Scalable Symbolic Execution of Systems Code

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Qiuchen Yan

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Stephen McCamant

Dec, 2024

© Qiuchen Yan 2024
ALL RIGHTS RESERVED

Acknowledgements

There are many people that have earned my gratitude for their contribution to my time in graduate school.

Abstract

Systems code, such as a CPU emulator or OS kernel, is pervasive and its quality is critical for all the software built on top of it. To monitor the code quality of systems code, automated program analysis is desirable as an assistant for human experts considering the size and complexity of the code. Compared to other automated program analysis techniques, symbolic execution can reason about more complicated code, but it is more difficult to scale to large systems. To take advantage of symbolic execution while circumventing the scalability issue, we combine symbolic execution with other techniques. We also explore solutions for the path explosion problem which is the major cause of this issue.

Software CPU emulators are difficult to implement correctly and extensive testing is desirable. Since a large number of test cases are required for full coverage, it is important to execute tests 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.

Loop misuse is one of the major sources of real world program bugs. However, symbolic execution of programs with loops can be challenging to scale due to the large number of execution paths introduced by loops. To address this issue, a previous loop summarization approach has been developed on a concolic system that is not publicly available. To further evaluate this approach and combine it with other techniques, we redesign and implement this approach on FuzzBALL, a publicly available symbolic execution system. According to our evaluation, this approach can summarize linear loops, but is limited for non-linear or nested loops and loops manipulating symbolic arrays. Based on the loop summarization implementation, we implement a prototype of function summarization which currently supports strlen and its variations. Compared to

existing function summarization approaches, our approach can identify functions more accurately without the help of debug information.

A kernel is the most important OS component that supports both the entire OS and user-space applications, while it is challenging to monitor code correctness of a kernel. As the last part of this thesis, we propose a novel approach to confirm candidate bugs of the Linux kernel found by static analysis, which combines fuzzing with symbolic execution. Given code locations of a candidate bug, we start with fuzzing to collect inputs that can get close to the those location. We then perform concolic execution directed by those inputs, and switch to symbolic execution when getting close to the bug. We implement a prototype of this approach, and evaluate it with known bugs. Unfortunately, we are not able to find a working example due to various limitations. By analyze the results, we identify various reasons for the failed experiment and propose improvements for this approach.

Contents

Acknowledgements i

Abstract ii

List of Tables vi

List of Figures vii

1 Introduction 1

1.1 Motivation 1

1.2 Background 2

1.2.1 Systems Code 2

1.2.2 Symbolic Execution 3

1.2.3 FuzzBALL 5

1.3 Overview 7

2 CPU Emulator Testing 8

2.1 Introduction 8

2.2 Approach 9

2.2.1 Aggreration and looping 9

2.2.2 Reusing memory space with the Feistel Construction 10

2.3 Evaluation 11

2.3.1 Performance 11

2.3.2 Effectiveness 12

2.3.3 Historical bugs 13

2.4 Related work . . . 13
2.5 Discussion . . . 16

3 Loop Summarization **17**
3.1 Introduction . . . 17
3.2 Loop summary for concolic execution . . . 19
3.3 Approach . . . 21
3.4 Evaluation . . . 23
3.4.1 Compatible examples . . . 23
3.4.2 Examples of limitations . . . 29
3.5 Related work . . . 31

4 Loop-based function summarization **33**
4.1 Introduction . . . 33
4.2 Approach . . . 34
4.3 Evaluation . . . 36
4.4 Related work . . . 37

5 Fuzzing-directed symbolic execution for kernel bug confirmation **38**
5.1 Introduction . . . 38
5.2 Approach . . . 40
5.2.1 Candidate Reachable Nodes Collection . . . 40
5.2.2 Distance Analysis . . . 42
5.2.3 Symbolic Execution . . . 43
5.3 Evaluation . . . 44
5.3.1 Supporting experiments . . . 44
5.3.2 Known bug experiment . . . 48
5.3.3 Demonstration of extended symbolic execution . . . 51
5.4 Future Work . . . 53
5.5 Related Work . . . 54

References **56**

List of Tables

2.1 Runtime performance of Fast PokeEMU test execution in QEMU. Row 1 shows the result of vanilla PokeEMU. Row 2 through row 5 are aggregated tests combined with different features. 12
2.2 Effect of aggregation on QEMU behavior difference coverage. For most instructions (rows 1 and 2), an aggregated test case gives the same result as separated tests. 12
2.3 Historical bugs revealed by vanilla and Fast PokeEMU. The first two columns show the instruction variations that can trigger behavior difference and the QEMU commits associated with them. The remaining two columns exhibit results of pokeEMU and fast PokeEMU, where a * means the tool detects behavior differences. 14
3.1 Results of all compatible examples. All ratios have been rounded to three significant digits. 25
3.2 The major causes of limitations for SV-COMP examples under loops category 30
4.1 Summarization ratios 37

List of Figures

2.1 Architecture overview of (Fast) PokeEMU. 9

2.2 Feistel cipher (left) and PokeEMU tests (right) 11

3.1 multi-level choice in a FuzzBALL decision tree. `Use Summary` denotes the extra branch. The subtree in grey box demonstrates the multi-level choice branches when FuzzBALL decides to apply the loop summary (take true side of `Use Summary`). Alternatively, FuzzBALL can take the false side and explore the loop normally, which corresponds to the `Normal Execution Subtree`. 22

5.1 Architecture diagram of the fuzzing-directed symbolic execution system for kernel bug confirmation 41

5.2 Results of the switching point experiment. The x-axis stands for the position of the switching point function on call stack. The y-axis stands for the total running time in chart (a) and the total number of paths in chart (b). 45

5.3 Results of the edit distance experiment. 47

Chapter 1

Introduction

1.1 Motivation

Low level software such as OS kernels and standard libraries is pervasive and supports a variety of different applications. Such software is commonly called systems code. Since systems code plays an important role in the world of software, it may cause serious problems if bugs or vulnerabilities are introduced in it. Due to its size and complexity, it requires a huge amount of human efforts to detect and fix bugs in systems code. To efficiently monitor the quality of systems code, an intuitive solution is to develop automated tools that can assist developers.

Automated bug finding has become a common purpose of program analysis, and various techniques have been developed for this purpose. At a high level, those tools fall into two categories: static analysis and dynamic analysis. Static analysis targets the program code without executing it, while dynamic analysis executes the program with particular inputs generated or acquired elsewhere. As the most popular dynamic approach for bug finding, fuzzing systems execute programs with randomly generated inputs. Symbolic execution is on the borderline of static and dynamic analysis. A symbolic execution system executes the program after replacing inputs partially or completely with symbols which are variables that can represent any value, and construct constraints that represents the program. By solving those constraints, we can find inputs that satisfy complicated conditions, and reach code beyond those conditions.

A automated program analysis tool for systems code must be scalable: the tool not

only should support small programs, but should also work when the program grows in size and complexity. The state-of-art tools based on pure static or dynamic analysis are scalable. Techniques based on static analysis can scale to large systems, but the false positive rate can be high. Dynamic techniques are similarly scalable and can avoid false positives, but it may have trouble reaching certain pieces of code since the set of inputs that reach those locations would have very low probability to be generated randomly. Compared to pure static or dynamic analysis, symbolic execution trade-offs effectiveness with scalability: symbolic execution is more likely to trigger bugs that only occur under complicated conditions, but suffers from the scalability problem caused mostly by the path explosion issue (explained in section1.2.2).

As previously mentioned, systems code is complicated and bugs can be hidden under complicated conditions. Therefore, we would like to take advantage of symbolic execution to reason about complicated conditions. To achieve this goal, the major obstacle is the scalability issue. Our research focuses on the scalability issue of symbolic execution, especially when the program to analyze is systems code. We explore different approaches for this purpose, including improving symbolic execution itself to mitigate path explosion and combining symbolic execution with other techniques.

1.2 Background

1.2.1 Systems Code

As mentioned above, systems code is low level software that can be used to support a variety of different applications. For instance, we consider OS kernels as systems code, since all applications are built on top of the OS kernel. A standard library is another example of systems code, since most applications rely on library functions to some degree. A software CPU emulator is a special case of systems code, since a emulator itself is application level software. On the other hand, it is considered as systems code since it can simulate hardware, which support operating systems and all the applications running in it.

For systems code, binary analysis can be more desirable than source-based analysis due to some common properties. First, most systems code is written in a combination of high-level and low-level languages. For example, the Linux kernel is written in C and

assembly. Since source-based approaches are only applicable to the high-level language part, binary analysis is necessary if we want to cover the full program. In addition, some system code is only available in binary. Although the Linux kernel and QEMU are open source, the OS kernel of Windows and commercial emulators such as VMware are not.

1.2.2 Symbolic Execution

Symbolic execution is a widely used technique in software testing and vulnerability detection. Instead of concrete inputs, a symbolic execution system executes the tested program with symbols, which are also called symbolic variables. As a result, the outputs of a symbolically executed program are a set of expressions over symbols and concrete values instead of concrete outputs. Those expressions, which are called symbolic expressions, can summarize the behavior of the tested program precisely.

Symbolic execution becomes more complicated when code has branches that depend on symbolic inputs. Each branch is associated with a branch condition, which is a formula that represents the condition to take one or another side of a the branch. For each branch, the program checks the branch condition to decide which side to jump to. By making decisions at a sequence of branches, we say the program takes an execution path. In the rest of this thesis, we refer to this path as a program path, or path for short. For each program path, symbolic execution collects a list of branch conditions that can be negated. The collection of branch conditions for a single path is called a path condition. The path condition is a set of formulas over the symbolic inputs, which can be replaced by inputs that would direct the program to take that same path. In other words, any solution to the formula gives a test case for that path.

The way symbolic execution works can be depicted using the example shown in listing 1.1. As shown in the code snippet, function `emul_SUBT` simulates a subtract instruction written in C. Assume that we execute this function with symbolic `eax` and `ecx`, whose initial value are represented by symbols a and c respectively. To take the path that passing line 5 and line 12, the condition at line 2 and line 9 should be negated, which leads to a path condition of $(a \geq c) \wedge (a - c \neq 0)$. By solving this path condition, we can get one set of possible values over which the path condition evaluates to true, such as $a = 5$ and $c = 3$. This set of inputs or any other values that satisfy this path

```c
void emul_SUBT(struct regs *cpu) {
    if (cpu->eax < cpu->ecx)
        cpu->nf = 1;
    else
        cpu->nf = 0;

    cpu->eax = cpu->eax - cpu->ecx;

    if (cpu->eax == 0)
        cpu->zf = 1;
    else
        cpu->zf = 0;

    cpu->eip++;
}
```

Listing 1.1: A code example for symbolic execution.

condition can cause the program to take this path.

Concolic Execution

Concolic execution is a special type of symbolic execution proposed in CUTE [1] and DART [2]. Instead of exploring every possible program path, it directs symbolic execution using concrete inputs: the program takes the paths associated with given concrete inputs, and the symbolic execution system collects branch conditions on this path at the same time. Although the program behavior remain unchanged, by concolic execution we can get the condition of a certain program path.

Path Explosion Issue

Path explosion is a major issue of symbolic execution. the number of paths can increase quickly since it is exponential to the number of branches. For example, if there are `n` independent branches (the decision of every single branch does not effect the decision of any other branch) on a certain program path, the total path number of this program can be 2^n at least. Note that path explosion can happen even if there is only a small number of branches in the source code. One possible cause of this symptom is looping. When the loop condition is controlled by symbolic inputs, the loop body can execute by a nondeterministic number of times. As a result, the loop guard (a branch checking

whether or not to leave the loop) and branches in the loop body can appear in the program path for a nondeterministic number of times, which drastically increases the number of branches and leads to path explosion. For the same reason, recursion can also cause path explosion with a small number of branches.

1.2.3 FuzzBALL

The FuzzBALL [3] system is a binary symbolic execution system that currently supports x86-32, x86-64 and ARM. It is implemented as a machine code interpreter, which dynamically translates each machine code instruction into the BitBlaze [4] Vine intermediate language, and then interprets that representation to perform the instruction's behavior. When a program is running under FuzzBALL, any value stored in a register or memory can be a symbolic expression.

Among all the symbolic execution systems, KLEE [5] is the one that closest to FuzzBALL. The major design difference between FuzzBALL and KLEE is the way they handle branches. KLEE fork the program state at each symbolic branch, and execute all the child processes simultaneously. On the other hand, FuzzBALL executes each program path one after another: once it reaches the end of a path, FuzzBALL goes back to the start point (which is the first symbolic branch) and reruns the program. This design difference leads to a space-time tradeoff. KLEE requires more memory space, but can finish faster with enough memory since it completely avoids executing duplicated code. On the contrary, FuzzBALL can avoid heavy memory usage, but it has to spend extra time rerunning code between the start point and the first branch where there is a side that has never been explored before.

Decision Tree

To avoid exploring duplicated paths, FuzzBALL keeps track of previously explored program paths by maintaining a decision tree, and checks it before making branch decisions. The decision tree is a binary tree that records all the previously explored symbolic branches. Each node in a decision tree represents a symbolic branch and the corresponding program state. And each route in a decision tree, from the root node to a leaf, corresponds to a path of the explored program. A node may have a "true" child, a "false" child, or both, depending on which directions of the branch are feasible

in that state. The "true" child is the next symbolic branch that will be encountered if the branch condition is true, and similarly for the false child.

FuzzBALL constructs the decision tree dynamically while executing a program. Whenever Fuzzball reaches a symbolic branch for the first time, it creates a new decision tree node for this branch, and checks the feasibility of the branch condition as well as its negation. If only one side of the branch is feasible, FuzzBALL will chose the feasible side to further explore. Simultaneously, FuzzBALL moves down to the corresponding side of the current decision tree node so that it can create another node when reaching the next symbolic branch. The infeasible side will never be explored, and the corresponding subtree in the decision tree is pruned. If both the condition and its negation are feasible, FuzzBALL will randomly pick one side and leave the other side for future exploration. In this case, FuzzBALL moves down to the selected side of the tree, while the other side is marked as uncovered. Note that in the actual implementation the decision making process is pseudorandom and controlled by a random seed.

When FuzzBALL reaches the end of a program path and restarts, it traverses the tree again from the root node. With the information of previous execution paths stored in the decision tree, FuzzBALL can figure out whether a node has unexplored descendants when revisiting it. Since FuzzBALL only explores a node if it still has unexplored descendants, it is guaranteed that FuzzBALL will only explore each feasible path once. In addition, when revisiting known branches, FuzzBALL checks whether the states on current path are consistent with corresponding nodes on the decision tree in memory, and stops execution if any inconsistency is detected. This feature is used as a debug feature, since an decision tree inconsistency is most likely to be caused by bugs in FuzzBALL's implementation.

Solver Dependency

FuzzBALL uses an SMT (satisfiability modulo theories) solver to check feasibility for each branch. Given a list of constraints, an SMT solver tries to figure out whether any solution exists, and produces one if so. Those solutions are satisfying assignments to the constraints. Currently, FuzzBALL is compatible with STP [6], Z3 [7], or other SMT solvers that support the SMT-LIB 2 interface format, among which Z3 is mostly used in our work. To interact with an SMT solver, FuzzBALL translates symbolic

expressions into the SMT-LIB 2 language, and queries translated expressions with the solver. When the solver returns the results of this query, FuzzBALL parses it and converts the satisfying assignments back to symbolic expressions written in Vine.

1.3 Overview

The rest of this thesis is organized as follows. Our previous work is described in chapter 2 through chapter 4. In chapter 2, we describe our work to improve an existing testing framework for CPU emulators by exploring a better way to combines symbolic execution with plain testing. Our attempts to improve symbolic execution are illustrated in chapter 3 and 4, where we apply summarization techniques as a solution to path explosion brought by loops and loop-based functions respectively. Chapter 5 is the plan for the final thesis project, which combines symbolic execution with fuzzing.

Chapter 2

CPU Emulator Testing

2.1 Introduction

CPU emulators are widely used in various fields, while they are challenging to develop since they must follow the complicated architecture specification for software compatibility. In addition, emulator developers may change the emulator very often. For example, QEMU accepts 231 commits to the X86 translator per year on average 2018 through 2023. One way to mitigate those challenges is to build a automatic testing framework. An effective testing framework should generate tests that cover all the behaviours of all the instructions. Given the rapid change in emulator development, it is also important to finish the full test in a reasonable amount of time: considering how frequent QEMU changes, a testing system that can rerun a comprehensive relevant test suite nightly would be a sweet spot.

PokeEMU is an existing emulator testing framework that generates tests using symbolic execution. As depicted in figure 2.1, PokeEMU detects bugs by comparing the behaviors of the tested emulator with KVM, which runs most instructions using host hardware. To generate tests, PokeEMU first explores Bochs with symbolic execution to collect instruction variations, and then converts them to individual tests. PokeEMU then runs those tests on both QEMU and KVM, dumps the final machine state in memory dumps and compare them. For each test, different results on QEMU and KVM indicates a potential bug in QEMU. This test involves 76510 test cases, and it takes approximately 150 CPU hours to finish the full test. Among the 150 hours, it spends

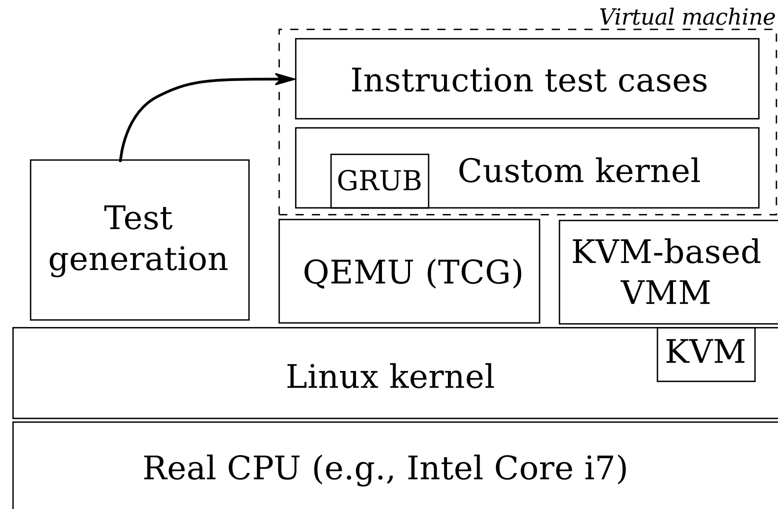

Figure 2.1: Architecture overview of (Fast) PokeEMU.

most time booting QEMU and making memory dump (529/583 ms according to our measurement.) Currently PokeEMU only support QEMU with X86-32bit target, but the same approach can be applied to other emulators with additional engineering efforts.

PokeEMU can rerun the complete test suite in about 24 hours. Although this performance is acceptable for running a thorough test of a CPU emulator, it is still beneficial if we can speed it up to meet the requirement of nightly testing. Besides, a faster testing framework allows us to run tests repeatedly, which may reveals bugs that are not detected when tests only run once. To improve the performance and effectiveness of PokeEMU, we implemented Fast PokeEMU, an advanced version of PokeEMU. We applied test aggregation and looping to PokeEMU tests, and handled issues related to those new features. In addition, we evaluate the performance and results of Fast PokeEMU on a comprehensive x86-32 instruction test suite executed on QEMU.

2.2 Approach

2.2.1 Aggreration and looping

Instead of running only one test for each QEMU session, we run a large number of tests after booting up QEMU, and dump the machine state after all those tests finish.

With this change, we only start QEMU and run tests for 1078 times, much less than the previous 76510 times. The simplest implementation of this idea would be just run a group of test cases one after another and compare the final machine state. One problem of this simple approach is that outputs of one test case can be overwritten by following test cases. As a workaround, we copy the output to unused memory before the next test case. In practice, we group tests by the instruction it tests, and aggregate each group.

Another way to increase the efficiency of PokeEMU is to reuse test case code. If we re-run each test for multiple times with different inputs, the total time for running a full test almost doesn't change, while the coverage of PokeEMU may further increase: when generating test cases using symbolic execution, we only set a essential subset of the whole machine state to symbolic (otherwise the symbolic execution phase will take forever.) Therefore, we probably can further increase the coverage by running each test case for multiple times with different random inputs, which is analogous to fuzz testing.

2.2.2 Reusing memory space with the Feistel Construction

Applying aggregation and looping increase memory usage, which can be a issue for PokeEMU. To save storage space and running time, vanilla PokeEMU run with 4 MB memory, and maps it repeatedly to 4GB memory address. This design works fine if we only run one test for one time, which only take 12 - 30 bytes of additional memory space. However, this number can keeping increasing as we increase the aggregation size or looping times. For example, if an aggregation contains 100 tests and we repeat each test 10,000 times, then it will take at least 10 MBs to store the outputs.

To work around this issue, we integrate the Feistel construction with PokeEMU tests. As depicted in Figure 2.2, the Feistel construction is a symmetric structure that can build invertible function out of repeated application of round functions F, which may not be invertible. We use the tested instruction as the round function, whose parameters serve as key to the function, and the output of one test is the input of the next test.

We choose the Feistel construction because of its ability to build invertible functions. Since the Feistel construction is invertible, this structure can guarantee that two ciphertexts with the same input will be different if there is only one different round function and everything else are the same. When there are more than one round functions that

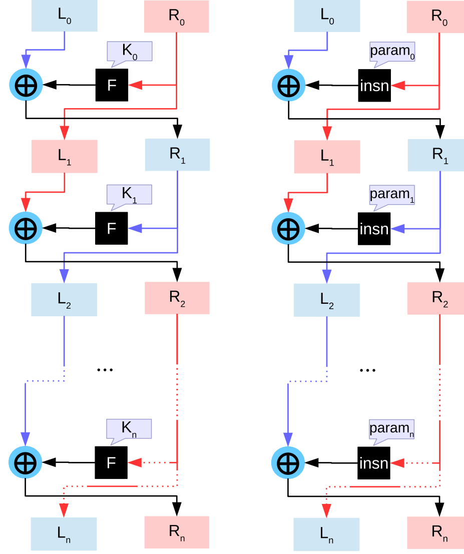

Figure 2.2: Feistel cipher (left) and PokeEMU tests (right)

behave differently, the probability that two or more tests cancel out the side effects of each other is also low because of the pseudorandom mixing performed by the Feistel construction. In Fast PokeEMU, we run the same test aggregation under both QEMU and real hardware (KVM), and compare the output on two platforms. If all but one test behave differently, it is guaranteed that the final outputs will be different. Even if there are more than one differences in the same aggregation, the chance to get different outputs is still high as long as only a small number of tests behave differently. Once we observe different results, we can identify the specific test that has different behaviours by running each single test.

2.3 Evaluation

2.3.1 Performance

We first evaluate how much faster the Fast PokeEMU system is compared with original PokeEMU. As shown in Table 2.1, testing aggregation under is significantly faster then non-aggregation tests all 4 modes (row 2 through 5). Among all 4 aggregation modes, Feistel cipher slightly increases the performance cost compared to simple aggregation

Mode	Total time (s)	Time per test (ms)
Separate	84871.8	583.528
Simple	334.7	2.313
Feistel	345.0	2.448
Loop (1)	345.17	2.672
Loop (10000)	1635.45	0.002

Table 2.1: Runtime performance of Fast PokeEMU test execution in QEMU. Row 1 shows the result of vanilla PokeEMU. Row 2 through row 5 are aggregated tests combined with different features.

Separated result	Separated result with extra code	Aggregated result	# of instructions
Match	Match	Match	577
Mismatch	Mismatch	Mismatch	273
Match	Match	Mismatch	8
Match	Mismatch	Match	10
Match	Mismatch	Mismatch	28
Mismatch	Match	Match	25
Mismatch	Match	Mismatch	28
Mismatch	Mismatch	Match	9

Table 2.2: Effect of aggregation on QEMU behavior difference coverage. For most instructions (rows 1 and 2), an aggregated test case gives the same result as separated tests.

without the Feistel construction. Although it takes longer when looping each test for more times, the average time per test decreases with a larger number of loop iterations.

2.3.2 Effectiveness

In this experiment we compared the testing results of vanilla PokeEMU (column 1), Fast PokeEMU without aggregation (column 2) and Fast PokeEMU (column 3.) Most time the results of all three cases are the same, which means Fast PokeEMU is as effective as original PokeEMU. When original PokeEMU reports a match while Fast PokeEMU reports a mismatch, it potentially can indicate that Fast PokeEMU found a previously unrevealed bug. Unfortunately, according to our investigation, all tests fall into this category are caused by bugs in Fast PokeEMU.

2.3.3 Historical bugs

In this experiment, we try to figure out whether Fast PokeEMU can find more real-world bugs with looping turned on. We run aggregated tests of all instruction variations on an old version of QEMU (version 1.0). For each test aggregation that reveals behavior differences, we first run each single test in it to identify the exact test that triggers those differences. We then search for the first QEMU commit where the differences are eliminated. This commit, if available, is likely to be a bug fixing patch. Since it requires additional effort to add (Fast) PokeEMU support to QEMU, we run this experiment on QEMU version 1.0 through 2.4, where 2.4 was the latest version of QEMU while this project was ongoing.

As shown in table 2.3, Fast PokeEMU found 15 tests that can trigger behavior differences. All the differences are associated with three bug fixing commits. Although vanilla PokeEMU can find all three bugs, Fast PokeEMU's random testing allowed it to find the problem in the `BT*` family of instructions (fixed by commit dc1823c) much more reliably.

2.4 Related work

Apart from PokeEMU, there are several other works that test either CPU emulators or similar tools translating binaries to specific intermediate representations. In addition, work has been done on undoing side effects when executing multiple tests in a row, which is analogous to an important aspect of test aggregation in our work.

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 [8] and KEmuFuzzer [9] systems are predecessors to PokeEMU. Instead of symbolic execution, those systems generate tests using random fuzz testing. EmuFuzzer targeted only user-mode emulators, while KEmuFuzzer developed most of the infrastructure for executing whole-system tests. The infrastructure of the later is also used in PokeEMU. As a result, the test execution performance of KEmuFuzzer is similar to vanilla PokeEMU.

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 2.3: Historical bugs revealed by vanilla and Fast PokeEMU. The first two columns show the instruction variations that can trigger behavior difference and the QEMU commits associated with them. The remaining two columns exhibit results of pokeEMU and fast PokeEMU, where a * means the tool detects behavior differences.

Though KVM primarily uses hardware virtualization, it also contains a software CPU emulator for some uncommon situations. Bugs in this emulator may have serious impacts since adversarial guests can make use of them to execute any instruction, as Amit et al. point out [10]. To detect bugs for this emulator, they apply a human-created test suite originally developed by Intel for testing real CPUs. By this approach, they discover a large number of bugs in the KVM emulator. Unfortunately, it requires a lot of effort to create such a test suite, and CPU vendors who created such a test suite usually decide not to release it to the public. In addition to urging vendors to release their test suite, it is also helpful to develop automated techniques that can reduce the cost of constructing such test suites.

Many emulators and binary analysis tools translate binaries to a simplified intermediate representation (IR) before executing or analyzing them, and bugs may appear in those translators. One way to detect those bugs is to compare the behaviour of several translators given the same input binaries. Although these IRs are typically tool-specific, some still can be translated to the same language for comparison. As described by Kim et al. [11], they cross-compare several x86 binary translators using symbolic execution, and discover a number of bugs in their CPU models. This approach is more powerful than generating test cases as (Fast) PokeEMU does, but is only applicable if IRs are compatible. On the other hand, test cases can be used to compare tools with incompatible IRs, such as interpreters like Bochs, and real CPUs.

Resetting Side Effects When executing tests, it is usually necessary to efficiently undo the side effects of one test before executing another. To achieve this goal, a testing system may run into a performance trade-off similar to the one we address. For example, in randomly testing Java methods, it would be inefficient to start a fresh JVM for each test since the tests are short. On the other hand, one test should not affect the next for reproducibility. JCrasher [12] has been developed to efficiently reset static state within a single JVM. Unfortunately, JCrasher's techniques would not apply in our context, given that JCrasher's techniques are specific to features of the JVM such as classloaders and static initialization.

2.5 Discussion

Apart from the performance issue, there is another inherent limitation in the general infrastructure of (Fast) PokeEMU: although Bochs is more high-fidelity than QEMU overall, behavior differences between those two emulators may affect the coverage of generated test cases. Instead of symbolically executing Bochs, a more straight forward way to create preliminary tests for QEMU is to symbolically execute QEMU itself. With this technique, we not only can generate more reliable tests for a system like (Fast) PokeEMU, but can also execute both emulators symbolically and compare the symbolic states, which is a more accurate way to compare them.

Unfortunately, symbolic execution of QEMU still requires a fair amount of engineering effort given that QEMU is a dynamic binary translator. QEMU translates the input binary to another binary targeting the the host's ISA, and executes the resulting binary. During the translation process, symbolic data in the input binary may propagate to the output binary. FuzzBALL currently doesn't support running symbolic code, but we can work around this issue by carefully adjust the way we use FuzzBALL. One approach also applied by (Fast) PokeEMU for Bochs symbolic execution is to split the execution into 2 parts. In the first part, we symbolically execute the binary translation code and stop before QEMU starts to execute the translated binary. Based on the result of part 1, we then set up symbolic execution for every instruction variation.

Although there is no fundamental challenge to run QEMU under FuzzBALL as far as we know, we may run into new issues given the scalability issue of FuzzBALL (and other major symbolic execution frameworks). To better execute a system as complicated as QEMU, we should either mitigate the scalability issue of symbolic execution, or combine it with other techniques such as fuzzing.

Chapter 3

Loop Summarization

3.1 Introduction

Loop misuse is one of the major sources of real world program bugs. For example, a loop processing each item of a buffer may overflow this buffer if the loop is not carefully written. In addition to buffer overflow, other bugs such as infinite loops and off-by-one errors are also closely related to loop misuse. To detect loop-based bugs, symbolic executors should be able to execute those loops accurately and efficiently.

However, executing loops can be challenging for symbolic executors, since it may cause path explosion. If the guard condition of a loop is symbolic, there will be a symbolic branch at the beginning of each iteration. As a result, when executing this loop, it will branch to as many paths as the maximum possible number of loop executions. Moreover, if there are symbolic branches in the loop body, the number of paths can be exponential in the number of loop executions.

Loop summary for concolic execution was first proposed by Godefroid et al. as a countermeasure against path explosion caused by loops. This technique has been implemented in SAGE, a trace-based tool that performs concolic execution (explained in 1.2.2.) To demonstrate the capability of this technique, the authors evaluated the ability to detect and summarize loops using an ANI parser embedded in Windows 7. Given that the version number of Windows is unspecific, and that SAGE has never been released, it is difficult to reproduce exactly the same experiment. Researchers may want to further evaluate this technique, compare it with other loop-related techniques,

or determine how effective it is in bug finding when combined with other techniques. Moreover, it is also interesting to understand the limitations of this technique and figure out how to improve it. For all the purpose mentioned above, it is necessary to reimplement the loop summary tool, given that neither SAGE nor the loop summary implementation on top of it is available.

Instead of implementing loop summary on another concolic system similar to SAGE, we decided to implement it on FuzzBALL, a publicly available standard (non-concolic) symbolic execution platform. Historically, our work was supported by DARPA and Air Force Research Laboratory under contract FA8750-14-C-0093, and was intended to be used in DARPA Cyber Grand Challenge competition. In this competition, our team used FuzzBALL to detect vulnerabilities in binaries. The preliminary goal is to improve FuzzBALL's loop handling capabilities by integrating the loop summary technique with it. Our hypothesis is that it is always possible to find a mechanical translation of an algorithm from a concolic system to a standard symbolic system (and vice versa.) However, the mechanical translation may not be the most elegant or efficient implementation. As a result, redesigning is necessary if we want a better translation. Due to the architectural difference between symbolic and concolic systems, it requires efforts to figure out the proper way to redesign a specific feature.

Reimplementing loop summary on a standard symbolic system also makes it possible to integrate this technique with other techniques that are more natural for a symbolic system. Among all guiding techniques for symbolic systems, it is more natural to implement some of them on a standard symbolic system and others on a concolic system. Intuitively, guidance controlling the current path [13, 14] fits into standard symbolic systems better. On the contrary, if a guiding technique select the next path based on previously execution rather than deciding the current path on the fly, it is more natural to implement this technique on a concolic system. Loop summary is a borderline case, since it relies on both previous paths and the current one. Once we implement the symbolic version of loop summary on FuzzBALL, it will be potentially possible to combine it with other existing techniques on FuzzBALL.

We redesigned the concolic-based loop summary to adapt it to standard symbolic systems. Based on the redesign, we implemented it on FuzzBALL. We then evaluated our version of loop summary using SV-COMP, an open source benchmark that is widely

available. Based on the evaluation results, we identified three major limitations of loop summarization, and discussed solutions for each of them.

The rest of this section is organized as follows. In subsection 3.2, we introduce the original loop summarization, which is the most important related work. After than, we provide a more detailed explanation of our symbolic reimplementation of loop summarization in subsection 3.3. Finally, we describe the experiments we designed to evaluate symbolic loop summarization as well as the results in subsection 3.4.

3.2 Loop summary for concolic execution

Our work is based on the trace based loop summary technique [15] implemented in concolic executor SAGE. SAGE starts the first execution with a seed input, and collects path condition at the end of this execution. After that, SAGE flips each branch condition in the collection. In this way, SAGE generates up to n new path conditions, where n is the total number of branches. SAGE then computes new inputs using those new path conditions, and uses them for further concolic execution. Note that SAGE keeps track of code coverage of each input. Inputs with higher coverage are used first, and inputs with no coverage contribution are deprioritized in further rounds of executions.

A trace-based loop summary approach [15] has been built on top of SAGE. Given an execution trace, the first step of loop summarizarion is to identify all the loops covered in this trace and the associated loop guards. For this purpose, SAGE constructs a control flow graph according to each trace, and identifies loops and guards according to the graph. To summarize a loop, it is necessary to figure out whether each of the in-loop variables is inductive. A variable is an inductive variable if it is always changed by the same amount during each loop iteration, and a loop is an inductive loop if all the in-loop variables are inductive. To summarize a loop, all the non-inductive variables are ignored, and each inductive variable is updated to $IV_0 + dV \cdot EC$ when exiting the loop. IV_0 stands for the initial value of this variable, dV stands for the difference per iteration, and EC is a symbolic expression that represents the total number of loop iterations. Given the loop guard condition and the change of guard condition per iteration dD, EC can be computed as $\lfloor (D - dD - 1)/ - dD \rfloor$, where D is the initial value of the loop guard. Since EC depends on symbolic inputs, the loop summaries can be used to

explore paths with different loop counts and exit points.

To apply a loop summary, SAGE simply replace part of the path condition generated from entering the loop to leaving the loop with summarized side effects, which are a set of extra branch conditions. After that, SAGE will compute new inputs by flipping each of those branch conditions, and use those inputs for future execution. All those inputs and any other inputs derived from them depends on loop summary. At the same time, it is possible to explore the unsummazied original loop at any time by not using those inputs. As mentioned in the previous paragraph, the summaries can be unsound, since all the non-inductive variables are ignored. However, SAGE is tolerant to unsound summaries by nature: When SAGE creates an unsound summary, the worst result is that no input derived from the summarized path increase coverage, and SAGE will not use those inputs for further execution due to its coverage based prioritizing.

For a loop with multiple guards G_1, G_2... G_n and assuming we want to leave from the k-th guard starting from the beginning of the loop body, a set of extra conditions as shown in equation 3.1 is added to the path conditions.

$$\neg \min(G_1) \wedge \neg \min(G_2) \wedge ... \wedge \neg \min(G_{k-1}) \wedge \min(G_k) \tag{3.1}$$

$\min(G_n)$ is defined by equation 3.2, where EC_k stands for the expected count of G_k, and EC stands for the expected count of any other guard.

$$\forall G \neq G_k \begin{cases} EC < EC_k & \text{if } G < G_k \\ EC \leq EC_k & \text{if } G_k < G \end{cases} \tag{3.2}$$

$\min(G_k)$ guarantees that the EC of the k-th guard is smaller than any other guards placed before G_k. In this way, it is guarantee that we leave at guard G_k. Since both EC and $\min(G_n)$ depends on symbolic inputs, the loop summaries can be used to explore paths with different loop counts and exit points. Each of those mutually exclusive conditions $\min(G_j)$ (where $j < k$) is later flipped to generate the new input for a path that takes guard G_j.

3.3 Approach

Since each trace in SAGE is equivalent to a program path, we can implement the same trace-based algorithm for each path under FuzzBALL: our tool constructs intraprocedural control flow graphs dynamically while executing each path, uses the graph to identify loops and guards, collects any changed registers or memory locations as inductive variables candidates, and checks whether they are inductive at the end of each loop iteration. Registers changed at different instructions are regarded as different variables. However, we have to apply loop summaries in a different way due to the design differences between SAGE and FuzzBALL.

While the original loop summary algorithm only guarantees soundness when the loop to summarize is an inductive loop (all the variable in the loop are inductive), it always attempts to summarize loops and apply those summaries whether this requirement holds or not. This behaviour is safe for SAGE. As a concolic executor, SAGE always verifies the newly generated inputs by using them to direct concolic execution, and any new input that doesn't execute the expected path is discarded. In contrast, FuzzBALL (as well as many other symbolic execution systems) is designed to always guarantee sound execution, and never double check soundness afterwards. Therefore, when implementing loop summary in FuzzBALL, we only summarize a loop if all the in-loop variables are inductive.

As another design difference, SAGE can replace path conditions without negative impacts, while in FuzzBALL it is more restricted. FuzzBALL keeps track of previous executions explicitly with more details (see the discussion of the decision tree in Section 1.2.3.) Therefore, it can only replace path conditions if the new conditions are logically equivalent to the old ones. Since loop summaries are generated according to previously executed paths, there is no guarantee that the loop summary conditions are complete representation of the loop. As a result, if we apply an incomplete loop summary in place, the rest of the execution will be incomplete (i.e. FuzzBALL may not be able to explore all program behaviours.)

Instead, we added an extra branch after the loop entrance, which we call summary-decision branch. The purpose of summary-decision branch is to check whether there is any feasible summary. If only one summary is feasible, FuzzBALL takes the true

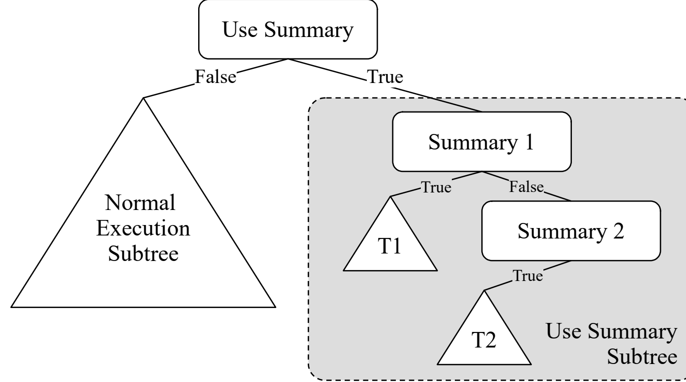

Figure 3.1: multi-level choice in a FuzzBALL decision tree. `Use Summary` denotes the extra branch. The subtree in grey box demonstrates the multi-level choice branches when FuzzBALL decides to apply the loop summary (take true side of `Use Summary`). Alternatively, FuzzBALL can take the false side and explore the loop normally, which corresponds to the `Normal Execution Subtree`.

side of the summary-decision branch, finds another branch corresponding to the feasible summary, and then takes the true side of that branch. When there are multiple feasible summaries, FuzzBALL randomly chooses one from them. A loop may have multiple summaries if there are in-loop branches and/or there are multiple guards. Therefore, FuzzBALL considers a summary as feasible if both conditions are satisfiable: 1. The summary is for the same in-loop branch taken on current path. 2. min(G_k): EC of this summary is smaller then any other summaries before it.

Figure 3.1 demonstrates the high-level structure of the decision tree with two known summaries. When a loop is executed for the first time, FuzzBALL always takes to the no-summary side since FuzzBALL hasn't created any summary. After exploring a few paths, FuzzBALL creates `Summary 1`. When FuzzBALL executes this loop again, it will apply `Summary 1` if it is feasible, or continue exploring the `Normal Execution Subtree` if not. By executing the loop when `Summary 1` is infeasible, FuzzBALL created `Summary 2`. Now that there are 2 summaries, FuzzBALL will check whether either of them is feasible, and choose one of them accordingly. Assuming FuzzBALL chooses `Summary 2`,

the current path will take the false side of `Summary 1` and true side of `Summary 2`. Note that FuzzBALL will also exclude a summary if all branches beyond the summary have been covered. For example, in Figure 3.1, if all the descendants of `Summary 1` (which are equivalent to the subtree T1) have been completely explored, FuzzBALL will only check whether `Summary 2` is feasible at `Use Summary` node.

3.4 Evaluation

To evaluate the applicability and benefit of loop summarization, we test our implementation using a set of programs designed for software testing and verification. The majority of tests used in our evaluation come from SV-COMP, a set of benchmarks used in the Competition on Software Verification [16]. Since not all tests in SV-COMP contain loops, we only focus on the sub category named “loops”. We also manually pick out tests whose loops are irrelevant to their inputs, since loop summarization can only apply if symbolic loops exist. In addition, we manually created tests for specially program behaviours that are not covered in SV-COMP loop tests.

Before starting larger scale experiments, we manually investigate a few interesting examples. Base on what we learned from the preliminary investigation, we can distinguish tests that are not likely to work from tests that may work. For compatible examples, we measure the coverage of generated loop summaries from several aspects. For tests that are not likely to work, we manually study the code and identity the causes.

Among all 63 tests under the loops category of SV-COMP, 43 of them contain at least one symbolic loop (loop controlled by inputs). Out of the 43 tests, 8 are likely to work. In addition, we create 4 tests in complement to SV-COMP benchmarks, both of which are expected to work to some extend. In the rest of this section, we demonstrate and discuss the results of tests that are likely to work, and then analyze the tests that are not likely to work due to the limitations of our tool.

3.4.1 Compatible examples

For compatible tests, we execute them under the loop summarization system with program inputs set to symbolic. SV-COMP tests need specific adjustment because they are designed mostly for evaluating source-code level static analysis techniques. Instead of

taking actually inputs, those tests annotate input dependent values using "verifier" functions, which can integrate into static analysis systems and provide information about program inputs. For example, an input-dependent integer is assigned by `__VERIFIER_nondet_int()`. To execute those tests with our system, we write empty verifier functions for them, replace those functions by a `nop` instruction and overwrite the return values with a symbolic variable when running under our tool.

To evaluate the applicability and efficiency of symbolic loop summarization, we run all 12 tests that are likely to work under our system. We introduce summarization ratio as an metric to demonstrate the benefit of loop summarization. Currently we compute the summarization ratio of time and numbers of paths. For each loop, the summarization ratio of path is defined as $\frac{N}{N_{sum}}$, where N_{sum} denotes the total number of paths where loop summaries are applied, and N denotes the total amounts of paths through the loop. For example, in figure 3.1, the total number of paths in the grey area is N_{sum}, while the total number of paths in the normal execution subtree is N. Similarly, we define summarization ratio on time as $\frac{T}{T_{sum}}$, where T_{sum} and T is the amount of time spent on N_{sum} and N paths, respectively. A loop summary is helpful if the ratio is higher than 1, and a higher ratio indicates greater benefits.

Given that there is no fundamental challenge when the input width is larger, we set restrictions on input widths according to the loop controlled by each input: the input is limited to 8-bit width if the iteration number of the loop is linear in it, and is limited to 4-bit if the iteration number increases faster (e.g. quadratic, cubic or exponential.) The main purpose of those limits is to stop the non-summarized executions from running too long. Without those limits, the running time and path number of the non-summarized execution increase drastically, while there is no significant change on summaries execution. As a result, the summarization ratio would be even higher if we remove those limits. Some tests only work with additional restrictions, which are discussed in the context of those tests

The general statistics are shown by table 3.1. Each row of this table shows the summarization ratio of one loops. Some tests contain multiple loops. Among all 12 tests, 8 can be completely summarized with respect to loops, and 2 can be partially summarized as shown in this table. Both path and time ratios are above 1 for all summarized loops except the one in `nested`, which indicates that loop summarization

<table>
<tr><th rowspan="2">Categories</th><th rowspan="2" colspan="2">Test</th><th colspan="2">Summarization ratio</th></tr>
<tr><th>Path number</th><th>Time</th></tr>
<tr><td rowspan="8">Our tests</td><td colspan="2">3-sum</td><td>15.8</td><td>42.3</td></tr>
<tr><td colspan="2">3-sum-partial</td><td>12.8</td><td>53.4</td></tr>
<tr><td rowspan="2">2-loops-1</td><td>loop 1</td><td>35.1</td><td>116.0</td></tr>
<tr><td>loop 2</td><td>N/A</td><td>N/A</td></tr>
<tr><td rowspan="2">2-loops-2</td><td>loop 1</td><td>N/A</td><td>N/A</td></tr>
<tr><td>loop 2</td><td>27.1</td><td>37.3</td></tr>
<tr><td rowspan="2">nested</td><td>inner loop</td><td>0.0396</td><td>0.0616</td></tr>
<tr><td>outer loop</td><td>N/A</td><td>N/A</td></tr>
<tr><td rowspan="10">SV-COMP loops</td><td colspan="2">count_up_down-1</td><td>127.0</td><td>505.0</td></tr>
<tr><td colspan="2">count_up_down-2</td><td>127.0</td><td>418.0</td></tr>
<tr><td colspan="2">sum01-2</td><td>127.0</td><td>2090.0</td></tr>
<tr><td colspan="2">terminator_01</td><td>228.0</td><td>80700.0</td></tr>
<tr><td colspan="2">terminator_03-1</td><td>227.0</td><td>717.0</td></tr>
<tr><td colspan="2">terminator_03-2</td><td>99.0</td><td>195.0</td></tr>
<tr><td colspan="2">trex02-1</td><td>N/A</td><td>N/A</td></tr>
<tr><td colspan="2">trex02-1(inlined)</td><td>31.8</td><td>44.4</td></tr>
<tr><td colspan="2">trex02-2</td><td>N/A</td><td>N/A</td></tr>
<tr><td colspan="2">trex02-2(inlined)</td><td>31.8</td><td>42.4</td></tr>
</table>

Table 3.1: Results of all compatible examples. All ratios have been rounded to three significant digits.

is beneficial in those cases. `trex02-1` and `trex02-2` are the only two tests that are not able to be summarized without assistance. For those two tests, we simplified them based on the issue and retest on those simplified variations.

In the rest of this section, we explain the experiments and results of all working examples. We discuss a few interesting examples as case studies, which either represent a specific feature of our system, or indicate a shortcoming of it. For all other tests, we give brief descriptions both for the experiment setup and for the results.

3-sum & 3-sum-partial `3-sum` is a test created at an early stage of this project. The initial purpose of `3-sum` was to test fundamental loop summarization and the capability to handle branching loops. As shown in listing 3.1, the loop in `3-sum` contains 2 guards depending on `x` and 2 paths depending on the parity of `x`. Both guards appear on the path when `x` is odd, while only the first guard appears on the even path. Despite being

```c
void func(int x){
    char str[30];
    int c = 0;
    while (1){
      if (x <= 0) // Guard 1 on both branch
         break;
      if (x % 2 == 0){
          c = c + 1;
      }
      else{
          c = c + 2;
          if (x == 15) // Guard 2 only on odd branch
          break;
      }
      x = x - 2;
    }
    if (c <= 40) str[c] = 0; // buffer overflow
}
```

Listing 3.1: 3-sum

a branching loop, the loop in `3-sum` is still completely inductive given that the parity of `x` never changes within the loop.

We execute `3-sum` with symbolic `x`. Three summaries are created from this loop and applied. Among all three summaries, two of them are for the odd branch, and these two summaries leave the loop at guard 1 and guard 2 respectively. The rest one summary represents the even path, and leave at guard 1.

The buffer overflow at line 17 demonstrates the benefit of loop summarization. With the default behavior, where paths are explored in a pseudorandom order controlled by a random seed, FuzzBALL detects this bug after exploring 51 paths, while the number is 10 when loop summarization is on. Once the loop summary is created, it will be applied in upcoming paths whenever feasible. After the loop summary has been applied, the buffer overflow will be detected immediately when executing line 17.

We also create `3-sum-partial`, a variation of `3-sum` which is partially inductive. As shown in listing 3.2, `3-sum-partial` is almost the same as `3-sum` except that `c` is not inductive on the even path. For this test, we can still create and apply the other two summaries on the odd path.

Terminator_03-1 `Terminator_03-1` is one of the tests from SV-COMP and is shown

```
1  void func(int x){
2      char str[30];
3      int c = 0;
4      while (1){
5        if (x <= 0) // Guard 1 on both branch
6           break;
7        if (x % 2 == 0){
8            c = 2*c + 1;
9        }
10       else{
11           c = c + 2;
12           if (x == 15) // Guard 2 only on odd branch
13           break;
14       }
15       x = x - 2;
16     }
17     ...
18 }
```

Listing 3.2: 3-sum-partial

in listing 3.3. Unlike most other tests, the in-loop variable `x` increases by a symbolic value `y` every loop iteration. Since `y` remains unchanged within the scope of the loop, this loop is sill a inductive loop and can be summarized.

There is no fundamental difficulty in supporting symbolic increments. Initially the system was built under the assumption that all in-loop variables increase by a concrete value. To support examples like this, we generalized the old implementation: instead of aborting summarization when an in-loop variable increases by a symbolic amount, we check whether the symbolic increment is always the same among all observed iterations, and summarize the loop if the symbolic increment is constant in this sense.

Our system can create one loop summary for the loop from line 6 through line 8. Solving the path constraints with the summary applied can be slow due to extra complexity introduced by the symbolic increment. For example, it takes 14.14 seconds to solve the condition check at line 10, while most solver queries finish in much less than 1 second. Despite the long solver time on one path, it is still worthwhile to apply the summary since executing this path is much faster than executing all 227 paths going through the loop.

```
1  int main(){
2      int x=__VERIFIER_nondet_int();
3      int y=__VERIFIER_nondet_int();
4  
5      if (y>0){
6          while(x<100){
7              x=x+y;
8          }
9      }
10     __VERIFIER_assert(y<=0 || (y<0 && x>=100));
11 
12     return 0;
13 }
```

Listing 3.3: Terminator_03-1

Nested This test is created to test nesting loop. As shown in 3.4, the inner loop at line 6 through line 8 is completely inductive. In the outer loop, `a` is inductive, but the outer loop itself is not inductive due to the side effects of the inner loop. Although the outer loop is currently not summarizable, it is interesting to know whether the inner loop can be summarized and how helpful the summary is.

We execute `nested` with `x` and `y` symbolic, and the integer width is set to 4 bits to finish the experiment within a reasonable amount of time. Our system can create and apply a summary for the inner loop. However, applying this summary is not helpful. Due to the nesting structure, the inner loop can be executed for multiple times within the same path. Each time the inner loop is entered, our system adds an extra branch that represents whether or not to apply a summary (the same as the one in figure 3.1 after `Use Summary`). As a result, the total number of paths increase from 51 to 1287.

Although nesting loops cannot benefit from the current implementation of loop summarization, technically a loop likes the one in `nested` can be summarized if we further generalize our tool. We discuss more details on this line in section 3.4.2 under "Nesting".

Other working examples

2-loops contains 2 independent loops one after another, one of which is inductive while the other is not. We test 2 variations of this test, `2-loops-1` and `2-loops-2`. Only the first loop is inductive in the former, and only the second loop is inductive in

```
1  int func(int x, int y){
2      int i, j, a=0, b=0;
3
4      for(i = 0; i < x; i++){
5          a += 1;
6          for(j=0; j<y; j++){
7              b+= 2;
8          }
9      }
10     ...
11 }
```

Listing 3.4: Nested

the latter. FuzzBALL can summarize the inductive loop regardless of the ordering, and the summarization ratios are similar.

trex02-1 contains a loop with a non-deterministic symbolic branch, while both sides of the branch call the same function. Given that our system can only summarize loops within the same function, this test can be summarized only if the in-loop function is compiled as an inline function. `trex02-2` is the same as `trex02-1` except a different SV-COMP assertion.

terminator_01, **sum01-2**, **count_up_down-1** and **count_up_down-2** contains standard loops with a symbolic iteration counter and inductive in-loop variables. Our system can completely summarize those loops, and the summarization ratios show that those loop summaries are effective.

3.4.2 Examples of limitations

Fundamentally, our system can only summarize loops that are completely inductive. However, a loop can be non-inductive for various reasons, and it is promising to summarize some types of non-inductive loops if we further extend our current system. We manually identified tests containing non-inductive loops, and group them by the cause of the limitations.

As shown in table 3.2, there are 3 major causes of unsuccessful loop summarization: non-linear arithmetic, array accessing and nesting. A loop is linear arithmetic if the output value of any in-loop variable can be represented as an linear expression of the input values of in-loop variables and loop iteration counts. Due to the nature of our current

Test	Array	Nested	Non-linear
string-1	*		
array-1	*		
array-2	*		
invert_string-1	*		
invert_string-2	*		
sum_array-1	*		
sum_array-2	*		
eureka_01-1	*	*	
bubble_sort-1	*	*	*
bubble_sort-2	*	*	*
insertion_sort-1	*	*	*
insertion_sort-2	*	*	*
matrix-2	*	*	*
linear_sea.ch	*		*
linear_search	*		*
n.c24		*	*
for_bounded_loop1			*
n.c11			*
sum01_bug02_sum01_bug02_base.case			*
terminator_02-1			*
terminator_02-2			*
trex01-1			*
trex01-2			*

Table 3.2: The major causes of limitations for SV-COMP examples under loops category

loop summarization approach, it is not possible to summarize a non-linear arithmetic loop. For a linear arithmetic loop, it can still be non-inductive if it loops through an array or has another loop nested inside. In the rest of this section, we explain those two cases with more details and discuss possible solutions for them.

Nesting A nested loop can be non-inductive even if both the inner and outer loops are linear arithmetic loops. The test discussed as a case study in a previous section (listing 3.4) is an example of non-inductive loop caused by nesting. For this example, the innermost loop can be summarized, but applying an incomplete summary for the nested loop shows no benefit overall.

As far as we known, there is no fundamental challenge to add support for nested

loops. To handle nested loops, the first step is to recover the nesting relationship of loops, which can be built on top of the existing dynamic loop detection. With more information about the structure of the nested loop, we can then extend the system so that it creates and applies summaries accordingly. For a two-level nested loop, an variable in outer loop IV_{out} is updated to $IV_{out} + EC_{out} \cdot dV_{out}$, while an variable in inner loop IV_{in} is updated to $IV_{in} + EC_{out} \cdot EC_{in} \cdot dV_{in}$.

Array accessing A linear arithmetic loop can be non-inductive if it accesses arrays. When accessing an array using a loop, a common pattern is to loop through the array and access loop elements one by one. Such an array-accessing loop is non-inductive since a different element is accessed in each loop iteration.

Listing 3.5 is an example of array-accessing loop, taken from the `sum_array-1` test of SV-COMP. In this test, array `A` and array `B` are added together and assigned to array `C`. The loop starting from line 7 is a linear arithmetic loop since `C[i]` is always a linear expression of `A[i]` and `B[i]`. However, the loop is non-inductive since the i-th element of `C` is only updated at the i-th iteration, while the increment at any other iterations is 0.

Similar to nested loops, it is also not fundamentally difficult to support array-accessing loops. However, to support a wide variety of array-accessing loops, a considerable amount of engineering work is required. For example, to summarize `sum_array-1`, the system must identify the array addition operation first. After that, a specific summarization that represents the array addition would be created and applied. Note that this implementation will only work for loops adding two array together. A different implementation is required if we want to support loops accessing and manipulating arrays in a different way.

3.5 Related work

Apart from the baseline loop summarization technique described in section 3.2, there are other works on loop summarization. As a trace based symbolic execution system, LESE [17] broadens the coverage of a single path by generalizing loops. Instead of creating loop summaries automatically, it requires an input grammar from users to figure

```
for(i=0;i<M;i++)
    A[i] = __VERIFIER_nondet_int();

for(i=0;i<M;i++)
    B[i] = __VERIFIER_nondet_int();

for(i=0;i<M;i++)
    C[i]=A[i]+B[i];
```

Listing 3.5: Array accessing loop

out the relationship between in-loop variables and loop counts. Proteus [18] presents a static loop summarization technique based on the flow graphs of the loops. This technique can accurately summarize linear loops and multi-path loops when different paths interleave in a sequential or periodic way. For multi-path loops taking different paths in a more complicated order, Proteus approximates them to loops that can be summarized. This technique can integrate with our system or other symbolic execution systems to improve the performance when executing loops.

Chapter 4

Loop-based function summarization

4.1 Introduction

The usage of standard library functions is pervasive in programs. However, it is challenging to reason about them with baseline symbolic execution, since some of the most commonly used functions suffer from path explosion issue caused by array accessing loops. Instead of executing library functions, some symbolic systems skip them and replace them with constraints created based on the semantics of those functions. This technique has been implemented in S2E [19], Angr [20] and its predecessor Firmalice [21]. In the rest of this section, we refer to this technique as function summarization.

To perform function summarization, it is necessary to identify the location of library functions, and to figure out whether it is a known function that can be summarized. Due to the presence of symbol tables, it is trivial to identify library functions called in dynamically linked binaries. On the contrary, this task can be challenging if the binary is statically linked, or if the name of a dynamically linked function does not match its implementation. However, statically linked binaries are common in many real-world program analysis scenarios: firmware is often linked statically for fewer dependency. Statically linked malicious programs are not only easier to spread but also more difficult to analyze. Given how common statically linked binaries are, an approach to identify

functions without symbol tables is necessary. Shoshitaishvili et al. proposed a test-case-based approach to identify functions in Firmalice [21]. Firmalice includes test case generation code that can randomly generate test inputs and expected outputs for each summarizable function. At each function call, Firmalice ran the callee function with those test cases. If all the test cases of a certain function are passed, Firmalice determines that the callee function is an implementation of this function, and applied the function summary accordingly.

While the test case based approach works correctly for the standard C library, false positives may occur when analyzing binaries linked with a modified library. For example, a modified version of strlen which treats both NULL and newline as terminators can still pass the tests for standard strlen as long as there is no newline in the input strings, but the function summary for standard strlen is not a complete representation of the modified strlen. In addition, since this approach identifies functions by call instructions, it doesn't work if a function is compiled as inline.

We propose and implement an alternative function summary approach based on the loop summarization technique described in chapter 3. Instead of identifying functions by test cases or reading function information directly from symbol tables, our tool identifies them by performing symbolic reasoning about their behaviors. We implement this approach as a extended version of loop summarization. Once the implementation for a specific function is done, it only requires minor developer efforts and no user effort to support a wider variety of this function.

4.2 Approach

Our function summarization starts from baseline loop summarization described in section 3.3: FuzzBALL detects loops while executing the target program, and collects guards and in-loop inductive variables for every known loop. With information about guards and in-loop inductive variables, FuzzBALL then check whether a loop is an implementation of a known function, and create a function summary accordingly. Due to a limited time budget, we only implemented loop-based function summary for standard strlen and two variations. While implementation details may vary for other functions, the same approach described below can be applied to other standard library functions

```
1 size_t strlen(const char *str){
2     size_t i;
3     for(i = 0; str[i] != '\0'; i++) ;
4     return i;
5 }
```

Listing 4.1: baseline strlen implementation

```
(str[0] == 0 ? 0 : (str[1] == 0 ? 1 :
    (str[2] == 0 ? 2 : (str[3]  == 0 ? 3 :
        ...
        (str[MAXLEN-1]==0 ? MAXLEN-1 : MAXLEN)...))))
```

Listing 4.2: function summary for a minimal implementation of standard strlen

or any array accessing loops as long as the array accessing index is inductive.

For a detected loop, we check the associated guards and inductive variables to determine whether it is an implementation of a function we can summarize. To determine whether a loop is strlen, we starts from the guard. As shown in listing 4.1, the guard of strlen should check whether $str[i]$ equals to a terminator (NULL in this implementation.) At binary level, this is equivalent to checking whether the zero flag (ZF) is assigned the true value, and the value of ZF is true if the character pointed by a certain address $addr$ is zero. Note that $addr = base + index$, where $base$ is the starting address of the input string of strlen, and $index$ is the array accessing index that increases by one at each iteration. In other words, $addr$ is inductive. To guarantee that the string is accessed by a inductive index, we search for $addr$ in the inductive variable table. If add is a known inductive variable, we can finally determine that the current loop has the same behavior as strlen.

Once a summarizable function is identified, the next step is to generate the function summary accordingly. The summary generation process can vary a lot for different functions. For standard strlen, the function summary is an nested if-then-else expression that represents its return value. As shown in listing 4.2, the expression checks each character in the string in order, and returns the proper length when the terminator is found for the first time. `MAXLEN` stands for the maximum length of the input string. Note that currently FuzzBALL cannot automatically figure out the length of a symbolic string. As a workaround, we set the maximum length of the input string when running

FuzzBALL. A complete solution to this issue is to over-approximate the maximum length of a string at runtime by looking for the first concrete null byte (or any other terminator) after the base address of this string.

The last step is to apply the function summary. This step is almost the same as applying a loop summary: each inductive variable IV is updated to $IV + EC \cdot dV$, where EC is the expected loop count and dV refers to the constant increment of IV. Instead of a uniform representation of EC, we need to figure out the function-specific way to compute it. For strlen, it is easy to compute EC, since EC equals to the return values of this function. Given the expected loop count, our tool computes the updated value for each IV and writes them back to memory or registers.

Compared to the test case approach, our approach is more resistant to false positives, due to the underlying loop summarization. For example, a strlen variation which supports both NULL and newline as terminators can be incorrectly identified and summarized as a standard strlen in test case based systems, while our system can tell the difference and decide to execute it without applying any function summary. Once the support for a standard library function is implemented, we can extend it to support more variations of the function with much less effort. For strlen, adding support for non-zero terminators or different element sizes on top of the baseline implementation is easy.

4.3 Evaluation

We implement function summarization on a modified version of FuzzBALL, which includes all the loop summarization features mentioned in chapter 3. Due to a limited time budget, we only implemented function summarization support for strlen and a limited set of its variations.

To test our implementation of function summarization, we created three motivating examples: strlen, strlen_n and wcslen. Strlen is a minimal implementation of standard library function strlen, as shown in listing 4.1. Strlen_n and wcslen are two variations of strlen, where strlen_n checks the size of newline-terminated strings instead of zero-terminated, and wcslen checks the size of a `wchar_t` array rather than a char array. The maximum string length is set to 64 entries with various entry sizes, and the entire input

Test	Summarization ratio	
	Path number	Time
strlen	21.3	8.97
strlen_n	16.0	6.98
wcslen	21.3	7.68

Table 4.1: Summarization ratios

string is set to symbolic. Consequently, strlen and strlen_n both take a 64 bytes symbolic string as the input, while the symbolic input size of wcslen is 256 bytes. All three strlen variations can be summarized correctly. We also calculate the summarization ratio for path number and time defined in the last chapter (section 3.4.1) to measure the effectiveness of functions summaries. As shown in table 4.1, applying function summaries is beneficial in all three tests.

4.4 Related work

Function summarization has been implemented in several symbolic execution systems such as Firmalice [21], Angr [20] and S2E [19]. The major difference between those systems and ours is the function identification approach. As discussed in section 4.1, Firmalice identifies functions using the test case based approach. This approach has been partially inherited by its successor Angr for CGC binaries, which are statically linked binaries used in DARPA Cyber Grand Challenge competition. For any other binary, users have to supply function names and locations to Angr. Similarly, S2E relies on user inputs to identify functions for any binary. Our system, on the other hand, can automatically identify functions without user input, whether the binary is statically or dynamically linked.

Chapter 5

Fuzzing-directed symbolic execution for kernel bug confirmation

5.1 Introduction

The kernel is the most important part among all components of an operating system (OS), which supports both the entire OS and user-space applications. As the most commonly used OS kernel, Linux is used on various platforms, including mobile devices, personal computers, servers and embedded systems. Given how widely used it is, the correctness of the Linux kernel is essential for the security and integrity of a large proportion of computer systems all over the world.

Nevertheless, monitoring the correctness of Linux is challenging. To provide up-to-date support for the latest devices and protocols, Linux is under active development and evolves rapidly: 91290 commits have been contributed to the Linux kernel in 2023. To continually assess and maintain the quality of Linux code, a good practice is to verify that no bug has been introduced for every patch applied to the Linux kernel. However, considering the large size of the Linux code base, it is impractical to do so all the time. Therefore, a scalable technique that automates bug detection and reproduction is desirable.

Static analysis is one of the most promising technique that can scale to OS kernels. Since the OS kernel is a software component that handles the interaction between hardware and software, testing it with dynamic analysis tools can be challenging. For example, to test drivers of various devices, a dynamic testing tool has to accurately emulate the behavior of those devices. This issue is less drastic for static analysis, since a static tool can analyze the code without executing it, and as a result no input from devices is required. Nonetheless, static analysis suffers from its own problems due to the nature of this technique. Without executing the program, it is difficult to keep track of pointers and indirect jumps. As a result, the false positive rate of a static analysis tool can be high. To determine whether a bug candidate reported by static analysis is a real bug, further confirmation is required. However, since static analysis can not provide inputs to reproduce the bug, a fair amount of work is required to confirm a candidate bug.

Fuzzing is another practical technique to detect kernel bugs. It has been widely applied and has found a large number of bugs in a variety of OS kernels. Apart from specific problems of dynamic analysis mentioned above, baseline fuzzers suffer from an efficiency issue since they can waste time on unrelated program components. Although we can spend most of the time budget on important code by directing fuzzers with specific targets, approaching the targets by random mutated inputs can be difficult if the expected side of some branches can only be taken by restricted inputs.

Symbolic execution is an alternative approach to detect kernel bugs. Instead of randomly mutating inputs to generate new inputs, it generates inputs based on branch conditions and explores every possible program path. Therefore, symbolic execution can do a better job finding inputs that pass complicated conditions. As a trade-off, baseline symbolic execution is difficult to scale to the full OS kernel due to the path explosion issue. Consequently, symbolic execution is more often combined with other techniques rather than used as a standalone OS kernel analyzer.

Motivated by the intuition that greybox techniques can approach a certain code location more efficiently compared to fuzzing, we proposed an alternative approach to confirm candidate kernel bugs found by static analysis, which combines directed fuzzing with symbolic execution. To evaluate this approach, we implemented a prototype of this system, and designed several experiments, including supporting experiments that

validate intuitions of this idea and an end-to-end experiment using known bugs. Unfortunately, we fail to find a working example of our system due to various obstacles. By analyzing the results, we identify possible issues of the failing end-to-end experiment, and proposed improvements for this system.

The rest of this section is organized as follows. In section 5.2, we explain our approach of fuzzing-directed symbolic execution in depth. In section 5.3, we describe supporting and end-to-end experiments, report the experiment results and discuss our analysis. Finally, we give a brief overview of the related work in section 5.5.

5.2 Approach

The general workflow of our system is presented in 5.1. At a high level, we generate candidate tests by fuzzing, and rank those tests based on their distance to the target location. We then direct symbolic execution using those tests, where tests that can get closer to the target are used first. The rest of this section explains each component with more details.

5.2.1 Candidate Reachable Nodes Collection

The system starts with taking bug candidates as inputs, which are targeted code locations that can be revealed by various static bug detection techniques. Since our goal is to find inputs that can reach target locations, it does not make sense to run tests for unrelated system calls (syscall for short). Instead, we can identify the syscalls that can reach a target location with the help of static analysis tools such as TyPM [22]. We use the static tool to generate a call graph of the Linux kernel. With the call graph, we can figure out the call stacks from syscalls to the target location. Those syscalls are entry points for fuzzing. Once we have identified those entry points, this system can set up the fuzzer accordingly, so that the fuzzer only generates tests for related syscalls and other syscalls required to set up the state for testing the targeted syscalls. We allow the fuzzer to run for a fixed amount of time (usually 24 hours at most), and pass all the tests and their coverage data to the next step. The coverage data will later be used to annotate corresponding nodes on a graph representation of the program. We define those nodes as reachable nodes for the rest of this chapter, which indicates that the

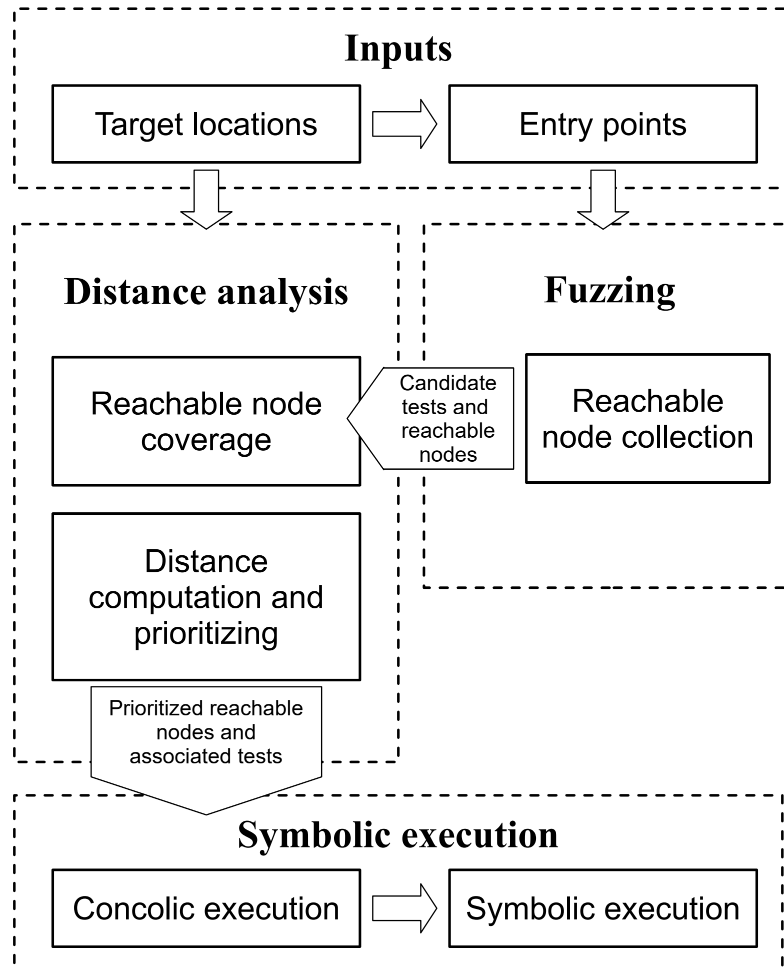

Figure 5.1: Architecture diagram of the fuzzing-directed symbolic execution system for kernel bug confirmation

corresponding locations of those nodes are reachable by known tests.

An alternative approach is to reuse existing tests generated by syzbot [23] instead of running Syzkaller by ourselves. Syzbot is an continuous fuzz testing platform targeting at Linux kernels. It has been operational since 2017 and reported more than 4,000 bugs. Instead of running Syzkaller for certain entry points, we can collect all the tests of a certain instance from syzbot with their coverage information, and perform the distance analysis described later. By using data from Syzbot, we no longer need to spend time running Syzkaller by ourselves. On the other hand, since Syzbot data contain tests of all syscalls, we either need to pick tests of the related syscalls or perform distance analysis using tests of all syscalls. The former approach requires extra engineering work, while the distance analysis can be much slower if we take the later approach.

5.2.2 Distance Analysis

The purpose of distance analysis is to compute the distance from each reachable node to the target location, and prioritize those nodes accordingly. This analysis can be done at the granularity of either basic blocks or functions. For function granularity analysis, we generate a call graph that only includes call edges. Note that if a function is reachable, then at least one of its callers is also reachable. Therefore, the return edges have no effect on distance computation and we decide to omitted them for simpler implementation. For basic block granularity, we take a slightly different approach. We first build an inter-procedural control flow graph, which includes edges from the call site to the callee function but not the corresponding return edge. The problem of this graph is that nodes after returning from callee are unreachable. To fix this problem, we add an extra edge pointing from the call site to the return address. With either a call graph or a control flow graph, we identify the corresponding nodes of the target location and covered code from the graph.

To prioritize reachable nodes, we use breadth first search to find the first `k` nodes (where `k` is a threshold) that are closest to the target. Those nodes are prioritized according to their distance: a node closer to the target has higher priority. All the nodes together with their priorities and associated tests are passed to the next phase. Note that one node is only associated with one test, while one test can be associated to multiple nodes.

5.2.3 Symbolic Execution

With all the prioritized nodes and associated tests, we can direct symbolic execution to the closest reachable code locations, and continue exploring the program from those locations. To take advantage of the distance analysis, nodes with higher priority are used first. For each prioritized node, we first run its associated test with concolic execution (explained in 1.2.2) mode, which follows the path directed by the test but also constructs symbolic expressions for data and collects path conditions. When we reach the location indicated by the node, the symbolic execution tool will switch from concolic mode to symbolic execution mode. In this mode, the symbolic execution tool explores both sides of each branch, and stops when it reaches the target. When the granularity is at function level, we switch at the beginning of the closet reachable function. Similarly, for basic block level granularity, the switch point is the starting address of the closest basic block.

We initially considered using FuzzBALL for symbolic execution. However, FuzzBALL lacks support for x86-64 bit kernel mode we target at. Considering that almost all the CPUs running Linux are 64 bit, and that it is less common to run a 32 bit kernel on 64 bit CPUs, we decided to switch to S2E, the state-of-art symbolic execution platform which supports both kernel and user-space code. S2E is a system-wide symbolic execution built on top of QEMU and KLEE. The programs executed by S2E are dynamically translated into QEMU TCG, which is then translated to LLVM IR and passed to KLEE for symbolic execution. Instead of exploring program paths one after another, S2E forks the current process whenever reaching a symbolic branch. To symbolically execute a test generated by Syzkaller, we set the test inputs to symbolic according to Syzkaller's syscall descriptions: an input value marked as symbolic if it is randomly mutated by Syzkaller. As an exception, memory addresses and lengths are always left concrete, since it requires extra engineering work to execute with symbolic memory address efficiently.

5.3 Evaluation

5.3.1 Supporting experiments

We designed supporting experiments to validate the idea of fuzzing-directed symbolic execution. Specifically, we would like to validate the following two intuitions: (1) Symbolic execution takes less time to reach the target if it starts in concolic mode and switches to symbolic mode when getting close to the target. (2) Tests ranked high by distance analysis are more likely to reach the targeted bug. We designed the edit distance experiment and the switching point experiment for those two intuitions respectively.

Switching point experiment Intuitively, we can save time on symbolic execution if we do not switch from concolic to symbolic mode until we are close to the target code region. In other words, the later we switch to symbolic mode, the more efficient our approach is. To demonstrate the benefit, we design the switching point experiment to study a known bug, WARNING in hugetlb_change_protection, which we will discuss with more details in section 5.3.2.

Given this known bug, we can reproduce it and get the call stack trace when the bug occurs (shown in 5.1.) We then execute the bug reproducer symbolically to simulate the symbolic execution process described in section 5.2.3: starting the execution in concolic mode and switching to symbolic mode at some point. With this setup, we can switch to symbolic mode at different functions in the call stack. The function at which we switch to symbolic is the switching point. To begin with, we switch to symbolic mode at the entry function of the mprotect system call, which is the system call that causes the bug. We then move the switching point one step deeper on the call stack, and repeat the experiment. In this way, we try all the functions in the stack trace, and collect the running time and total number program paths explored.

Figure 5.2 shows the results of the switching point experiment. The x-axis in both charts stands for the position of the switching point function on call stack, where a smaller position number refers to a function deeper on the call stack. For example, the switching point at position 0 stands for the function where the target bug occurs, 1 stands for the function that calls it, and 14 is the entrance function of the system call. The y-axis of chart (a) stands for the total running time corresponding to different

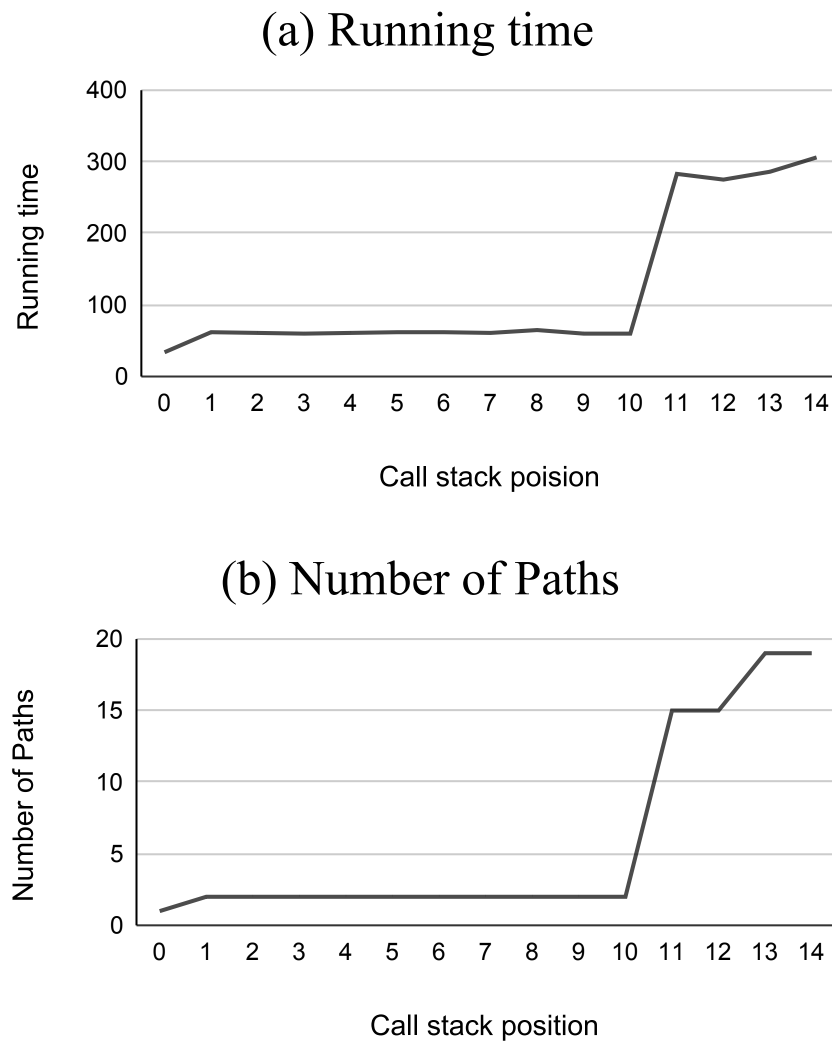

Figure 5.2: Results of the switching point experiment. The x-axis stands for the position of the switching point function on call stack. The y-axis stands for the total running time in chart (a) and the total number of paths in chart (b).

switching points, and the y-axis of chart (b) stands for the total number of paths explored. As shown in the two charts, both the running time and the total number of paths decrease as we switch to symbolic at a closer point to the target, which is consistent with the intuition mentioned at the beginning of this section.

Edit distance experiment To evaluate the effect of distance analysis, we use known bugs reported on syzbot to measure the relationship between call graph distance and the quality of selected tests, where ideally a test that is more likely to reach the target bug is higher quality. Based on our observation of known bugs, a test is more likely to reach a known bug by symbolic execution if it is similar to the bug reproducer. For example, if a test makes a sequence of system calls in the same order as the reproducer, the chance is high that this test can reach the bug if we change the arguments of those system calls. Therefore, we measure the quality of selected tests using the edit distance between a test and the reproducer of the targeted bug: a test is higher quality if its edit distance to the target bug reproducer is lower. The edit distance is measured by Levenshtein distance, and we compare the test or reproducer in syz format test (a Syzkaller-specific format that describes a sequences of system calls) to avoid irrelevant noise.

Given a known bug, we select reachable nodes whose distance ranges from 0 (which is the function where the bug occurs) to 10 using the distance analysis approach described in section 5.2.2, and measure the edit between each test selected by our algorithm and the bug reproducer. We then group tests by call graph distance and compute the average edit distance for each group. For this experiment, we randomly select 8 known bugs that can be reproduced without a large amount of data (such as bugs of file systems or the USB subsystem), since those large reproducers can have much larger average edit distance and may dominate the experiment results if not excluded.

Figure 5.3 shows the results of the experiment. To plot the overall trend, we compute the average of the per target bug result described earlier for all 8 bugs, as well as the first and third quartiles. As shown in this figure, the average edit distance increases as call graph distance increases, though the effect is smaller when the call graph distance is small.

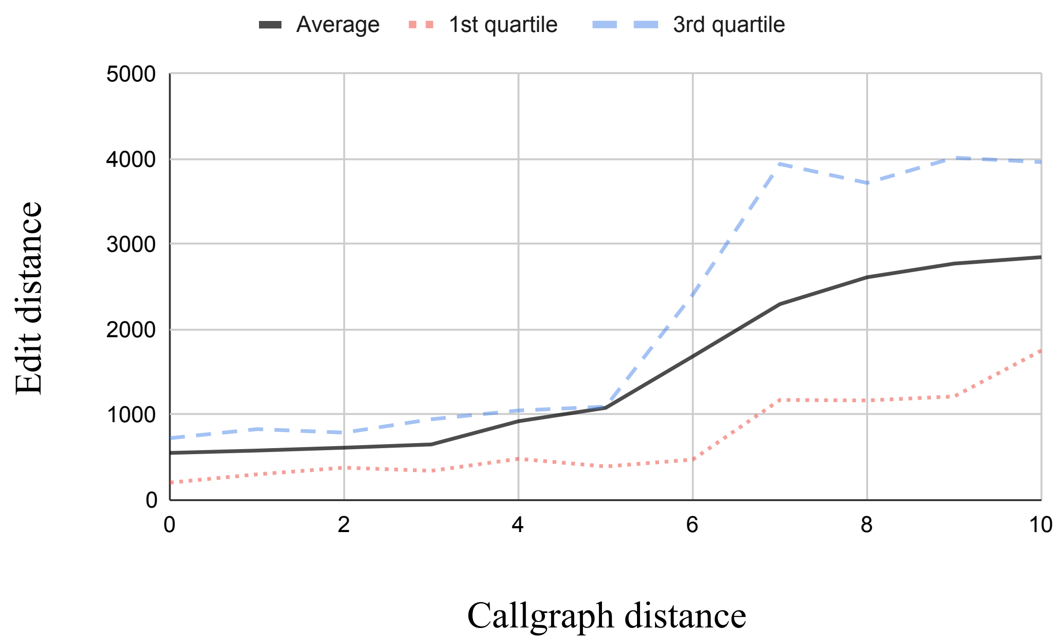

Figure 5.3: Results of the edit distance experiment.

5.3.2 Known bug experiment

We implement a prototype of the fuzzing-directed symbolic execution that works end-to-end, and use known bugs reported by Syzbot to evaluate it. Given a known bug, we can identify the code location where the bug is triggered. The function in which the bug occurs is the target function, and is provided to our system to simulate the static analysis results that would be provided to our system. To begin with, we collect reachable nodes that are close to the target function. We then rank those reachable nodes according to the result of distance analysis, and execute tests associated with top-ranked nodes symbolically. We declare success if the known bug is reproduced by symbolic execution of a test.

Unfortunately, we are not able to find a working example in the available time. In the rest of this section, we provide more details about the experiment, and discuss about the various reasons for the failed experiment.

Reproduce known bugs on S2E

Before we reproduce a known bug by symbolic execution, we must confirm that the bug is reproducible under S2E. To collect valid bugs for the rest of this experiment, we manually execute the reproducers of known bugs under S2E, and collect those that can be successfully reproduced.

A Linux kernel bug is reproducible only if the Linux kernel includes the code causing the bug and a patch to fix it is not yet applied. Therefore, we need different versions of Linux kernels to reproduce bugs introduced or fixed in different time. On the other hand, adding S2E support for different versions of Linux kernels requires a non-negligible amount of work: to support a new version of Linux kernel, S2e supporting code must be ported to this kernel, and the kernel configuration should be tailored to meet S2E's requirement. Instead of adding support for a variety of Linux kernels in different versions, we focus on bugs that are likely to be reproduced in Linux kernel 6.8.2, which is the built-in version of Linux kernel of S2E. Since Linux kernel 6.8.2 is released on May 26th this year, we only check open bugs reported this year until May 26th, or fixed bugs reported from May 26th to April 30th.

Among all 185 bugs in the selected time period, 132 bugs have at least one C

```
0   hugetlb_change_protection+0x77a/0x1330
1   mas_store_prealloc+0xd3/0x1c0
2   uprobe_apply+0x120/0x120 (uprobe_mmap)
3   change_protection+0xc4c/0x1f50
4   uprobe_apply+0x120/0x120 (uprobe_mmap)
5   vma_complete+0x42b/0xe80
6   vma_prepare+0x404/0x620
7   __split_vma+0x8dc/0xbb0
8   can_change_pte_writable+0x2b0/0x2b0(change_protection)
9   vma_needs_dirty_tracking+0x1f0/0x1f0(vma_wants_writenotify)
10  vma_set_page_prot+0x94/0x100
11  mprotect_fixup+0x373/0x900
12  apparmor_file_mprotect+0x215/0x3a0
13  do_mprotect_pkey+0x727/0x9d0
14  __x64_sys_mprotect+0xe9/0x290
```

Listing 5.1: Call stack trace of the target bug

reproducer. We run each of them under S2E and identified 10 reproducible bugs. Among those bugs, we select WARNING in hugetlb_change_protection[1] as the target bug, given that the reproducer of this bug is simpler compared to other reproducible bugs.

End-to-end experiment with the target bug

The bug WARNING in hugetlb_change_protection is a warning triggered unexpectedly after the new function to mark memory ranges as poisoned is implemented. As shown in the call stack (Figure 5.1), the bug occurs at function `hugetlb_change_protection`. The most relevant code is shown in Listing 5.2. Before memory poisoning was implemented, the only possible PTE mark is UFFD_WP. Therefore, the warning at line 8 should never be triggered. However, with the newly introduced poison feature, the PTE mark can be either UFFD_WP or POISON. Instead of raising a warning and continuing execution, the code should check the mark, and only continue changing protection for the huge TLB if the mark is UFFD_WP.

Figure 5.3 shows the Syzkaller internal reproducer of the bug, which involves a sequence of system calls. To trigger the target bug, a memory region is allocated by `mmap`, and the region is made up of huge pages. In addition, the allocated memory region is poisoned by `ioctl`. After that, the following `mprotect` can trigger this bug if

[1]https://syzkaller.appspot.com/bug?extid=b07c8ac8eee3d4d8440f

```
1  long hugetlb_change_protection(struct vm_area_struct *vma,
2                  unsigned long address, unsigned long end,
3                  pgprot_t newprot, unsigned long cp_flags)
4  {
5      ...
6      } else if (unlikely(is_pte_marker(pte))) {
7              /* No other markers apply for now. */
8              WARN_ON_ONCE(!pte_marker_uffd_wp(pte));
9              if (uffd_wp_resolve)
10             /* Safe to modify directly (non-present->none). */
11                     huge_pte_clear(mm, address, ptep, psize);
12     } else if (!huge_pte_none(pte)) {
13     ...
```

Listing 5.2: Target function of the bug

```
mmap$IORING_OFF_SQ_RING(&(0x20400000/0xc00000)=nil, 0xc00000, 0x0,
    0x5d032, -1, 0x0)
r0 = userfaultfd(0x80001)
ioctl$UFFDIO_API(r0, 0xc018aa3f, &(0x200000c0))
ioctl$UFFDIO_REGISTER(r0, 0xc020aa00, &(0x20000040)=
    {{&(0x20400000/0xc00000)=nil, 0xc00000}, 0x5})
ioctl$UFFDIO_POISON(r0, 0xc020aa08, &(0x20000080)=
    {{&(0x20400000/0xc00000)=nil, 0xc00000}})
mprotect(&(0x20000000/0x800000)=nil, 0x800000, 0x6)
```

Listing 5.3: Syz reproducer of the target bug

it attempts to change the protection flags of a huge page (page larger than 2MB) that overlaps with the poisoned one.

We start the experiment by collecting reachable nodes to `hugetlb_change_protection` and sort them by call graph distance. For all 12 reachable nodes whose distance is no more than 2, we execute the associated test under S2E in concolic mode, and switch to symbolic mode at the function pointed by the reachable node. Unfortunately, none of them can reach the target bug. More details of the results are demonstrated and discussed in the next section.

Analysis of the results

In this section, we analyze the results to discuss possible causes of the failed experiment.

Engineering limitations As previously mentioned, only 10 bugs reported by Syzbot are reproducible in S2E. In addition, most of them are unsuitable for symbolic execution, since reproducing them requires inserting a large chunk of symbolic data, which can significantly decrease the performance of S2E. On the other hand, we may be able to find a working example if we have more reproducible tests. One way to collect more reproducible bugs is to run the experiment in larger scale: instead of running tests manually on Linux kernel 6.8.2, we port S2E support to various different versions of Linux kernels, and automatically run each bug reproducer on a compatible version of Linux kernel. In addition, the kernels used by Syzkaller are built with allyesconfig, where most kernel components are compiled. For S2E, it is recommended to turn off unnecessary components unless you want to test them, since a kernel compiled with allyesconfig can decrease the performance of S2E. If we can find a workaround for the performance issue, we will be able to collect more reproducible bugs.

Limitation of symbolic execution When directed by a test, symbolic execution only executes the same sequence of system calls. If none of the directing tests involves all the system calls that are required to trigger a bug, then it will be impossible to reach the target bug by our approach. For example, to trigger the bug WARNING in hugetlb_change_protection, it is necessary to execute `mmap`, `mprotect` and the three `ioctl` commands. However, the tests that can get closest to hugetlb_change_protection only call `mmap` and `mprotect`. One way to overcome this limitation is to extend symbolic execution so that it can insert new system calls and run them symbolically. In the next section, we demonstrate the effectiveness of this idea by creating and experiment with a proof-of-concept example.

5.3.3 Demonstration of extended symbolic execution

As previously discussed, extended symbolic execution tool that can insert new system calls is helpful for our purpose. Instead of making changes to the symbolic execution system, we simulate the expected behavior in the test program.

Figure 5.4 shows the simplified version of the proof-of-concept example. All variables with sym_ as their prefix are symbolic values. The `mmap` at line 22 and the`mprotect` at line 31 are from a top-ranked test in distance analysis. Since the test never poison

```
1  void run_syscall(uint8_t seed){
2      uint8_t s = seed % 4;
3      switch(s){
4          case 0:
5              // run userfaultfd
6              ...
7          case 1:
8              // run ioctl$UFFDIO_API symbolically
9              ...
10         case 2:
11             // run ioctl$UFFDIO_REGISTER symbolically
12             ...
13         case 3:
14             // run iotcl$UFFDIO_CONTINUE symbolically
15     }
16 }
17
18 int main(void){
19     // Initialize the test and declare variables
20     ...
21
22     syscall(__NR_mmap, 0x20400000ul, 0xc00000ul, sym_prot, sym_flags, -1,
        0);
23
24     // Simulation of additional system calls inserted by extended
25     // symbolic execution system
26     run_syscall(sym_rand1);
27     run_syscall(sym_rand2);
28     run_syscall(sym_rand3);
29     run_syscall(sym_rand4);
30
31     syscall(__NR_mprotect, 0x20000000ul, 0x800000ul, sym_prot_2);
32 }
```

Listing 5.4: A simplified version of the proof-of-concept example

memory, it is impossible to reach the WARNING in hugetlb_change_protection bug by symbolically executing it.

The 4 occurrences of `run_syscall()` simulate extra system calls inserted by the extended symbolic execution system. `run_syscall()` takes a symbolic byte as input, and selects one system call to execute symbolically according to the symbolic input. With this design, the symbolic execution system can try different combinations of system calls in `run_syscall()`. For example, when `run_syscall()` contains 4 candidate system calls, S2E can explore all 256 combinations of them.

We execute this example under S2E and measure the time it takes to trigger the target bug for the first time. We run the experiment on a server with 16 Intel(R) Xeon(R) CPU E5-2623, and parallelize S2E in 16 processes. This example takes 42 minutes to trigger the bug for the first time. However, this example is created based on the assumption that we know what system calls are needed and how many additional system calls should be inserted. In a real-world example, we will have to involve more candidate system calls, and try more combinations. For example, if there are 7 candidate system calls instead of 4, we will need to try 2401 combinations at most, which can take 9 times as long as the 4-candidate example.

5.4 Future Work

As mentioned in section 5.3.2, we may be able to find working examples if more known bugs are reproducible on S2E. To achieve this goal, we can either port S2E support to more versions of Linux kernels or build S2E with more subsystems turned on. Once the system can stably reproduce known bugs, the next step is to integrate our system with a static analysis system: instead of bug locations reported on Syzbot, we take bug candidates reported by a static bug-detecting system and try to find tests that trigger this bug by our system.

Currently, our system only executes each step once, and there is no feedback from later steps to earlier steps. Alternatively, we can build a system that is more similar to a hybrid fuzzing system: we can take new tests generated by symbolic execution and feed them back to Syzkaller as seed tests. By doing that, Syzkaller may be able to generate tests that can get close to the target in less time.

5.5 Related Work

Greybox OS kernel fuzzing Greybox fuzzing has proved to be a practical approach for kernel bug finding. In addition to general fuzzing, greybox fuzzers prioritize generating tests by code coverage. As the most well-known greybox OS kernel fuzzer, Syzkaller has discovered thousands of Linux kernel bugs [24]. Syzkaller replies on KCOV to collect code coverage, which requires tested kernels to be instrumented at compile time. To avoid invalid inputs for kernel syscalls, Syzkaller generates tests according to manually created syscall descriptions. Unlike Syzkaller, kAFL [25] applies hardware-based coverage-collection approach using Intel's Processor Trace. This approach can drastically increases performance of the fuzzer, and compiler instrumentation is no longer required. FUZZNG [26] is a more recent greybox fuzzer that is similar to Syzkaller but does not require syscall descriptions. Instead of generating inputs based on syscall descriptions, FUZZNG reshapes inputs associated with file descriptors and pointers by populating data from user-space applications. For other input, FUZZNG still uses random data produced by Syzkaller.

Hybrid fuzzing Hybrid fuzzing is a promising program analysis technique mostly used in vulnerability detection. The idea is to combine symbolic execution with fuzzing. One way to integrate symbolic execution with fuzzing is to switch to symbolic execution when fuzzing no longer increases coverage. In this way, the hybrid system attempts to explore code only reachable after passing complicated conditions. Those two techniques complement each other: symbolic execution is strong at reasoning about complicated conditions compared to fuzzing, while using a fuzzer for less complicated code can avoid the scalability issues of symbolic execution. Hybrid systems such as Driller [27] have been proposed for user-space program bug finding. As a closer analogue to our system, HFL [28] applied OS kernel hybrid fuzzing with the help of Syzkaller and S2E.

Our system is inspired by hybrid fuzzing but different in two ways. Firstly, our system does not switch back and forth between fuzzing and symbolic execution. Instead, we perform fuzzing until a reachable node, and use symbolic execution to find the path from those nodes to the target. Secondly, the goal of hybrid fuzzing is to achieve high coverage and discover new bugs, while our system is aimed at verifying a candidate bug by reaching its code location.

Directed fuzzing Fuzzers can be directed by target locations of bug candidates. The basic idea is to compute the distance between currently reachable code and those targets, and prioritize tests that can cover code closer to targets. Many directed fuzzing systems have been proposed since the idea was first introduced by AFLgo [29]. Among all those directed fuzzing systems, some of them are specifically designed for OS kernels, with SyzDirect [30] being the state-of-art OS kernel fuzzer. Our technique is related to directed fuzzing to the extent that we also perform distance based analysis. Nonetheless, the system we are going to build is fundamentally different from a directed fuzzer: instead of directing fuzzing by target locations, we use both the targets and fuzzing to direct symbolic execution. Given the nature of symbolic execution, it is promising that our system can get through complicated conditions more easily compared to fuzzers, which is the major reason that motivates this project.

References

[1] Koushik Sen, Darko Marinov, Gul Agha, Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. *10th European software engineering conference and 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE'05)*, 2005.

[2] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. *SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005.*

[3] Stephen McCamant et al. Fuzzball: Vine-based binary symbolic execution. `https://github.com/bitblaze-fuzzball/fuzzball`.

[4] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, 2008.

[5] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008.

[6] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification, 19th International Conference, (CAV 2007)*, 2007.

[7] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[8] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing CPU emulators. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, 2009.

[9] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing system virtual machines. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, 2010.

[10] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. Virtual cpu validation. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, 2015.

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

[12] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 2004.

[13] Domagoj Babi, Lorenzo Martignoni, and Dawn Song. Statically-directed dynamic automated test generation. *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11.*

[14] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Stephen McCamant, and Dawn Song. HI-CFG: Construction by Binary Analysis and Application to Attack Polymorphism. In *Computer Security – ESORICS 2013*. Springer Berlin Heidelberg, 2013.

[15] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*, 2011.

[16] SoSy-Lab. SV-COMP. https://sv-comp.sosy-lab.org/.

[17] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. *Proceedings of the eighteenth international symposium on Software testing and analysis ISSTA 09*, 2009.

[18] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. Proteus: computing disjunctive loop summary via path dependency analysis. *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering(FSE'16)*, 2016.

[19] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E : A Platform for In-Vivo Multi-Path Analysis of Software Systems. *Asplos*, 2011.

[20] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, 2016.

[21] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings 2015 Network and Distributed System Security Symposium*.

[22] Kangjie Lu. Practical Program Modularization with Type-Based Dependence Analysis. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P'23)*, San Francisco, CA, May 2023.

[23] Google. syzbot. https://syzkaller.appspot.com/upstream.

[24] Google. Fixed bugs detected by syzbot. https://syzkaller.appspot.com/upstream/fixed.

[25] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. *26th USENIX Security Symposium (USENIX Security '17)*.

[26] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. No Grammar, No Problem: Towards Fuzzing the Linux Kernel without System-Call Descriptions. In *Proceedings 2023 Network and Distributed System Security Symposium.*

[27] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings 2016 Network and Distributed System Security Symposium*, 2016.

[28] Kyungtae Kim, Dae R Jeong, Chung Hwan, Kim Yeongjin, Jang Insik, and Shin Byoungyoung. HFL : Hybrid Fuzzing on the Linux Kernel. In *Proceedings 2020 Network and Distributed System Security Symposium.*

[29] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[30] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. SyzDirect: Directed Greybox Fuzzing for Linux Kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, Copenhagen Denmark.