Path-Exploration Lifting: Hi-Fi Tests for Lo-Fi Emulators

Lorenzo Martignoni
UC Berkeley
martignlo@gmail.com

Stephen McCamant
UC Berkeley
smcc@cs.berkeley.edu

Pongsin Poosankam
CMU & UC Berkeley
ppoosank@cmu.edu

Dawn Song
UC Berkeley
Berkeley, CA, USA
dawnsong@cs.berkeley.edu

Petros Maniatis
Intel Labs
Berkeley, CA, USA
petros.maniatis@intel.com

Abstract

Processor emulators are widely used to provide isolation and instrumentation of binary software. However they have proved difficult to implement correctly: processor specifications have many corner cases that are not exercised by common workloads. It is untenable to base other system security properties on the correctness of emulators that have received only ad-hoc testing. To obtain emulators that are worthy of the required trust, we propose a technique to explore a high-fidelity emulator with symbolic execution, and then lift those test cases to test a lower-fidelity emulator. The high-fidelity emulator serves as a proxy for the hardware specification, but we can also further validate by running the tests on real hardware. We implement our approach and apply it to generate about 610,000 test cases; for about 95% of the instructions we achieve complete path coverage. The tests reveal thousands of individual differences; we analyze those differences to shed light on a number of root causes, such as atomicity violations and missing security features.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Testing tools

General Terms Reliability, Security, Verification

Keywords Symbolic binary execution, CPU emulators, cross-validation

1. Introduction

Processor emulators are widely used in systems, to provide privacy [29], integrity [17], instrumentation, replay, sandboxing [9], and program analysis [3, 27]. In principle, any component meant to *mediate transparently* between an operating system and the hardware needs to emulate the hardware to some degree; consequently, processor emulators have been linchpin components for a large class of critical applications.

Unfortunately, our critical reliance on emulators has not been met by a commensurate assurance of their correctness. Emulators are just too hard to implement correctly, for several reasons. First, emulators are large, complex pieces of software meant to mirror complex emulated layers such processors. Second, as components that often appear in the critical path of applications, emulators must be optimized aggressively, often using pervasive, brittle optimizations involving self-modifying code and multiple intermediate representations. Third, typical emulated systems exhibit great variance in how their different features are exercised, which leads to emulators with many corner cases that are infrequently exercised by common workloads.

In this paper, we tackle the problem of increasing the assurance of processor emulators, hoping to generalize what we learn to broader emulation assurance. Specifically, we present PokeEMU, a systematic framework for high-coverage testing and cross-validation of processor emulators such as Bochs[1] and QEMU[2]. Our work is inspired by two observations. First, there are several emulators for common processor architectures such as IA-32, each of which achieves a different point in the complexity-fidelity spectrum. For example, the Bochs interpreter is a faithful but relatively slow implementation of the processor (a "Hi-Fi" emulator), whereas QEMU, a dynamic binary translator, is faster but much more complex, buggier, and less complete ("Lo-Fi")[3]. Second, exploration based on symbolic execution has matured significantly in recent years, allowing us to leverage such techniques to enable high-coverage path exploration and test-case generation.

We capitalize on these observations by exploiting the following insight: one can use symbolic execution on a Hi-Fi emulator to generate high-quality test cases for a Lo-Fi emulator. Analysis of the Hi-Fi emulator extracts all the distinct behaviors and corner cases it implements; those are useful because we assume the Hi-Fi emulator's behavior is closer to the ideal processor specification than the behaviors implemented by the Lo-Fi emulator. Using then the Hi-Fi emulator's behaviors to generate automatically a test suite for the Lo-Fi emulator, one can detect and fix deviations of the Lo-Fi emulator from the behavior of the (presumed "more correct") Hi-Fi emulator or the emulated hardware. We call this methodology *path-exploration lifting*, since it automatically "lifts" definitions of program behaviors—as captured by distinct *code paths*—from one emulator to another.

ASPLOS'12, March 3–7, 2012, London, England, UK.

Reprinted from ASPLOS'12,, Proceedings, March 3–7, 2012, London, England, UK., pp. 1–12.

[1] `http://bochs.sourceforge.net/`.

[2] `http://www.qemu.com/`.

[3] We use *fidelity* loosely to denote how closely an emulator approximates the target architecture; a buggier, less complete emulator has lower fidelity than a more correct, more complete emulator.

Interestingly, although in this paper we use path-exploration lifting from a Hi-Fi emulator to a Lo-Fi emulator, hoping to "rub off" some of the higher fidelity of the Hi-Fi emulator to the Lo-Fi one, the technique is more general. It can be used in the opposite direction, from Lo-Fi to Hi-Fi, to see how the Hi-Fi emulator would behave for the distinct cases implemented by the Lo-Fi emulator developers. Beyond emulation, for any two implementations of the same precise specification (e.g., SQL query engines, SSL implementations, etc.), it makes sense to analyze one implementation to generate test cases for comparison to the other implementation.

Certainly neither cross-validation nor path exploration via symbolic execution are new. However, path exploration on an artifact to test that same artifact can at best trigger its own corner cases, but not capture those unimplemented in it, which our approach can achieve. More importantly, cross-validation alone can, at best, do random directed testing ("fuzzing") without capitalizing on the fundamental differences between the different tested artifacts. In contrast, applying systematic program analysis for path exploration, such as symbolic execution, to amplify the benefits of cross-validation is novel to our knowledge. Ideally, one would want to apply path-exploration lifting to a hardware specification (e.g., the register-transfer language specification of a circuit); unfortunately, such specifications for commodity hardware are proprietary and extremely well guarded. By applying the methodology to a Hi-Fi emulator, we capture corner cases from the next best thing. Conveniently, the Hi-Fi emulator need not be perfect, only complete: we use the actual hardware to test our emulators against, so correctness bugs in the Hi-Fi emulator do not impact our results, and can be discovered through our approach as side effects.

Although path-exploration lifting is a general concept, its implementation is challenging, often losing generality. In this paper we apply and customize the technique for Bochs and QEMU, dealing with several fundamental challenges. While in the past others have used symbolic execution to generate high-coverage test cases for programs, those programs were applications with scalar or other simple input types. In contrast, PokeEMU must generate test cases for emulators, whose input is a starting machine state and a test instruction, a staggeringly large state space to explore. Furthermore, even after PokeEMU makes sense out of the state space and generates some test cases with starting states for a given test instruction, we must figure out how to *lead* the emulated system to the desired start state: how to get it to set its registers, configuration, program counter, and execution mode to the values required, which is not straightforward since certain parts of the machine state cannot be set directly and most instructions have multiple side effects that may undo prior state setup. Finally, it may not always be possible to analyze the source code of an emulator due to intellectual property restrictions and, even without such restrictions, the emulator may manipulate system state through multiple intermediate representations, via just-in-time compilers, etc.; operating on the binary executable may be the only option for testing system emulators.

Contributions. This paper proposes path-exploration lifting, a new methodology for *exhaustively* exploiting the correctness of one artifact to improve the correctness of another. The paper presents the design, implementation, and evaluation of the methodology in the PokeEMU system for processor emulators. PokeEMU consists of several key components. First, the paper presents a symbolic execution engine for x86 binaries, FuzzBALL, used to explore paths from binaries, as opposed to source code. Second, it describes a novel exploration strategy for processor emulators, which starts with the instruction decoders, generating instructions to iterate over, and then explores the instruction emulator to identify paths per instruction, with optimizations to reduce the state space. Third, it details an essential tool for processor testing: an input-state generator that, given an input state for a test case, automatically creates a program to bring an emulator or physical system to that state, so that the test can take place. Finally, it conducts the first systematic study of the approach using symbolic execution, assessing QEMU's emulation fidelity using test cases lifted from Bochs and comparing it to both Bochs and real hardware, identifying several deviations from expected behavior.

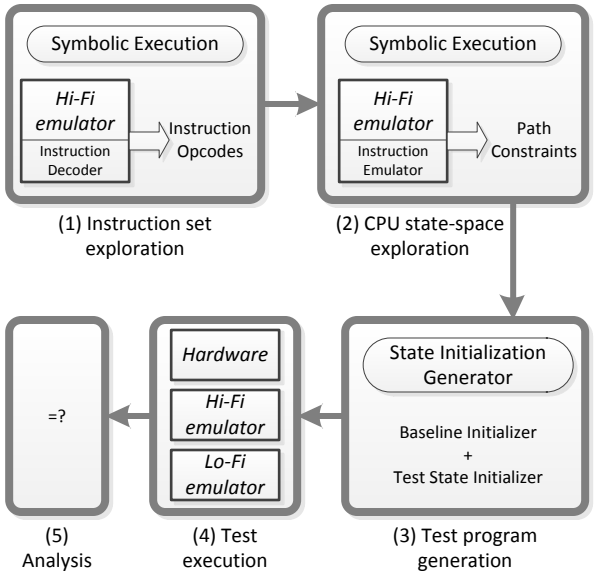

Figure 1. Overview

Our evaluation establishes four key points. First, for more than 95% of the instructions, PokeEMU achieved complete path coverage. Second, PokeEMU found a large number of deviations among the emulators tested and real hardware: out of about 610,000 programs, more than 72,000 triggered differences, and we have identified a number of root causes, some of which affect many instructions. Third, many of the deviations found could not have been found by prior emulator-testing approaches, such as random testing. Finally, at least two of the deviations found could lead to significant security problems, when those emulators are used as the basis for a security tool.

2. Overview

A CPU or instruction-set emulator is a program that runs on one architecture (*host*), but whose functionality is to simulate the processor of a potentially different architecture (*guest*). Our goal in this work is to discover differences between the behavior of a CPU emulator, and the behavior of another emulator or real hardware: such a difference enables us to find bugs in emulators.

To be precise, we define the *(machine) state* of a CPU emulator or hardware system to be all the values, such as in general purpose registers, control registers, flags, or memory, that can affect the execution of a future instruction. We say that systems show a behavioral difference if we start them in the same machine state (called the *test state*), and they execute the same instruction (called the *test instruction*), but after the instruction executes they are in different machine states (called the *final states*). Example differences include having a different value in a register, or not raising an exception when the hardware would.

Approach. At a high level, our approach is to discover differences by constructing high-coverage tests that trigger them, using the

methodology of *path-exploration lifting*. Furthermore, unlike traditional program testing where the tests are simply scalar test input values, a test for the emulator would specify a test state and a test instruction. And in practice, we generate *test programs*: stand-alone programs that can run on an emulator to set up the test states and then execute the test instructions.

Our design goals are (1) to maximize the coverage of our testing, subject to the constraints of (2) producing a practical number of tests, while (3) requiring relatively little human guidance in modifying the emulators or configuring the tests. Next we discuss how we apply these principles to the key technical challenges.

Challenges and Techniques. First, the space of possible instructions and machine states is astronomically large, so it would not be practical to individually test every possible instruction and initial state. However this space also has complex structure, so choosing instructions and states at random, or based on typical usage, would miss differences that occur in corner cases. We address this challenge by using symbolic execution to explore the space based on how components of the state are used by a tested emulator. First, we apply symbolic execution on the instruction decoder of an emulator to select test instructions. Then for each test instruction, we apply symbolic execution to the emulator's implementation of that instruction. Specifically, we choose a subset of the machine state as relevant (this choice is discussed in detail in Section 3.3.1 and Figure 3). Symbolic execution determines a test state for each execution path through the implementation that can be triggered by some assignment of values to the selected state components.

A second challenge is that CPU architectures do not provide a uniform interface to initialize all the components of their state. For instance control registers must be initialized using specialized instructions, and some kinds of initialization are either prerequisites for or conflict with other kinds. To address these challenges, we write a fixed piece of code to initialize a machine to a baseline state. Then our tool automatically instantiates a sequence of *test state initializers* to set the remaining parts of the state. Thus the test program consists of the baseline state initializer, then the test state initializers, then the test instruction.

A third challenge is that many emulators use inline assembly code or perform just-in-time (JIT) compilation, so they cannot be properly analyzed at the source code level. We address this challenge by using binary-level symbolic execution, which applies uniformly to an interpreter compiled from source code and the machine code created by a just-in-time compiler. Although we did have source code for the emulators we studied, we took on this challenge so as to prepare for also studying closed-source, commercial emulators and VMMs.

System Architecture. Our approach has four steps: exploration, test program generation, test program execution, and difference analysis (Figure 1).

In the first step, exploration, we use symbolic execution to explore an emulator. To efficiently partition the space of possible instruction executions, we do the exploration in two steps: we first explore to generate legal instructions (Figure 1(1)), and then explore the execution of each instruction separately (Figure 1(2)). We progressively explore all the execution paths of an instruction implementation, given a selected set of symbolic state components, and generate one test for each path. Thus the output of this step is a set of pairs of test instructions and test states. The symbolic execution is implemented using our tool FuzzBALL, which we have extended with optimizations for this problem domain. The second step, test program generation, constructs complete test programs from the results of exploration. For each input pair of a test instruction and an test state found in the exploration phase, we construct as output a test program consisting of the baseline initializer, the test state initializer for the test state, and the test instruction (Figure 1(3)). In the third step, test program execution, we take the test programs as input and run them on emulators and real hardware (Figure 1(4)). We instrument the emulators and a hardware-based virtual machine to save as output the machine state after executing the test program (the *final state*). In the fourth step, difference analysis, we compare the final states from the different executions of a test (Figure 1(5)). If the results differ between emulators or between an emulator and the real hardware, we have triggered a behavior difference.

For our evaluation, we have selected two emulators that support x86 guest code: Bochs is an interpreter, and QEMU is a JIT compiler. We compile the emulators for the Linux/x86 host platform. In our experiments, we apply symbolic execution to Bochs, and then use the generated tests for a three-way behavior comparison of Bochs, QEMU, and an Intel® Core™ i5 * workstation virtualized by KVM. We test the processor's 32-bit protected mode.

3. Path-Exploration Lifting

In this section we describe the main technical aspects of how our system explores the space of possible instruction executions in an emulator. We start by describing our core technology of lightweight binary-level symbolic execution (Section 3.1). Then we describe the two ways we apply it: first, to discover possible instructions (Section 3.2), and then to find machine states that trigger various behaviors in an emulated instruction (Section 3.3). Finally, we describe a difference-minimization technique that we use to simplify the machine states discovered by symbolic execution (Section 3.4).

3.1 Lightweight Symbolic Execution

The core of our system's state-space exploration is a lightweight engine for binary-level symbolic execution, named FuzzBALL. We start our description with a review of the key concepts of symbolic execution in general, then describe the *online* approach our tool takes, and some of the particular challenges that arise when operating on binaries. At a high level, FuzzBALL implements similar functionality to previous symbolic execution systems such as KLEE [6]. But in contrast, it takes a simpler approach in some areas that can be performance or code complexity challenges in other systems, and it is designed for a binary-level, rather than a source-level, program representation.

3.1.1 Background: Symbolic Execution

The basic principle of symbolic execution is to replace certain *concrete* values in a program's state with *symbolic variables*. As these symbolic values are used in later computations, they produce more complex symbolic expressions. These symbolic expressions are valuable because they can summarize the effect of many concrete executions.

When a symbolic expression is used in a control-flow instruction, we call the formula that controls the target a *branch condition*. On a complete program run, the conjunction of the conditions for all the symbolic branches is the *path condition*. Thus the values for the symbolic variables that satisfy a path condition are ones that would cause the program to execute the same control-flow path as the one executed symbolically. Similarly, by taking a prefix of the path condition with the final branch condition negated, we obtain a condition corresponding to a different control-flow path. Solving such a path condition lets us obtain a new set of concrete values that would cause the corresponding path to be executed.

* Intel® Core™ i5 is a trademark of Intel Corporation in the U.S. and/or other countries. Other names and brands may be claimed as the property of others.

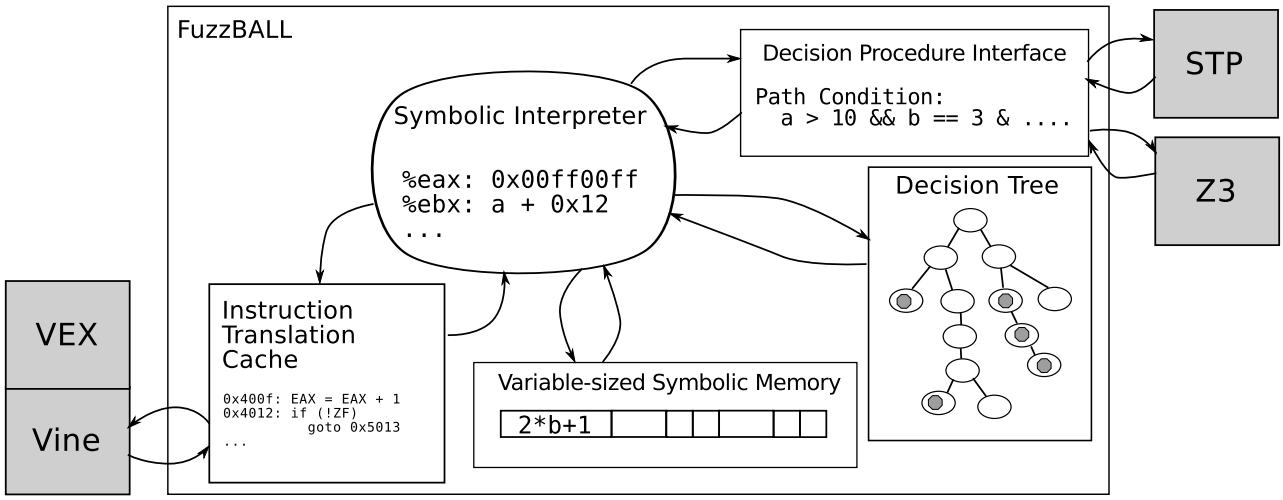

Figure 2. An overview of some of the key components of our symbolic execution engine FuzzBALL. The interior components are described in Section 3.1; the gray-shaded components are off-the-shelf libraries.

3.1.2 Lightweight, Online Approach

At its core, FuzzBALL is an interpreter for x86 instructions, but one in which the values in registers and memory can be symbolic expressions rather than just concrete bit patterns. A graphical overview of FuzzBALL's architecture is shown in Figure 2. As it executes an x86 binary program, FuzzBALL translates each instruction into a streamlined intermediate representation (IR), then performs each action specified by this IR on a symbolic state representation. For instance, suppose that `%eax` has the symbolic value $a - 20$, and the location pointed to by `%ebx` has the value 5. Then after FuzzBALL interprets the instruction `add (%ebx),%eax`, the symbolic value of `%eax` will be $a - 15$.

Suppose that the following instruction is `jz a04`, a conditional branch that checks whether `%eax` is zero. The branch condition $a - 15 = 0$ depends on the symbolic variable a, so to decide which instruction to execute next, FuzzBALL checks whether either of the formulas $a - 15 = 0$ or $a - 15 \neq 0$ is satisfiable, using a decision procedure. In this case both are, so FuzzBALL can choose freely. Suppose it chooses to take the true case ($a - 15 = 0$); then it will record that the false case is available to explore later, and go on to the instruction at address `0xa04`. As it encounters other symbolic branches, it makes similar choices and records them. When it reaches the end of the program or another specified stopping point, FuzzBALL will mark that state as explored, and start again from the beginning of the program or a specified starting point. On the next run, FuzzBALL uses recorded decisions to ensure that the path it executes is different from those explored before. FuzzBALL continues in a loop, executing new paths, until all the possible paths have been executed. The choice of which program values are marked symbolic (typically the inputs to some computation) controls how many paths FuzzBALL explores: more symbolic values mean more execution paths.

Many uses of symbolic execution start by running a program with a pre-existing concrete input. In such applications, a simplification is to have the symbolic execution run in parallel with, but not modify, a concrete execution, so that the path condition describes the path the concrete execution took. In our context, by contrast, we wish to completely explore an execution space, so it does not matter which path we execute first. On the other hand, it is useful to have more flexibility in choosing which paths to symbolically execute. Therefore our system performs *online* symbolic execution. During execution we start with no commitment to a concrete value for any symbolic variable; when the value at a memory location has a symbolic expression, this is instead of, rather than in addition to, a concrete value. This gives the system the flexibility to choose either direction for a symbolic branch, subject only to feasibility.

Thus FuzzBALL is an interpreter for machine instructions, where registers and memory contain symbolic formulas. The representation of memory uses a two-level data structure similar to a page table, in which each page of memory is an array of formulas rather than an array of integers.

Online Decision Making. When a branch condition is symbolic, an online symbolic execution tool can execute either the true or false side of the branch, but to make this decision it must reason about the path condition before execution can proceed. Our symbolic execution engine can choose directions subject to two constraints: the branch direction must be *feasible*, and it should lead to a path that has not been explored before. A branch direction is feasible if it is logically consistent with the previous branches in the path. For instance, in `if (x>y) x=y; if (x>y) abort();`, there is no feasible path in which both `if` conditions are true. FuzzBALL checks feasibility by giving the path condition to a decision procedure, which determines whether the condition is satisfiable, and if so, supplies a satisfying assignment.

Specifically, FuzzBALL interfaces with the decision procedures STP [13] and Z3 [12]. After simplifying symbolic expression, it translates them into STP or Z3's syntax for quantifier-free formulas over fixed-sized bit-vectors (representing bounded machine arithmetic). STP and Z3 are well tuned for applications like FuzzBALL: their results are precise but produced quickly, with most queries completing in a fraction of a second. When using Z3, FuzzBALL can also solve path conditions incrementally: i.e., if it previously solved $b_1 \wedge b_2$, Z3 can reuse information from that solving process when solving $b_1 \wedge b_2 \wedge b_3$.

Decision Tree. To avoid exploring the same paths repeatedly, FuzzBALL uses a *decision tree*. The decision tree is a binary tree in which each node represents the occurrence of a symbolic branch on a particular execution path, and a node has children labeled "false" and "true" representing the next symbolic branch that will occur in either case. Each tree node records whether the false and true branches have been checked for feasibility, as well as whether any additional unexplored branches appear below this node in the tree.

On each execution, FuzzBALL examines the decision tree to choose a random path within the part of the tree that has not been completely explored, then adds on to the tree for the part of the path being explored that is new. When creating a new node, FuzzBALL checks whether both the false and true branch directions are feasible, and if so, it can choose arbitrarily (either randomly or according to a supplied heuristic). After reaching the end of the path, FuzzBALL propagates the bit indicating that a subtree has been fully explored back up the tree until it reaches a node with an unexplored branch. The decision tree grows as more paths are explored, so FuzzBALL uses a compact in-memory representation and can optionally store it on disk instead, but this is not needed for runs of the length used in this project.

Branches that come from `if` statements and branches for loop exit conditions are treated uniformly, since at the instruction level they look the same. Thus FuzzBALL considers a different number of executions of a loop as distinguishing a different execution path. In other applications, this can lead to a significant state-space explosion to manage, but it is not a major obstacle to PokeEMU because instructions usually do not contain unbounded input-dependent loops.

Thus the decision tree ensures that each path FuzzBALL explores is different, and that exploration stops if no further paths are possible. Similarly to systems that duplicate execution state at a symbolic branch [6, 8], the decision tree saves (expensive) invocations of the decision procedure when the tool already knows which branch direction is feasible. As a tradeoff, our approach repeats (relatively inexpensive) concrete and symbolic execution on the repeated path, to avoid keeping multiple states at once, which would increase memory usage and implementation complexity.

Extension to Word-sized Values. When execution requires a concrete value for a word-sized expression, like a `switch` statement argument or an array index, FuzzBALL applies the same mechanisms described above for two-way branches, once for each bit in the word, most-significant first. This reduction carries over the key properties from two-way branches: FuzzBALL chooses only feasible values, and eventually tries all feasible values.

3.1.3 Operating at the Binary Level

Since FuzzBALL targets binaries rather than source, it must address challenges including instruction-set complexity and variable-sized memory accesses.

To factor out instruction-set complexity, FuzzBALL uses the BitBlaze[4] Vine library [27], which in turn builds on the VEX library which is also used by the Valgrind debugging tool [25]. First VEX translates an x86 instruction into the VEX intermediate representation, and then Vine translates from this into its own language which is even simpler; these translations are cached for efficiency. To handle memory accesses of different operand sizes (bytes, words, etc.), FuzzBALL tries when possible to represent values in their natural size, so that splitting and reassembly are required only when the program itself accesses memory in an inconsistent way. To achieve this, FuzzBALL's representation of memory can contain symbolic values of differing sizes. We describe some additional implementation challenges that, in particular, are inspired by use with emulators in Section 3.3.2.

3.1.4 Impact of FuzzBALL's Correctness

At this point, one might worry about seemingly circular reasoning in our approach. Our goal is to check the correctness of one x86 interpreter (that in an emulator), but our technique relies on another x86 interpreter (that inside FuzzBALL). What if FuzzBALL has bugs similar to those we find (Section 6) in other emulators?

In fact there are several reasons why our approach is still effective. First, any such bugs in FuzzBALL would be unlikely to significantly affect our results, because emulators *use* in their own implementation a much smaller and better-exercised subset of processor functionality than they *emulate*. Second, the differences we discover are real, independent of the test generation process. We use symbolic execution to improve coverage, but the behavior differences are validated by test cases that run on their own. Third, FuzzBALL can be used to validate many emulators, so efforts towards strengthening or verifying the correctness of FuzzBALL would be amplified through its use in a tool such a PokeEMU.

3.2 Instruction Set Exploration

The x86 instruction set is complex enough that even just enumerating all the possible instructions is non-trivial. But we would like exactly such an enumeration, in order to partition the later exploration so that we consider each instruction separately and exactly once. Therefore our first, and relatively simpler, application of symbolic execution is to discover a set of byte sequences representing instructions to test.

We observe that emulators contain *instruction-decoding functionality* to parse a byte sequence, check whether the sequence is a legal instruction, and if so, decide which code in the emulator will process it. This later code might be the implementation itself in an interpreter, or a code-generation routine in an IR-based or JIT-compiler emulator; we will refer to it as *per-instruction code*. By exploring the instruction decoder with symbolic execution, we can discover which byte sequences the emulator considers to be instructions, and group byte sequences that are the "same" instruction in the sense that they have common per-instruction code in the emulator. In particular, we start symbolic execution at the entry point of the emulator's instruction parser, mark the bytes that are the input to this parser as symbolic, and explore execution paths up to the selection of the per-instruction code.

An x86 instruction is between 1 and 15 bytes, consisting of optional prefix bytes, an opcode that is usually 1 or 2 bytes, and trailing fields. Those trailing fields can specify a sub-opcode, register operands, addressing modes, and immediate values. The total number of possible instruction byte sequences is astronomical (though less than $2^{8 \cdot 15} \approx 1.3 \cdot 10^{36}$, because not all instructions allow all possible prefixes and operands). To select a more manageable number of byte sequences, we conceptually partition the byte sequences according to which per-instruction code they trigger, and select a bounded number of byte sequences (currently 1) from each cell of the partition. Intuitively, we select one byte sequence per instruction, for the definition of "instruction" given by the emulator's per-instruction code. Selecting more byte sequences per instruction would slightly improve our coverage of functionality selected by flags within the instruction, such as different addressing modes, but we estimate that the incremental benefit would be relatively low.

The instructions defined by emulator implementations are not in one-to-one correspondence with the 1-2 byte instruction opcode field: some opcode values correspond to multiple implementations depending on prefixes or an extra sub-opcode field, and some distinct opcodes share a single implementation. But we observe that generally at most either a single prefix byte or the sub-opcode within the next byte after the opcode is also relevant, and any other prefix bytes are optional, so every implementation has a unique representative based on the first three bytes of an instruction byte sequence. As shown in the results of Section 6, this approach allows us to cut down an original space of 2^{24} (16.8 million) three-byte sequences into less than 1000 unique instructions.

[4] `http://bitblaze.cs.berkeley.edu/`

3.3 Machine State-Space Exploration

Our system's more critical (and more complex) use of symbolic execution is to explore how the state of the emulated machine before execution of an instruction (the *input state*) affects the instruction execution of the Hi-Fi emulator, and the state after its execution (the *output state*). At a high level, we mark the input state as symbolic, symbolically execute the instruction implementation, and for each execution path record the behavior and output state. Specifically, each execution path starts at the beginning of the code implementing an instruction and ends when the Hi-Fi emulator is about to raise an exception or execute the next instruction. Here we discuss how we select and mark machine state as symbolic, and some optimizations that make this state space manageable.

3.3.1 Machine State

Our key control over the exploration performed by symbolic execution is the choice of which parts of the machine state we treat as symbolic. Symbolic execution will explore all of the code paths that can be reached for some assignment of values to the symbolic parts of the input state. On the other hand, those parts of the machine state that are left concrete will be treated as fixed in exploration. Thus the more state we mark as symbolic, the larger a state space we will explore. We would generally like to explore as large a state space as possible, except that we would like to avoid repeatedly exploring large numbers of executions that are effectively identical. For instance the page table can have 2^{20} possible base addresses, but while the contents of the page table are significant, its location is not, so it is enough to use only one such position for exploration.

The state of the guest machine is represented by data structures in the memory of the host program, so it is with respect to these data structures that we specify symbolic locations. FuzzBALL supports a mode in which the entire state of the host machine is symbolic, and we also considered inferring this data layout from execution. However, neither of those approaches has proved necessary so far: in our observation the data structures that represent the machine state have a straightforward layout in the Hi-Fi emulator.

Symbolic values are specified to FuzzBALL by giving their address, so we write C test code to print the locations of the fields that we make symbolic. For uniformity, all of the symbolic locations are specified as bytes, but specifying 4 consecutive bytes as symbolic is equivalent to specifying a single symbolic word. Conversely, we can make a subset of the bits in a byte symbolic by marking the entire byte as symbolic, and then adding a side constraint that fixes the concrete bits.

Overall, our strategy has been to mark as much of the machine state as symbolic as possible, except for locations that have many effectively equivalent values. Thus we make most of the machine registers and tables such as the page table and the global descriptor table (used for segment accesses) symbolic. A graphical presentation of the registers we mark symbolic is shown in Figure 3. However, we keep concrete values for parts of registers that are effectively just pointers to other data in memory, such as the portion of the CR3 register that is a pointer to the page table, as described above. We also leave concrete the flags that specify the CPU is operating in 32-bit protected mode, since that is the target of our testing. And of course the instruction pointer (EIP in x86 terminology) needs to be concrete along with the bytes of the instruction to be executed. In total, our symbolic machine state consists of approximately 400 bytes in registers and tables, along with all of the unused bytes in physical memory.

3.3.2 Optimizations for Exploration

Even after restricting to a single instruction and carefully selecting which machine state should be symbolic, the space of possible executions is still quite large. Here we discuss two classes of further optimizations that make our symbolic execution more effective by avoiding repeated exploration of similar paths.

Summarizing Common Computations. Some parts of instruction execution involve multiple execution paths, but are consistent over a large number of instructions. We would like to avoid repeated execution of these code regions, particularly since each such multi-path region has a multiplicative effect on the total number of paths. To avoid path explosion, we take a divide and conquer approach: we identify the common code region, perform symbolic execution on it separately to build a precise summary, and then use that summary in place of the common code.

Using a preliminary run of symbolic execution, we explore all the paths of a computation and record the outputs as a function of the inputs. Next, we combine the symbolic expressions for the path conditions and the output values for a value into a single large formula: for instance, if such a computation had an output v_i on the path with path condition p_i, the summary formula is `p1 ? v1 : (p2 ? v2 : ...)`. Then, in the main symbolic execution runs, we skip execution of the computation and instead add the pre-computed constraint to the path condition.

For instance, we use this treatment for code in Bochs that computes a cached copy of a segment descriptor. An x86 processor allows the specification of six memory regions, called segments, each with a chosen base address and size (these were once used for memory management, and are still used for thread-local memory regions and some security applications). The locations and other attributes of each segment are stored in a packed data structure called the *segment descriptor*, but since this descriptor rarely changes, emulators often cache its information in a data structure with their own choice of layout. The layout of the cache is emulator-specific, so we mark the state symbolic in the emulator-independent descriptor structure and let the emulator recompute its cache. However the code that updates the cache has 23 paths, so executing all the possible paths through all of the possible cache updates would increase the search space by a factor of $23^6 = 148035889$. Instead, we summarize the cache-update computation with a single set of symbolic expressions, automatically computed from an exploration, in advance, of just the cache-update function.

Indexing Memory and Tables. As described in Section 3.1.2 above, FuzzBALL's default behavior, when a symbolic value is used as an index, is to exhaustively explore each index value. While this would be suitable for a small table in which each entry is different, it is impractical for large arrays such as the page table or the guest memory (which is generally represented as a single or multi-level table in the host). For these arrays, we instead direct FuzzBALL to select a single index value at random, and not explore any other values. Like making a pointer in the initial state concrete rather than symbolic, this on-the-fly concretization narrows the search space that FuzzBALL explores, but it is justified by the observation that for most purposes, all 2^{32} locations in memory are equivalent.

Another important consideration for large tables in the machine state is how they are initialized for symbolic execution. For the page table, we concretely initialize the parts of the table that are pointers to sub-tables or page frames, but make all of the flag bits symbolic. For main memory, we treat each byte as a separate symbolic variable, but we modify FuzzBALL to create those variables on demand only when a location is accessed.

3.4 State Difference Minimization

For each path explored in symbolic execution, the decision procedure computes an assignment of bits to the symbolic variables that would cause the emulator to execute that path: this assignment is then the basis for constructing a test state. If any bits are symbolic,

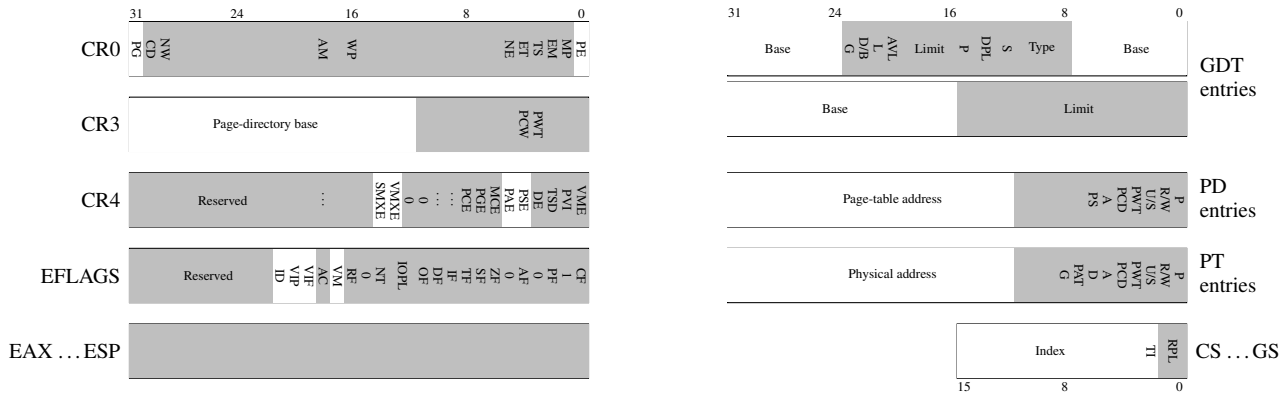

Figure 3. Symbolic machine state (grayed bits are symbolic, the remaining ones are concrete).

but not constrained by checks on the explored path, the decision procedure will choose values for them arbitrarily. However, this flexibility is inconvenient for two reasons. First, it makes the generated tests harder to understand, because they contain extra state differences that are irrelevant to the execution of the emulator, and caused only by the decision procedure's arbitrary choices. Second, these irrelevant differences can cause test execution to fail when they affect state that is checked in the test execution but not during the symbolic execution. As an example, we start symbolic execution in Bochs after it has fetched and decoded an instruction, so the permissions on the code segment CS are not relevant for most instructions. But in a real execution, the test instruction must be fetched using CS, so a change that makes that segment inaccessible will cause the test to fail before executing the instruction.

To avoid these problems, we wish to base test states not on an assignment where unconstrained bits are arbitrary, but on one where unconstrained bits are left the same as in a baseline machine state that "just works." In other words we want to find an assignment that is minimally different from the baseline state. We implement this minimization using a simple and efficient greedy approach. Starting with a working assignment equal to the one produced by the decision procedure, we iterate over each of the bits that are different from the baseline state. For each bit that is different, we check whether setting it to its value in the baseline state still satisfies the path condition; if so, we make the change in the working assignment. Potentially making multiple passes could further reduce the size of the difference, but a single pass is sufficient for the problem of unconstrained variables, which is our main motivation.

We also explored implementing this minimization by excluding variables from the assignment that do not appear in the path condition. However, particularly in the presence of bitwise operations, FuzzBALL's symbolic expressions sometimes retain irrelevant variables. It would have required a complex additional analysis to reliably remove such variables. By comparison our current approach based on evaluation was simple to implement and requires no approximation.

4. Generating Test Programs

Figure 4 shows the execution of a test program, which is a standalone disk image that boots an emulator, initializes a test state, executes a test instruction, and either halts normally or raises an exception. To simplify the process of constructing code to set up the test state, we divide it into two steps. First we write a *baseline state initializer*, code that sets up a single baseline state that is a starting point for any state in a given processor mode. Then we use an automated code generation process to construct, for each specific test state, the additional initializations needed to reach the test state from the baseline state: we call these the *test state initializers*. The advantage of this two-step approach is that because the test states are similar to the baseline state, we require relatively little new code specific to each test.

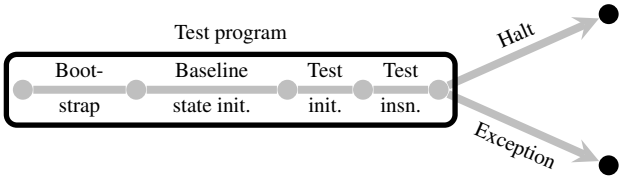

Figure 4. Execution of a test (the black circles denote when we take a snapshot of the CPU state and of the physical memory; the rectangle delimits the test program).

We choose a bootable disk image as the easiest way to load and run code in an emulator. So in summary, a test consists of a bootable disk image containing an off-the-shelf boot loader, the fixed baseline state initializer, the test state initializers for a particular test state, and the test instruction.

Next we describe in detail baseline-state initialization and test-state initializer generation.

4.1 Baseline State Initialization

The baseline state is a minimalist execution environment necessary for successfully running all possible tests in a specific operating mode. This baseline state corresponds to the concrete state used during the exploration stage (described in Section 3.3). We now describe specifically the baseline initializer we use for 32-bit protected mode with paging enabled, the most common mode for x86 processors and the one used in our evaluation. We could construct similar baseline initializers for other modes.

The off-the-shelf boot loader we use happens to already configure the machine in 32-bit protected mode. The remainder of the initialization consists of populating the global descriptor table, the page table, the interrupt descriptor table, and enabling paging and interrupts. More precisely we initialize the global descriptor table to use a flat segmentation model. That is, the code, data, and stack segments have a zero base and a 4-GByte limit. We configure the page table to map the 4-GByte virtual address space linearly to a 4-MByte physical memory, repeating every 4-MBytes so that each physical page backs 1024 virtual pages. All pages are initially marked as readable and writable and accessible to both user and kernel mode. This configuration ensures that, unless the global de-

scriptor table or the page table are modified later during the execution, any access to the memory will succeed. The interrupt descriptor table is configured to dispatch exceptions to handlers that halt the CPU, and interrupts to handlers that effectively ignore the interrupt by resuming execution immediately.

4.2 Building Test Programs

Each output of the exploration phase is a test instruction and a test state: a precise configuration for a subset of the CPU state and of the physical memory, necessary to trigger a specific path during the emulation of the test instruction. In order to exercise the same behavior at run-time, we need to run the test instruction starting from the same machine state discovered during the exploration. To do so, we need to generate code that, starting from the baseline state described in the previous section, sets up the machine state for running the test instruction. This code, the *test state initializer*, will be executed immediately after the baseline state has been initialized. The test program then executes the test instruction. If the test program is executed successfully the CPU halts; otherwise an exception is raised.

To organize the process of constructing the test state initializer, we divide it according to the various parts of the machine state we initialize. For each part of the state that must be initialized, we design a *gadget*, a short code snippet, that encapsulates how to initialize the state component. Then we have an algorithm that automatically selects and instantiates appropriate gadgets and assembles them into a complete test state initializer.

In more detail, each gadget consists of an assembly-language instruction sequence that initializes a specific state component, plus additional constraints specifying its prerequisites and side effects. The prerequisites specify that one gadget must execute before another. The side effects specify that in addition to the intended initialization, a gadget has other effects such as modifying scratch registers or causing a cache to become out of date. We have built gadgets to initialize general purpose registers, segment registers, control registers, and memory locations.

Then, our algorithm to generate the test state initializer operates as follows. First we instantiate a gadget for each component of the test state. Next, for each side effect of this first set of gadgets, we instantiate an additional gadget to correct the side effect, and we continue this process until there are no more outstanding side effects. Then, based on the prerequisite structure between all of these gadgets, we construct a dependency graph, and topologically sort the gadgets to respect the dependencies among them. Finally we append to this initializer sequence the test instruction itself and a halt instruction, and assemble the whole sequence to a binary. If our mechanism fails to find an initializer (e.g., because of a cyclic dependency or a side effect for which no gadget exists), we abort and ask for user assistance. A benefit of state-difference minimization (Section 3.4) is that none of the test cases generated by our system caused the initializer generator to fail.

Figure 5 shows a sample test program (without the baseline initializer) generated by our system for the instruction `push %eax`. The left hand side (Figure 5(a)) shows the output of the machine state space exploration. The system returned an assignment for the stack pointer and for two bytes of the physical memory, corresponding respectively to the fifth and sixth byte of the tenth entry in the global descriptor table. This descriptor is used for the stack segment, thus the purpose of the test is to exercise checks on the type and on the limit of the stack segment. The right hand side of the figure (Figure 5(b)) shows the code generated by our system to initialize the state, to execute the `push` instruction, and to mark the end of the execution. Line 1 initializes the stack pointer. Lines 2 and 3 overwrite the type and the default operand size attributes of the tenth segment descriptor in the global descriptor table. Lines 4 and 5 force the CPU to reload the descriptor of the stack segment (the tenth descriptor). Line 6 restores the original value of the `%eax` register, used by the previous instructions. Line 7 is the test instruction and line 8 halts the CPU. Our sequence generation algorithm automatically determines that lines 2 and 3 require lines 4 and 5, and that lines 4 and 5 require line 6.

5. Executing Test Programs

In the test program execution step, we take as input a test program as constructed in the previous step, and execute it on either an emulator or the real hardware. We monitor its execution to catch any exceptions raised by the test instruction, and to record the final state; these are the outputs that we will then compare. We now explain how we implement this execution step for emulators and for real hardware.

5.1 Emulator Execution

To run a test on a given emulator, we boot the emulator using the virtual disk image created for the test. After booting, the test runs automatically. The execution of the test can terminate in two different ways: the CPU either halts or throws an exception. To detect the termination of a test, we intercept those events; when one occurs, we generate a snapshot of the state of the CPU and of the physical memory. We achieve this by instrumenting the main execution loop of the emulator, enabling the interception after the baseline initialization has completed.

The test state initialization might modify critical portions of the CPU state (e.g., the page table or the global descriptor table). If any hardware interrupt is delivered during this stage of the initialization, the CPU might be unable to execute the interrupt handler successfully and will throw an exception. Different emulators simulate different devices and adopt different strategies to deliver hardware interrupts. Therefore, to prevent a spurious difference in the state, we ignore all interrupts after the baseline state has been initialized, again by instrumenting the main execution loop of the emulator.

We have had to implement very little instrumentation to intercept hardware interrupts, exceptions, and halt requests. Bochs already offers an API to instrument various types of events. QEMU does not provide an API, but we only needed to write a 10-line patch to intercept the events.

We create snapshots of the CPU state and of the physical memory with instrumentation code in the emulator that uses the emulator's APIs for memory access. The advantage of this approach, as opposed to attempting to create the snapshot as part of the test program, is that it is effective even if the emulated CPU is in an invalid state. Bochs and QEMU have similar built-in snapshot capabilities, but we implement our own file format to simplify comparison.

5.2 Hardware Execution

As with the emulator, we would like to record the state of the CPU and of the physical memory obtained at the end of the test program on real hardware Unfortunately, this is not easy to do because the execution environment would require special support for creating snapshots of the state at the end of the execution and because each test would have to reset the hardware.

Our strategy to overcome this problem is to leverage the closest approximation of the real hardware available: a hardware assisted virtual machine. With a hardware assisted virtual machine, based on Intel® VT-x, or AMD-V* [1, 24], we can execute tests in a guest virtual machine and supervise the execution of the guest from the virtual machine monitor. The majority of the instructions (including most privileged ones) can be executed directly on the hardware from the guest. The mediation of the virtual machine monitor is required only for a small subset of the instructions and events,

```
%esp    : 0x002007dc
00208055: 0x13  (gdt 10)
00208056: 0xcf  (gdt 10)
```

(a)

```
1  movl $0x002007dc,%esp
2  movb $0x13,0x00208055   // modify segment type and
3  movb $0xcf,0x00208056   //   default operation size (gdt 10)
4  movw $0x0050,%ax        // force reload of stack segment
5  movw %ax,%ss
6  movl $0x00000000,%eax   // restore killed %eax
7  .byte 0xff,0xf0         // push %eax
8  hlt                     // the end
```

(b)

Figure 5. Sample test-case generated by FuzzBALL (a) and corresponding x86 code of the test program (b), for the instruction `push %eax`.

and is triggered by traps. Hardware interrupts, exceptions, and halt requests that occur while executing guest code directly on the hardware can be intercepted by configuring the CPU to trap into the virtual machine monitor whenever they occur. When a trap occurs, the virtual machine monitor, having complete visibility to the state of the guest virtual machine, can create a snapshot of the state of the CPU and of the physical memory. Finally, the hardware guarantees a separation of the guest from the virtual machine monitor. Thus, the virtual machine monitor is always able to regain control of the execution, it can reset the state of the guest, and multiple tests can be run without having to reset the machine physically.

All the guest instructions in the test program that can be directly executed on the hardware are guaranteed to be correct. In other words, the state at the end of their execution corresponds to the state we would obtain if we executed the same instructions without the virtualization layer. On the other hand, for the instructions that require the mediation of the virtual machine monitor we do not have the same guarantee. However the number of such instructions is very small (just those that load and store a few privileged control registers), and their semantics simple, so we have checked by hand that the code in the virtual machine monitor responsible for the mediation complies with the real semantics.

Our implementation is based on KVM [19] (Kernel-based Virtual Machine), a virtual machine monitor for GNU/Linux. Only a few modifications were necessary to the original KVM codebase in order to intercept all traps that occur after the baseline state has been initialized. We handle different types of traps differently. If the trap originates from an exception or a halt request, we take a snapshot of the guest CPU state and physical memory and terminate the guest. If the trap originates from a hardware interrupt, we ignore the trap and resume the execution of the guest. Another class of traps are used to simulate exceptions: these occur when an instruction that would normally cause an exception (in the absence of the virtualization layer), instead generates a virtualization trap. Thus for all other types of trap, we let the virtual machine monitor handle the trap, but, before resuming the execution of the guest, we check whether an exception will be injected into the guest at the next resume. If so, this indicates that the trap was simulating an exception, so we take a snapshot and terminate as for a direct exception.

6. Evaluation

We evaluated PokeEMU by comparing the behaviors of the latest versions of QEMU (0.14.0) and Bochs (2.4.6), with the behavior of an Intel® Core™ i5 processor. On the latter we used a customized version of KVM (2.6.37) to automate the execution of the experiments. Since the i5 processor has hardware support for memory virtualization (extended, or nested, page tables), the vast majority of the instructions could be executed natively by the hardware without the need for software emulation.

As the Hi-Fi emulator we used a slightly earlier version of Bochs (2.4.5), the latest available at the time we started working on this project. We slightly customized this emulator to ease symbolic execution (e.g., we disabled the devices and the user interface).

We generated test cases using virtual machines running on Amazon EC2. We then used the same virtual machines to run the test cases in QEMU and Bochs and to compare their behaviors. The generation of the test cases required 545.4 CPU hours on 3 8-core instances on EC2 (total cost was about 135 US dollars in Amazon EC2 charges during the summer of 2011). Generation is highly parallelizable, since the bulk of its execution cost lies in the invocations of the solver, and multiple paths can be explored at the same time. We estimate that, with proper scheduling, test-case generation would take about 33.0 hours on 3 instances.

Test-case execution took totals of 198.7, 391.9, and 48.5 CPU hours on QEMU, Bochs, and the real hardware, respectively, and results comparison took 175.9 CPU hours. Test execution is also highly parallel, but our real-hardware testing approach is incompatible with EC2's para-virtualization; for the present results we used a local workstation. By combining 13 EC2 instances and 3 bare-metal instances from another provider, and accounting for the network transfer between them, we estimate that a complete set of test executions and the comparison of their results would take 7.8 hours and $100.19. This is already fast enough to use for nightly regression testing, so we believe that execution time is not a limiting factor for our approach or the PokeEMU prototype.

Our system was able to identify several differences in the behaviors of the emulators, some of which were not known before. We argue that our system can successfully be used in the future to validate the implementation of the currently missing security features in QEMU (i.e., the enforcement of segments' limits and rights) and the other issues (such as those caused by the lack of atomicity during emulation) we found.

6.1 Completeness of the Testing

To generate test instructions we explored the instruction set using a 15 byte input buffer. The first three bytes of this buffer were made symbolic (for the reasons explained in Section 3.2) and the remaining ones were set to zero. We identified 68,977 candidate byte sequences encoding valid instructions and then selected 880 unique instructions. This set of instructions covered all the unique instructions supported by the emulator, with the exception of a few SIMD instructions whose opcodes are longer than three bytes; we also excluded floating point instructions since our symbolic execution engine does not support them.[5]

We used each of these instructions to explore the machine state-space and to generate test programs. For the exploration we treated the entire machine state as symbolic, with the exception of the bytes in memory representing pointers (as shown in Figure 3), the FPU state, the MMX registers, and the contents of the interrupt descriptor table. As concrete inputs we used a snapshot of the

[5] Some of the techniques used for floating-point equivalence checking by Collingbourne et al. [11] might help us remove the floating-point restriction from PokeEMU in our future work.

baseline machine state. For each test instruction we executed the emulator until we explored all paths or we reached a limit on the maximum number of paths (currently 8192).

In this setting, our system explored 610,516 different paths. We observed that the number of explored paths per instruction mainly depends on the type of instructions and on the type of operands (e.g., whether the operand represents a register or a memory location). We exhaustively explored the machine state-space for about 95% of the instructions. The remaining 5% of the test instructions were not exhaustively explored because we either hit the limit on the maximum number of paths or because of a limitation of our current concretization strategy. Thus, for the exhaustively explored instructions, our system generated test programs that covered *all the possible behaviors* of the Hi-Fi emulator that can be triggered by varying the symbolic machine state. (It does not follow that our tests achieved 100% block or branch coverage of all of the code within the exhaustively-explored instructions, because for instance code that would only execute outside of protected mode was not included in the exploration. But in the cases we examined manually the static coverage appeared very high.)

6.2 Analysis of Differences

Overall, we observed quite a high number of differences: out of the 610,516 test programs generated by our system, 60,770 of these programs produced distinguishable behaviors in QEMU and 15,219 of them produced distinguishable behaviors in Bochs.

Not all discovered differences represent a distinct bug. Some are caused by undefined CPU behaviors; since those behaviors are undocumented, there may be no single correct behavior (and even different physical CPUs may produce different results). Among the remaining differences, many are imputable to the same root cause. We used scripts to filter out differences due to undefined behaviors (we reused filters from our prior work [20, 21]). We then clustered the differences according to root cause; this clustering identified different executed paths that triggered the same behavior difference. We then examined representative tests to understand each root cause. In the remainder of this section we briefly summarize some of the root causes we identified, and we discuss their implications.

Hardware CPUs execute instructions atomically. On the other hand, in a software emulated CPU, the execution of an instruction requires executing multiple instructions on the real hardware. Thus, to emulate the execution of an instruction atomically, special care is needed to ensure that the original state is preserved (or restored) when the execution of an instruction is interrupted by an exception. Non-atomic execution of instruction can produce incorrect program behaviors and open opportunities for attacks. The test programs generated by our system confirmed that both emulators execute the majority of the instructions atomically. However, our system identified instructions for which the atomicity property is not guaranteed in QEMU. More precisely, our system found that this problem occurs with the instructions `leave` (high level procedure exit) and `cmpxchgw` (compare and exchange). The former corrupts the stack pointer when the page containing the top of the stack is not accessible. The latter corrupts the source operand when the destination operand represents a memory location and this memory location is read-only. Indeed, the lack of write permissions is detected only after the source operand has been updated, and the original value of the source operand is not checkpointed. We speculate that such issues, although not easy to trigger, might lead to serious security consequences.

Paging and segmentation are the two main security mechanisms provided by the CPU; an emulator has to support these mechanisms to be considered trustworthy. Our system identified that QEMU does not implement segmentation properly because it does not enforce segment limits and rights with the majority of instructions, which can have serious security implications. The lack of segmentation support renders security mechanisms that rely on this feature [28] completely useless. This problem is known to QEMU's developers and was previously, in part, found by applying random fuzzing to manually written test programs [21]. However, our system was able to generate test programs to exercise all the checks the CPU could possibly do and to identify all the cases in which limits and rights are not properly enforced. Thus, the test programs we have generated can be used again in the future to validate the implementation when this currently missing feature is available.

We also found other less dangerous discrepancies in the behaviors of the tested emulators. For example our system generated test programs showing that QEMU does not raise a general protection fault exception when the `rmsr` (read machine specific register) instruction is used to read the value of an invalid machine status register. Moreover, our system found that QEMU, Bochs, and the hardware fetch data from memory in different orders. For example, the order in which the emulators pop items from the stack while emulating the `iret` (interrupt return) instruction differs (QEMU accesses stack items from the outermost to the innermost, Bochs and the hardware in the opposite order). Similarly, for the `lfs` instruction (load far pointer), Bochs fetches the two operands from the memory in the opposite order as QEMU and the hardware. This difference could cause different exceptions. Again, we found that QEMU does not consider valid certain instruction encodings and that it does not properly update the segments' "accessed" flag. Finally, both in QEMU and Bochs, some arithmetic and logical instructions differently update some status flags (documented as undefined). Since emulators are widely used to dynamically analyze malicious software, malicious developers could embed into their software anti-emulation tricks that fingerprint emulators by exploiting these subtle differences in their behaviors.

Many of these differences would have been difficult to find using purely random testing, and were in fact not found by a previous such study [20]. For instance, the difference in `iret` read ordering can be significant only if the values read lie on different pages or across a segment boundary, either of which would have a very low probability if the address and segment limit were chosen uniformly at random. Random testing can generate tests more quickly than PokeEMU, but this would leave the cost dominated by the time to execute the tests.

7. Limitations and Future Work

By necessity, our work reduced the size of the problem by narrowing its scope. We examine the resulting limitations and the direction of our future work next.

Other operating modes and extended instruction sets. Although x86 CPUs support multiple operating modes, we focus on testing only the 32-bit protected mode with paging enabled. Other operating modes (e.g. real and virtual 8086) are more prone to buggy behavior, since they are less commonly used. Our system could be easily extended for testing these operating modes as well. We plan to do that in the future. We also plan to extend our system to support floating-point, MMX, and SSE instructions.

Multiple-Instruction Sequences. We focus on testing each instruction separately, rather than sequences of several instructions together. In principle, doing so is completely sufficient if we can construct an initializer for every possible machine state, and the execution of every instruction is independent, properties that have held in our experiments so far. Under these observations, any difference caused by a multi-instruction sequence can be divided into one or more single-instruction differences.

In practice, however, emulators may themselves compose individual instructions incorrectly, especially in the case of QEMU, which performs dynamic binary translation for multi-instruction sequences. In our future work, we plan on studying how multi-instruction sequences are treated by emulators.

Symbolic Execution of JIT Compilers and Hardware Specifications. We have based our system on binary-level symbolic execution so that in the future we can apply it to emulators based on just-in-time compilation, such as QEMU. For example, it would be interesting to perform the converse of the comparison in Section 6 by generating tests from QEMU and using them to evaluate Bochs. Since Bochs is generally more complete, our expectation is that this would produce only a few more differences than our current experiments, but it is important if there are cases where QEMU implements a check and Bochs fails to.

In the limit, it may be possible to apply our path-exploration lifting methodology to the highest-Fi emulator there is: the hardware specification itself. Although we have no hope of obtaining (and publishing about) specifications of commercial hardware, it might be possible to apply this methodology to open-source hardware architectures, like the SPARC Leon processor.

Before we reach that desirable remote limit, we hope to study higher-level interpreters, e.g., for high-level languages such as Java.

Other Virtual Machines. We currently make some use of source code to simplify the workflow of our study, but our binary approach allows us to tackle emulators for which we have no source code at all, e.g., commercial virtual machine monitors that incorporate emulation in one or more execution modes. To facilitate this, we would like to further automate the process of determining which host locations hold guest machine state. For instance the location of `%eax` is the one where the emulator writes 42 when executing the instruction `mov $42, %eax`.

Equivalence Checking. Despite its promise, our approach only provides tests, not proofs of correctness. A further direction to improve the completeness of our emulator checking would be to perform a complete equivalence check between our set of symbolic execution results. Starting with a single Hi-Fi emulator path, we could identify all paths in the Lo-Fi emulator exercised by the same input states. Then we could symbolically combine the results for all Lo-Fi paths into a single large formula (as in the summary-building technique described in Section 3.3.2). Then we would check with a decision procedure whether the formula for the single Hi-Fi path is equivalent to the formula for the few Lo-Fi paths on all possible inputs. It may be difficult to make such an approach scale to all instructions, but when it works it provides a very strong statement about the absence of differences. This has been tried successfully for smaller, restricted programs, like processor microcode [2].

8. Related Work

Next we discuss two classes of previous research that are related to our work here: first, other projects that have searched for bugs in emulators, and then other systems for symbolic execution.

Testing of Emulators. Emulator authors presumably perform testing internally, but there has been relatively little research on techniques to make that testing more automated and effective. A series of two recent papers by Martignoni et al. show the practical value of third-party comparative testing of emulators. They first tested CPU emulators specifically, with randomly generated instructions [20]. Later they tested whole-system virtual machines (based on emulation and other technologies) using hand-written templates that were then automatically expanded to create a larger number of instruction sequences [21]. To generate a set of legal instruction byte sequences (the same challenge we face in Section 3.2), they perform a concrete exploration using the CPU as a black-box correctness oracle. They also execute tests using techniques similar to the ones we describe in Section 4: either with a user-space program [20] or a custom-written kernel [21]. However, random testing on its own does not provide the same kind of coverage guarantees that symbolic execution does. First, PokeEMU completed test generation with measurable path coverage: complete path coverage for 95% of the tested instructions, a precise quantitative measure of coverage, which random-testing methods cannot provide. Second, as shown by the comparison of Section 6, our approach revealed some bugs that these previously state-of-the-art studies based on random testing did not find. Therefore, we consider PokeEMU a demonstrated improvement over the state of the art.

Symbolic Execution. Though our primary motivation in this work is the practical problem of trustworthy emulation, our results there are made possible in part by improvements in the underlying technology of symbolic execution.

Symbolic execution was first proposed in the 1970s [18]. It has been the subject of renewed interest in the last decade thanks to a new generation of approaches [7, 15] and advances in constraint solving and increased computing power that have allowed it to be more widely applied. We can classify symbolic execution systems according to the relationship between concrete and symbolic execution. In systems that are called trace-based, dynamic, or concolic [26], the program chooses branch directions based on a concrete input, but records a path so that it can generate an different input later. By contrast online systems, of which FuzzBALL is an example, maintain symbolic values without a corresponding concrete value, and so can be free to choose either direction at a branch.

Another online symbolic execution tool is KLEE [6], which generates test cases for C programs using a symbolic interpreter for LLVM byte code. KLEE is similar to FuzzBALL in many ways, but has two key design differences. First, KLEE "forks" and maintains multiple execution states at once when both sides of a branch are feasible, whereas FuzzBALL executes just one path to completion and returns to other paths later. Second, KLEE's symbolic constraints can contain array expressions, while FuzzBALL avoids them by choosing concrete values for indexes. KLEE's approach produces fewer execution paths, but it requires additional knowledge and assumptions about the way a program manages memory. Also, decision procedure queries that contain large arrays can be significantly difficult to solve. Though a more symbolic approach could be added to FuzzBALL, our current approach works sufficiently well for many applications, including the present one.

Particularly for security applications, it is important to be able to perform symbolic execution at the binary level, as we do. SAGE [16] is a trace-based symbolic execution system for x86 that is used for extensive testing within Microsoft, but is not publicly available; SmartFuzz [22] is open-source and based on Valgrind. However trace-based systems tend to be geared to exploring just a few paths in a program, rather than the exhaustive exploration we perform. Another capability that is important in some security applications is to be able to symbolically execute a program in the context of a complete operating system. In a trace-based tool one can collect traces with a whole system emulator, but maintain symbolic information for a single process, as in the BitFuzz [5] system, based on QEMU. Most recently, S^2E [10] is an online system that integrates KLEE with QEMU, allowing more flexible combination of symbolic and concrete execution across multiple components. However, our emulators do not make significant use of the operating system when executing instructions, so a lighter-weight single-process approach was appropriate for us.

Some of the optimizations we perform are also related to previous approaches in symbolic execution. For instance, the technique of summary construction described in Section 3.3.2 is similar in spirit to compositional symbolic execution techniques [14].

It is important to distinguish our work on validating processor emulators (using symbolic execution) from work on using processor emulators (possibly to implement symbolic execution). For example, Anubis [3] uses a CPU emulator (based on QEMU) to analyze malware and can perform symbolic path exploration [23]. Similarly, Minesweeper [4] uses an emulator to discover trigger-based behaviors in malware. Although our work concerns itself with security, emulators, and symbolic execution, it aims instead to provide assurances that the emulator itself is correct. The challenges of executing an emulator symbolically may have some similarities to executing malware symbolically. However, we have two additional problems to address: mapping a CPU state identified by symbolic execution to a sequence of instructions that allow to reach the state, and identifying anomalous behaviors in an emulator.

9. Conclusion

We perform high-coverage testing of emulators by using binary-level symbolic execution to explore the space of legal instructions and machine state that could influence their execution in a high-fidelity emulator (Bochs). The system generates a test, in the form of a bootable disk image, for each of the 610,516 explored paths. We can use these tests to lift the exploration to test a low-fidelity emulator (QEMU), and to cross-validate with a hardware processor. In analyzing the differences, we see that many reveal systematic implementation oversights, such as atomicity violations in QEMU: the tests will be valuable both for understanding the failures and verifying that the problems have been fixed. Though this is only the first application, it demonstrates a practical and powerful tool to make emulators more trustworthy.

Acknowledgments: We thank the anonymous reviewers, as well as our shepherd, Jim Larus, for their comments and helpful suggestions. We are grateful to Jim Grundy for his detailed feedback on our drafts. This work was in part supported by a gift from Intel Corporation, by the Air Force Office of Scientific Research (AFOSR) under MURI award FA9550-09-1-0539, by the Air Force Research Laboratory under grant no. P010071555, by DARPA under award HR0011-12-2-005, and by the National Science Foundation under grants CCF-0424422 and 0842695. Our use of Amazon EC2 was supported by the Amazon Web Services in Education research grant program. Any opinions, findings, and recommendations expressed herein are those of the authors and do not necessarily reflect the views of Intel or the US Government.

References

[1] Advanced Micro Devices. AMD64 virtualization: Secure virtual machine architecture reference manual. AMD Publication no. 33047 rev. 3.01, 2005.

[2] T. Arons, E. Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Vardi, and L. Zuck. Formal Verification of Backward Compatibility of Microcode. In *Computer Aided Verification (CAV)*, 2005.

[3] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *European Institute for Computer Antivirus Research (EICAR)*, 2006.

[4] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In W. Lee, C. Wang, and D. Dagon, editors, *Botnet Detection*, volume 36 of *Advances in Information Security*. Springer, 2008.

[5] J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song. Input generation via decomposition and re-stitching: Finding bugs in malware. In *CCS*, 2010.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[7] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Model Checking Software (SPIN Workshop)*, 2005.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. Engler. EXE: automatically generating inputs of death. In *CCS*, 2006.

[9] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *ASPLOS*, 2008.

[10] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[11] P. Collingbourne, C. Cadar, and P. H. J. Kelly. Symbolic Crosschecking of Floating-Point and SIMD Code. In *EuroSys*, 2011.

[12] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[13] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV)*, 2007.

[14] P. Godefroid. Compositional dynamic test generation. In *POPL*, 2007.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

[16] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security (NDSS)*, 2008.

[17] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection using Demand Emulation. In *EuroSys*, 2006.

[18] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[19] Kernel-based Virtual Machine (KVM). `http://linux-kvm.org/`.

[20] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU emulators. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[21] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing system virtual machines. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2010.

[22] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *USENIX Security Symposium*, 2009.

[23] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy (Oakland)*, 2007.

[24] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), 2006.

[25] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[26] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *European Software Engineering Conference / Foundations of Software Engineering (ESEC/FSE)*, 2005.

[27] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *International Conf. on Information Systems Security (ICISS)*, 2008. Keynote.

[28] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy (Oakland)*, 2009.

[29] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. Neon: system support for derived data management. In *Virtual Execution Environments (VEE)*, 2010.