Fast PokeEMU: Scaling Generated Instruction Tests Using Aggregation and State Chaining

Anonymous submission

Abstract

Software that emulates a CPU has many applications, but is difficult to implement correctly and requires extensive testing. Since a large number of test cases are required for full coverage, it is important that the tests execute efficiently. We explore techniques for combining many instruction tests into one program to amortize overheads such as booting an emulator. To ensure the results of each test are reflected in a final result, we use the outputs of one instruction test as an input to the next, and adopt the "Feistel network" construction from cryptography so that each step is invertible. We evaluate this approach by applying it to PokeEMU, a tool that generates emulator tests using symbolic execution. The combined tests run much faster, but still reveal most of the same behavior differences as when run individually.

CCS Concepts •**Software and its engineering →Software testing and debugging;**

Keywords Symbolic binary execution, CPU emulators, cross-validation

ACM Reference format:
Anonymous submission. 2018. Fast PokeEMU: Scaling Generated Instruction Tests Using Aggregation and State Chaining. In *Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Williamsburg, Virginia USA, March 2018 (VEE'2018),* 13 pages.
DOI: 10.475/123_4

1 Introduction

Emulators are widely used in a variety of systems as an approach to provide transparency between different operating systems or processor architectures. For instance, developers of mobile applications use emulators to develop software for the ARM CPU architecture on more powerful x86 workstations. Malware analysis uses emulation to execute untrusted software in an isolated and controlled environment and to support features like whole-system taint analysis [10, 21]. Software emulation supplements hardware virtualization for certain instructions [1], or to provided flexible nesting [8].

While high-quality CPU emulators have many uses, developing one is a difficult undertaking. For compatibility with the software that runs on it, an emulator must match the full behavior of a real system. Processors like modern x86 CPUs have complex specifications that run to thousands of pages of documentation, with many instructions and backwards-compatibility features. Developing an accurate software re-implementation of this behavior is a large task. Another major source of complexity is optimization techniques: to achieve the most efficient execution, emulators deploy techniques like just-in-time compilation and compiler-style intermediate representations. Optimizations can add even more special cases that must be handled correctly. Thus it is still common to find bugs even in widely used emulators, especially in aspects of CPU behavior that are less frequently exercised.

In the abstract, testing an emulator is straightforward: one creates tests that cover the relevant aspects of CPU behavior, executes those tests on both an emulator and a real hardware platform, and checks whether the results are the same. Since the large number of tests required would be costly to create by hand, several previous research projects have explored automated test generation and execution techniques for emulators or other virtual machines. However several limitations of these systems stand in the way of their widespread use. Some systems can be used only with user-space instructions [15], or require substantial manual effort in specifying interesting test cases [17]. Among highly-automated, whole-system approaches like KEmuFuzzer [14] and PokeEMU [13], a key limitation is the speed with which the generated tests can be executed. Typically the test suite for an emulator will only need to be re-generated occasionally, but we would like to execute the generated test suite efficiently. When tests run faster, developers can run tests more frequently and/or run more comprehensive test suites, allowing bugs to be found sooner. When bugs are found and fixed sooner, the cost of doing so is less because the code is fresher on developers' minds and less other code will need to be reworked.

A sweet spot for a system such as the open-source emulator QEMU is to rerun a comprehensive relevant test suite nightly or after each commit (these frequencies are similar; in the past 5 years, the software x86 emulator of QEMU was updated on average 231 times per year). Of course latency can also be improved by running tests in parallel, but in a parallel setting more efficient execution allows the same latency

VEE'2018, Williamsburg, Virginia USA

DOI: 10.475/123_4

with fewer CPU resources. And if latency and CPU resources are both held constant, more efficient test execution allows us to run more test cases and so achieve higher coverage. Allowing tests to take up less space can help them run faster, as well as easing other tasks such as the management of old test results. Therefore, the key challenge we address in this paper is how to generate a test suite that can exercise CPU instructions with millions of controlled test inputs, but that can still run and be checked for correct results using little time and space.

We build on PokeEMU [13], an existing tool for testing x86 emulators that generates test cases using symbolic execution. Formerly PokeEMU booted an emulated virtual machine for each instruction test. Instead we explore techniques for combining many instruction tests into a single program that runs in one virtual-machine session, but with the property that the failure of any individual test can still be detected in the final test results. The result of our Fast PokeEMU system is tests that retain the coverage properties of PokeEMU's tests, but which produce their results much more quickly. We can also supplement PokeEMU's targeted tests with a large number of random tests, to further improve coverage.

PokeEMU embeds each instruction test into a bootable virtual floppy disk image, which executes the test whenever booted in a x86-compatible virtual machine. This design makes the approach easily portable for testing different emulators, but we observe that most of the time in testing is spent booting the virtual machine before the test and collecting an image of its memory state after the test (used for results comparison). Our overall approach is to keep this same structure of a testing session, but to execute many more instruction tests in between the boot and the final state collection, to amortize the cost of the non-instruction parts of test execution. The first step, which we refer to as "simple aggregation," is to take the code for several instruction tests as generated by PokeEMU and combine them into a single program. To achieve this, each test has to clean up its side-effects to the machine state after executing (returning most registers and memory to what we call a "baseline state"), and save the instruction outputs in an otherwise-unused area of memory.

The tests that PokeEMU generates with symbolic execution give high coverage, by construction, of the emulator from which they were generated: PokeEMU generates a test to cover each feasible execution path through a high quality implementation. However this property does not fully transfer when the same tests are used on another emulator: for instance if the original emulator is missing a check, the tests may not exercise it. For this reason we also supplement the symbolic-execution-based tests with tests that randomize the instruction inputs. Random tests also have the advantage of running quickly because they only need one copy of the test, a loop, and a way to generate (pseudo-)random inputs.

However, especially when we combine aggregation and repeated random tests, the amount of space used by all of the test results becomes a limiting factor. Our current implementation stores the test results in the limited-sized RAM of a virtual machine, but increasing this memory size or sending the results over a virtual I/O device would still involve transfer costs and storing the results somewhere. The common case is tests that pass, so we would like to combine the results of many instruction tests into a small area of memory, which will have the same contents when the tests run correctly, and with high probability will be different if one or more tests produce the wrong results.

Our approach for combining the test results is to chain instruction tests together. Rather than each test having a separate input and a separate output location, we connect the output of one test to be an input of a subsequent test. A naive approach to such chaining could cause instruction failures to be missed. For instance, consider a test with two instructions, where the first should generate the output 20, and the second instruction is a right shift by two bits. The expected result of the chain of these two instructions would be 5, but the result would also be 5 if the first instruction incorrectly produced 21, because 21 >> 2 is also 5. This problem would not occur if the second instruction were an invertible operation like increment, but of course we would like to be able to test all machine instructions, not just those that are invertible.

Another naive approach would be to simply XOR the results of all the tests together in the same output location. Because XORing implements a bijection, any change in the output of a single test would produce a change in the final output. But this approach has the problem that for instance if the same bit difference appeared in an even number of tests, they would cancel out and the final result would be the same.

In order to combine the results of all the tests, but make cancellation unlikely, we chain our instructions together in an more complex way based on the Feistel network construction from cryptography, which produces an invertible transformation out of arbitrary functions. This ensures that the effect of any single instruction failure is visible in the final test output, and that multiple changes are detected with high probability, without the output size needing to grow as more tests are aggregated. The Feistel network can also serve as the random number generator for random testing, and it produces a final output that is only twice the size of any individual test output.

To evaluate the benefits of our approach, we reproduce and extend a previous experiment using PokeEMU to compare the behavior of the QEMU emulator against real Intel x86 hardware with KVM hardware virtualization. Comparing the results of PokeEMU-style individual tests with Fast PokeEMU's Feistel-aggregated versions of the same tests, Fast PokeEMU's test suite ran much faster but revealed most of the same behavior differences.

```
int test0(short a, short b)
{
        return (a + b);
}

int test1(short a, short b)
{
        return (a << b);
}

...

int test99(short a, short b)
{
        return (a * b);
}
```

Listing 1. Single test case

Contribution In summary, this paper makes the following contributions:

- We demonstrate that aggregating many instruction tests into a single program allows other testing costs to be significantly amortized.
- We introduce the use of a Feistel network to aggregate test cases, which allows many tests to be combined within limited space, without sacrificing coverage.
- We implement our techniques in an enhanced Fast PokeEMU system which generates tests for x86 emulators such as QEMU.
- We evaluate the performance and results of Fast PokeEMU on a comprehensive x86-32 instruction test suite executed on QEMU (software emulation) as compared with KVM (hardware).

2 Overview

In this section, we give an intuitive walkthrough example to explain our main idea before involving details about emulator testing.

Assuming we want to test a list of 100 arithmetic functions, as shown in Listing 1. One simple approach would be to compile each function into its own binary, and check that each program gives the correct answer. However this would be inefficient, because there is significant additional overhead in running each program.

Instead of compiling each test function into a separate binary and testing each of them, more efficient is to create one big binary containing all 100 functions to test them all together. The idea of our Feistel aggregation technique is shown via source code analogy in Listing 2. We execute each test function, but instead of using separate inputs and outputs, the inputs of each test function are derived from the outputs of the previous test, mixed together with previous state using a combination of swapping and XOR called a Feistel network (we give more details about the Feistel construction in 4.3.1.) The mixing means that no test output is overwritten: they all have some effect on the final result. In particular each step of the loop is an invertible transformation on the state, so if there is a difference in the output of a single test it is guaranteed to be reflected in a change in the final output. In this way, we aggregate 100 tests into one large test, and can tell whether all the tests have passed by comparing one 8-byte value instead of 100 4-byte values.

```
void (*test_ptr[100]) = {test0, test1, ... test99};

long long aggreg(short a, short b)
{
    int i;
    int r = a + (b << 16);
    int l = 0;

    // Feistel construction
    for (i = 0; i < 100; i++){
        short x = r;
        short y = r >> 16;
        int output = (*test_ptr[i])(x, y);
        int r2 = l ^ output;
        int l2 = r;
        r = r2;
        l = l2;
    }

    return (r + (l << 32))
}
```

Listing 2. Aggregated test cases

3 Background

This section provides background on some technologies used in PokeEMU and Fast PokeEMU. First we briefly introduce symbolic execution, and the binary-level symbolic execution tool FuzzBALL, which are used for test case generation by PokeEMU and Fast PokeEMU. Then, we discuss the operation of vanilla (i.e., non-Fast) PokeEMU in more detail.

3.1 Symbolic Execution

Symbolic execution is a technique that is widely used in software testing. Instead of concrete inputs, a symbolic execution tool executes the tested program with symbols (also called symbolic variables). Symbolic execution gives the outputs of the program as expressions over symbols and concrete values, called symbolic expressions. These symbolic expressions precisely summarize the behavior of the tested program.

Symbolic execution is made more complicated when code has branches that depend on the symbolic input. For each branch, the program checks a formula called a branch condition to decide which side to jump to. By making a decision at each branch, we say the program takes an execution path. For any particular execution, symbolic execution collects a list of possibly-negated branch conditions, collectively called a path condition. The path condition is a formula over the

symbolic inputs, which holds for inputs that would cause the program to take that same path. This means, for instance, that any solution to the formula gives a test case for that path.

3.2 FuzzBALL

The FuzzBALL[1] system for binary symbolic execution is at its core an interpreter for machine code (x86-32, x86-64, ARM) in which any value stored in a register or memory can be a symbolic expression. FuzzBALL operates by dynamically translating each machine code instruction into the BitBlaze [19] Vine intermediate language, and then interprets that representation to perform the instruction's behavior.

FuzzBALL explores one execution path at a time. It maintains a decision tree, a binary tree that records all the previously explored paths, to avoid exploring any path more than once. Each decision tree node represents one occurrence of a branch, and the node can have a "true" child, a "false" child, or both, depending on which directions of the branch are feasible in that context. The "true" child is the next symbolic branch that will be encountered if the branch condition is true, and similarly for the false child. Each route from the root node to a leaf corresponds to a path of the explored program, and a collection of all the branch conditions along this route is the path condition of this path. FuzzBALL only explores a node if it still has unexplored descendants. Therefore, it is guaranteed that FuzzBALL will only explore each feasible path once.

FuzzBALL checks the feasibility of each branch condition and its negation to prune infeasible paths. Each time FuzzBALL reaches a new branch, it will compute the feasibility of both the true side and the false side of this branch. If both sides are feasible, FuzzBALL will randomly pick one side to explore, mark the other side as "feasible," and leave the other direction for future exploration. If there is only one feasible child, FuzzBALL will tag the other side as infeasible and never explore it again.

FuzzBALL uses an SMT (satisfiability modulo theories) solver to decide branch feasibility. Given a list of constraints, an SMT solver tries to figure out whether any solution ("satisfying assignment") exists, and if so it produces one. FuzzBALL is compatible with STP [9], Z3 [6], or other SMT solvers that support the SMT-LIB 2 interface format (the experiments reported here use Z3). FuzzBALL queries those solvers with symbolic expressions, translated into their appropriate syntax, and then parses the result.

A special case of branches is branching on word-sized expressions. The branch condition of a two-way branch is a Boolean expression that evaluates to either true or false. But some other kinds of control flow, like a jump table implementing a C `switch` statement, correspond to a larger number of choices. FuzzBALL reduces to the Boolean case by branching on each bit of the expression representing the choice, starting with the most significant. For instance a jump table with 10 entries will be expanded into a sub-tree of two-way choices within the decision tree with 10 feasible paths.

3.3 PokeEMU

PokeEMU is an emulator testing framework that generates high coverage test cases automatically, and compares an emulator with a real machine by running the test cases on both of them. It generates test cases by binary-level symbolic execution of an emulator that is chosen for high fidelity (currently Bochs[2]). Then the generated test cases can be used with any other compatible emulator. Previous work used PokeEMU's generated tests to compare Bochs and QEMU[3] to each other and to a hardware-based virtual machine using KVM in which most instructions run on the real processor. For simplicity we will focus just on comparing QEMU (in its "TCG" binary translation mode) against KVM in our experiments.

Before creating test cases, PokeEMU first automatically generates a list of instructions to be tested. For this purpose, it runs Bochs under FuzzBALL with the first three bytes of an instruction symbolic (enough to cover all opcodes). Of course information about x86 instructions is also available from other sources, but this step would be important for an emulator for an architecture that lacked a machine-readable instruction set specification.

For each instruction, PokeEMU generates a list of a test cases that are machine states (contents of registers and memory) in which to execute that instruction. The goal is that the test cases should cover as many behaviors of the instruction as possible. In this step, PokeEMU again symbolically executes Bochs, but now the instruction is concrete and parts of the CPU state are symbolic. For PokeEMU one chooses a subset of the full CPU state to be symbolic, which typically includes most of the registers, flags, and memory contents, but excludes some pointer-like data whose actual value is not likely to affect behavior. (For instance the tests always have the top-level page directory at a particular address, since it is the contents but not the location of the directory which is significant.) FuzzBALL explores all the paths caused by branching on those parts of the CPU state, and generates one test case for each. In more detail, FuzzBALL's results for each execution path consists of a testing input set that assigns values to bytes in memory or registers; any locations not mentioned stay in their baseline state. As a simple example, if a testing input set consists of just "in_reg_ESP__4_0=0x9", we can take the corresponding path if we replace the lowest byte of `%esp` with `0x9` and keep the remaining bytes in the CPU state unmodified.

[1] https://github.com/bitblaze-fuzzball/fuzzball

[2] http://bochs.sourceforge.net/

[3] http://www.qemu.com/

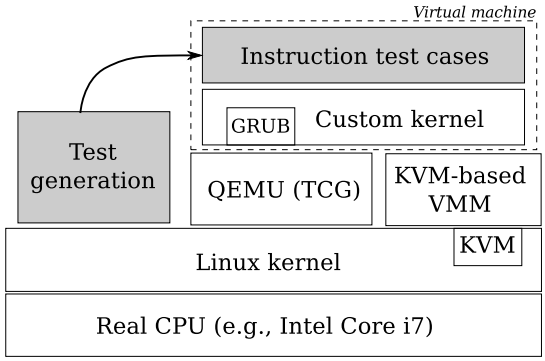

Figure 1. Architecture overview of (Fast) PokeEMU. We automatically generate instruction test cases which run inside a virtual machine supported by an emulator or a hardware VMM.

PokeEMU then converts each test input set into a virtual floppy disk image that can be used to boot QEMU (or another emulator) and run the test. The disk image uses the GRUB bootloader to start a small custom "kernel" that initializes the machine to the baseline state and then runs the test case. The test case makes the state changes to turn the baseline state into the desired test state, and the executes the tested instruction. After the tested instruction completes normally or raises an exception, the kernel halts the emulated machine, and the emulator writes the virtual machine's memory and register state as the test result (e.g., for comparison with other emulators).

The KVM-based hardware test case execution works similarly, but to keep the KVM-based virtual machine simple it does not implement I/O hardware: it just starts its execution from a machine state snapshot right before execution of the test case. KVM uses hardware-based virtualization so that most instructions run on the real host CPU: it serves as the standard for comparison because it is as close to direct hardware execution as one can easily get with fully automatic execution.

A summary of the PokeEMU architecture is shown in Figure 1. The dashed box shows the code that executes inside the virtual-machine abstraction. If the VMM implementations were perfect, this code would always behave the same; differences in behavior are usually caused by bugs in software emulation. The shaded boxes indicate the parts of the system we enhance for Fast PokeEMU.

4 Approach

We describe our approach to efficiently aggregate test cases in this section. We use the terminology of a testing "session" to refer to the whole period from starting an emulator for testing through saving its final machine state. Thus, aggregation refers to our techniques for combining many instruction test cases into one test session.

4.1 Simple Aggregation

It is intuitive to try to run multiple test cases within one session, considering the significant time taken by booting up and other unproductive parts of a testing session. Aggregating tests into larger groups gives the most time savings, but it is also sometimes convenient to rerun just a subset of tests. Though our approach could support any choice of grouping, we have found it convenient to rerun all the test cases for a single instruction, so this is the granularity of aggregation we generally use.

The key challenge presented by simple aggregation is that the aggregated tests no long run independently from each other. In vanilla PokeEMU with one test case per session, each test case naturally starts from the baseline state. In a multiple test case session, a test case starts not from the baseline state but from the state modified as by previous test cases. This difference can be problematic because a different machine state may change what a test covers; in the worst case a test case might disrupt the machine state so severely that subsequent tests cannot execute.

To solve this problem, we generate code at the end of a test case to clean up the effect of this test case, restoring the machine close to the baseline state (Figure 3(b)). The general work-flow is to run one test case, save its output, reset changes made by this test case, and then run the next one. Each instruction output that we wish to check as a test result is copied from the place the instruction left it (e.g., a register) to an otherwise-unused area of memory (selected randomly or sequentially from an area not used by the kernel or testcase code). After that, we reset everything overwritten by the tested instruction. We also undo any CPU state changes that set up for the tested instruction by generating code to reset those same locations to their baseline values. For example, if a FuzzBALL testing input set the lowest byte of eax to 0x2, we both generate code to set the register byte before running the tested instruction and code setting this byte back to the baseline value after the tested instruction. If an instruction would normally modify control flow, like an indirect jump, we choose its operands so that control-flow is still under our control, such as by jumping to another code sequence of our choosing.

Another kind of change to the machine state is raising a hardware exception such as for a page fault or a divide by zero. Instead of an exception being the end of a testing session, we would like to treat whether or not an instruction raises an exception as another output, and then go on to test the next instruction. To do this we implement exception handlers which record information about an exception and return control to the cleanup phase of the current test case. We then treat exception information as part of the output of the test case: exception handlers store it in memory, so that we can tell whether there is any different behavior on raising exceptions or the exception type.

Generating appropriate clean up code requires our tool to know the locations that an instruction will write to. In our current implementation we use the Intel XED[4] library for this purpose, via a Python interface[5]. XED is an instruction encoding and decoding library that produces detailed information about instruction operands. Our tool uses this information to determine which written operands need to be cleaned up. We have encountered very few situations where an instruction changes part of the machine state not mentioned by XED: a rare example is the `ltr` instruction's side effect of setting the busy flag in the task's segment descriptor. To extend our implementation to other architectures without comparable libraries, we might collect operand accesses during symbolic execution of the high-fidelity emulator.

Once we can tell whether an aggregation contains test cases that reveal inconsistent behaviors, we can pinpoint the exact test case(s) using binary search. Since each test case saves its outputs in memory, we can detect inconsistency by comparing two final memory dumps, one from the tested emulator and the other from a real machine. Any difference between those two memory dumps indicates an inconsistency between the emulator and the real machine. To track down the exact test case(s) that trigger this inconsistency, we split this aggregation into two smaller aggregations, each of which contains half of the test cases involved in the original one. We then execute those two smaller aggregations, and track down any of them that reveals inconsistent behaviors by further splitting it. In this way, we finally can find a single test case that triggers inconsistent behaviors. Note that this approach works as long as we can tell whether an aggregation involves test cases that reveal inconsistency. Therefore, we can use this approach together with the memory reusing techniques discussed in the next section.

4.2 Looping and Random Testing

PokeEMU's symbolic execution produces a fixed number of test cases, and each test case requires storage for the test inputs (in our case, in the form of code that sets up the machine state), so the space needed to store the tests is a limiting factor. Since the time needed to run a test is small relative to I/O costs, we could run more tests if we could get more tests out of a limited amount of code. An obvious direction is to run code in a loop, but since CPU instructions are mostly deterministic, repeating the same instruction on the same inputs would not be productive.

Instead, we use another common testing approach and execute instructions with inputs chosen (pseudo-) randomly (Figure 3(c)). The test inputs chosen by vanilla PokeEMU have high path coverage on Bochs, and so also cover many code paths in a tested emulator. But they may not cover other distinctions, such as a corner case that is missing from Bochs. If we have a larger testing budget (perhaps thanks to the speed improvement of aggregation), we can spend additional instruction executions by varying instruction inputs randomly. Though simple, random or fuzz testing has a long history of finding bugs [15, 16, 20].

4.3 Reusing Memory Space

For several practical reasons, PokeEMU test cases were designed to be small and run in small virtual machines: the test case code fits in a floppy-disk-sized image, and the testing virtual machine has 4MiB of RAM. (The 4GiB 32-bit virtual address space can be conveniently mapped to 4MiB of RAM by having all 1024 entries in the top-level "page directory" point to the same second level "page table.") With this design, we can save time creating and comparing memory dumps, and 4MiB is more than enough for booting our simple kernel and running a few instruction test cases. However the memory size becomes a limiting factor if we want to aggregate many test cases and perform random testing on each of them for a large number of times using a loop: space is required both for the code to set up inputs and run the instruction, and to store the instruction outputs.

Increasing the memory size of the VMs would be one way to get around the size limitation, but it is not a fully satisfying solution because large amounts of test output data still take time for I/O, and are unwieldy to store. Though we have not implemented a variant of PokeEMU that uses large VM sizes, we can estimate the costs by extrapolating from our existing experiments. Suppose we want to apply simple aggregation and random testing to a set of 300,000 instruction tests, running each test in a loop 10,000 times. The code for each test case requires about 200 bytes, and each set of test output takes around 20 bytes. So the total code for the set suite will about 60MB, which is relatively manageable, but the total size of the output will be about 60GB, large enough that it would become inconvenient to send a full set of results over the network, or to store a month's worth of nightly results on a workstation. The (virtual) I/O of large amounts of data also has a time cost. Dumping one of PokeEMU's 4MiB virtual machine images to disk currently takes about 90ms, which is almost unnoticeable, but processing 60GB of data in the same way would take a significant part of the testing time.

Creating a test suite that runs a large number of test cases but still produces only a small amount of output will result in tests that are easier to use. In the following subsections, we discuss how we reuse data space by chaining the outputs of one instruction into the inputs of another with a Feistel network construction, and how we use the Feistel construction as a random input generator.

4.3.1 Reusing Data Space

We assume that in the common case, the key result of testing is a yes/no result telling whether or not the test passes: in our

[4] https://software.intel.com/en-us/articles/xed-x86-encoder-decoder-software-library

[5] https://github.com/huku-/pyxed

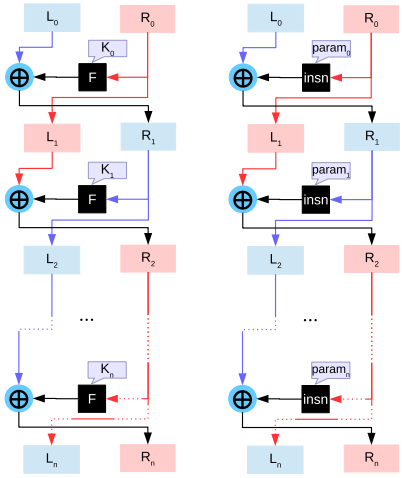

Figure 2. An visual description of the Feistel construction (left) and the way it combined with test aggregation (right). Note that `param` in the right-most diagram refers to the inputs pre-designed by symbolic execution.

case, whether or not the emulator has the correct behavior on a set of tests. When a test fails unexpectedly, it is also useful to know more details such as what instruction failed and what the unexpected output was. But since this occurs less frequently, it is not necessary for it to be supported by an aggregated test: developers can re-run a test without aggregation, or use binary search as described above, to pinpoint the cause of a failure first detected by an aggregated test. Under this assumption, we design our aggregated tests so that instead of recording the output of every instruction, they produce a result that is affected by the output of every instruction, so that comparing this shorter result between runs is enough to detect a problem.

Of course it is mathematically impossible to compress a long sequence of results into a small space without the possibility of losing some information. Since we expect that tests succeeding will be the more common case, we present a scheme were the final result is only twice as large as the output of any one instruction, and a failure is guaranteed to be detected if only one tested instruction produces an incorrect output, and the others behave as expected. If more than one tested instruction within an aggregated test produces an incorrect result, the best we can guarantee is that it will be detected with a high probability.

Our basic approach is to use the output of one tested instruction as part of the input to the next tested instruction. If every instruction were invertible (i.e., information-preserving, implementing a bijective function), this would be sufficient to detect any single failure. But because many instructions overwrite data or compute many-to-one functions, we need to chain the instructions together in a slightly more complicated way.

The construction we use was first invented in cryptography, where a similar need arises of building an invertible function (e.g., an encryption function that can also be decrypted) out of repeated application of inner functions that may not be invertible. It is called a Feistel network or Feistel cipher, after the designer responsible for its first public use in the development that lead to the cipher DES [7]. The key property of the construction for cryptographic purposes, later proved formally [12], is that if the inner function meets a definition of cryptographic strength known as being a "pseudorandom function" and there are enough rounds (at least 3), then constructed function has the analogous property of being a "pseudorandom permutation". This cryptographic definition of pseudorandomness is stronger than, but implies, the properties of (pseudo-) randomness that would be expected for random testing.

Figure 2 gives a graphical overview of the Feistel construction. The state in a Feistel network is divided into two equal-sized halves, commonly called L and R (from "left" and "right"). Suppose that the starting input (e.g. the plaintext in encryption) is L_0 and R_0, and that the smaller non-invertible functions indexed as F_i operate on a value the size of an L or R. In each round, the state evolves via the relations:

$$L_i = R_{i-1} \quad \text{and} \quad R_i = L_{i-1} \oplus F_i(R_{i-1})$$

where $\oplus$ represents bitwise XOR. A single round is invertible because R_{i-1} can be recovered from L_i, and then L_{i-1} computed as $R_i \oplus F_i(R_{i-1})$. Any number of rounds can be inverted by repeating this process.

The reason any single incorrect test output can be detected is that the function representing all future rounds of the computation is invertible. Suppose that R'_i in an incorrect execution is different from R_i in the correct execution, but that the rest of the construction implements the same function, because by assumption no other failures are triggered. Then the final output of each test is an invertible function of R_i or R'_i respectively, but if $R_i \neq R'_i$, this implies that the outputs are also different.

If there are multiple incorrect executions, this can still be detected with high probability, because of the pseudorandom mixing performed by the Feistel construction. For instance, suppose that there are two instructions which produce incorrect outputs; then there is a chance that the two differences could cancel out, leaving the final output the same. However, there is only one incorrect result from the second instruction which will conceal the failure, and because of the mixing, we can model it as a value that is selected randomly, independent of the implementation. Since we generally have a block size that is at least 64 bits, the chance of an incorrect execution generating a result that would conceal a previous error is no more than 2^{-64}.

To complete the analogy of using a Feistel network construction for aggregating test cases, Fast PokeEMU chooses an L/R block size large enough to accommodate either all

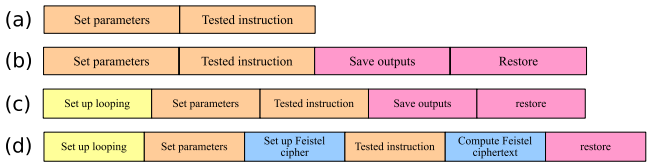

Figure 3. Change of test case structure. The 4 diagrams from top to bottom are test cases of (a) vanilla PokeEMU, (b) with simple aggregation, (c) with looping, and (d) with all three new features.

the inputs or all the outputs of any tested instruction in an aggregation. Each round function F_i copies the inputs to the locations expected by a tested instruction, executes the instruction, and copies the outputs back to an appropriately sized memory block, padding with zero bits if needed. The inputs to the first tested instruction can be arbitrary as in simple aggregation, since they are just R_0; while L_0 can just be zero.

4.3.2 Feistel construction as a random input generator

Conveniently, we do not need a new mechanism for producing random inputs, because the Feistel network construction described in Section 4.3.1 already produces unpredictable pseudo-random values from the interacting behavior of previous instructions, which we supplement with other mixing operations. To repeatedly execute an instruction with varying inputs, we simply execute it repeatedly with outputs chained to following inputs thorough the Feistel construction. The choice of which test inputs to take from the Feistel construction chooses between which parts of the CPU state are random versus set from a pre-generated test case. Commonly the test execution budget will be larger than the number of tests created by vanilla PokeEMU. In this case we use the natural combination of taking a fully PokeEMU-generated test case as the first test of an instruction, and then modifying the test with random data for subsequent loop iterations. In this way the random testing can only improve the error detection of pure symbolic execution.

5 Experiments

In this section we describe the experiments we conduct to evaluate the performance and error coverage of Fast PokeEMU's tests as compared with those generated by vanilla PokeEMU, when used to compare the behavior of QEMU and KVM.

5.1 Implementation Details

Of the 336,798 instruction tests generated by PokeEMU, we select 128,742 which do not raise triple fault or other fatal exceptions. For aggregation we group the test cases according to which of 870 instructions they test: the number of test cases per instruction varies from 1 to a configurable upper bound of 4096.

For comparison with the previous evaluation of vanilla PokeEMU, we generate tests from the same version of Bochs (2.4.5) and test a similar version of QEMU (0.12.4). To evaluate the impact of random testing with looping, we also run PokeEMU with historical versions of QEMU ranging from 1.0 to 2.4. We have ported the KVM-based test execution environment to the version of KVM in a more recent Linux kernel (4.4.0-45-generic). The tests were generated using an Intel Core i7-4770, and the KVM tests were executed on an Intel Core i5-6200U, both running Ubuntu Linux 16.04.

Though our long-term goal is to treat all exceptions similarly to other results of an instruction execution, our current implementation can only recover from some exceptions. The test cases run in kernel mode (ring 0) of the virtual machine, so our first implementation has the exception handlers run at this level as well; the test case and the exception handler share a stack which the CPU uses to store exception information. Unfortunately this design does not work if an exception occurs while the stack is unusable, for instance a page fault that occurs when an instruction tries to write to the stack. The CPU encounters an exceptional situation while trying to set up the stack for the exception handler; this then triggers a second exception handler which fails for the same reason. Instead of an infinite loop of interrupted exception handlers, Intel's design simply halts the processor in this condition (sometimes referred to as a "triple fault"). We have begun to investigate an approach to avoiding this problem by putting the exception handlers in a separate task. This is not a fundamental problem, but it has some level of complexities, and requires more engineering work. As a temporary solution, we restrict our evaluation to instruction test cases that do not raise unsupported exceptions in either QEMU or KVM when executed individually. We select these tests by running all of the tests generated by PokeEMU individually. Unless otherwise noted, reported results describe only this subset of the tests.

5.2 Performance experiment

Since our major goal is to improve the performance of PokeEMU, the very first thing to evaluate is how much faster the Fast PokeEMU is compared with the vanilla PokeEMU. In this experiment, we compare the performance of PokeEMU and Fast PokeEMU with combinations of different features.

Simple Aggregation To evaluate the performance of simple aggregation, we compare the execution time to run a group of test cases one by one against running all of them at once. For each instruction, we generate a large test equivalent to all the test cases of this instruction, and execute it. We call this large test an "aggregation" in the rest of this paper. In addition, we also run each test case of this instruction

	Mode	Total time (s)	Time per test (ms)
1	Separate	84871.8	583.528
2	Simple	334.7	2.313
3	Feistel	345.0	2.448
4	Loop (1)	345.2	2.672
5	Loop (10000)	1635.4	0.002

Table 1. Runtime performance of Fast PokeEMU test execution in QEMU. Row 1 shows the result of vanilla PokeEMU, while the row 2 through row 5 are aggregated tests combined with different features

separately and report the sum of the executing times for comparison.

Feistel Aggregation Even when the number of test cases is the same, Feistel aggregation may be slower than simple aggregation because of the overhead of maintaining and mixing extra state. To measure this, we also generate aggregations with the Feistel chain.

Looping For the final configuration of performance experiments, we use the mode of Fast PokeEMU that executes instructions with a loop as well as chaining their inputs and outputs. For a direct overhead comparison, we perform one set of runs with an execution count of 1, which is the same as Feistel aggregation without looping except for overhead. Then we also perform a set of runs with an execution count of 10000, which shows the incremental cost of executing instructions more times.

The results of our performance experiments are shown in Table 1. The most obvious performance difference is that all forms of aggregated tests are much faster than running tests separately. Because sometimes an aggregated test will fail even when its constituent cases did not, the number of cases is slightly different for the different modes, so the average time per test is the best measure for comparison. Despite the varying total number, we still can see that the Feistel aggregation and looping impose overheads over simple aggregation by comparing time per test. Because the overheads are further amortized, looping with a larger execution count can execute many more instruction tests in not much more time.

5.3 Error coverage experiment

Table 2 shows the effect of test case aggregation on whether test cases show a difference in behavior between QEMU and KVM. Recall that we aggregate the test cases in groups according to the instruction they test. When we run the tests separately, we classify a match if all of the tests show the same result between QEMU and KVM, and a mismatch if at least one shows a differing result. The aggregated result is from a single test case that is the Feistel-aggregation combination of the separate tests; it yields just one result.

The design goal of aggregation is to not affect the results of tests; to evaluate how well we achieve this, we run each set of tests in three ways. First we run each test separately, with no extra features, comparable to the previous evaluation of vanilla PokeEMU. Second we run each test individually, but with the extra code such as to support the Feistel construction. Finally we run the set of tests aggregated together using the Feistel construction (but without random testing). The ideal result would be for all treatments of an instruction test set to give the same match/mismatch results; as can be seen from the first and last lines of the table, this was the most common result.

Some of the remaining entries in other lines are likely limitations of our aggregation implementation. Unlike tests that run separately, a test in an aggregation may not start running from the baseline machine state if the side effect of previous tests is not totally cleaned up. In addition, since the test cases of PokeEMU become more complicated as we add looping and the Feistel construction (as shown in Figure 3), there are more engineering challenges in performing correct clean up.

One class of limitations we have found in analyzing these results relates to the flags in the page table and segment descriptors that are set when memory regions are accessed. Fast PokeEMU does not currently record changes to these flags as outputs of an instruction (and they are not treated as such by XED). This leads to several of the mismatch/match/-match (row 5) results, because the fact that an access bit is not set can be seen in the machine state right after the instruction executes, but is overwritten by later accesses in cleanup code or later tests in an aggregation. Specifically a problem of this sort with the accessed bit of the code segment descriptor applies to `in (%dx),%al` and likely 16 other variants of `in` and `out` instructions in row 5. Ideally the fix for this problem would be to treat the accessed bits as instruction outputs, but this would be somewhat complex because PokeEMU would need to simulate or otherwise determine which page table entry a memory access uses. It would also sometimes be tricky to read accessed bits without disturbing them, because any memory access might set some accessed bit.

Some other results show cases in which mismatches come from real behavior differences in QEMU, but which are found only sporadically because of limitations of Fast PokeEMU. One example from row 2, match/match/mismatch, is the instruction `setbe (%eax)`, which sets a byte in memory to either 0 or 1 based on a comparison result. The mismatch is caused by a limitation of QEMU that it does not enforce segment-limit checks on memory accesses: the test accesses a memory location that is below the end of a segment, which correctly causes a general-protection exception on the real processor but does not in QEMU. However the address intended in PokeEMU's test case would not have triggered the exception; in the current test results the exception happens

	Separated result	Separated result with extra code	Aggregated result	# of instructions
1	Match	Match	Match	578
2	Match	Match	Mismatch	20
3	Match	Mismatch	Match	5
4	Match	Mismatch	Mismatch	13
5	Mismatch	Match	Match	18
6	Mismatch	Match	Mismatch	0
7	Mismatch	Mismatch	Match	31
8	Mismatch	Mismatch	Mismatch	292

Table 2. Effect of aggregation on QEMU behavior difference coverage. For most instructions (rows 1 and 8), an aggregated test case gives the same result as separated tests.

Fix	Instruction	PokeEMU	Fast PokeEMU
321c535	BSF_GdEdR	*	*
	BSR_GdEdR	*	*
dc1823c	BTR_EdGdM	*	*
	BTR_EdGdR		*
	BTR_EdIbR		*
	BTC_EdGdR		*
	BTC_EdIbR		*
	BT_EdGdR		*
	BT_EdIbR		*
	BTS_EdGdR		*
	BTS_EdIbR		*
5c73b75	MOV_CdRd	*	*
	MOV_DdRd	*	*
	MOV_RdCd	*	*
	MOV_RdDd	*	*

Table 3. Historical bugs revealed by vanilla and Fast PokeEMU

only because the address in `%eax` is accidentally overwritten. (A separate test case that should have consistently caused the exception was excluded from this evaluation because it did not run correctly on its own under KVM.) After analyzing it, we plan to fix this problem by ensuring that Fast PokeEMU correctly saves and restores `%eax` when the register is used as part of a memory operand (e.g., `(%eax)`), in addition to saving and restoring the pointed-to memory location. 4 other `SETcc` instructions with memory operands in row 2 would likely also be addressed by the same fix.

5.4 Historical bug experiment

The previous evaluation of PokeEMU did not directly compare to random testing, and it did not associate discovered bugs with their fixes. Our next experiment looks at which behavior differences found by PokeEMU and/or Fast PokeEMU in an older version of QEMU can be confirmed as real bugs because they were fixed in a later QEMU version. We also evaluate which of these failures were only found with the addition of random testing to vanilla PokeEMU based only on symbolic execution.

In particular, we do binary search among a range of historical QEMUs to identify a related bug. Given a test that reveals a behavior differences on a buggy version of QEMU, we find an more recent version of QEMU in which this test no longer reveals differences. The chance is high that this bug has been fixed before or at this version. We than start binary search between the buggy version and known fixed version, until we find the first historical QEMU in which the test reveals no difference. By studying the code and comments, we can confirm whether the changes made in this version of QEMU is the fix to a bug, and whether this bug is associated with the behavior differences detected by the test.

We start the experiment by running all the aggregations on an old version of QEMU, both with and without random testing. For random testing, we set the loop count to 10,000, namely repeating each test with 10,000 sets of random inputs the instruction.

With aggregations that reveal new differences, the next step is to check whether those results are associated with bugs. For each aggregation, we rerun all the single tests it includes with random testing, and collect a single test that reveals differences when running on old QEMU but matches on the latest version of QEMU. Those tests are eligible for the rest of the experiment, since the cause of differences in old QEMU revealed by them have been fixed in latest QEMU. We then pick one eligible test for each instruction, and do binary search from the old QEMU through a more recent one, to find the first version of QEMU that reveals no difference.

The range of QEMU versions we use in the historical bug experiment is 1.0 to 2.4. The reason for this range is that PokeEMU relies on a small modification to QEMU to produce machine state dumps in the common format used for comparison, and this modification must be ported to a historical version to use it. (We also considered implementing a converter from one of QEMU's native state dump formats, but this would have had similar engineering challenges.) Most of

the porting of the changes to support the 39,685 revisions between 1.0 and 2.4 was accomplished automatically using Git merging. However the process was not fully automatic since we had to occasionally resolve merge conflicts manually or update the dumping functionality for changes in QEMU's internals. Because the merged versions were stored in a Git repository, we also found it convenient to use the "bisect" feature of Git to perform the binary search over versions.

The results of the experiment are summarized in Table 3. Fast PokeEMU's results for QEMU 1.0 reveal later-fixed behavior differences across 15 instructions as we aggregated them (7 unique mnemonics), that were fixed by three different commits (indicated by prefixes of their commit hashes in the official repository; see for instance https://git.qemu.org/?p=qemu.git;a=commit;h=5c73b75). The developers' one-line summaries of the fixing commits were "Implement tzcnt and fix lzcnt", "Preserve the Z bit for bt/bts/btr/btc", and "mov to/from crN/drN: ignore mod bits".

Many of the differences were already found by vanilla PokeEMU, but Fast PokeEMU's random testing allowed it to find the problem in the BT* family of instructions much more reliably. Intel's documentation specifies that the value of the OF flag after these instructions is undefined, and most real processors leave it unmodified, but older versions of QEMU set OF based on another bit in the bit-vector, sharing code with the way the flags are set after a shift instruction. Bochs has the correct behavior, but since the correct implementation does not require a branch, symbolic execution provides no particular coverage guarantee. On the other hand, the results show how this problem can be detected with high probability by random testing.

The number of behavior differences that this experiment can attribute to later bug fixes seems small relative to the total number of differences Fast PokeEMU reports. One likely reason is the limited range of QEMU versions we have tested so far; bugs fixed before version 1.0 or after 2.4 would not be found in this experiment. Expanding the range of covered versions is something we plan for the future: of course it would also be interesting to know what differences remain in the most recent QEMU version. Note also that the current experiment starts only with behavior differences that (Fast) PokeEMU does report, so it does not provide information about whether there are bugs that PokeEMU should find, but does not (false negatives). A further investigation of this question using historical bugs is also in our future plans.

6 Related Work

We mention here some of the most closely related work in symbolic execution and emulator testing, as well as other software testing.

Symbolic Execution Symbolic execution is becoming increasingly common as an approach to explore the behavior of software systems and generate test cases. There are a number of other symbolic execution tools that could be used for the same purpose that PokeEMU and Fast PokeEMU use FuzzBALL.

KLEE [3] may be the best-known system for symbolic execution. Like FuzzBALL it performs symbolic execution online, rather than using instruction traces, but when KLEE reaches a branch with both directions feasible, it "forks" to execute both sides in parallel, whereas FuzzBALL only ever explores one path at a time. Because KLEE operates on LLVM bitcode produced by a compiler, it could be applied to Bochs whose source code is available, but not to a high fidelity emulator available only as a binary.

S^2E [4] is a whole-system binary-level symbolic execution system which is implemented using both KLEE and QEMU. It translates instructions first into QEMU's TCG intermediate representation (IR), and then into the LLVM IR for symbolic execution. S^2E's namesake selective symbolic execution features are not needed because PokeEMU explores instruction implementations exhaustively, but it could apply to binaries in the same was as FuzzBALL.

Another more recently released binary symbolic execution tool is angr [18]. The angr system is structured as a collection of loosely-coupled libraries which can implement a variety of symbolic execution algorithms. For instance among these veritesting [2] might be useful to address challenges with path explosion in branching, such as a case in Bochs segment cache initialization that PokeEMU currently handles via a specialized procedure summary.

Emulator Testing One of the most commonly discussed challenges in emulator testing is achieving coverage of emulator behaviors. The performance of test cases, our main concern in this paper, has received relatively less attention.

The EmuFuzzer [15] and KEmuFuzzer [14] systems are predecessors to PokeEMU that perform random fuzz testing instead of using symbolic execution to generate tests. EmuFuzzer targeted only user-mode emulators, while KEmuFuzzer developed most of the infrastructure for executing whole-system tests that is also used in PokeEMU, so its test execution performance is similar to vanilla PokeEMU.

Though KVM primarily uses hardware virtualization, it also contains an emulator that is used in some uncommon situations. Amit et al. point out that bugs in this emulator are more serious that had been previously appreciated, because its use can be triggered for any instruction by an adversarial guest [1]. They propose the approach of applying a human-created test suite originally developed by Intel for testing real CPUs, and with it they discover a large number of bugs in the KVM emulator. Unfortunately such a test suite is expensive to produce, and would also be useful to competing hardware manufacturers, so such suites are usually not shared publicly. Automated techniques that can reduce the cost of constructing such tests are still required.

Another use for differences in behavior between emulators and real hardware is for malicious software to detect whether it is running on real hardware; in this context the differences are sometimes called "red pills". Inspired by this application, Shi et al. use six weeks of researcher effort to develop testing templates that capture interesting operands for x86 instructions [17]. They find that testing with these chosen inputs reveals more red pills per test than EmuFuzzer's random fuzzing.

Many emulators and binary analysis tools (including both QEMU and FuzzBALL) work by translating instructions into a simplified intermediate representation (IR). Though these IRs are typically tool-specific, some are similar enough that they can be translated into a common language for comparison purposes. Kim et al. [11] describe how to cross-compare several x86 tools of this sort using symbolic execution, and discover a number of bugs in their CPU models. When compatible IRs are available, comparing the IR expressions directly is more powerful than generating test cases as (Fast) PokeEMU does, but test cases can also be used to compare with tools with incompatible IRs, interpreters like Bochs, and real CPUs.

Resetting Side Effects Efficiently undoing the side effects of one test before executing another is a common concern in test execution in many application contexts. For instance, a performance trade-off similar to the one we address occurs in randomly testing Java methods: each test case is short, so it would be inefficient to start a fresh JVM for each test, but for reproducibility one test should not affect the next. Ways to efficiently reset static state within a single JVM have been developed for Java testing tools, such as JCrasher [5]. However JCrasher's techniques are specific to features of the JVM such as classloaders and static initialization, so they would not apply in our context.

7 Future Work

The implementation limitation of Fast PokeEMU we would like to address next is support for test cases that raise fatal exceptions including triple faults and other unsupported exceptions. We will explore handling exceptions using a different task with its own stack segment and data segments, so that record the information about one exception without triggering a new one.

8 Conclusion

Ensuring the correctness of a CPU emulator requires a large number of instruction tests, but to be practical the test suite must execute quickly. We propose techniques to speed up the execution of whole-system emulator tests by testing many instructions at the same time, and saving space by chaining test outputs to subsequent inputs. We have implemented these techniques in Fast PokeEMU, an automatic test generation system for x86-32. Compared to the prior PokeEMU system without these improvements, Fast PokeEMU's tests run much more quickly, but reveal most of the same behavior differences in QEMU.

Acknowledgments

This research was supported by the National Science Foundation under grant no. 1514444.

References

[1] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. 2015. Virtual CPU Validation. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 311–327. DOI: http://dx.doi.org/10.1145/2815400.2815420

[2] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing symbolic execution with veritesting. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 1083–1094. DOI: http://dx.doi.org/10.1145/2568225.2568293

[3] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[4] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*. 265–278. DOI: http://dx.doi.org/10.1145/1950365.1950396

[5] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An Automatic Robustness Tester for Java. *Softw. Pract. Exper.* 34, 11 (Sept. 2004), 1025–1050. DOI: http://dx.doi.org/10.1002/spe.602

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[7] Horst Feistel. 1971. Block cipher cryptographic system. US Patent 3,798,359. (1971).

[8] Alex Fishman, Mike Rapoport, Evgeny Budilovsky, and Izik Eidus. 2013. HVX: Virtualizing the Cloud. In *5th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX, Berkeley, CA. https://www.usenix.org/conference/hotcloud13/workshop-program/presentations/Fishman

[9] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *Computer Aided Verification, 19th International Conference, (CAV 2007)*. 519–531.

[10] Andrew Henderson, Aravind Prakash, Lok-Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. 248–258. DOI: http://dx.doi.org/10.1145/2610384.2610407

[11] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing intermediate representations for binary analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 353–364. https://doi.org/10.1109/ASE.2017.8115648

[12] Michael Luby and Charles Rackoff. 1988. How to Construct Pseudorandom Permutations from Pseudorandom Functions. *SIAM J. Comput.* 17, 2 (1988), 373–386. DOI: http://dx.doi.org/10.1137/0217022

[13] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration lifting: hi-fi tests for lo-fi emulators. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*. 337–348. DOI: http://dx.doi.org/10.1145/2150976.2151012

[14] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. Testing System Virtual Machines. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*. ACM, New York, NY, USA, 171–182. DOI: http://dx.doi.org/10.1145/1831708.1831730

[15] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09)*. ACM, New York, NY, USA, 261–272. DOI: http://dx.doi.org/10.1145/1572272.1572303

[16] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44. DOI: http://dx.doi.org/10.1145/96267.96279

[17] Hao Shi, Abdulla Alwabel, and Jelena Mirkovic. 2014. Cardinal Pill Testing of System Virtual Machines. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 271–285. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/shi

[18] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. 138–157. DOI: http://dx.doi.org/10.1109/SP.2016.17

[19] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper*. Hyderabad, India.

[20] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 283–294. http://doi.acm.org/10.1145/1993498.1993532

[21] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of ACM Conference on Computer and Communication Security*.