjecture that a more sophisticated test generation technique could probably find these bugs.

CSmith \([47]\) is a C compiler testing tool that generates random C programs, avoiding ones whose behavior is undefined by the C99 standard. When generating programs, CSmith does not attempt to model the current state of the machine; instead, it chooses program fragments that are correct with respect to some static safety analysis (including type-, pointer-, array-, and initializer-safety, etc.). We found that modeling the actual state of our (much simpler) machine to check that generated programs were hopefully well-formed, as in our generation by execution strategy, made our test-case generation far more effective at finding noninterference bugs. In order to get smaller counterexamples, Regehr et al. present C-Reduce \([38]\), a tool for reducing test-case C programs such as those produced by CSmith. They note that conventional shrinking methods usually introduce test cases with undefined behavior; thus, they put a great deal of effort and domain specific knowledge into shrinking well-defined programs only to programs that remain well-defined. To do this, they use a variety of search techniques to find better reduction steps and to couple smaller ones together. Our use of QuickCheck’s double shrinking is similar to their simultaneous reductions, although we observed no need in our setting for more sophisticated searching methods than the greedy one that is guaranteed to produce a local minimum. Regehr et al.’s work on reduction is partly based on Zeller and Hildebrandt’s formalization of the delta debugging algorithm \textit{ddmin} \([49]\), a non-domain-specific algorithm for simplifying and isolating failure-inducing program inputs with an extension of binary search. In our work, as in Regehr et al.’s, domain-specific knowledge is crucial for successful shrinking.

Another relevant example of testing programs by generating random input is Randoop \([36]\), which generates random sequences of calls to Java APIs. Noting that many generated sequences crash...
after only a few calls, before any interesting bugs are discovered, Randoop performs feedback directed random testing, in which previously found sequences that did not crash are randomly extended. This enables Randoop to generate tests that run much longer before crashing, which are much more effective at revealing bugs. Our generation by execution strategy is similar in spirit, and likewise results in a substantial improvement in bug detection rates.

A state-machine modeling library for (an Erlang version of) QuickCheck has been developed by Quviq [29]. It generates sequences of API calls to a stateful system satisfying preconditions formulated in terms of a model of the system state, associating a (model) state transition function with each API call. API call generators also use the model state to avoid generating calls whose preconditions cannot be satisfied. Our generation-by-execution strategy works in a similar way for straightforward code.

A powerful and widely used approach to testing is symbolic execution—in particular, concolic testing and related dynamic symbolic execution techniques [15, 35]. 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 [25] and PathCrawler [46], a variety of tools and methods have appeared; some of the state-of-the-art tools include CUTE [43], CREST [12], and KLEE [13] (which evolved from EXE [14]). We wondered whether dynamic symbolic execution could be used instead of random testing for finding noninterference bugs. As a first step, we implemented a simulator for a version of our abstract machine in C and tested it with KLEE. Using KLEE out of the box and without any expert knowledge in the area, we attempted to invalidate various assertions of noninterference. Unfortunately, we were only able to find a concrete example 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 [16], so we don’t cover enough of it to reach the particular IFC-violating configurations.

Balliu et al. [4] 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 [21] 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 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 interactive theorem provers, automatically generating counterexamples for false conjectures can prevent wasting time and effort on proof attempts doomed to fail. Dybjer [22] propose a QuickCheck-like tool for the Agda/Alfa proof assistant. Berghofer and Nipkow [8] proposed a QuickCheck-like tool for Isabelle/HOL. This was recently extended by Bulwahn [10] to also support exhaustive and narrowing-based symbolic testing [17, 34, 39]. 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 [11]. Eastlund [23] implemented DoubleCheck, an adaption of QuickCheck for ACL2. Chamari et al. [37] later proposed a more advanced counterexample finding tool for ACL2s, which uses the full power of the theorem prover and libraries to simplify conjectures so that they are easier to falsify. While all these tools are general and only require the statement of the conjecture to be in a special form (e.g., executable specification), so they could in principle be applied to test noninterference, our experience with QuickCheck suggests that for the best results one has to incorporate domain knowledge about the machine and the property being tested. We hope to compare our work against these tools in the future and provide experimental evidence for this intuition.

On the dynamic IFC side Birgisson et al. [9] 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 [48] 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 [27] 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. [9] 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 [45] and later Barthe et al. [6] 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 $\hat{P}$ with a renaming of itself. Self-composition enables the use of standard (i.e., not relational) 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 [41], relational program logics [5, 7], and self-composition based tools [6].

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. The techniques we present bring many orders of magnitude improvement in the rate at which bugs are found, and for the hardest-to-find bugs (to EENI) the minimal counterexamples are 10-15 instructions long—well beyond the scope of naive exhaustive testing. Even if we ultimately care about full security proofs [3], 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 CRASHSAFE, an ambitious hardware-software co-design effort underway at Penn, Harvard, Northeastern, and BAE Systems [19, 20], 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 ma-
chines and for checking that the code running on lower-level ab-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-

Just how well do our methods need to perform to find bugs ef-
\[ i = \begin{array}{c|c|c|c|c} \hline pc & m & s & i(pc) \\ \hline 0@⊥ & 0@⊥, 0@⊥ & 1@⊥ & Push 1@⊥ \\ 1@⊥ & 0@⊥, 0@⊥ & 1@⊥ & Push 0@⊥ \\ 2@⊥ & 0@⊥, 0@⊥ & 0@T, 1@⊥ & Push 1@⊥ \\ 3@⊥ & 0@⊥, 0@⊥ & 1@⊥, 0@T, 0@⊥ & Store \\ 4@⊥ & 0@⊥, 0@⊥ & 1@⊥ & Push 1@⊥ \\ 5@⊥ & 0@⊥, 0@T & 1@⊥, 0@⊥ & Store \\ 6@⊥ & 0@⊥, 0@T & 1@⊥, 1@⊥ & \text{Halt} \\ \hline \end{array} \]

Figure 19. A counterexample showing that it is wrong to make high atoms be equivalent to all other atoms.
different labels. As we saw earlier, such a counterexample can be extended to one in which values also differ.

\[ i = \left[ \text{Push } 2 \oplus T, \text{Call } 0 \oplus 0, \text{Halt, Return} \right] \]

<table>
<thead>
<tr>
<th>pc</th>
<th>m</th>
<th>s</th>
<th>( i(pc) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0@⊥</td>
<td>[]</td>
<td>[]</td>
<td>Push 2 ⊕ T</td>
</tr>
<tr>
<td>1@⊥</td>
<td>[ ]</td>
<td>[ ]</td>
<td>Call 0 0</td>
</tr>
</tbody>
</table>

Machine 1 continues...

2@⊥ | [ ] | [ ] | Store 0@⊥

9@T | [ ] | [ ] | Return

Machine 2 continues...

3@T | [ ] | [ ] | Return

Figure 21. A counterexample justifying the change to \( \text{EENI}_{\text{init, Halt, \text{Low}, \text{SSNI}, \text{max}}} \) in §5.2.

A.3 Indistinguishability for stack elements when adding 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 (as for values) \( R(x_1@L_1) \approx R(x_2@L_2) \) if either \( L_1 = L_2 = \top \) or \( x_1 = x_2 \) and \( L_1 = L_2 = \bot \). If instead we considered high return addresses and high values to be indistinguishable, QuickCheck would find a counterexample. This counterexample requires quasi-initial states (and \( \approx_{\text{low}} \)) and is listed in Figure 22. The first machine performs only one \( \text{Return} \) that throws away two elements from the stack and then halts. The second machine returns twice: the first time to the same \( \text{Return} \), unwinding the stack and raising the \( pc \); and the second time to the \( \text{Halt} \) instruction, labeling the return value high in the process. The final states are distinguishable because the elements on the stack have different labels. As we saw earlier, such a counterexample can be extended to one in which values also differ.

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

<table>
<thead>
<tr>
<th>pc</th>
<th>m</th>
<th>s</th>
<th>( i(pc) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0@⊥</td>
<td>[ ]</td>
<td>[ ]</td>
<td>( \text{R}(0,0@\top) )</td>
</tr>
</tbody>
</table>

Machine 1 continues...

1@⊥ | [ ] | [ ] | Halt

Machine 2 continues...

0@T | [ ] | [ ] | Return

1@T | [ ] | [ ] | Halt

Figure 22. A counterexample that motivates the indistinguishability of stack elements for the machine with calls and returns.

A.4 Counterexamples justifying indistinguishability for SSNI

The indistinguishability relation high states used for SSNI needs to be strong enough to ensure that when both machines return to low states, those low states are also indistinguishable. Since \( \approx_{\text{low}} \) 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 \( \approx_{\text{full}} \) (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 \( \text{Return} \) instruction.

B. 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 \( a \)

\[
\text{data Variation} \ a = \text{Variation} \ (\Rightarrow) \ a
\]

with an invariant that the components should be indistinguishable, plus a class of \( \text{observable types} \) for which we can generate a

\[ i = \left[ \text{Return} \right] \]

<table>
<thead>
<tr>
<th>pc</th>
<th>m</th>
<th>s</th>
<th>( i(pc) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0@⊤</td>
<td>[ ]</td>
<td>[ ]</td>
<td>( \text{R}(0,0@\top) ) Return</td>
</tr>
</tbody>
</table>

Machine 1 continues...

0@⊥ | [ ] | [ ] | Return

Machine 2 continues...

1@⊥ | [ ] | [ ] | _

Figure 23. A counterexample showing that \( \approx_{\text{low}} \) is too weak for SSNI. Since the \( pc \) is initially high, \( \approx_{\text{low}} \) 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 \( pcs \).
A counterexample that shows that treating high states exactly like low states in the indistinguishability relation over machine states is too strong and breaks condition 2 in Definition 6.4.1. When a machine steps from a high state to another high state the contents of the stack can change.

\[ i = \left[ \text{Push } 10 @ \top, \text{Push } 0 @ \top, \text{Add, Store, Halt} \right] \]

Figure 24. A counterexample to PUSH*

\[ i = \left[ \text{Push } 00 @ \top, \text{Push } 00 @ \top, \text{Store, Halt} \right] \]

Figure 26. Counterexample to PUSH*

\[ i = \left[ \text{Push } 10 @ \top, \text{Push } 00 @ \top, \text{Store, Push } 00 @ \top, \text{Load, Push } 00 @ \top, \text{Store, Halt} \right] \]

Figure 27. Counterexample to LOAD*

\[ i = \left[ \text{Push } 00 @ \top, \text{Push } 00 @ \top, \text{Store, Push } 00 @ \top, \text{Push } 00 @ \top, \text{Store, Push } 00 @ \top, \text{Store, Halt} \right] \]

Figure 28. Counterexample to STORE*A

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, EENIInit,Halted,≈_{form}, which corresponds to the first column in Figure 18.

C.1 ADD* (Counterexample in Figure 25)

\[ i(pc) = \text{Add} \]

\[ \begin{array}{c|c|c|c} pc & m & s & i(pc) \\ \hline 00 @ \top & 00 @ \top & [] & \text{Push } 10 @ \top \\ 10 @ \top & 00 @ \top & [10] & \text{Push } 00 @ \top \\ 20 @ \top & 00 @ \top & [0000 @ \top, 10] & \text{Push } 00 @ \top \\ 30 @ \top & 00 @ \top & [0000 @ \top, 00] & \text{Add} \\ 40 @ \top & 00 @ \top & [0000 @ \top, 00] & \text{Store} \\ 50 @ \top & 00 @ \top & [0000 @ \top, 00] & \text{Push } 00 @ \top \\ 60 @ \top & 00 @ \top & [0000 @ \top, 00] & \text{Store} \\ 70 @ \top & 00 @ \top & [0000 @ \top, 00] & \text{Push } 00 @ \top \\ \end{array} \]

C.2 PUSH* (Counterexample in Figure 26)

\[ i(pc) = \text{Push } x@L \]

\[ \begin{array}{c|c|c|c} pc & s & m & i(pc) \\ \hline 00 @ \top & x@L & y@L & s \Rightarrow pc \Rightarrow pc+1 \Rightarrow (x+y)@L s \Rightarrow m \end{array} \]

C.3 LOAD* (Counterexample in Figure 27)

\[ i(pc) = \text{Load} \]

\[ \begin{array}{c|c|c|c} pc & x@L & s & m \Rightarrow pc+1 \Rightarrow m(x) \Rightarrow m \end{array} \]

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

\[ i(pc) = \text{Store} \]

\[ \begin{array}{c|c|c|c} pc & x@L & y@L & s \Rightarrow m(x) \Rightarrow pc \Rightarrow pc+1 \Rightarrow s \Rightarrow m(x) \Rightarrow y@L(\ell y \cap \ell \ell_p) \end{array} \]
\[ i = \left[ \text{Push } 0@⊥, \text{Push } 1@⊥, \text{Store, Halt} \right] \]

\[ \text{pc} \quad m \quad s \quad i(\text{pc}) \]

\begin{array}{cccc}
0@⊥ & 0@⊥, 0@⊥ & \{ \} & \text{Push } 0@⊥ \\
1@⊥ & 0@⊥, 0@⊥ & \{ 0@⊥ \} & \text{Push } 1@⊥ \\
2@⊥ & 0@⊥, 0@⊥ & \{ 2@⊥ \} & \text{Store} \\
3@⊥ & 3@⊥, 0@⊥ & \{ \} & \text{Halt} \\
\end{array}

**Figure 29.** Counterexample to STORE*B

\[ i = \left[ \text{Push } ½ @ ⊤, \text{Push } 0@⊥, \text{Store, Halt} \right] \]

\[ \text{pc} \quad m \quad s \quad i(\text{pc}) \]

\begin{array}{cccc}
0@⊥ & 0@⊥ & \{ \} & \text{Push } ½ @ ⊤ \\
1@⊥ & 0@⊥ & \{ 0@⊥ \} & \text{Push } 0@⊥ \\
2@⊥ & 0@⊥ & \{ 0@⊥, 1@⊥ \} & \text{Push } ½ @ ⊤ \\
3@⊥ & 0@⊥ & \{ 3@⊥, 1@⊥ \} & \text{Jump} \\
\end{array}

**Figure 30.** Counterexample to STORE*C

\[ i = \left[ \text{Push } 1@⊥, \text{Push } 0@⊥, \text{Push } ¼ @ ⊤, \text{Jump, Store}, \text{Halt} \right] \]

\[ \text{pc} \quad m \quad s \quad i(\text{pc}) \]

\begin{array}{cccc}
0@⊥ & 0@⊥ & \{ \} & \text{Push } 1@⊥ \\
1@⊥ & 0@⊥ & \{ 0@⊥ \} & \text{Push } 0@⊥ \\
2@⊥ & 0@⊥ & \{ 0@⊥, 1@⊥ \} & \text{Push } ¼ @ ⊤ \\
3@⊥ & 0@⊥ & \{ 3@⊥, 0@⊥, 1@⊥ \} & \text{Store} \\
\end{array}

**Figure 31.** Counterexample to JUMP*A

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

\[ i(\text{pc}) = \text{Store } L_{pc} \sqsubseteq L_{m(x)} \]  

\[ pc \xrightarrow{x@L_x, y@L_y, s \land m \Rightarrow pc+1, s \land m \{ x \mapsto y@L_x \sqcup L_y \}} \]  

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

\[ i(\text{pc}) = \text{Store } L_{pc} \sqcup L_x \sqsubseteq L_{m(x)} \]  

\[ pc \xrightarrow{x@L_x, y@L_y, s \land m \Rightarrow pc+1, s \land m \{ x \mapsto y@L_x \sqcup L_{pc} \}} \]  

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

\[ i(\text{pc}) = \text{Jump} \]  

\[ pc \xrightarrow{x@L_x, s \land m \Rightarrow x@L_{pc} \land s \land m} \]  

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

\[ i(\text{pc}) = \text{Jump} \]  

\[ pc \xrightarrow{x@L_x, s \land m \Rightarrow x@L_x \land s \land m} \]  

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

\[ i(\text{pc}) = \text{Store } L_{pc} \sqcup L_x \sqsubseteq L_{m(x)} \]  

\[ pc \xrightarrow{x@L_x, y@L_y, s \land m \Rightarrow pc+1, s \land m \{ x \mapsto y@L_x \sqcup L_y \}} \]
\[
i = \left[ \begin{array}{l}
\text{Push } 5@\bot, \text{Jump, Push } 0@\bot, \text{Store, Return,}
\text{Push } 0@\bot, \text{Push } 2@\top, \text{Call } 1@0, \text{Halt}
\end{array} \right]
\]

\[
\begin{array}{c|c|c|c}
\text{pc} & m & s & i(pc) \\
\hline
0@\bot & 0@\bot & [] & \text{Push } 5@\bot \\
1@\bot & 0@\bot & 5@\bot & \text{Jump} \\
5@\bot & 0@\bot & [] & \text{Push } 0@\bot \\
6@\bot & 0@\bot & 0@\bot & \text{Push } 2@\top \\
7@\bot & 0@\bot & \frac{2}{1}@\top, 0@\bot & \text{Call } 1@0 \\
\end{array}
\]

Machine 1 continues...
\[
\begin{array}{c|c|c|c}
2@\top & 0@\bot & 0@\bot, R(8, 0)@\bot & \text{Push } 0@\bot \\
3@\top & 0@\bot & 0@\bot, 0@\bot, R(8, 0)@\bot & \text{Store} \\
4@\top & 0@\top & R(8, 0)@\bot & \text{Return} \\
8@\top & 0@\top & [] & \text{Halt}
\end{array}
\]

Machine 2 continues...
\[
\begin{array}{c|c|c|c}
4@\top & 0@\bot & 0@\bot, R(8, 0)@\bot & \text{Return} \\
8@\top & 0@\bot & [] & \text{Halt}
\end{array}
\]

C.10 STORE*E (Counterexample in Figure 34)
\[
i(pc) = \text{Store } L_x \subseteq L_{m(x)}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
\text{pc} & x@L_x & y@L_y & s & m & \Rightarrow & pc+1 & s & m \\
\hline
0@\bot & 0@\bot & 0@\bot, R(8, 0)@\bot & & & \text{Push } 0@\bot \\
1@\bot & 0@\bot & 0@\bot, 0@\bot, R(8, 0)@\bot & & & \text{Store} \\
2@\bot & 0@\top & R(8, 0)@\bot & & & \text{Return} \\
3@\bot & 0@\bot & [] & & & \text{Push } 2@\top \\
4@\bot & 0@\bot & [] & & & \text{Jump}
\end{array}
\]

C.11 CALL*A (Counterexample in Figure 35)
\[
i(pc) = \text{Call } n \ n' \ n' \in \{0, 1\}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
\text{pc} & x@L_x & v_1 : \ldots : v_n : s & m & \Rightarrow & x@L_{pc} & v_1 : \ldots : v_{n'} : R(x_{pc}+1, n')@L_{pc} & s & m \\
\hline
0@\bot & 0@\bot & 0@\bot & & & \text{Push } 0@\bot \\
1@\bot & 0@\bot & 1@\bot & & & \text{Push } 0@\bot \\
2@\bot & 0@\bot & 0@\bot, 1@\bot & & & \text{Store} \\
3@\bot & 1@\bot & [] & & & \text{Push } 2@\top \\
4@\bot & 1@\bot & [] & & & \text{Jump}
\end{array}
\]

C.12 RETURN*A (Counterexample in Figure 36)
\[
i(pc) = \text{Return } k \geq n'
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
\text{pc} & v_1 : \ldots : v_k : s & m & \Rightarrow & x@L_x & v_1 : \ldots : v_{n'} : s & m \\
\hline
0@\bot & 0@\bot & 0@\bot & & & \text{Push } 0@\bot \\
1@\bot & 0@\bot & 1@\bot & & & \text{Push } 0@\top \\
2@\bot & 0@\bot & 0@\bot, 1@\bot & & & \text{Call } 1@1 \\
3@\bot & 0@\bot & [] & & & \text{Push } 0@\bot \\
4@\bot & 0@\bot & [] & & & \text{Store} \\
5@\bot & 0@\bot & [] & & & \text{Halt}
\end{array}
\]

C.13 CALL*a and RETURN*B (Counterexample in Figure 37)
\[
i(pc) = \text{Call } n \ \ L = L_x \lor L_{pc}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
\text{pc} & x@L_{pc} & x@L_x & v_1 : \ldots : v_n : s & m & \Rightarrow & x@L_x & v_1 : \ldots : v_{n'} : R(x_{pc}+1@L_{pc}) & s & m \\
\hline
0@\bot & 0@\bot & x@L_x & v_1 : \ldots : v_n : s & m & \Rightarrow & x@L_x & v_1 : \ldots : v_{n'} : s & m \\
1@\bot & 0@\bot & x@L_x & v_1 : \ldots : v_n : s & m & \Rightarrow & x@L_x & v_1 : \ldots : v_{n'} : s & m \\
2@\bot & 0@\bot & x@L_x & v_1 : \ldots : v_n : s & m & \Rightarrow & x@L_x & v_1 : \ldots : v_{n'} : s & m \\
3@\bot & 0@\bot & x@L_x & v_1 : \ldots : v_n : s & m & \Rightarrow & x@L_x & v_1 : \ldots : v_{n'} : s & m \\
4@\bot & 0@\bot & x@L_x & v_1 : \ldots : v_n : s & m & \Rightarrow & x@L_x & v_1 : \ldots : v_{n'} : s & m \\
5@\bot & 0@\bot & x@L_x & v_1 : \ldots : v_n : s & m & \Rightarrow & x@L_x & v_1 : \ldots : v_{n'} : s & m \\
\end{array}
\]

C.14 POP* (Counterexample in Figure 38)
\[
i(pc) = \text{Pop}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
\text{pc} & e : s & m & \Rightarrow & pc+1 & s & m \\
\hline
0@\bot & 0@\bot & [] & & & \text{Push } 1@\bot \\
1@\bot & 0@\bot & [1@\bot] & & & \text{Push } 0@\top \\
2@\bot & 0@\bot & [0@\bot, 1@\bot] & & & \text{Call } 1@1 \\
3@\bot & 0@\bot & [0@\bot, 1@\bot] & & & \text{Push } 0@\bot \\
4@\bot & 0@\bot & [0@\bot, 1@\bot] & & & \text{Store} \\
5@\bot & 0@\bot & [1@\bot] & & & \text{Halt}
\end{array}
\]

Machine 1 continues...
\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
6@\top & 0@\bot & [1@\bot, R(3, 1}@\bot] & & & \text{Push } 0@\bot \\
7@\top & 0@\bot & [0@\bot, 1@\bot, R(3, 1}@\bot] & & & \text{Return} \\
8@\top & 0@\bot & [0@\bot, 1@\bot] & & & \text{Push } 0@\bot \\
9@\top & 0@\bot & [0@\bot, 1@\bot] & & & \text{Store} \\
10@\bot & 0@\bot & [1@\bot] & & & \text{Halt}
\end{array}
\]

Machine 2 continues...
\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
7@\top & 0@\bot & [1@\bot, R(3, 1}@\bot] & & & \text{Return} \\
8@\top & 0@\bot & [0@\bot, 1@\bot] & & & \text{Push } 0@\bot \\
9@\top & 0@\bot & [0@\bot, 1@\bot] & & & \text{Store} \\
10@\bot & 0@\bot & [1@\bot] & & & \text{Halt}
\end{array}
\]
\[
\begin{align*}
&i = \left[ \begin{array}{c}
\text{Push } 0@\bot, \text{ Push } \frac{6}{10}@\top, \text{ Call } 0, \text{ Push } 8@\bot, \text{ Call } 1, \\
\text{Halt, Push } 0@\bot, \text{ Return } 1, \text{ Push } 0@\bot, \text{ Store}, \\
\text{Return } 0
\end{array} \right]
\end{align*}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
pc & m & s & i(pc) \\
\hline
0@\bot & [0@\bot] & [\ ] & \text{Push } 0@\bot \\
1@\bot & [0@\bot] & [0@\bot] & \text{Push } \frac{6}{10}@\top \\
2@\bot & [0@\bot] & \left[\frac{6}{10}@\top, 0@\bot\right] & \text{Call } 0 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Machine 1 continues...} & & & \\
\text{6@\top} & [0@\bot] & \left[R(3@\bot), 0@\bot\right] & \text{Push } 0@\bot \\
\text{7@\top} & [0@\bot] & [0@\bot, R(3@\bot), 0@\bot] & \text{Return } 1 \\
\text{8@\bot} & [0@\bot] & [0@\bot, 0@\bot] & \text{Push } 8@\bot \\
\text{4@\bot} & [0@\bot] & \left[8@\bot, 0@\top, 0@\bot\right] & \text{Call } 1 \\
\text{5@\bot} & [0@\bot] & [0@\bot, R(5@\bot), 0@\bot] & \text{Push } 0@\bot \\
\text{9@\bot} & [0@\bot] & [0@\bot, 0@\bot, R(5@\bot), 0@\bot] & \text{Store} \\
\text{10@\bot} & [0@\bot] & \left[R(5@\bot), 0@\bot\right] & \text{Return } 0 \\
\text{5@\bot} & [0@\bot] & [0@\bot] & \text{Halt} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Machine 2 continues...} & & & \\
\text{10@\top} & [0@\bot] & \left[R(3@\bot), 0@\bot\right] & \text{Return } 0 \\
\text{3@\bot} & [0@\bot] & [0@\bot] & \text{Push } 8@\bot \\
\text{4@\bot} & [0@\bot] & \left[8@\bot, 0@\bot\right] & \text{Call } 1 \\
\text{8@\bot} & [0@\bot] & [0@\bot, R(5@\bot)] & \text{Push } 0@\bot \\
\text{9@\bot} & [0@\bot] & [0@\bot, 0@\bot, R(5@\bot)] & \text{Store} \\
\text{10@\bot} & [0@\bot] & [0@\bot] & \text{Return } 0 \\
\text{5@\bot} & [0@\bot] & [\ ] & \text{Halt} \\
\hline
\end{array}
\]

Figure 37. Counterexample to CALL*B and RETURN*B
### Machine 1 continues...

<table>
<thead>
<tr>
<th>pc</th>
<th>m</th>
<th>s</th>
<th>i(pc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5@T</td>
<td>0@⊥, 5@T</td>
<td>R(11, 1)@⊥, 0@⊥, R(17, 0)@⊥</td>
<td>Push 0@⊥</td>
</tr>
<tr>
<td>6@T</td>
<td>0@⊥, 5@T</td>
<td>0@⊥, R(11, 1)@⊥, 0@⊥, R(17, 0)@⊥</td>
<td>Return</td>
</tr>
<tr>
<td>11@⊥</td>
<td>0@⊥, 5@T</td>
<td>0@⊥, 0@⊥, R(17, 0)@⊥</td>
<td>Push 0@⊥</td>
</tr>
<tr>
<td>12@⊥</td>
<td>0@⊥, 5@T</td>
<td>0@⊥, 0@⊥, R(17, 0)@⊥</td>
<td>Store</td>
</tr>
<tr>
<td>13@⊥</td>
<td>0@⊥, 5@T</td>
<td>0@⊥, R(17, 0)@⊥</td>
<td>Pop</td>
</tr>
<tr>
<td>14@⊥</td>
<td>0@⊥, 5@T</td>
<td>R(17, 0)@⊥</td>
<td>Return</td>
</tr>
<tr>
<td>17@⊥</td>
<td>0@⊥, 5@T</td>
<td></td>
<td>Halt</td>
</tr>
</tbody>
</table>

### Machine 2 continues...

| pc  | m         | s                                      | |
|-----|-----------|----------------------------------------||
| 13@T| 0@⊥, 13@T | R(11, 1)@⊥, 0@⊥, R(17, 0)@⊥           | Pop                  |
| 14@T| 0@⊥, 13@T | 0@⊥, R(17, 0)@⊥                        | Return               |
| 17@⊥| 0@⊥, 13@T |                                        | Halt                 |

---

**Figure 38.** Counterexample to POP"