CONCOLIC EXECUTION TAILORED FOR HYBRID FUZZING

A Dissertation
Presented to
The Academic Faculty

By

Insu Yun

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computing
Department of Computer Science

Georgia Institute of Technology

December 2020

CONCOLIC EXECUTION TAILORED FOR HYBRID FUZZING

Date approved: November 18, 2020

ACKNOWLEDGEMENTS

First of all, I would like to thank to Prof. Taesoo Kim for his guidance and support through my entire Ph.D. program. He was indeed a perfect advisor. He nurtured me to become an independent researcher in system security, and he kindly taught me how to discover a worthwhile research problem, how to explore it, and how to illustrate my idea technically. He is not only an exceptional advisor but also a wonderful person. I always enjoyed chatting with him, and our talks helped me relax from the stress of graduate studies. I hope one day I can become an outstanding advisor to my future students like Prof. Taesoo has been for me.

I also would like to acknowledge my thesis committee: Dr. Weidong Cui, Prof. Wenke Lee, Prof. Mayur Naik, and Prof. Alessandro Orso, for their invaluable comments and suggestions for my dissertation. Dr. Weidong Cui mentored me to develop a background in system security at Microsoft Research, and I am grateful to Prof. Mayur Naik for his help on my first research at Georgia Institute of Technology.

I have been fortunate to collaborate with many brilliant and friendly researchers at System Software and Security Lab and Microsoft research: Prof. Sanidhya Kashyap, Prof. Meng Xu, Dr. Ming-Wei Shih, Dr. Steffen Maass, Dr. Mohan Kumar, Dr. Sangho Lee, Prof. Changwoo Min, Prof. Byoungyoung Lee, Prof. Chengyu Song, Prof. Kangjie Lu, Prof. Hong Hu, Wen Xu, Jinho Jung, Ren Ding, Dr. Chanil Jeon, Dr. Woonhak Kang, Soyeon Park, ChangSeok Oh, Fan Sang, Seulbae Kim, Dr. Daehee Jang, Dr. Hyunjoon Koo, Hanqing Zhao, Yechan Bae, Sujin Park, Mansour Alharthi, Ammar Askar, Jungwon Lim, Yonghwi Jin, Prof. Hyungon Moon, Jungyeon Yoon, Dr. Xinyang Ge, Dr. Ben Niu, and Prof. Ryuou Wang. I also want to express my special thanks to Prof. Yeongjin Jang for being a good friend, a great collaborator, and an attentive mentor since my undergraduate days.

I would like to thank my parents, Nanhui Lee and Deokgwon Kang, and my parents-inlaw, Byeongseok So and Hyunju Lee. I am especially grateful to my mother for her endless love and support. I also wish to thank my wife, Eunhye, and my lovely daughter, Lily. I could not finish this journey without my wife. Like the meaning of her name, she is an amazing grace. Lastly, I give my thanks to God for everything in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Recently, hybrid fuzzing, which combines fuzzing and concolic execution, has been highlighted to overcome limitations of both techniques. Despite its success in contrived programs such as DARPA Cyber Grand Challenge (CGC), it still falls short in finding bugs in real-world software due to its low performance of existing concolic executors.

To address this issue, this dissertation suggests and demonstrates *concolic execution tailored for hybrid fuzzing* with two systems; QSYM and HYBRIDRA. First, we present QSYM, a binary-only concolic executor tailored for hybrid fuzzing. It significantly improves the performance of conventional concolic executors by removing redundant symbolic emulations for a binary. Moreover, to efficiently produce test cases for fuzzing, even sacrificing its soundness, QSYM introduces two key techniques: optimistic solving and basic block pruning. As a result, QSYM outperforms state-of-the-art fuzzers, and, more importantly, it found 13 new bugs in eight real-world programs, including `file`, `ffmpeg`, and `OpenJPEG`.

Enhancing the key idea of QSYM, we discuss HYBRIDRA, a new concolic executor for file systems. To apply hybrid fuzzing for file systems, which are gigantic and convoluted, HYBRIDRA employs compilation-based concolic execution to boost concolic execution leveraging the existing of source code. Moreover, HYBRIDRA introduces a new technique called staged reduction, which combines existing heuristics to efficiently generate test cases for file systems. As a result, HYBRIDRA outperforms a state-of-the-art file system fuzzer, Hydra, by achieving higher code coverage, and successfully discovered four new bugs in `btrfs`, which has been heavily tested by other fuzzers.

CHAPTER 1
INTRODUCTION

1.1 Problem Statement

The computer science community has developed two notable technologies to automatically find vulnerabilities in software: coverage-guided fuzzing [1, 2, 3] and concolic execution [4, 5]. Coverage-guided fuzzing can quickly explore the input space at nearly native speed, but it is only good at discovering inputs that lead to an execution path with loose branch conditions because of its random exploration. On the contrary, concolic execution is good at finding inputs that drive the program into tight and complex branch conditions, but it is very expensive and slow to formulate and solve these constraints. More seriously, it suffers from a fundamental limitation, namely, path explosion; the number of paths of a program grows exponentially based on the program size. To take advantage of both worlds, a hybrid approach [6, 7, 8], called *hybrid fuzzing*, was recently proposed. It combines both fuzzing and concolic execution, with the hope that the fuzzer will quickly explore trivial input spaces (i.e., loose conditions) and the concolic execution will solve the complex branches (i.e., tight conditions). By selectively applying concolic execution, hybrid fuzzing can avoid its high cost and path explosion problem. For example, Driller [8] demonstrates the effectiveness of the hybrid fuzzing in DARPA Cyber Grand Challenge (CGC) binaries—generating six new crashing inputs out of 126 binaries that are not possible when running either fuzzing or concolic execution alone.

Unfortunately, these hybrid fuzzers still suffer from scaling to find bugs in non-trivial, real-world applications. We observed two main performance bottlenecks of their concolic executors. First, the symbolic emulation of concolic executors is too slow in formulating path constraints. This overhead has been underestimated in classical concolic execution

```c
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
  very_complicated_logic(buf);

  if (x * x == 1234 * 1234)
    crash();
}
```

(a) This example shows that completeness of concolic execution may result in inefficiency in test case generation. In particular, `very_complicated_logic()` in concolic execution blocks its further exploration, failing in discovering `crash()`.

```c
// 'x' is symbolic and 'x' == 0 in a given input
int soundess(int x) {
  if (x == 0)
    do_something();

  if (x * x == 1234 * 1234)
    crash();
}
```

(b) Similarly, this example shows a negative impact of soundness in concolic execution. Since `x` is concretely less than zero, it makes a path for finding `crash()` unsatisfiable. To discover this crash, concolic execution needs to be re-executed, which requires non-trivial efforts.

Figure 1.1: Examples that shows negative impacts of completeness and soundness in hybrid fuzzing. Completeness often blocks further exploration of concolic execution in hybrid fuzzing (Figure 1.1a), and soundness incurs significant overhead due to repetitive analysis (Figure 1.1b).

because it can be alleviated using state forking [9]. State forking, which utilizes knowledge from previous executions in exploring neighboring execution paths, can avoid the recurrent cost from the symbolic emulation. However, hybrid fuzzing explores paths randomly, which is different from concolic execution's systematic exploration; therefore, state forking is limited in hybrid fuzzing because of the few neighboring paths. Thus, concolic execution in hybrid fuzzing needs to repeat symbolic emulation, making its overhead more serious than the classical one.

Second, classical concolic executors aim to maximize completeness and soundness, resulting in inefficient test case generation for hybrid fuzzing. In any analysis including concolic execution, completeness and soundness are desirable but not without trade offs. As shown in Figure 1.1, complete concolic execution attempts to analyze every logic in a program. This can block further exploration of concolic execution if concolic execution

Table 1.1: Summary of techniques used to build concolic execution tailored for hybrid fuzzing in QSYM and HYBRIDRA.

	Fast symbolic emulation	**Heuristics for generating test cases**
QSYM	Instruction-level concolic execution (For binary, §3.3.1)	Optimistic solving (§3.3.2) Basic block pruning (§3.3.3)
HYBRIDRA	Compilation-based concolic execution (For source code, §4.3.2)	Staged reduction (§4.3.3) + Heuristics from QSYM

encounters an extremely complicated routine, making it give up other interesting, yet accessible, paths. Moreover, soundness requires conservative analysis; This often incurs redundant re-executions, whose costs can be significant in hybrid fuzzing. For example, in Figure 1.1b, if `x` is concretely defined as zero, concolic execution will conclude that a branch in Line 7 is unsatisfiable because `{x == 0}` and `{x * x == 1234 * 1234}` conflict. However, in this example, if a program hits Line 7 regardless of `do_something()`, it is safe to ignore the conflicting constraint, `{x == 0}`, from Line 4. This relaxation is not always acceptable because of implicit data or control flow. Therefore, classical concolic execution waits for a new input that can satisfy Line 7 without breaking its completeness. In hybrid fuzzing, this strategy significantly delays the discovery of `crash()`, particularly in a large program; concolic execution can encounter such a good input after consuming enormous number of other test cases from fuzzing.

1.2 Research Outline

To overcome the aforementioned issues, this thesis proposes *concolic execution tailored for hybrid fuzzing*. Unlike classical concolic execution, this specialized concolic execution employs two fundamental techniques: ① systematic approaches for fast symbolic emulation and ② heuristics for generating test cases in hybrid fuzzing. We demonstrate our ideas with two systems: QSYM (§3) and HYBRIDRA (§4).

Table 1.1 summarizes techniques used for hybrid fuzzing in this dissertation. In particular, this dissertation presents ① instruction-level concolic execution (§3.3.1) and an alternative design for compilation-based concolic execution (§4.3.2) [10, 11, 12] for fast symbolic emulation in binary-only and open-source applications, respectively. Moreover, it proposes ② several heuristics for test case generations: optimistic solving (§3.3.2), basic block pruning (§3.3.3), and staged reduction (§4.3.3). Optimistic solving and basic block pruning makes concolic execution efficiently generate test cases while sacrificing soundness and completeness of concolic execution. This is based on our key observation in hybrid fuzzing, in which coverage-guided fuzzing can act as an efficient validator to filter out incorrect test cases. Moreover, staged reduction combines existing heuristics for test case generation to take an specific advantage of each mechanism.

In the following, we briefly introduce QSYM and HYBRIDRA. QSYM is a fast concolic execution engine to support hybrid fuzzing in real-world user applications. Its key idea is to tightly integrate the symbolic emulation with the native execution using dynamic binary translation, making it possible to implement more fine-grained, and thus faster, instruction-level symbolic emulation. Additionally, QSYM loosens the strict soundness requirements of conventional concolic executors for better performance, but takes advantage of a faster fuzzer for validation, providing unprecedented opportunities for performance optimizations, e.g., optimistically solving constraints and pruning uninteresting basic blocks. Our evaluation shows that QSYM does not just outperform state-of-the-art fuzzers (i.e., found 14× more bugs than VUzzer in the LAVA-M dataset, and outperformed Driller in 104 binaries out of 126), but also found 13 previously unknown security bugs in eight real-world programs like Dropbox Lepton, ffmpeg, and OpenJPEG, which have already been intensively tested by state-of-the-art fuzzers, AFL and OSS-Fuzz. QSYM was published in USENIX Security Symposium 2018 in collaboration with Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim [13].

HYBRIDRA extends this idea to a more gigantic system, file systems in Linux Kernel. To reduce the overhead from binary translation, HYBRIDRA benefits from the existence of source code by instrumenting it for concolic execution. Unlike SymCC [10], HYBRIDRA adopts shadow-memory-based memory modeling from Kirenenko [14], which is faster and more resilient to multi-threading. Moreover, HYBRIDRA adopts a new test case generation mechanism, called stage reduction, which gradually applies various reduction mechanisms to generate test cases. We applied HYBRIDRA to four popular file systems: `btrfs`, `ext4`, `ftfs`, and `xfs`; HYBRIDRA discovered 10 crashes with four new bugs. Hybrid fuzzing directly helps HYBRIDRA discover these bugs by understanding buggy conditions in Linux Kernel. Moreover, our evaluation shows that stage reduction outperforms other reduction mechanisms, and HYBRIDRA can achieve more code coverage than the fuzzing-only approach [15], which shows its effectiveness.

CHAPTER 2
RELATED WORK

In this chapter, we introduce the current state-of-the-art approaches for fuzzing, concolic execution, and hybrid fuzzing and compare them with our approaches, QSYM and HYBRIDRA.

2.1 Coverage-guided Fuzzing

Fuzzing is a dynamic software testing technique that controls a program's execution by randomly generating an input. Fuzzing runs a program with mutated inputs and hopes that the input will eventually drive the program to an erroneous program state, such as a crash or an assertion error (i.e., a bug). Thanks to its simple nature, fuzzing is largely scalable and applicable to any type of program. Consequently, fuzzing has been largely used for finding bugs in various fields [16], including file parsers [1, 17, 18], compilers [19, 20, 21, 22], browsers [23, 24, 25, 26, 27, 28], and operating systems [29, 30, 31, 32, 15, 33].

Coverage-guided fuzzing has become popular, especially since AFL [1] has shown its effectiveness. AFL prioritizes inputs that likely reveal new paths by collecting coverage information during program execution to assess generated inputs, enabling quick coverage expansion. Also, AFLFast [34] uses a Markov chain model to prioritize paths with low reachability, and CollAFL [35] provides accurate coverage information to mitigate path collisions. libFuzzer [36] provides a useful set of library functions for in-memory, coverage-guided fuzzing. Entropic [37] suggests an entropy-based power schedule to boost the efficiently of coverage-guided fuzzing.

However, fuzzing has a fundamental limitation: it cannot traverse paths beyond narrow-ranged input constraints (e.g., a magic value). To overcome such a limitation, BuzzFuzz and VUzzer [38, 39] develop application-aware mutation techniques by performing taint analysis. Steelix [40] recovers the correct magic values by collecting comparison progress information during program execution. FairFuzz [41] discovers magic values and prevents their mutations with program analysis and heuristics. Angora [42] adopts taint tracking, shape and type inference, and a gradient-descent-based search strategy to solve path constraints efficiently. Neuzz [43] suggests neural program smoothing and utilizes gradient-guided fuzzing instead of using the evolutionary algorithm. REDQUEEN [44] suggests efficient fuzzing using input-to-state correspondence, in which part of a given input appears in the program state without any modification. TaintScope [45] suggests an automatic way to avoid a checksum routine in a program leveraging the inherent properties of a checksum — imbalanced dependencies between the checksum field and the calculated one. Eclipser [46] suggests gray-box concolic testing, which can efficiently handle linear and monotonic constraints. These approaches, however, can only handle certain types of constraints. In contrast, QSYM and HYBRIDRA rely on symbolic execution such that it has a chance to satisfy any kinds of constraints.

2.2 Concolic execution

Symbolic execution [47, 48, 49, 50] has been widely used for software testing; it represents a program's execution in the form of symbolic expressions instead of concretely running it. Then, it systematically explores a program by flipping each symbolic constraint and generates a corresponding test case using a constraints solver — a tool that engenders a test case from symbolic constraints. CUTE [12] and DART [11] suggest concolic execution to overcome the limitations of their underlying constraint solver, which combines symbolic execution with a concrete one. In particular, their solver fails to support non-linear arithmetic or symbolic array indexing; therefore, they replace these unsupported expressions with concrete values from concrete execution. Equipped with a more powerful solver — STP [51] — EXE [52] successfully handles non-linear arithmetic expressions and symbolic dereferences, and its successor, KLEE [9], redesigns EXE to use LLVM IR and found 56 bugs in GNU coreutils. Even with such a powerful solver, concolic execution is still helpful; KLEE [9] relies on concrete values to emulate unmodeled operations, including system calls. S2E [5] suggests full system concolic execution by leveraging KLEE and QEMU [53]. Accordingly, S2E is capable of analyzing any part of a system, including third-party libraries and even a kernel. Mousse [54] further supports concolic execution even in an untamed environment, which cannot be emulated.

However, concolic execution suffers from an infamous limitation, called path explosion, in which the number of paths to explore grows exponentially with program size. To mitigate this problem, SAGE [4, 55] proposes a generational search to maximize the number of test cases in one execution and applies unrelated constraint solving [12]. Cloud9 [56] and Veritesting [57] merge states to reduce their numbers using static analysis and dynamic symbolic execution. Dowser [58] and DASE [59] suggest techniques to prioritize concolic execution for more interesting paths; to guide concolic execution, they leverage the characteristics of buffer overflow vulnerabilities and documentations, respectively. Mayhem [60] combines forking-based symbolic execution and re-execution-based symbolic execution to balance performance and memory usage.

On the contrary, QSYM and HYBRIDRA use (1) fuzzing to explore most paths to avoid the path explosion problem, (2) generic heuristics (e.g., basic block pruning) without assuming any specific bug type or domain-specific knowledge, and (3) re-execution-based symbolic execution for better performance. QSYM and HYBRIDRA are highly inspired by whitebox fuzzers (e.g., SAGE) and have adopted SAGE's design, including the generational search algorithm and unrelated constraint solving. However, this dissertation has several differences from whitebox fuzzers. First, our goal is to make interesting test cases for fuzzing but not to explore the program individually. This makes a big difference in our design decisions; for example, if an algorithm can generate a number of interesting test cases, which can also be discovered by fuzzing, the algorithm would not be useful for our purpose. Second, in hybrid fuzzing, our exploration of a program is random, not systematic. Thus, optimization techniques based on systematic exploration [9, 60] are no longer useful in hybrid fuzzing. Third, coverage-guided fuzzing [1] allows us to adopt more aggressive strategies for test case generation. Thanks to its high performance for evaluating incorrect test cases, we can apply several techniques that are expected to generate many incorrect test cases, e.g., optimistic solving in QSYM.

2.3 Hybrid Fuzzing

The concept of hybrid fuzzing was first proposed by Majumdar and Sen [6]. Later, Driller [8] demonstrated its effectiveness in DARPA CGC with a refined implementation. In both studies, the majority of path exploration is offloaded to the fuzzer, while concolic execution is selectively used to drive execution across the paths that are guarded by narrow-ranged constraints. Pak [7] also proposes a similar idea, but it is limited to the frontier nodes that are mainly magic value checks at early execution stages. However, these hybrid fuzzers use general concolic executors that are not only slow but also incompatible with hybrid fuzzing. On the contrary, QSYM is tailored for hybrid fuzzing, so that it can scale to detect bugs from real-world software. Similar to HYBRIDRA, SymCC [10] recently revisits compilation-based concolic execution from CUTE [12] and DART [11]. HYBRIDRA takes an alternative design for compilation-based concolic execution using shadow memory, which has lower overhead than SymCC's [61]. HFL [62] combines Syzkaller with S2E for kernel hybrid fuzzing; however, it fails to address the technical challenges in a file system fuzzing unlike HYBRIDRA. DigFuzz [63] and MEUZZ [64] suggest special seed selection algorithms for hybrid fuzzing, which can be applied to QSYM and HYBRIDRA.

2.4 File system fuzzing

To discover bugs in file systems, numerous fuzzing frameworks for file systems have been proposed. Because of the popularity of file systems, generic kernel fuzzing frameworks [65, 29, 66, 67] such as Syzkaller include file systems as part of their fuzzing targets; however, such systems are limited to evaluate system calls without considering file system images, which also heavily determine the behaviors of file systems. To overcome this issue, Janus [68] proposes two-dimensional fuzzing for file systems; Janus randomly generates not only system calls but also file system images to discover memory corruption vulnerabilities in file systems. Hydra [15] further extends Janus to discover other types of file system bugs beyond memory corruptions. Compared to these solutions, HYBRIDRA improves two-dimensional fuzzing by supporting powerful concolic execution that can resolve the fundamental limitations of existing fuzzing.

CHAPTER 3
QSYM: A BINARY-LEVEL CONCOLIC EXECUTION ENGINE TAILED FOR HYBRID FUZZING

3.1 Introduction

In this work, we first systematically analyze the performance bottlenecks of concolic execution and then overcome the problem by tailoring the concolic executor to support hybrid fuzzing (§3.2). The key idea of QSYM is to tightly integrate the symbolic emulation to the native execution using dynamic binary translation. Such an approach provides unprecedented opportunities to implement more fine-grained, instruction-level symbolic emulation that can minimize the use of expensive symbolic execution (§3.3.1). Unlike our approach, current concolic executors employ coarse-grained, basic-block-level taint tracking and symbolic emulation, which incur non-negligible overheads to the concolic execution.

Additionally, we alleviate the strict soundness requirements of conventional concolic executors to achieve better performance as well as to make it scalable to real-world programs. Such incompleteness or unsoundness of constraints is not a problem in a hybrid fuzzer where a co-running fuzzer can quickly validate the newly generated test cases; the fuzzer can quickly discard them if they are invalid. Moreover, this approach makes it possible to implement a few practical techniques to generate new test cases, i.e., by optimistically solving some parts of constraints (§3.3.2), and to improve the performance, i.e., by pruning uninteresting basic blocks (§3.3.3). These new techniques and optimizations together allow QSYM to scale to test real-world programs.

Our evaluation shows that the hybrid fuzzer, QSYM, —built on top of our concolic executor, and the state-of-the-art fuzzer, AFL—outperforms all existing fuzzers like Driller [8] and VUzzer [39]. QSYM achieved significantly better code coverage than Driller in 104 out of 126 DARPA CGC binaries (tied in five challenges). Further, QSYM discovered 1,368 bugs out of 2,265 bugs in the LAVA-M test set [69], whereas VUzzer found 95 bugs.

More importantly, QSYM scales to testing complex real-world applications. It has found *13 previously unknown vulnerabilities* in *eight* non-trivial programs, including ffmpeg and OpenJPEG. It is worth noting that these programs have been thoroughly tested by other state-of-the-art fuzzers such as AFL and OSS-Fuzz, highlighting the effectiveness of our concolic executor. OSS-Fuzz running on a distributed fuzzing infrastructure with hundreds of servers [70] was unable to find these bugs, but QSYM found them by using *a single workstation*. For further research, we open-source the prototype of QSYM at https://github.com/sslab-gatech/qsym.

This work makes the following contributions:

- **Fast concolic execution through efficient emulation.** We improved the performance of concolic execution by optimizing emulation speed and reducing emulation usage. Our analysis identified that symbol generation emulation was the major performance bottleneck of concolic execution such that we resolved it with instruction-level selective symbolic execution, advanced constraints optimization techniques, and tied symbolic and concolic executions.
- **Efficient repetitive testing and concrete environment.** The efficiency of QSYM makes re-execution-based repetitive testing and the concrete execution of external environments practical. Because of this, QSYM is free from snapshots incurring significant performance degradation and incomplete environment models resulting in incorrect symbolic execution due to its non-reusable nature.
- **New heuristics for hybrid fuzzing.** We proposed new heuristics tailored for hybrid fuzzing to solve unsatisfiable paths optimistically and to prune out compute-intensive back blocks, thereby making QSYM proceed.
- **Real-world bugs.** A QSYM-based hybrid fuzzer outperformed state-of-the-art automatic bug finding tools (e.g., Driller and VUzzer) in the DARPA CGC and LAVA test

sets. Further, QSYM discovered *13 new bugs* in eight real-world software. We believe these results clearly demonstrate the effectiveness of QSYM.

The rest of this work is organized as follows. §3.2 analyzes the performance bottleneck of current hybrid fuzzing. §3.3 and §3.4 depict the design and implementation of QSYM, respectively. §3.5 evaluates QSYM with benchmarks, test sets, and real-world test cases. §3.7 explains QSYM's limitations and possible solutions.

3.2 Motivation: Performance Bottlenecks

In this section, we systematically analyze the performance bottlenecks of the conventional concolic executor used for hybrid fuzzers. The following are the main reasons that block the adoption of hybrid fuzzers to the real world beyond a small-scale study.

3.2.1 P1. Slow Symbolic Emulation

The emulation layer in conventional concolic executors that handles symbolic memory model is extremely slow, resulting in a significant slowdown in overall concolic execution. This is surprising because the community believes that symbolic and concolic executions are slow due to path explosion and constraint solving. Table 3.1 shows this significant overhead in symbolic emulation when we execute several programs without branching out to the other paths (no path explosion) or solving constraints on the path in widely-used symbolic executors, KLEE and angr. Compared to the native execution, KLEE is around 3000 times slower and angr is more than 321 000 times slower, which are significant.

Why is symbolic emulation so slow?. In our analysis, we observed that the current design of concolic executors, particularly adopting IR in their symbolic emulation, makes the emulation slow. Existing concolic executors adopt IR to reduce their implementation complexity a lot; however, this sacrifices the performance. Additionally, optimizations that speed up this use of IR prohibit further optimization opportunities, particularly by translating the program into IRs in a basic-block granularity. This design does not allow skipping

Table 3.1: The emulation overhead of KLEE and angr compared to native execution, which are underlying symbolic executors of S2E and Driller, respectively. We used `chksum`, `md5sum`, and `sha1sum` in coreutils to test KLEE, and `md5sum (mosml)` [74] to test angr because angr does not support the `fadvise` syscall, which is used in the coreutils applications.

Executor	chksum	md5sum	sha1sum	md5sum(mosml)
Native	0.008	0.014	0.014	0.001
KLEE	26.243	32.212	73.675	0.285
angr	-	-	-	462.418

the emulation that does not involve in symbolic execution instruction by instruction. We describe the details of these in the following.

Why IR: IR makes emulator implementation easy. Existing symbolic emulators translate a machine instruction to one or more IR instructions before emulating the execution. This is mainly to make the implementation of symbolic modeling easy. To model symbolic memory, the emulator needs to interpret how an instruction affects the symbolic memory status when supplied with symbolic operands. Unfortunately, interpreting each machine instruction is a massive task. For instance, the most popular Intel 64-bit instruction set architecture (i.e., the `amd64` ISA) contains 1795 instructions [71] described in a 2000-page manual [72]. Moreover, the `amd64` ISA is not machine-interpretable, so human effort is required to interpret each instruction for its symbolic semantic.

To reduce this massive complexity in implementation, existing emulators have adopted the IR. For example, KLEE uses the LLVM IR and angr uses the VEX IR. These IRs have much smaller sets of instructions (e.g., 62 for the LLVM IR [73]) and are simpler than native instructions. Consequently, the use of IR significantly reduces the implementation complexity because the emulator will have a much smaller number of interpretation handlers than when it directly works with machine instructions (e.g., 1795 versus 62).

Why not: IR incurs additional overhead. Despite making implementation easy, the use of IR incurs overhead in symbolic emulation. First, the IR translation itself adds overhead. Because the `amd64` architecture is a complex instruction set computer (CISC), whereas the IRs model a reduced instruction set computer (RISC), in most cases, a translation of a

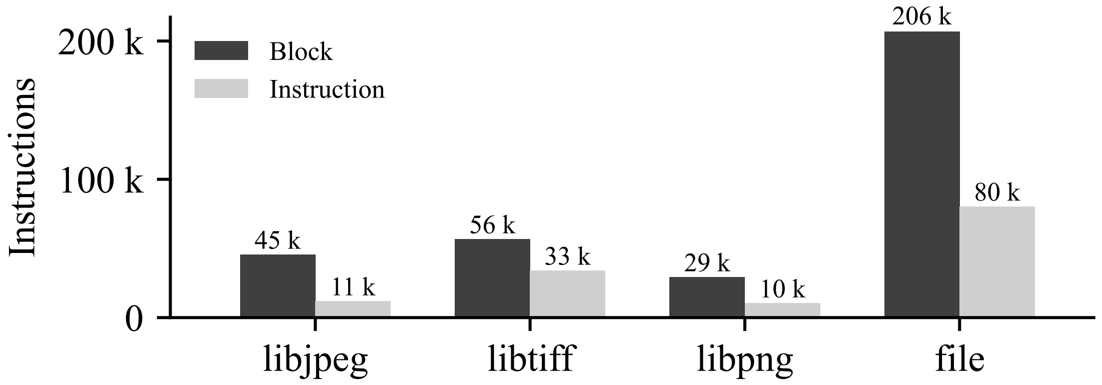


Figure 3.1: The number of instructions in symbolic basic blocks and the number of symbolic instructions in popular open-source software. More than half of the instructions in the basic blocks are not symbolic instructions, which can be executed natively.

machine instruction results in multiple IR instructions. For instance, based on our evaluation, the VEX IR [75], used by angr, increases the number of instructions by 4.69 times on average (versus machine instructions) in the CGC binaries, resulting in much symbolic emulation handling.

Why not: IR blocks further optimization. Second, using IR prohibits further optimization opportunities. For example, existing symbolic emulators have an optimization strategy that minimizes the use of emulation because it is slow. Particularly, they do not execute a basic block in the emulator if the block does not deal with any symbolic variables. Although this effectively cuts off the overhead, it still has room for optimization. According to our measurement with the real-world software (Figure 3.1), such as libjpeg, libpng, libtiff, and file, only 30% of instructions in symbolic basic blocks require symbolic execution. This implies that an *instruction-level approach* has an opportunity to reduce the number of unnecessary symbolic executions. However, current concolic executors cannot easily adopt this approach due to IR caching. To use IR, they need to convert native instructions into IR, which has significant overhead. To avoid repetitive overhead, they transform and cache basic blocks into IRs, instead of individual instructions, to save space and time for cache management. This caching forces existing symbolic emulators to execute instructions in a basic block level and prevent further optimization.

Our approach. Remove the IR translation layer and pay for the implementation complexity to reduce execution overhead and to further optimize towards the minimal use of symbolic emulation.

3.2.2 P2. Ineffective Snapshot

Why snapshot: eliminating re-execution overhead. Conventional concolic execution engines use snapshot techniques to reduce the overhead of re-executing a target program when exploring its multiple paths. The snapshot mechanism is also mandatory for hybrid fuzzing whose concolic re-execution is significantly slow, such as Driller. For example, we measured the code coverage by turning off the snapshot mechanism in Driller with 126 CGC binaries and given proof of vulnerabilities (PoVs) as initial seed files. As a result, Driller with snapshot achieved more code coverage in *76* binaries, but without snapshot achieved more code coverage in only *17* binaries, and others are the same.

Why not: fuzzing input does not share a common branch. Snapshots in hybrid fuzzing are not effective because concolic executions in hybrid fuzzing merely share a common branch. In particular, for conventional concolic engines, a snapshot is taken when the engine splits the path exploration from one conditional branch (i.e., the taken and untaken paths). The main purpose of taking a snapshot is to reuse a symbolic program state when exploring both paths at the same branch. In this regard, the engine backs up the symbolic state of the program in one branch, and then explores one of the paths (e.g., the taken path). When the path is exhausted or stuck, the engine restores the symbolic state to the previous state at the branch and moves to another path (i.e., the untaken path). The engine can explore the path without paying overhead for re-executing the program to the branch.

On the contrary, the concolic execution engine in hybrid fuzzing fetches multiple test cases from the fuzzer with which they are associated different paths of the program (i.e., sharing no common branch). This is because random mutation generates such test cases. This could 1) lead the program to a different code path or 2) concretize values differently on handling symbolic memory access [76]. Therefore, snapshots taken from one test case path cannot be re-used in the other test case path such that they do not optimize the performance.

Why not: snapshot cannot reflect external status. Worse yet, the snapshot mechanism becomes problematic in supporting external environments since it breaks process boundaries. Supporting external environments is required since the program heavily interacts with the external environment during its execution. Such interactions include the use of a file system and a memory management system, and these would be able to change the symbolic status of the program. When a program is being executed, it does not consider external environments since the underlying kernel maintains internal states related to them. Unfortunately, the snapshot mechanism breaks the assumption that the kernel holds: when a process diverges through `fork()`-like system calls, the kernel no longer maintains the states. Thus, concolic execution engines should maintain the states by itself.

Existing tools try to solve this problem through either *full system concolic execution* or *external environment modeling*, but they result in significant performance slowdown and inaccurate testing, respectively.

Full system concolic execution. Concolic testing tools such as S2E apply concolic execution for both the target program and the external environment. Although this approach ensures completeness and correctness, the tools cannot test the program in a reasonable time because conventional concolic executors are too slow and the complexity of the external environment is high. Moreover, a full system concolic execution requires expensive state backup and recovery. This overhead could be mitigated by copy-on-write under normal circumstances, but it is not applicable for hybrid fuzzing due to its non-shareable nature.

External environment modeling. Hybrid fuzzers, such as Driller, model or emulate the execution in the external environment. This approach has clear performance benefits by avoiding concolic execution, but it results in inaccurate models because it is almost impossible to completely and correctly model all system calls in practice. For example, Linux kernel 2.6 has 337 system calls, but angr only supports 22 system calls out of

```c
// @funcs.c:221 in file v5.6
if ((ms->flags & MAGIC_NO_CHECK_COMPRESS) == 0) {
  m = file_zmagic(ms, &b, inname); // zlib decompress
  ...
}

// other interesting code
```

```c
// @funcs.c:177 in file v5.6
// looks_ascii()
if (ch >= 0x20 && ch < 0x7f)
  ...
// file_tryelf()
if (ch == 0x7f)
  ...
```

Figure 3.2: The first example shows that collecting complete constraints for complicated routines such as `file_zmagic()` could prohibit finding new paths. The second example shows that if a given concrete input follows a true path of `looks_ascii()`, it over-constrains the path not to find a true path of `file_tryelf()`.

them. Further, despite excessive efforts of the developers, angr models many functions incompletely, such as `mmap()`. The current implementation of `mmap()` in angr ignores a valid file descriptor given to the function. It just returns empty memory instead of memory containing the file content.

Our approach. Optimize repetitive concolic testing, remove the snapshot mechanism that is inefficient in hybrid fuzzing, and use concrete execution to model external environments.

3.2.3 P3. Slow and Inflexible Sound Analysis

Why sound analysis?. Concolic execution tries to guarantee soundness by collecting *complete* constraints. This completeness assures that an input satisfying the constraints will lead the execution to the expected path. Thus, concolic execution can produce inputs to explore other paths of a program without worrying about false expectations.

Why not: never-ending analysis for complex logic. However, computing complete constraints could be expensive in various situations. In particular, computing the constraints for complex operations such as cryptographic functions or compression is often problematic.

The upper part of Figure 3.2 shows a code snippet of the file program. If concolic execution visits `file_zmagic()`, it sticks there to compute complex constraints for `zlib` decompression and cannot search other interesting code.

Why not: sound analysis could over-constraint a path. The complete constraints can also over-constrain [5] a path that limits concolic execution to find future paths. In particular, a constraint that is inserted to follow the native execution can cause the over-constraint problem. In the lower code of Figure 3.2, if `ch` is defined as 'A' by a given concrete input, concolic execution will put the constraint, `{ch >= 0x20 ∧ ch < 0x7f}`, at `looks_ascii()` because the native execution will execute the true branch of the `if` statement. When it arrives at `file_tryelf()`, the concolic execution cannot generate any test case because the final constraint is unsatisfiable, which is `{ch >= 0x20 ∧ ch < 0x7f ∧ ch == 0x7f}`. However, if `file_tryelf()` does not depend on the true branch of `looks_ascii()`, this is the over-constraint problem because an input generated by concolic execution without caring about the path constraint, `ch == 0x7f`, will explore a path in `file_tryelf()`.

Our approach. Collect an incomplete set of constraints for efficiency and solve only a portion of constraints if a path is overly-constrained.

3.3 Design

In this section, we explain our design decisions to realize QSYM. Figure 3.3 shows an overview of QSYM's architecture. QSYM aims at achieving fast concolic execution by reducing the efforts in symbolic emulation, which is the major performance bottleneck of existing concolic executors. To this end, QSYM first instruments and then runs a target program utilizing Dynamic Binary Translation (DBT) along with an input test case provided by a coverage-guided fuzzer. The DBT produces basic blocks for native execution and prunes them for symbolic execution, allowing us to quickly switch between two execution models. Then, QSYM selectively emulates only the instructions necessary to generate symbolic constraints, unlike existing approaches that emulate *all* instructions in the tainted

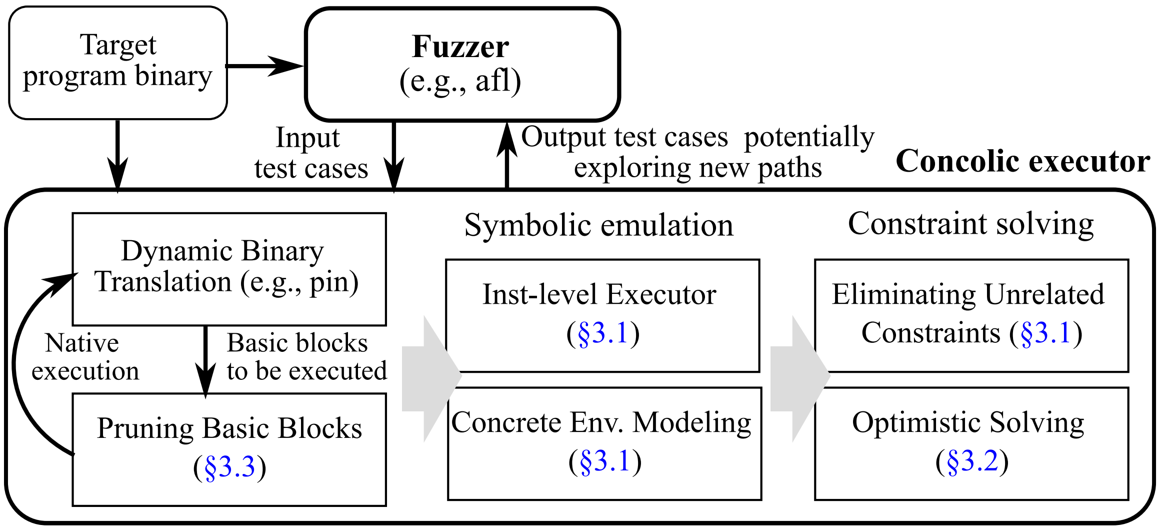
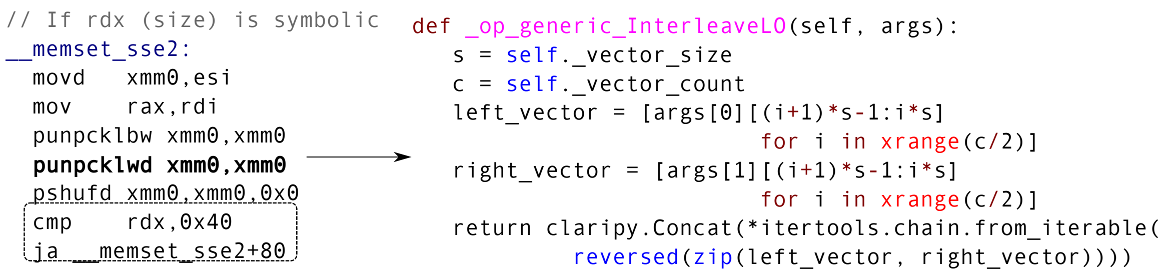

Target program binary
Fuzzer (e.g., afl)
Input test cases
Output test cases potentially exploring new paths
Concolic executor
Dynamic Binary Translation (e.g., pin)
Native execution
Basic blocks to be executed
Pruning Basic Blocks (§3.3)
Symbolic emulation
Inst-level Executor (§3.1)
Concrete Env. Modeling (§3.1)
Constraint solving
Eliminating Unrelated Constraints (§3.1)
Optimistic Solving (§3.2)

Figure 3.3: Overview of QSYM's architecture as a hybrid fuzzer. QSYM takes a test case and a target binary as inputs and attempts to generate new test cases that might explore new paths. It uses Dynamic Binary Translation (DBT) to natively execute the input binary as well as to select basic blocks for symbolic execution. Since QSYM applies various heuristics to trade strict soundness for better performance in constraint solving, the new test cases will be validated later by the fuzzer.

basic blocks. By doing this, QSYM reduced the number of symbolic emulations by a significant magnitude (5×, see Figure 3.9 in §3.5.3) and hence achieved a faster execution speed. Thanks to its efficient execution, QSYM can execute symbolic execution repeatedly instead of using snapshots that require external environment modeling. In particular, QSYM can interact with the external environment in a concrete fashion instead of relying on the contrived environment models. To improve the performance of constraint solving, QSYM applies various heuristics that trade off strict soundness for better performance. Such a relaxation provides an unprecedented opportunity to the concolic executor for a hybrid fuzzer, in which the paired-up fuzzer can quickly validate the newly produced test cases—it will simply discard them if they are not interesting. The rest of this section describes our approaches to scale the concolic executor for the hybrid fuzzer to test real-world programs.

3.3.1 Taming Concolic Executor

We explain in detail four new techniques to optimize the concolic executor for the hybrid fuzzer.



```
// If rdx (size) is symbolic
__memset_sse2:
  movd    xmm0,esi
  mov     rax,rdi
  punpcklbw xmm0,xmm0
  punpcklwd xmm0,xmm0
  pshufd xmm0,xmm0,0x0
  cmp     rdx,0x40
  ja __memset_sse2+80
```

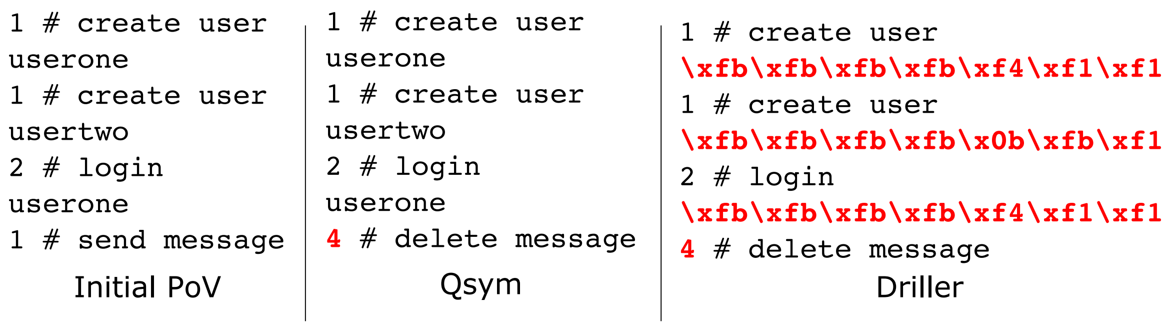
```
def _op_generic_InterleaveLO(self, args):
   s = self._vector_size
   c = self._vector_count
   left_vector = [args[0][(i+1)*s-1:i*s]
                         for i in xrange(c/2)]
   right_vector = [args[1][(i+1)*s-1:i*s]
                         for i in xrange(c/2)]
   return claripy.Concat(*itertools.chain.from_iterable(
           reversed(zip(left_vector, right_vector))))
```

Figure 3.4: An example that shows the effect of instruction-level symbolic execution. If a size is symbolic at `__memset_sse2()`, the instruction-level symbolic execution only executes symbolic instructions, which are in the dashed box. However, the basic-block-level one needs to execute other instructions that can be executed natively, including `punpcklwd`, which is complex to handle as shown in the right-side angr code.

Instruction-level symbolic execution. QSYM symbolically executes a small set of instructions that are required to generate symbolic constraints. Unlike existing concolic executors, which apply a block-level taint analysis and so symbolically execute *all instructions* in the tainted basic blocks, QSYM employs an instruction-level taint tracking and symbolic execution on the tainted instructions. The existing concolic executors take such a coarse-grained approach because they suffer from high performance overheads when switching between native and symbolic executions. However, for QSYM, the efficient DBT makes it possible to implement a fine-grained, instruction-level taint tracking and symbolic execution, helping us to avoid unnecessary emulation overheads.

This method significantly improves the performance of QSYM's symbolic execution in practice. Take `memset()` as an example (Figure 3.4), where only its size parameter (`rdx`) is tainted. Unlike a block-level approach, such as angr, that should symbolically execute all instructions, QSYM can generate symbolic constraints by executing only the last two instructions. This problem is more critical in real-world problems where modern compilers produce highly optimized code to minimize control-flow changes (e.g., using a conditional move like `cmov`). For example, in angr, any symbolic arguments to the `memset()` can prevent its symbolic execution because `memset()` relies on complex instructions like `punpcklbw`.

QSYM runs both native and symbolic executions in a single process by utilizing the DBT, making such mode switches extremely lightweight (i.e., a normal function call). It is



```
1 # create user
userone
1 # create user
usertwo
2 # login
userone
1 # send message
```

Initial PoV

```
1 # create user
userone
1 # create user
usertwo
2 # login
userone
4 # delete message
```

Qsym

```
1 # create user
\xfb\xfb\xfb\xfb\xf4\xf1\xf1
1 # create user
\xfb\xfb\xfb\xfb\x0b\xfb\xf1
2 # login
\xfb\xfb\xfb\xfb\xf4\xf1\xf1
4 # delete message
```

Driller

Figure 3.5: The test cases generated by QSYM and Driller that explore the same code path from the same seed. They are different because QSYM uses unrelated constraint elimination as their underlying optimization techniques whereas Driller uses incremental solving. Unrelated constraint elimination can remove unnecessary constraints, for example, constraints for the user names, on the existence of a concrete input.

worth noting that this approach is drastically different from most of the existing concolic engines, such as angr, where two execution modes should make non-trivial communications such as updating memory maps to make a mode switch. Accordingly, many optimizations made by angr are to reduce such mode switching, e.g., striving to run one mode as long as possible.

Solving only relevant constraints. QSYM solves constraints relevant to the target branch that it attempts to flip, and generates new test cases by applying the solved constraints to the original input. Unlike QSYM, other concolic executors such as S2E and Driller incrementally solve constraints; that is, they focus on solving the updated parts of constraints in the current run by utilizing lemmas learned from the previous execution. For pure symbolic executors that do not have any initial inputs for exploration, this incremental approach is effective in enumerating all possible input spaces [77]. However, this is not a favorable design for hybrid fuzzers for the following two reasons.

First, the incremental approach in hybrid fuzzers repeatedly solves the constraints that are explored by other test cases. For example, Figure 3.5 shows an initial test case and new test cases generated by QSYM and Driller when exploring the same code paths: the red marker shows the differences between the original input and the generated test cases. By solving only constraints relevant to the branch (i.e., selecting a menu for deleting a message), QSYM generates the new test case by updating a small part of the initial input. However, Driller generates new test cases that look drastically different from the original input. This indicates that Driller wastes time on solving irrelevant constraints that are repeatedly tested by fuzzers (e.g., constraints on usernames).

Second, the incremental approach is effective only when complete constraints are provided. Unfortunately, due to the emulation overheads, existing concolic executors cannot formulate symbolic constraints for complex, real-world programs. However, focusing only on relevant constraints gives us a higher chance to solve the constraints and produce new test cases that potentially take different code paths. For example, the test cases that are only relevant to the command menu will not be affected by the incomplete constraints generated for usernames (Figure 3.5). Moreover, due to its environment support (§3.3.1) or various heuristics (§3.3.2, §3.3.3), QSYM tends to generate more relaxed (i.e., incomplete) forms of constraints that can be easily solved. This makes QSYM scale enough to test real-world programs.

Preferring re-execution to snapshoting. QSYM's fast concolic execution makes re-execution much preferable to taking a snapshot for repetitive concolic testing. The snapshot approach, which creates an image of a target process and reuses it later, is chosen to overcome the performance bottleneck of the concolic execution; re-executing a program to reach a certain execution path with a valid state can take much longer than restoring the corresponding snapshot. However, as QSYM's concolic executor becomes faster, the overhead of the snapshotting is no longer smaller than that of re-execution.

Concrete external environment. QSYM avoids problems resulting from an incomplete or erroneous modeling of external environments by concretely interacting with external environments. Since the incompleteness and incorrectness of modeling deviate symbolic execution and native execution and mislead additional exploration, we should avoid them for further analysis. Instead of these erroneous models, QSYM considers external environments as "black-boxes" and simply executes them by concrete values. This is a common way to handle functions that cannot be emulated in symbolic execution [4, 11], but it is difficult to apply to forking-based symbolic execution, which breaks process boundaries [78]. Since QSYM can achieve performance without introducing forking-based symbolic execution [60], QSYM can utilize the old but complete technique to support external environments. However, this approach can result in unsound test cases that do not produce any new coverage, unlike its claim. If QSYM blindly believes concolic execution, QSYM will waste its resources to explore paths using test cases that do not introduce any new coverage. To alleviate this, QSYM relies on a fuzzer to quickly check and discard the test cases to stop further analysis.

3.3.2 Optimistic Solving

Concolic execution is susceptible to over-constraint problems in which a target branch is associated with complicated constraints generated in the current execution path (Figure 3.2). This problem is prevalent in real-world programs, but existing solvers give up too early (i.e., timeout) without trying to utilize the generated constraints, which took most of their execution time (Figure 3.9). In hybrid fuzzing, a symbolic solver's role is to assist a fuzzer to get over simple *obstacles* (e.g., narrow-ranged constraints like `{ch == 0x7f}` in Figure 3.2) and go deeper in the program's logic. Thus, as a hybrid fuzzer, it is well justified to formulate potentially new test inputs, regardless of reaching unexplored code via the current path or other paths.

QSYM strives to generate interesting new test cases from the generated constraints by optimistically selecting and solving some portion of the constraints, if not solvable as a whole. As the emulation overheads dominate the overheads of constraint solving in complex programs, it economically makes sense to leverage this opportunity. In particular, QSYM chooses the last constraint of a path for optimistic solving for the two following reasons. First, it typically has a very simple form, making it efficient for constraints solving. Another candidate would be the complement of `unsat_core`, which is the smallest set of constraints that introduces unsatisfiability. However, computing `unsat_core` is very expensive and sometimes crashes the underlying constraint solver [79]. Second, test cases generated from solving the last constraint likely explore the target path as they at least meet the local constraints when reaching the target branch. Since QSYM first eliminates constraints that are not related to the last constraint, all irrelevant constraints do not impact the result of the optimistic solving.

3.3.3 Basic Block Pruning

We observed that constraints repetitively generated by the same code are not useful for finding new code coverage in real-world software. In particular, the constraints generated by compute-intensive operations in a program are unlikely solvable (i.e., non-linear) at the end even if their constraints are formulated. Even worse, they tend to block the possibility of exploring other parts that are not relevant yet are interesting enough for further exploration. For example, in the second example of Figure 3.2, even though concolic execution produces constraints for the `zlib` decompression, a constraint solver will not be able to solve the constraints because of their complexity [80].

To mitigate this problem, QSYM attempts to detect repetitive basic blocks and then prunes them for symbolic execution and generates only a subset of constraints. More specifically, QSYM measures the frequency of each basic block execution at runtime and selects repetitive blocks to prune. If a basic block has been executed too frequently, QSYM stops generating further constraints from it. One exception is when a block contains *constant* instructions that do not introduce any new symbolic expressions, e.g., `mov` instructions in the `x86` architecture and shifting or masking instructions with a constant.

QSYM decides to use *exponential back-off* to prune basic blocks since it rapidly truncates overly frequent blocks. It only executes blocks whose frequency number is a power of two. However, if it excessively prunes basic blocks, it could miss some of the solvable paths and thus could fail to discover new paths. To this end, QSYM builds two heuristic approaches to prevent excessive pruning: *grouping multiple executions* and *context-sensitivity*.

Table 3.2: QSYM's main components and their lines of code.

Component	**Lines of code**
Concolic execution core	12,528 LoC of C++
Expression generation	1,913 LoC of C++
System call abstraction	1,577 LoC of C++
Hybrid fuzzing	565 LoC of Python

Grouping multiple executions is a knob that minimizes excessive pruning of basic blocks. When we count the frequency of a basic block's execution, we regard a group of executions as one in frequency counting. For instance, suppose the group size is *eight*. Then, only after executing the block *eight* times, we count the frequency as *one*. This will allow QSYM to execute the block *eight* times once it decided not to prune. This helps not to lose constraints that are essential to discover a new path and also does not affect much on the symbolic execution because running such basic blocks a small number of times would not make the constraints too complex.

Context-sensitivity acts as a tool for distinguishing running the same basic block in a different context for frequency counting. If we do not distinguish a context (i.e., at which point is this basic block called?), we may lose essential constraints by pruning more blocks. For example, when there are two `strcmp()` calls, say `strcmp(buf, "GOOD")` and `strcmp(buf, "EVIL")`, these two calls must be considered as a different basic block execution for frequency counting. Otherwise, the execution of the same block in the other part of the program, which is irrelevant to the current execution, could affect pruning. QSYM maintains a call stack of the current execution, and uses a hash of it to differentiate distinct contexts.

3.4 Implementation

We implement the concolic executor from scratch. QSYM consists of 16K lines of code (LoC) in total, and Table 3.2 summarizes the complexity of each of its components. QSYM relies on Intel Pin [81] for DBT, and its core components are implemented as Pin plugins written in C++: 12K LoC for the concolic execution core, 1.9K LoC for expression generation, and 1.5K LoC for handling system calls. QSYM also exposes Python APIs (0.5K LoC) such that users can easily extend the concolic executor; the hybrid fuzzer is built as a showcase using these APIs. QSYM uses libdft [82] in handling system calls while adding support for the 64-bit environments. The current implementation of QSYM supports part of Intel 64-bit instructions that are essential for vulnerability discovery such as arithmetic, bitwise, logical, and AVX instructions. QSYM will be open-sourced and support different types of instructions, including floating point instructions in the future.

3.5 Evaluation

To evaluate QSYM, this section attempts to answer the following questions:

- **Scaling to real-world programs.** How effective is QSYM's approach in discovering new bugs and achieving better code coverage when fuzzing complex, real-world software? (§3.5.1, §3.5.2)
- **Justifying design decisions.** How effective are the design decisions made by QSYM in terms of bug finding? (§3.5.3, §3.5.4, §3.5.5)
    1. **Instruction-level symbolic execution.** How effective is our fine-grained, instruction-level symbolic execution in terms of the number of instructions saved and the overall performance of the hybrid fuzzer? (§3.5.3)
    2. **Optimistic constraints solving.** How reasonable is QSYM's optimistic constraints solving in terms of finding bugs? (§3.5.4)
    3. **Pruning basic blocks.** How effective is our approach to prune basic blocks in terms of the overall performance and code coverage? (§3.5.5)

Experimental setup. We ran all the following experiments on Ubuntu 14.04 LTS equipped with Intel Xeon E7-4820 (having eight 2.0GHz cores) and 256 GB RAM. We used three cores respectively for master AFL, slave AFL, and QSYM for end-to-end evaluations (§3.5.1, §3.5.2, and §3.5.4) and one core for testing concolic execution only (§3.5.3 and §3.5.5).

Even though we used a server machine with many cores, we did not exploit all cores to run QSYM, but we aimed to run multiple experiments concurrently.

3.5.1 Scaling to Real-world Software

QSYM's approach scales to complex, real-world software. To highlight the effectiveness of our concolic execution engine, we applied QSYM to non-trivial programs that are not just large in size but also well-tested by the state-of-the-art fuzzer for a longer period of time. Thus, we considered all applications and libraries tested by OSS-Fuzz as ideal candidates for QSYM: libjpeg, libpng, libtiff, lepton, openjpeg, tcpdump, file, libarchive, audiofile, ffmpeg, and binutils. Among them, QSYM was able to detect *13 previously unknown bugs* in *eight* programs and libraries, including stack and heap overflows, and NULL dereferences (as shown in Table 3.3). It is worth noting that Google's OSS-Fuzz generated 10 trillion test inputs a day [85] for a few months to fuzz these applications, but QSYM ran them for three hours using a single workstation. In other words, all the bugs found by QSYM require the accurate formulation of inputs to trigger, showing the effectiveness of our concolic executor. §3.6 provides in-depth analysis of some of the bugs that QSYM found.

Compared to QSYM, other hybrid fuzzers are not scalable to support these real-world applications. We tested Driller, a known state-of-the-art hybrid fuzzer, for comparison. For testing purpose, we modified Driller to accept file input because these applications receive input from files, while the original Driller accepts only the standard input. We followed the direction of Driller's authors for this modification. As a result, Driller was able to generate only a few test cases due to its slow emulation. Driller generated less than 10 test cases on average for 30 minutes of running, whereas QSYM generated hundreds (more than 10×) of test cases in the same duration. Moreover, Driller was not able to support 5 out of 11 applications for lack of environment modelings and system call supports as shown in Table 3.4.

Table 3.3: Bugs found by QSYM and known fuzzers that are previously used to fuzz the binaries, and the reason they cannot be detected by the existing fuzzer and hybrid fuzzer. CVE-2017-11543* and CVE-2017-1000249* are concurrently found by QSYM before being patched [83, 84]. The failure of the fuzzer in the tcpdump bug marked by * is not crucial since a fuzzer also can find the bug, but in our experiment, QSYM found the bug 3 hours earlier than pure fuzzing.

Program	**CVE**	**Bug Type**	**Fuzzer**	**Fail (Fuzzer)**	**Fail (Hybrid)**
lepton	CVE-2017-8891	Out-of-bounds read	AFL	Meet complex constraints	Explore deep code paths
openjpeg	CVE-2017-12878	Heap overflow	OSS-Fuzz	Meet complex constraints	Support external environments
	Fixed by other patch	NULL dereference			
tcpdump	CVE-2017-11543*	Heap overflow	AFL	Find where to change*	Support external environments
file	CVE-2017-1000249*	Stack overflow	OSS-Fuzz	Meet complex constraints	Explore deep code paths
libarchive	Wait for patch	NULL dereference	OSS-Fuzz	Meet complex constraints	Support external environments
audiofile	CVE-2017-6836	Heap overflow	AFL	Multi-bytes magic values	Explore deep code paths
	Wait for patch	Heap overflow $\times$ 3			
	Wait for patch	Memory leak			
ffmpeg	CVE-2017-17081	Out-of-bounds read	OSS-Fuzz	Meet complex constraints	Support external environments
objdump	CVE-2017-17080	Out-of-bounds read	AFL	Meet complex constraints	Explore deep code paths

Table 3.4: Incomplete or incorrect system call handling by Driller that prohibits from applying Driller to real-world software. Driller's `mmap()` had an error: it ignored a file descriptor. We detected these errors dynamically using basic test cases in each project. Therefore, other incorrect or unsupported system calls could exist in unexplored paths.

Program	**Bug Type**	**Syscall**
libtiff	Erroneous system calls	`mmap`
openjpeg	Unsupported system calls	`set_robust_list`
tcpdump	Erroneous system calls	`mmap`
libarchive	Unsupported system calls	`fcntl`
ffmpeg	Unsupported system calls	`rt_sigaction`

3.5.2 Code Coverage Effectiveness

To show how effectively our concolic executor can assist a fuzzer in discovering new code paths, we measured the achieved code coverage during the fuzzing process by using QSYM (a hybrid fuzzer) and AFL (a fuzzer) with a varying number of input seed files. We selected libpng as a fuzzing target because it contained various narrow-ranged checks (e.g., checking the 4-byte magic value for chunk identification) that were non-trivial to satisfy without proper seeding inputs in the fuzzing-only approach. As seeding inputs, we collected high-quality (i.e., including various types of chunks) 141 PNG image files from the libpng project and incrementally (by 20%) applied to the fuzzers. For the 0% case, we provided a dummy ASCII file containing 256 'A's as a seeding input as both fuzzers required at least one input to begin with. For fair comparisons with the fuzzing-only approach, we prepared a hybrid fuzzer consisting of one master and one slave AFL instance with QSYM, and a fuzzer consisting of one master and two slave AFL instances so that both fuzzers utilized the same computing resources given the execution time. We ran both fuzzers for six hours and measured the explored code coverage.

The hybrid fuzzing approach was particularly effective in discovering new code paths when no or limited initial inputs were provided (Figure 3.6). In the 0% case (only with a dummy input), AFL did not make much progress as libpng checked the PNG header identifier in an early phase of execution. On the contrary, QSYM not only formulated and

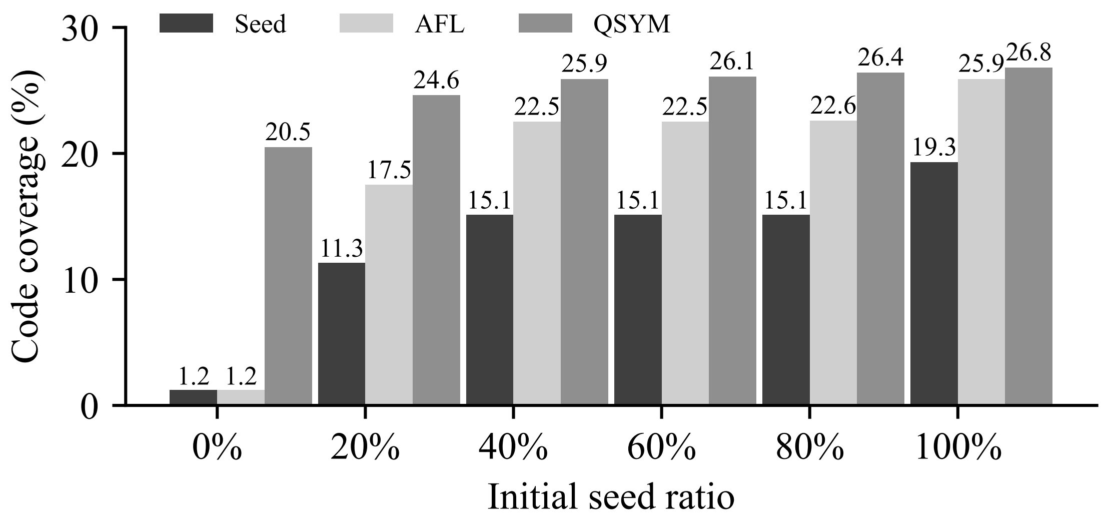


Figure 3.6: Code coverage of libpng after a six-hour run of QSYM and AFL (two AFL instances for a fair comparison) with an increasing number of seeding inputs. In the 0% case, we put an invalid PNG file consisting of 256 'A's as an initial input. The 100% case includes 141 sample PNG image files provided by the libpng project. This experiment result highlights the effectiveness of code coverage that the concolic execution approach contributes to hybrid fuzzing, depending on the availability of quality seeding inputs.

solved the constraints for checking the PNG's magic header identifier but also explored more than 20% of code paths of libpng, which was 3% higher than the code coverage of fuzzing with valid images, i.e., the 20% AFL case. Even when enough seeding inputs were provided, the concolic executor still allowed fuzzers to find more interesting paths. For example, the `hIST` chunk was not included in any of the 141 test cases, but QSYM was able to successfully generate new test cases by solving the symbolic constraints. It is worth noting that the `hIST` chunk needs to satisfy complex pre- and post-conditions to be a valid chunk in PNG: the `hIST` chunk should come after the `PLTE` chunk but before the `IDAT` chunk [86]. This example also hints at the difficulty of constructing complete test cases that cover all the features implemented in software, where we believe QSYM's approach can shed some light on.

3.5.3 Fast Symbolic Emulation

To show the performance benefits of QSYM's symbolic emulation, we used the DARPA CGC dataset [87] to compare QSYM with Driller, which placed third in the CGC competi-

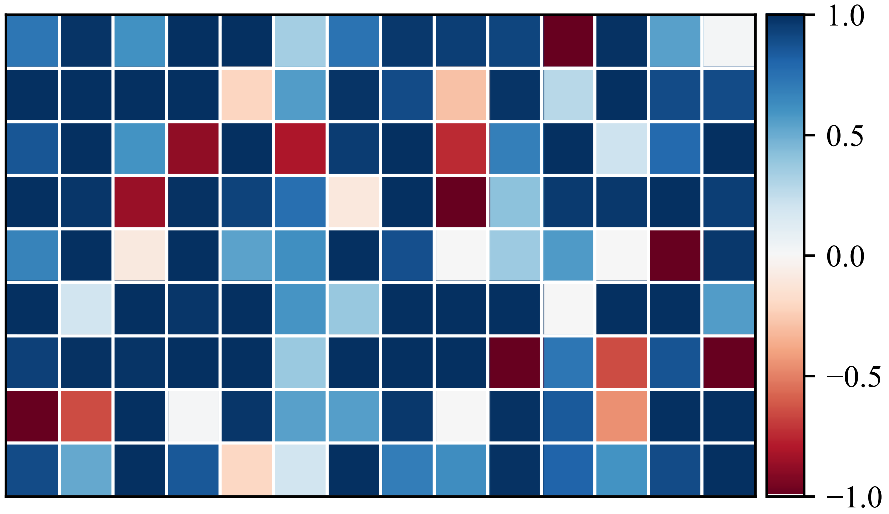


Figure 3.7: This color map depicts the relative code coverage for five minutes that compares QSYM's with Driller's: the blue color means that QSYM found more code than Driller, and the red color means the opposite (see §3.5.3 for the exact formula). Each cell represents each CGC challenge in alphabetical order (from left to right and top to bottom). QSYM outperforms Driller in discovering new code paths; QSYM results in better code coverage in 104 challenges (82.5% cases) and Driller does better in 18 challenges (14.3% cases) out of 126.

tion [8]. The CGC dataset included a wide range of programs from simple login services to sophisticated programs that attempt to mimic real-world protocols. CGC has released 131 challenge programs used in the CGC qualification event with PoVs—the inputs that trigger the vulnerabilities of the target program. Among the 131 challenge programs, we ignored five programs requiring Inter-Process Communication (IPC) that both QSYM and Driller did not support. We chose the PoVs as initial seed inputs because challenge writers intentionally hid bugs in the deep code path, so that PoVs tend to have good code coverage. To make our analysis simpler, we selected the first PoV (only one) as a seeding input for both fuzzers.

To show the fuzzing result, we used the code coverage that we measured from all the test cases generated while fuzzing each CGC challenge. Since the CGC programs did not support `libgcov`, a de-facto standard tool to measure code coverage, we used the AFL bitmap [88] instead to indicate their code coverage. The AFL bitmap consists of 65 536 entries to represent code coverage, which is reasonable enough for our comparison purpose.

Since the direct comparison of simple code coverage numbers might not properly indicate which fuzzer explored more and different code paths, we relatively compared their code coverage (see below). Additionally, we removed the bitmap entries that are already covered by initial PoVs for a fair comparison of newly explored paths. Based on this, we used the following formula to compare and visualize both coverage results relatively. For code coverage A (QSYM) and B (Driller), we can quantify the coverage differences by using:

$$d(A, B) = \begin{cases} \frac{|A-B|-|B-A|}{|(A\cup B)-(A\cap B)|} & \text{if } A \neq B \\ 0 & \text{otherwise} \end{cases}$$

It intuitively represents how many more unique paths that A explored out of the total discrete paths that only either A or B explored. For example, if QSYM found more unique paths than Driller, $d(A, B)$ will render a positive number, and it will be 1.0 when QSYM not only found more paths than Driller, but also covered all the paths that Driller found.

Figure 3.7 visualizes the results of the CGC code coverage for five minutes. Each cell represents each CGC challenge we tested in alphabetical order (from left to right and top to bottom). For example, the top-most left cell represents `CROMU_00001` and the bottom-most right cell represents `YAN01_00012`. The blue color represents the cases in which QSYM resulted in better code coverage, and the red color represents the ones that Driller did better. The darkest colors indicate that one fuzzer dominated the code coverage of another.

QSYM outperforms Driller in terms of code coverage; QSYM explored more code paths in 104 challenges (82.5%) out of 126 challenges, whereas Driller did better only in 18 challenges (14.3%). More importantly, QSYM fully dominated Driller in 37 challenges, where QSYM also covered all paths explored by Driller. It is worth noting that increasing the timeout for Driller (i.e., giving more time for constraints solving) does not help to improve the result of the code coverage. To show this, we ran Driller with varying timeouts from 5 to 30 minutes while fixing the timeout of QSYM to 5 minutes (Figure 3.8). Even with the

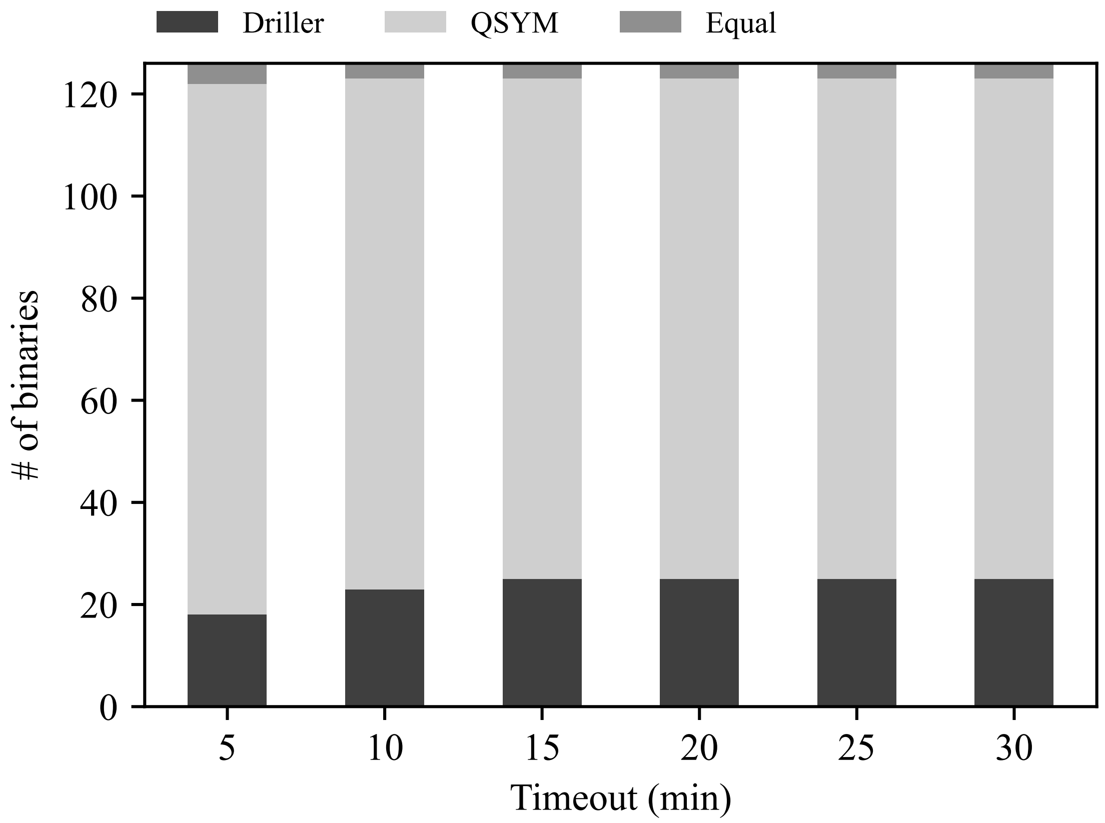


Figure 3.8: Comparing QSYM (5-min timeout) with Driller while increasing the time for constraints solving (from 5-min to 30-min). It shows that the reason Driller could not generate new test cases is not due to the limited time budget for solving the generated constraints.

30-min timeout of Driller, QSYM explored more paths in 98 out of 126 binaries, whereas Driller's coverage map was more or less saturated after the 10-min of the timeout.

Instruction-level symbolic execution. To understand how QSYM achieves a better performance than Driller, we break down the performance factors of QSYM and Driller. At a high level, Driller spent 27% of its execution time for creating snapshots and 70% for symbolic emulation (see, Figure 3.9(a)) In other words, Driller spent 2× more time than QSYM for concolic execution, but most of its time was spent for emulation and snapshot.

The instruction-level symbolic execution implemented in QSYM played a major role in speeding up the symbolic emulation. One way to demonstrate the effectiveness of this technique is to measure the number of instructions symbolically executed by both systems. However, QSYM and Driller took a different notion of symbolic instructions, making it hard

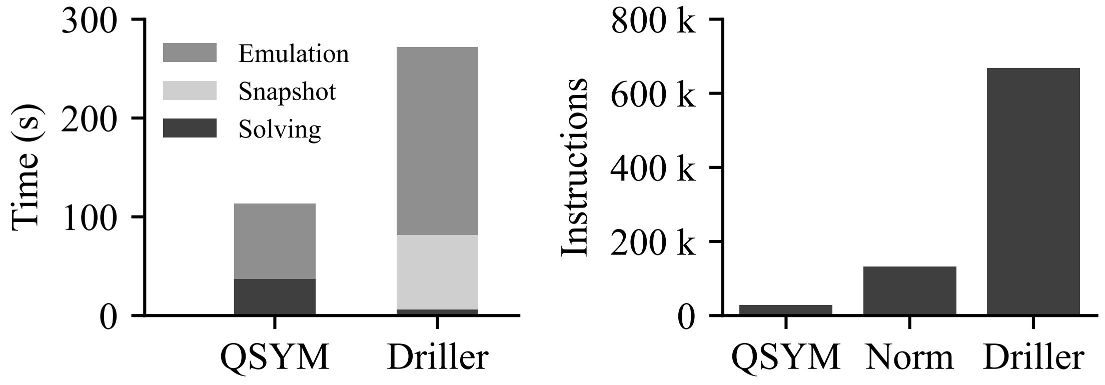


Figure 3.9: Average time breakdown of QSYM and Driller for 126 CGC binaries with initial PoVs as initial seed files, and the number of instructions that are executed symbolically. ‘Norm’ is the product of the number of instructions of QSYM and the average rate of increase of VEX IR, 4.69.

to compare both directly: QSYM uses the native x86 instructions, whereas Driller uses VEX IR for symbolic execution. Instead of counting and comparing the symbolically executed instructions, we took the amplification factor (i.e., 4.69) into consideration, the conversion rate from x86 to VEX IR when lifting all CGC binaries to use VEX IR. Even with this amplification factor (assuming an instruction in `amd64` is equivalent to 4.69 instructions), QSYM executed only $1/5$ of instructions symbolically when compared with Driller. Moreover, QSYM’s fast emulator helps us eliminate the ineffective snapshot mechanism. All these improvements applied together make constraints solving another important factor for the overall performance of the concolic execution.

Further case analysis. We could find several tendencies from further investigation of the results:

1) QSYM explores more paths than Driller in large programs and with long PoVs (i.e., in exploring deeper path). For example, QSYM covers more code coverage than Driller in `NRFIN_00039`, whose binary size is the largest among the challenges, about 12 MB. Moreover, QSYM can find test cases that cover code deep in the binaries. For example, `CROMU_00001` is a service that can send messages between users. To read a message, an attacker should go through the following process: *(1)* create a new user (user1), *(2)* create another user (user2), *(3)* log in as user1, *(4)* send a message to user2, *(5)* logout, *(6)* log in as

Table 3.5: The number of instructions in the CGC challenges that are not emulated due to the limitation of QSYM: no floating point operation supports.

Challenge	Not emulated	Total
`NRFIN_00026`	4 (0.02 %)	24 315
`NRFIN_00032`	4 (0.00 %)	4 784 433
`CROMU_00016`	18 (0.06 %)	31 988
`KPRCA_00045`	25 (0.00 %)	81 920 092
`KPRCA_00009`	27 (0.23 %)	11 512
`NRFIN_00027`	178 (0.73 %)	24 449
`CROMU_00028`	1154 (0.01 %)	18 626 977
`CROMU_00010`	1467 (0.18 %)	811 819
`CROMU_00020`	3492 (11.15 %)	31 306
`KPRCA_00013`	4589 (0.02 %)	18 746 620
`CROMU_00002`	14 977 (3.92 %)	381 793
`NRFIN_00021`	18 821 (33.26 %)	56 583
`KPRCA_00029`	31 800 (0.16 %)	19 604 258

user2, and *(7)* read a message by sending a message id to read. QSYM reaches the 7th step that reads a message and generates test cases in the function, but Driller fails to reach the function. This shows that QSYM's efficient symbolic emulation is effective in discovering sophisticated bugs hidden deeper in the program's path.

2) With a limited time budget (5 to 30 minutes), Driller gets more coverage in applications with multiple nested branches within quickly reachable paths (i.e., shallow paths) because its snapshot mechanism is optimized for this case. Due to its slow emulation, Driller can search only the branches close to the start of a program in a limited time (5 to 30 minutes). When Driller reaches a nested branch (i.e., a chunked multiple `cmp` instructions), Driller can fully leverage its snapshot to quickly explore these branches without involving re-execution. In contrast, QSYM should re-execute the emulation with a newly generated input to reach to the next branch. However, QSYM can gradually find the path via re-execution, and this exploration will be efficient since the branches are also easily reachable by QSYM.

Incomplete emulation. Currently, QSYM does not completely emulate all instructions (e.g., it cannot emulate floating point operations with symbolic operands), so that one can think that its performance improvement is due to non-emulated instructions. To refute

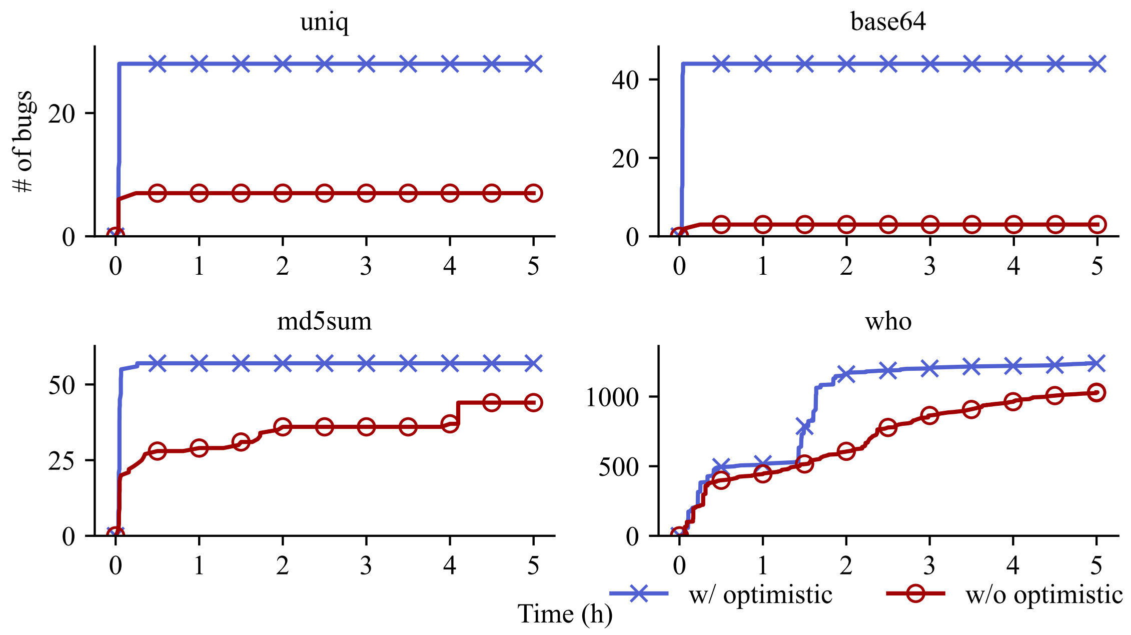


Figure 3.10: The cumulative number of bugs found in the LAVA dataset with or without optimistic solving by time.

Table 3.6: The number of bugs found by existing techniques and QSYM in the LAVA-M dataset. VUzzer (R) represents the number of bugs that are found by VUzzer in our machine settings, and VUzzer (P) represents the number of bugs in the VUzzer paper.

	uniq	**base64**	**md5sum**	**who**
FUZZER	7 (25 %)	7 (16 %)	2 (4 %)	0 (0 %)
SES	0 (0 %)	9 (21 %)	0 (0 %)	18 (1 %)
VUzzer (R)	27 (96 %)	1 (2 %)	0 (0 %)	23 (1 %)
VUzzer (P)	27 (96 %)	17 (39 %)	0 (0 %)	50 (2 %)
QSYM	28 (100 %)	44 (100 %)	57 (100 %)	1238 (58 %)
Total	28	44	57	2136

this hypothesis, we measured the number of instructions that were not emulated by QSYM (Table 3.5). Note that only 13 binaries out of 126 binaries have at least one instruction that is not handled by QSYM. Moreover, only three of them have not-emulated instructions that are more than 1% of their total instructions. Thus, we conclude that the performance improvement was not due to the incompleteness of QSYM's instruction modeling but to our instruction-level symbolic execution.

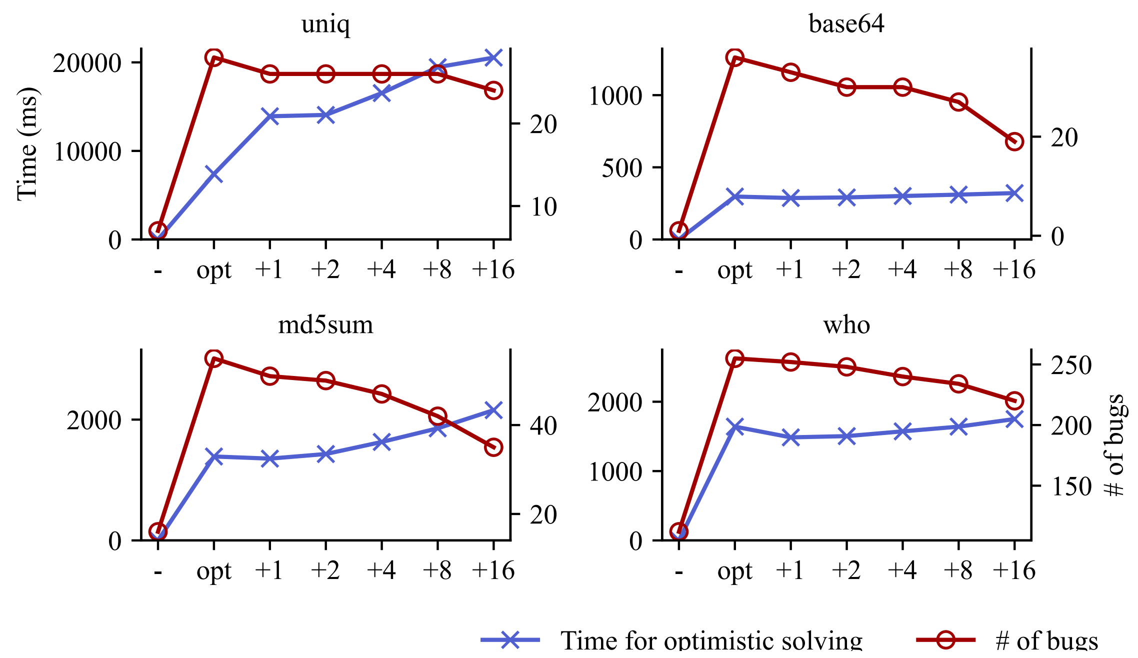


Figure 3.11: Time elapsed for optimistic solving and the number of unique bugs found in the LAVA dataset in a single execution of QSYM with an initial test case according to the number of constraints in optimistic solving. The minus symbol (–) represents the absence of optimistic solving; therefore, its elapsed time is zero in every case. Opt is our optimistic solving that only uses the last constraint in an execution path, and the number after the plus symbol (+) represents the number of additional constraints used for optimistic solving. For example, +1 represents that QSYM uses one additional constraint; therefore, it uses two constraints for optimistic solving, the last one and the additional one. The graph shows that our decision uses the last constraint helps QSYM find the most bugs while spending less time.

3.5.4 Optimistic Solving

To evaluate the effect of optimistic solving, we compared QSYM with others using the LAVA dataset [69]. LAVA is a test suite that injects *hard-to-find* bugs in Linux utilities to evaluate bug-finding techniques, so the test is adequate for demonstrating the fitness of the technique. LAVA consists of two datasets, LAVA-1 and LAVA-M, and we decided to use LAVA-M consisting of four buggy programs, `file`, `base64`, `md5sum` and `who`, which have been used for testing other systems such as VUzzer. We ran QSYM with and without the optimistic solving on the LAVA-M dataset for five hours, which is the test duration set by the original LAVA work [69]. To identify unique bugs, we used built-in bug identifiers provided by the LAVA project.

The optimistic solving helps QSYM find more bugs by relaxing over-constrained variables. Figure 3.10 shows the cumulative number of unique bugs found by QSYM with or without optimistic solving. In all test cases, running QSYM with optimistic solving supersedes the run without it by finding more bugs even at an early stage (within three minutes). This result supports our design hypothesis that relaxing overly constrained variables would benefit path exploration, and fuzzing will assist this well to pruning out false-positive cases due to missing constraints. Take an example in `base64`; the program decodes an input string using a table lookup (i.e., `table[input[0]]`) and further comparisons will be restricted by that concrete value. In such a case, concolic execution concretizes the entire symbolic constraints to the current input because the table lookup over-constrains input symbols to have only one solution that is identical to an initial test case. Therefore, without optimistic solving, although QSYM arrived at branches that must pass to trigger crashes, constraint solver will return unsatisfiability. However, with the optimistic solving, even if the constraint is unsatisfiable, the solver will solve only the last constraint and generate a potential crash input, which helps fuzzer move forward if this optimistic speculation is correct.

We also compared QSYM with other state-of-the-art systems; QSYM outperformed them (Table 3.6). At first, we tested VUzzer [39] in our environment. However, our results were either equal (in `md5sum` and `uniq`) or worse (in `base64` and `who`) than the original paper's results because our workstation has slow cores (2.0GHz). Instead, we decided to borrow the original results. We also borrowed the other results from the evaluation of LAVA [39] due to its anonymized testing systems. In Table 3.6, FUZZER represents the results of a coverage-oriented fuzzer and SES represents the results of the symbolic execution. QSYM found 14$\times$ more bugs than VUzzer and any other prior techniques in the LAVA-M dataset.

To evaluate our decision for optimistic solving that uses only the last constraint among constraints in an execution path, we measured the elapsed time and the number of bugs found in the LAVA-M dataset while changing the number of additional constraints. When we include additional constraints, we chose constraints in the order in which they were

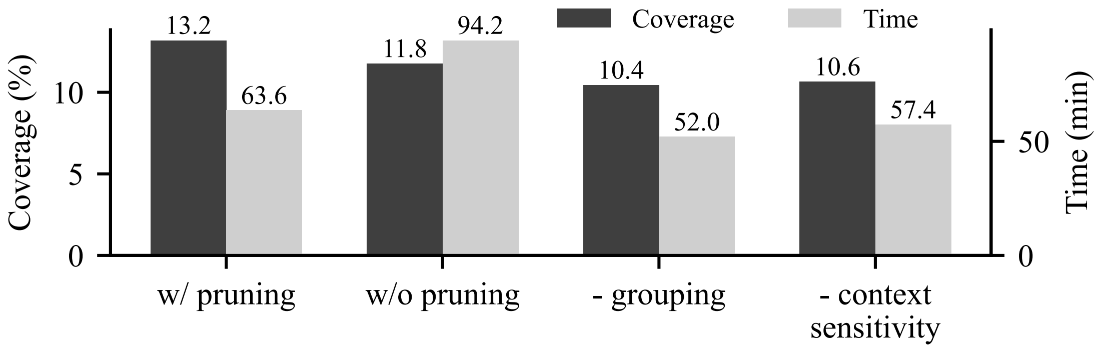


Figure 3.12: Total newly found coverage and elapsed time for libjpeg, libpng, libtiff, and file with five seed files, except for libjpeg, which has only four files, that have the largest code coverage in each project.

recently added. We used a single execution with the initial test case given by the dataset author instead of end-to-end evaluation to limit the impact by fuzzing. The results are shown in Figure 3.11. QSYM with optimistic solving always found more bugs than QSYM without optimistic solving. However, considering additional constraints did not help find more bugs and just increased solving time in most cases. In certain cases, adding more constraints can reduce the time required for optimistic solving. This is not surprising since adding more constraints might help to decide unsatisfiability.

3.5.5 Pruning Basic Blocks

To show the effect of the basic block pruning, we evaluated this technique with four widely-used open-source programs, namely, libjpeg, libpng, libtiff, and file. We chose five seed test cases that exhibit the largest code coverage (libjpeg has only four test cases so used just four) from each project. We ran QSYM with 5-min timeout for running concolic execution per each test case (19 cases in total, 5-min timeout for each test case, and up to 95 minutes) and then measured execution time and newly found code coverage.

Figure 3.12 shows that basic block pruning not only reduced execution time (63.6 min versus 94.2 min) but also helped to find more code coverage (13.2% versus 11.8%) in the real-world software. Take an example of libtiff; the function `TIFFReadDirectoryFindFieldInfo()` keeps introducing new constraints because it contains a loop with a symbolic branch. Basic block pruning made QSYM concretely execute the function and focus on other interesting code, whereas running without it made the emulation stuck there for generating constraints.

The other design decisions, context-sensitivity and grouping, are essential to increase code coverage. Figure 3.12 also shows code coverage and time when we disabled each grouping and context-sensitivity. If we disable grouping and use the AFL's algorithm as is, the pruning is too fine-grained, so it harms code coverage. A similar result was observed when we disabled context-sensitivity. In this case, QSYM prunes basic blocks too aggressively, prohibiting the generation of solvable constraints. Thus, these two design decisions are necessary to minimize the loss of code coverage.

3.6 Analysis of New Bugs Found

Out of 13 new bugs QSYM found, we took two interesting cases from ffmpeg and file in which we can clearly convey our idea. For each case, we attempt to answer how QSYM was able to find them, which features of QSYM helped find them, and most importantly, why OSS-Fuzz missed them.

3.6.1 ffmpeg

Figure 3.13 shows the simplified code of the ffmpeg bug that QSYM found, and the test case generated by QSYM to trigger it. To trigger the bug, a test case should meet very complicated constraints (Lines 3–10), which is nearly impossible for fuzzing. In contrast, QSYM successfully generated a new test case that can pass the complicated branch by modifying the seven bytes of a given input. AFL was able to pass the branch with the new test case and eventually reached the bug.

3.6.2 file

Figure 3.14 shows the simplified code of the file bug that QSYM found. The bug is that the check of `descsz` becomes a tautology because of the incorrect use of the logical OR operator

```
 1 // @libavcodec/x86/mpegvideodsp.c:58 (ffmpeg 3.4)
 2 if ( ((ox ^ (ox + dxw))
 3      | (ox ^ (ox + dxh))
 4      | (ox ^ (ox + dxw + dxh))
 5      | (oy ^ (oy + dyw))
 6      | (oy ^ (oy + dyh))
 7      | (oy ^ (oy + dyw + dyh))) >> (16 + shift)
 8     || (dxx | dxy | dyx | dyy) & 15
 9     || (need_emu && (h > MAX_H || stride > MAX_STRIDE)))
10 { ... return; }
11 // the bug is here
```

```
// input
< 00000010: 0120 0040 7800 000e 0001 0000 0820 8403
< 00000020: 0747 013f 303f 3f3f 7f7f 7fff 0080 8080
---
// output
> 00000010: 0120 0040 7800 000e 0008 0020 0020 47c3
> 00000020: 4040 013f 303f 3f3f 7f7f 7fff 0080 8080
```

Figure 3.13: The ffmpeg code about the bug found by QSYM and the test case generated by QSYM to reach it. AFL alone was unable to reach the bug because it is almost infeasible to randomly generate input to pass the complicated condition in Lines 3–10.

```
// @src/readelf.c:513 (file 5.31)
if (namesz == 4
    && strcmp((char *)&nbuf[noff], "GNU") == 0
    && type == NT_GNU_BUILD_ID
    && (descsz >= 4 || descsz <= 20)) {...}
```

Figure 3.14: The file bug that QSYM found. The check for `descsz` is always true due to the incorrect use of logical OR operator.

while parsing the ELF's note section. Interestingly, even though the bug is triggered when parsing an ELF file, initial seed files that we extracted from the `tests` directory in the file project do not contain any ELF files. In other words, QSYM successfully crafted a valid ELF file with a note section and triggered the vulnerability. This bug is difficult to be detected by a fuzzer because randomly crafting a valid ELF file with a note section starting with "GNU" is almost infeasible. Note that a concurrent bug report [84] detected this bug using a static analysis tool `cppcheck` [89].

3.7 Discussion

We discuss the potentials of QSYM's technique beyond hybrid fuzzing, using QSYM with other fuzzers, and the limitations of QSYM.

Adoption beyond fuzzing. Basic block pruning (§3.3.3) can directly be applied to the other concolic executors as a heuristic path exploration strategy. Take an example of testing file parsers; this technique allows QSYM to focus on control data (i.e., headers), which leads to new code coverage [90], rather than payloads, which will consume a lot more time to analyze but do not discover any new code coverage. We envision that the same strategy may help other concolic executors on testing programs with complex data processing logic such as data compression, Fourier transform, and cryptographic logic. By adopting this, concolic executors can automatically truncate such complex yet irrelevant logic and stay focused on the input fields that determine a program's control flow.

Optimistic solving (in §3.3.2) could also be applied to other domains to speed up symbolic execution, with a condition if the domain runs an efficient validator like a fuzzer. This cannot be directly applied to general concolic executors because optimistic solving relaxes an overly-constrained path to generate some potentially correct inputs. It will generate a haystack of false positives that deviate the program state from the expected state. However, in hybrid fuzzing like QSYM, because the fuzzer can efficiently validate whether the input drives the program to an expected state (i.e., finding a new code coverage) or not, we can quickly extract some useful results from the haystack. Likewise, other domains, for instance, automatic exploit generation, can adapt this technique to speed up for quickly reaching to the vulnerable state and crafting an exploit. After that, it could also efficiently validate a crafted exploit by just executing it and observe the core dump to check if it is a false positive.

Complementing each other with other fuzzers. Hybriding QSYM with other fuzzers better than AFL will show better results. While other fuzzers exist that enhance AFL, such as VUzzer [39] and AFLFast [34], in this work, we applied QSYM to AFL in order to fairly present the enhancement only by the concolic execution. QSYM can complement the others by quickly reaching the branch with narrow-ranged, complex constraints and solving them to generate test cases for that point. Moreover, QSYM can also be complemented by other fuzzers. Frequency-based analysis step and Markov chain modeling in AFLFast, as well as error-handler detection in VUzzer, could generate more meaningful input, which would result in using QSYM's concolic executor more efficiently.

Limitations. Although fast, QSYM is a concolic executor, so its performance is still bound to theoretical limits like constraint solving. Currently, QSYM is specialized to test programs that run on the `x86_64` architecture. Unlike other executors that adopted IR, QSYM cannot test programs that run on other architectures. We plan to overcome this limitation by improving QSYM to work with architecture specifications [71, 91] rather than a specific architecture implementation. Additionally, QSYM currently supports only memory, arithmetic, bitwise, and vector instructions, all of which are essential for vulnerability discovery. We plan to support other instructions including floating-point operations to extend QSYM's testing capability.

CHAPTER 4
HYBRIDRA: HYBRID FUZZING FOR KERNEL FILE SYSTEMS

4.1 Introduction

In this work, we propose HYBRIDRA, a hybrid fuzzer for Linux file systems. HYBRIDRA improves the state-of-the art file fuzzer, Hydra, by introducing concolic execution. To tackle the challenges in concolic execution for file systems, we have reformulated Hydra's approach for concolic execution; HYBRIDRA focuses on the metadata of file system images like Hydra and fixes the checksum using its file-system-specific modules. Moreover, to reduce emulation overhead from large software like Linux Kernel, HYBRIDRA adopts a technique called *compilation-based concolic execution* [10, 11, 12], instruments a binary for concolic execution to eliminate the overhead from symbolic emulation in concolic execution. Unlike SymCC [10], which recently proposed the same technique, HYBRIDRA chooses another design from Kirenenko [14], which uses different memory modeling using shadow memory. This design is more preferable for HYBRIDRA because of its support of multi-threading as well as better performance (i.e., 1.57× in our evaluation).

Additionally, HYBRIDRA suggests new heuristics called *staged reduction* to efficiently generate test cases for hybrid fuzzing. Instead of relying on a single strategy in test case generation, staged reduction combines multiple strategies in the process of hybrid fuzzing. For example, the current HYBRIDRA uses three reduction mechanisms: linear reduction [4], basic block reduction, and no reduction. By gradually applying cheaper strategies to more expensive ones, HYBRIDRA can generate diverse test cases and thereby explore input space more efficiently, while preserving the completeness of concolic execution.

Our evaluation shows that HYBRIDRA outperforms the state-of-the-art file system fuzzer Hydra by achieving higher code coverage in all file systems — `btrfs`, `ext4`, `ftfs`, and `xfs`—

that we tested. More importantly, HYBRIDRA discovered 10 bugs, and four of which are new ones in `btrfs`, which have been highly tested by other fuzzers, Hydra and Syzkaller. Notably, four out of the 10 bugs were directly discovered by our newly introduced concolic execution module, which shows its effectiveness in bug finding for file systems.

This work makes the following contributions:

- **System design of hybrid fuzzing for file systems** We design and implement a full-fledged hybrid fuzzer for file systems, HYBRIDRA, by applying concolic execution to the state-of-the-art file system fuzzer, Hydra. For efficient concolic execution, it instruments Hydra's executor based on library OS(LibOS) using Kirenenko, which is a compilation-based concolic executor supporting multi-threaded applications. Moreover, we reformulate Hydra's module to overcome the challenges in concolic execution; this makes HYBRIDRA focus on metadata of a file system image to complement the limited scalability of concolic execution, and it uses pre-defined rules to correct the checksum, which is difficult for concolic execution due to its complex constraints [45].
- **New heuristics for hybrid fuzzing** We propose new heuristics to combine existing heuristics for test case generation in concolic execution, called *staged reduction*. Staged reduction gradually applies multiple heuristics for test case generation from cheaper to more expensive ones. Consequently, it quickly explores the input space of file systems and also eventually reaches deep states that are only accessible using the expensive analysis.
- **Practical impacts** HYBRIDRA found 10 bugs including four new ones in `btrfs`, which has been thoroughly tested by fuzzing. Our evaluation shows that concolic execution directly discovers many of these bugs, which shows its usefulness in bug finding.

The rest of this work is organized as follows. §4.2 shows motivation and challenges to design a hybrid fuzzer for file systems. §4.3 and §4.4 depict the design and implementation

Table 4.1: The size of a raw file image and its metadata in each file system. Even though Hydra's approach significantly reduces the size of data to fuzz (from a raw image to metadata), it is still huge for effective fuzzing.

File system	btrfs	ext4	f2fs	xfs
Image size (MB)	100	2	38	16
Metadata size (KB)	40	197	48	50

of HYBRIDRA, respectively. §4.5 evaluates HYBRIDRA's design decisions and compare it with the-state-of-the-art file system fuzzer, Hydra. §4.6 explains HYBRIDRA's limitations and potential applications.

4.2 Motivation and Challenges

A file system is one of the most essential features of operating systems. It provides abstraction of hardware resources for files (e.g., disks) for efficiency and security. Users can utilize it by mounting a disk image and can interact using system calls for file systems (e.g., `open()`, `read()`, and `write()`). Most commodity OSes implement the file system as part of the kernel; therefore, a single vulnerability in the file system can subvert the entire system because of its high privilege [92, 93, 94].

Many file system fuzzers have been proposed to discover vulnerabilities in file systems. They randomly generate either file images or system calls until a system crashes. Recently, Janus [68] combined them to explore this two-dimensional space of a file system: images and system calls. Hydra [15] improves on Janus's method to discover other types of bugs beyond memory corruptions; it provides an extensible framework for supporting multiple types of checkers for various bug types.

Fuzzing for file systems is fundamentally weaker than one for other applications due to its large input size. Even though Hydra focuses only on the metadata of file system images, the metadata are still too large (i.e., a few KB); therefore, Hydra has disabled deterministic fuzzing, which is effective in discovering immediate test cases from the current one. Concolic execution can help fuzzing by systematically exploring paths from the current test case and generating immediate test cases; however, it is challenging to apply concolic execution in file systems due to its large image size, use of checksums, and gigantic code size.

4.2.1 P1. Large and checksum-protected file system images

Concolic execution is challenging to test file systems due to its special form of input, a file system image. Unlike the inputs of other common software, a file system image has several restrictions for its minimum size and a sector size alignment, which inflates its size. Consequently, a raw file image size often exceeds the size of the fuzzing input suggested by AFL (1MB), as shown in Table 4.1. Moreover, a file system introduces checksums to protect its data from unexpected corruptions. This is troublesome for concolic execution due to its complex constraints. More seriously, an existing automated method to fix the checksum [45] cannot be applied to a file system because it assumes the unique checksum, but a checksum routine in a file system is often hierarchical to support extremely large disk data.

Our approach. Reformulate Hydra's approach to tame a file system image for fuzzing to concolic execution. In particular, HYBRIDRA selectively symbolizes the metadata of a file system image, which is only a few KBs (see Table 4.1). Moreover, we have adopted Hydra's checksum correction module, which is implemented by a pre-defined rule for each file system.

4.2.2 P2. Gigantic code size

Concolic execution on a file system is also non-trivial because of its large code size. The file system is implemented as part of the most gigantic and complex software in the world: operating systems. For example, Linux consists of more than 27 million lines of code [95], and each file system is much larger than usual applications, as shown in Table 4.2. We have compared file systems with four popular applications — libjpeg, libpng, libtiff, and file — and conclude that file systems have 4.5 $\times$ more code on average. It is worth noting that this

Table 4.2: The lines of code for four popular user-space applications — libjpeg, libpng, libtiff, and file — and file systems in Linux Kernel. It shows that a file system is much larger than a typical application (on average 4.5 $\times$) even without considering other features in Kernel.

Type	Name	Version	Lines
Userspace	libjpeg	v9d	28,420
	libpng	v1.6.37	25,876
	libtiff	v4.1.0	34,303
	file	v5.39	16,288
File systems	ext4	v5.30	90,619
	btrfs		150,181
	xfs		146,212
	f2fs		79,922

result excludes core functionalities in Linux Kernel, which needs to be analyzed for correct concolic execution. In particular, if we fail to analyze a library function from Linux Kernel that is used in a file system, it engenders incomplete constraints, making it lose interesting test cases for the file system.

To analyze file systems, full system emulation has been used, which is fundamentally inefficient. S2E and its successors [5, 62, 54] aim at generic concolic execution tools for kernel and fully emulate it using QEMU [53]. This is beneficial to test various features in an operating system; however, it is overkill for testing file systems, which can be emulated by library OSes(LibOS) such as LKL [96]. Since LKL is an application in userland, existing binary concolic executors [97, 63, 98, 64] can be applied. Unfortunately, these solutions fail to take advantages of having source code like Linux Kernel, which can make concolic execution more efficient.

Our approach. Instrument a LibOS-based executor for concolic execution, which is known as compilation-based concolic execution [10, 12, 11], using Kirenenko [14]. This allows us to perform concolic execution on a file system more efficiently by utilizing the underlying host kernel and existence of source code. We choose Kirenenko thanks to its several advantages for file systems: better performance and multi-threading support.

4.2.3 P3. Complex constraints

Theoretically, concolic execution is a desirable solution for test case generation; however, it is limited due to complex constraints from real-world software. Concolic execution first extracts symbolic constraints from its execution and make test cases using a technique called constraint solving. Constraint solving is inherently inefficient; it is NP-complete, according to computational complexity theory. Thus, constraint solving often fails to generate interesting test cases even with complete constraints.

To address this problem, many constraint reduction mechanisms have been proposed. For example, SAGE [4] reduces constraints into linear expressions, which have an efficient algorithm to solve [99]. Moreover, QSYM discards constraints from repeatedly executed code based on the practical observation that such constraints are less interesting in bug finding and often stall further exploration of concolic execution. Unfortunately, there is no single winner among these mechanisms; they have their own pros and cons. For example, SAGE's linear reduction can generate test cases quickly; however it limits the power of concolic execution to support only certain types of expressions and thereby loses other interesting cases. On the contrary, QSYM's basic block reduction is more expressive, but it is more expensive than SAGE's without any algorithmic improvement.

Our approach. We suggest a new mechanism called *staged reduction*, inspired by ensemble systems [100, 101]. Staged reduction takes multiple reduction mechanisms ordered by their speed and gradually applies them. Consequently, it can efficiently explore the input space for file systems while preserving the power of concolic execution.

4.3 Design

To address the limitations of classical fuzzing, we have applied concolic execution to Hydra, designing a new hybrid fuzzer, HYBRIDRA. Figure 4.1 shows an overview of HYBRIDRA.

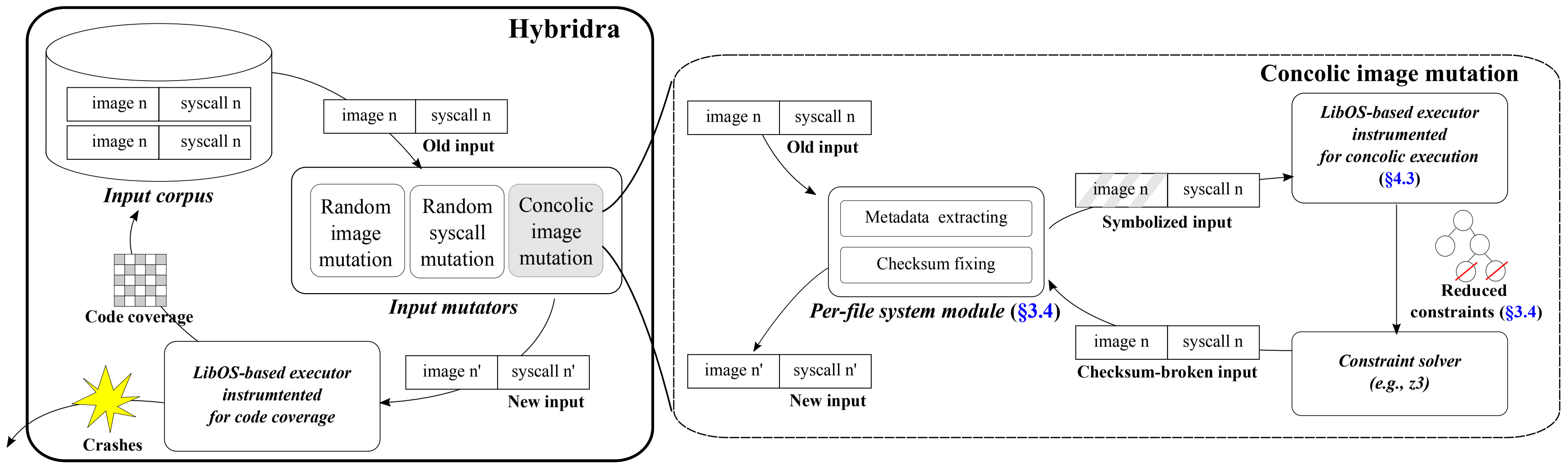


Figure 4.1: Overview of HYBRIDRA and details for its concolic execution module. HYBRIDRA generally adopts Hydra's design for exploring two dimensional space of file systems: an image and system calls. HYBRIDRA improves Hydra with its concolic execution. Its concolic execution selectively symbolizes metadata of a given file image and craft symbolic expressions using its LibOS-based executor instrumented for concolic execution. Then, HYBRIDRA invokes constraint solving after reducing its complexity for efficient solving. HYBRIDRA finally returns a new input by fixing broken checksum using its per-file system module.

HYBRIDRA mostly adopts Hydra's design, which is the state-of-the-art file system fuzzer. Unlike other file system fuzzers [32, 65, 102, 29] that modify either file system images or system calls, Hydra mutates both of them to explore the two-dimensional input space of file systems. From its input corpus, Hydra picks an input and modifies its images or system calls to generate new test cases. Similar to other gray-box fuzzing, Hydra runs an instrumented binary to efficiently measure the code coverage of its execution. After execution, Hydra reports crashing inputs or inserts interesting test cases to its input corpus for further exploration. Hydra's binary is based on LibOS, particularly LKL, for its efficiency and reproducibility.

HYBRIDRA improves Hydra by introducing a new concolic mutation module. HYBRIDRA's concolic module consists of three components: per-file system module (§4.3.1), a LibOS-based executor that is compiled for concolic execution (§4.3.2), and constraint reduction (§4.3.3). HYBRIDRA works as follows: First, HYBRIDRA's per-file system module symbolizes the metadata of its input image to reduce its scope of analysis. Second, it runs a LibOS-based executor instrumented for concolic execution to craft symbolic constraints. From its symbolic constraints, HYBRIDRA reduces their complexity according to the current strategy, which is described in §4.3.3. Then, a constraint solver will generate a new input by solving the given constraints and fixes its checksum to make it legitimate.

4.3.1 Per-file system module

HYBRIDRA's per-file system module is in charge of (1) selectively symbolizing a given image and (2) fixing the checksum of a new input from a constraint solver. HYBRIDRA only symbolizes the metadata of its image because most of the interesting behaviors are determined by its metadata even though a raw file system image is extremely large (e.g., 100MB in btrfs). Moreover, it fixes the checksum of inputs from a constraints solver. Similar to other concolic executors [9, 5], HYBRIDRA relies on an external constraint solver to make test cases from constraints. However, its constraint is incomplete for efficiency (see, §4.3.3);

therefore, the checksum of its test case is often incorrect. To prevent rejection due to this incorrect checksum, HYBRIDRA's per-file system module repairs it using domain specific rules.

Interface. To support a specific file system, we need to write the following two functions.

- `compress`: Returns metadata and their offsets from a given image.
- `fix_checksum`: Corrects the checksum of a given image.

HYBRIDRA universally defines the `decompress` function; it recovers an image from given metadata. It can be constructed using an original image and the above two functions: `compress` and `fix_checksum`. In particular, `decompress` overwrites metadata to the original image at the offsets from `compress` and repairs the checksum using `fix_checksum`.

Complexity. Even though such a module needs to be implemented for each file system, it is still tractable. To specify metadata and checksum information in a file system, we need to follow the specification for a file system; however, this requires only a shallow understanding of the file system format, and many file system utilities (e.g., `f2fsprogs` or `xfsprogs`) already have code base regarding this information. Therefore, we have reused these utilities and successfully implemented this module with only a few hundred lines of code, as shown in Table 4.5. We also have observed that these issues (i.e., huge image size and checksum) are also problematic for fuzzing; Hydra already has implemented similar functionalities for fuzzing. Therefore, we mostly reformulate its component for concolic execution. The current HYBRIDRA supports four file systems — `btrfs`, `ext4`, `ftfs`, and `xfs`— which Hydra includes in its open-source code.

4.3.2 Compilation-based Concolic Execution

To improve the execution performance of concolic execution, HYBRIDRA adopts compilation-based concolic execution [10, 11, 12]. Unlike emulation-based concolic execution, which relies on Dynamic Binary Translation (DBT) to produce symbolic expressions, compilation-based concolic execution instruments and compiles a program for concolic execution. This

Table 4.3: The memory layout of HYBRIDRA for shadow memory.

Start	**End**	**Description**
0x000000000000	0x000000010000	Reserved by Kernel
0x000000010000	0x400000000000	Shadow memory
0x400000000000	0x400c00000000	Symbol table
0x4000c0000000	0x400d00000000	Hash table
0x400d00000000	0x700000008000	Unused
0x700000008000	0x800000000000	Application memory

can improve performance significantly because it can (1) completely eliminate complex analyses in DBT such as IR transformation and (2) execute non-symbolic instructions natively without interpretation. Compilation-based concolic execution has been widely adopted in early concolic execution works [11, 12], and recently, SymCC [10] has revisited this concept with modern compiler techniques. As a result, it speeds up the existing concolic execution by up to three orders of magnitude compared to QSYM.

Workflow. Instead of using SymCC, HYBRIDRA employs Kirenenko [14], which is another compilation-based concolic executor. As shown in Figure 4.2, Kirenenko first converts code written in C/C++ to LLVM IRs. Then, following concrete execution, it inserts functions for symbolic execution, which starts with `_kirenenko` in the example. These functions build symbolic expressions based on the symbolic and concrete values of current execution. If a binary hits symbolic and is worth solving, it generate test cases by invoking a constraint solver. HYBRIDRA evaluates this worthiness based on code coverage similar to other works [4, 8].

Comparison with SymCC. Conceptually, SymCC and Kirenenko are equivalent; however, they are different in implementations, particularly for memory modeling and symbol generation as shown in Figure 4.3. Concolic execution needs to maintain mappings between an address and a symbolic expression to obtain corresponding symbolic expressions in memory. SymCC models memory by emulating a page table, while Kirenenko uses linear shadow memory similar to AddressSanitizer [61]. Since shadow memory only requires constant time complexity, shadow memory is more efficient than a page table for memory modeling,

```c
int is_double(int* a, int b) {
  return *a == 2 * b;
}
```

(a) C code

```llvm
define i32 @is_double(i32*, i32) {
  %3 = load i32, i32* %0
  %4 = shl nsw i32 %1, 1
  %5 = icmp eq i32 %3, %4
  %6 = zext i1 %5 to i32
  ret i32 %6
}
```

(b) LLVM code

```llvm
define i32 @is_double(i32*, i32) {
  ; symbolic execution (kirenenko)
  %3 = call i32 @_kirenenko_get_parameter_expression(i8 0)
  %4 = call i32 @_kirenenko_get_parameter_expression(i8 1)
  %5 = call i32 @_kirenenko_read_memory(i32* %3, i32 32)
  %6 = call i32 @_kirenenko_build_integer(i64 1)
  %7 = call i32 @_kirenenko_build_shift_left(i32 %5 , i32 %6)
  %8 = call i32 @_kirenenko_build_equal(i32 %4 , i32 %7)
  %9 = call i32 @_kirenenko_build_bool_to_bits(i32 %8)

  ; concrete execution (as before)
  %10 = load i32, i32* %0
  %11 = shl nsw i32 %1, 1
  %12 = icmp eq i32 %10, %11
  %13 = zext i1 %12 to i32

  call void @_kirenenko_set_return_expression(i32 %9)
  ret i32 %13
}
```

(c) Kirenenko-instrumented code

Figure 4.2: Overview of compilation-based concolic execution in HYBRIDRA. HYBRIDRA relies on Kirenenko for its concolic execution; it instruments code for concolic execution following semantics from concrete execution. This is conceptually equivalent to SymCC [10]. It is worth noting that this example code is almost same with the one in the SymCC paper [10] to clarify its difference from Kirenenko.

```
int         _last_symbol;
int         *_shadow_memory;
SymExpr     _symbols[MAX_SYMBOLS];

int _kirenenko_read_memory(void* memory) {
  return _shadow_memory[memory];
}

int _kirenenko_build_symbol(Kind kind, int symbol1, symbol2) {
  int new_symbol = _last_symbol++;
  _symbols[new_symbol] = {kind, symbol1, symbol2};
  return new_symbol;
}
```

(a) Pseudocode for instrumentation in Kirenenko.

```
std::map<void*, SymExpr*> _shadow_memory;

SymExpr* _symcc_read_memory(void* memory) {
  return _shadow_memory[memory];
}

SymExpr* _symcc_build_symbol(Kind kind, SymExpr* symbol1, SymExpr* symbol2) {
  return new SymExpr(kind, symbol1, symbol2);
}
```

(b) Pseudocode for instrumentation in SymCC.

Figure 4.3: Differences between Kirenenko and SymCC in their instrumentation. Both code snippets are semantically same; however, Kirenenko improves performance by avoiding logarithmic `std::map` lookup and memory allocation.

which requires logarithmic time [61]. Moreover, HYBRIDRA uses pre-allocated memory for making symbols. This can eliminate expensive memory allocations for symbol generation; however, it limits the number of symbols in concolic execution. Currently, HYBRIDRA allows 2^{30} symbols, which is enough for evaluating file systems in Linux. Therefore, we did not adopt any advanced technique to resolve the shortage of limited symbols [103]. SymCC claims that its performance difference is insignificant because page table lookup is logarithmic in the number of pages; however, it is fairly slow (i.e. $1.57\times$ according to our evaluation) for large software such as Linux Kernel.

Memory Layout. To utilize shadow memory and pre-allocated symbols, we used a specially designed memory layout, as shown in Table 4.3. Theoretically, an application can

Table 4.4: Failures from SymCC to run HYBRIDRA's LibOS-based executor. In most of cases, it fails to support multi-threading and suffers from deadlock.

File system	Failure
btrfs	z3 exception
ext4	Deadlock
f2fs	Deadlock
xfs	Deadlock

use 2^{56} memory, which will have one-to-one mapping with its shadow memory. Since each symbol is 4-bytes, the shadow memory is $4\times$ larger than the application's memory. The pre-allocated symbol table occupies 48 GB of memory, and each symbol is 48-bytes;therefore, the symbol table can hold 2^{30} symbols. The hash table in the memory layout serves as a cache for generating a symbol. To reduce the number of symbols, HYBRIDRA checks the hash table before making a new symbol whether the equivalent one has been already constructed. In this case, HYBRIDRA reuses the existing symbol instead of making a new one. This design is equivalent to SynFuzz [103] as well as LLVM's DataflowSanitizer [104].

Multi-threading. Kirenenko's design is more suitable for HYBRIDRA because of its multi-threading support. Notably, the Linux Kernel Library (LKL) uses multi-threading to emulate kernel. Therefore, our concolic execution needs to support multi-threading to run our LKL-based executor. Unfortunately, SymCC does not support multi-threading. More seriously, its design is fundamentally limited because its centralized memory requires exclusiveness for every memory operation. In particular, its memory is managed by `std::map`, which is essentially a red-black tree. This type of data structures requires locking whenever its content is being modified, which is equivalent to an application's memory write in concolic execution. Therefore, SymCC currently fails to support HYBRIDRA's LibOS-based executor for file systems for various reasons (see Table 4.4). Unlike SymCC, HYBRIDRA's flat memory model is safe if a program itself is thread-safe (i.e., no race condition). In HYBRIDRA, each address has its own slot in the shadow memory; there is no concurrent memory modification without race condition.

However, the current implementation of HYBRIDRA is still incomplete for full thread-safety; it assumes a single user for its internal data structures, e.g., a symbol hash table. We leave full thread-safety as future work because current design is acceptable for HYBRIDRA; only a main thread in LKL performs file system operations and thus uses our internal data structures to introduce new symbols even though memory can be modified by multiple threads. We believe that this issue can be resolved by replacing existing data structures into thread-safe ones [105].

Improvements from Kirenenko. For HYBRIDRA, we improved Kirenenko largely in three aspects. First, we introduce several heuristics for test case generation, including staged reduction (§4.3.3) and unrelated constraint elimination [4, 12]. In particular, unrelated constraint elimination significantly improves performance of hybrid fuzzing; it reduces the number of constraints to solve by leveraging concrete inputs [4]. Second, we introduce partial symbolization to Kirenenko. Similar to other concolic executors [10], Kirenenko can only symbolize an entire file. To symbolic only metadata of a file system image, we introduce new APIs (i.e., `_kirenenko_set_concolic`) for specifying a symbolic range. Lastly, we improve Kirenenko's *sequence symbol simplification* [4]; if t is a 16-bit symbol, and $t_1 = extract(t, 0, 8)$ and $t_2 = extract(t, 8, 16)$, Kirenenko converts $concat(t_1, t_2)$ into t for optimization. Such sub-symbols are frequent in concolic execution because memory is byte granularity but registers are multi-byte one. Therefore, if concolic execution attempts to store a symbolic value from a register to memory, it first needs to split the symbol into byte-size sub-symbols. Because of its popularity, all concolic executors, including QSYM and KLEE, have implemented this feature, particularly, in their symbol generation. However, this can introduce a temporary symbol (e.g., 24-bit symbol in computing 32-bit one), which is restricted in Kirenenko because of its limited number of symbols. Thus, Kirenenko adopts SynFuzz's [103] method; it optimizes load/store directly from symbolic inputs. Particularly, Kirenenko introduces one special symbol, called `Load`, when it is just sequence of symbolic inputs instead of making multiple byte-per-byte concatenations. We further optimize this by checking the most frequent cases in symbol generation; a final symbol can be represented with the parent of sub-symbols (e.g., the previous t_1 and t_2 case). This can further reduce the number of symbols even sub-symbols are not directly from symbolic inputs. As an immature project, we also helped to fix several bugs and other minor improvements in Kirenenko [106, 107, 108].

4.3.3 Staged Reduction

Constraint solving, which is the final step for making test cases in concolic execution, is arguably expensive. Theoretically, constraint solving is NP complete; no polynomial solution has been discovered. Therefore, constraint solving often becomes a bottleneck in concolic execution, and many techniques have been suggested for making constraint solving efficient in concolic execution [109, 110].

One of the common ways to speed up constraint solving is reduction, so-called concretization or pruning, which simplifies the constraints to solve. Historically, there are two types of reduction: (1) reduction for complexity of constraints and (2) reduction for the number of constraints. Definitely, such reduction has trade-offs; it breaks the soundness and completeness of concolic execution. As pointed out in QSYM, broken soundness is acceptable in hybrid fuzzing because coverage-guided fuzzing will efficiently filter out incorrect test cases from unsound concolic execution. However, incompleteness could be troublesome; it makes concolic execution miss interesting test cases for fuzzing. Existing concolic executors sacrifice this loss of information because of the benefits of reduction.

In this section, we discuss several reduction mechanisms as well as our newly proposed one, *staged reduction*, which benefits from multiple reduction mechanisms inspired from ensemble systems [101].

Linear reduction. Linear reduction converts non-linear expressions into linear ones by substituting concrete values from a given test case. A linear expression is attractive for software testing because it turns constraints solving into linear system solving, which has

```
def evaluate_concolic_linear(self, tc, expr):
    if self.is_non_linear(expr.kind)            \
        and self.is_symbolic(expr.left)         \
        and self.is_symbolic(expr.right):
            return self.evaluate_concrete(tc, expr)
    else:
        return self.evaluate_symbolic(tc, expr)
```

(a) Pseudocode for linear reduction. For brevity, we assume that there are only two types of expressions: addition and multiplications; however, this can be easily generalized to other operations.

```
def evaluate_concolic_bb(self, tc, expr):
    if self.is_too_frequent(expr):
        return self.evaluate_concrete(tc, expr)
    else:
        return self.evaluate_symbolic(tc, expr)
```

(b) Pseudocode for basic block reduction. Currently, `is_too_frequent` is defined in the same way of QSYM using bucketization based on call stacks and a current program counter.

```
def evaluate_concolic_stage(tc, expr):
    if tc.stage == 1:
        return self.evaluate_concolic_linear(tc, expr)
    elif tc.stage == 2:
        return self.evaluate_concolic_bb(tc, expr)
    else:
        assert(tc.stage == 3)
        return self.evaluate_symbolic(tc, expr)
```

(c) Pseudocode for staged pruning.

Figure 4.4: Pseudocode for multiple reduction mechanisms.

an efficient algorithm (e.g., Simplex method [99]). Moreover, many branch constraints in real-world software can be covered by linear expressions, as shown in various research [111, 112, 46]. One of the most famous projects using this linear reduction is SAGE [4], and recently, Eclipser [46] has suggested a novel way to solve this problem without using constraint solving, which is inherently expensive.

Figure 4.4a shows the pseudocode for linear reduction. HYBRIDRA considers several linear expressions, including arithmetic addition, logical addition, symbol extraction, and symbol concatenation. The algorithm works as follows: if our symbolic expression is non-linear with symbolic operands, it returns a concrete value based on a current input. Consequently, the final constraints should contain only linear expressions, which can be efficiently solvable.

Basic block reduction. On the contrary, QSYM suggests another way, so-called *basic block reduction*, which limits the number of constraints from the frequently executed basic blocks. It is based on a practical observation for bug finding; repeatedly executing code is generally not interesting and makes analysis complex. A few such examples are encryption, hashing, and decompression, which can stall further exploration of concolic execution. Unlike linear reduction, it can express arbitrary types of expressions but has no theoretical promise for efficient solving.

Figure 4.4b shows the pseudocode for basic block reduction. Similar to linear reduction, it concretizes a symbolic expression if its code is too frequently executed. The definition of `is_too_frequent` depends on the internal policy of concolic execution; HYBRIDRA borrows one from QSYM. In more details, HYBRIDRA relies on bitmap-style tracking for coverage with context sensitivity; it symbolizes only a logarithmic number of expressions from one basic block.

Staged reduction. To achieve both effectiveness of reduction and completeness of analysis, HYBRIDRA leverages a new technique called *staged reduction*. The key idea of staged reduction is to chain multiple reduction mechanisms ordered by their difficulty, similar to ensemble [101, 100]. As shown in Figure 4.4c, HYBRIDRA gradually moves towards a more expensive strategy only when its exploration is blocked (i.e., no progress from one stage). Currently, HYBRIDRA uses three reduction mechanisms: linear reduction, basic block reduction, and no reduction. Consequently, it can boost the exploration of concolic execution using more efficient algorithms and can escape blockage using more powerful ones.

Figure 4.5 shows how HYBRIDRA schedules multiple mutation strategies. Largely, HYBRIDRA has three mutation strategies: random image mutation, concolic image mutation, and system call mutation. Following Hydra, HYBRIDRA performs image mutation before system call one; it makes HYBRIDRA fully explore corrupted images before executing

```
  # class Hybridra
  def fuzz_one(self):
      tc = self.corpus.pick()
      found_new = self.random_img_mutator.fuzz(tc)
      if found_new: return

      # Newly introduced in Hybridra compared to Hydra
+     found_new = self.concolic_img_mutator.fuzz(tc)
+     if found_new: return

      found_new = self.syscall_mutator.fuzz(tc)
      if found_new: return

  # class ConcolicImageMutator
  def fuzz(self, tc):
      if not will_concolic(tc): return

      timedout = self.fuzz_internal(tc)

      # Move to the next stage if concolic execution successfully finishes,
      # or it fails to complete even after pre-defined trials.
      if not timedout or tc.trial >= MAX_TRIAL:
          tc.trial = 0
          tc.stage += 1
          self.update_stage()
      else:
          tc.trial += 1

  def will_concolic(self, tc):
      # 'favored' test case is the most efficient one for specific code coverage
      return tc.favored and tc.stage <= self.cur_stage

  def update_stage(self):
      # Prioritize lower stages for efficient test case generation
      self.cur_stage = MAX_STAGE
      for tc in self.corpus:
          self.cur_stage = min(self.cur_stage, tc.stage)
```

Figure 4.5: Pseudocode for scheduling in HYBRIDRA. HYBRIDRA first applies random mutation (i.e., fuzzing) for finding new test cases. If it fails, it starts concolic execution, which has internal stages. HYBRIDRA only runs concolic execution to favored test cases, which are the most efficient ones for covering specific code. It prefers concolic execution in earlier stages, which is cheaper than later ones.

system calls to make erroneous cases. In concolic execution, HYBRIDRA prefers concolic execution with a lower stage, to quickly discover useful test cases for fuzzing. Accordingly, HYBRIDRA maintains the globally minimum stage (i.e., `self.cur_stage`) and runs concolic execution only if the stage for the test case is equal to the minimum one. Moreover,

Table 4.5: HYBRIDRA's components and their lines of code.

Component	LoC	Language
Concolic Execution		
Instrumentation	1,767	C++
Runtime library	4,254	C++
Compiler wrapper	135	C
Hydra Modification		
Concolic support	144	C++
	285	Python
AFL Integration	162	C++
Per file system module		
btrfs	346	C++
ext4	176	C++
f2fs	273	C++
xfs	130	C++
Total	7,672	

HYBRIDRA limits its concolic execution for favored test cases, which are the most efficient ones for specific coverage [1]. HYBRIDRA also limits the number of trials for each stage to prevent HYBRIDRA from staying in a single stage due to an exceptionally slow test case.

4.4 Implementation

HYBRIDRA is implemented by 7.7K lines of code (LoC), as shown in Table 4.5: 6.2K lines of code for concolic execution, which is based on Kirenenko [14], 0.6K lines of code for supporting concolic execution in the state-of-the-art file system fuzzer, Hydra, and 0.9 lines of code for per file system modules.

In the remainder of this section, we discuss the unique implementation details of HYBRIDRA.

Assembly-defined symbols. Linux Kernel heavily uses assembly-specific features, which have no equivalent concepts in C, including symbol definitions. For instance, by co-utilizing `.weak` and `.set` directives in assembly, Linux Kernel can define a weak function that has a default behavior without duplicating code. Using this technique, Linux Kernel specifies system calls to invoke `sys_ni_syscall` by default if they have no architecture-dependent implementations.

To successfully compile Linux Kernel for concolic execution, HYBRIDRA needs to specially handle assembly-defined functions. In particular, HYBRIDRA modifies the ABI of each function in the module for instrumentation [104]; however, HYBRIDRA cannot detect an assembly-defined function because it relies on an IR-level analysis, resulting in a linking error. To respond to this issue, we have manually modified Linux Kernel to change the ABI of exceptional functions if a certain macro (i.e., `CONFIG_CONCOLIC`) is defined. Then, when compiling Linux Kernel for concolic execution, we specify the macro to avoid the linking error.

Kernel de-optimization. Compiler optimizations, which focus on speeding up native execution, are often harmful to concolic execution. For example, compilers introduce several intrinsics to optimize code. If LLVM identifies code for determining whether a number is a power of two, LLVM introduces `llvm.ctpop` for optimization, which is an intrinsic to count the number of bits set [113]. Without correctly defining the semantics of such intrinsics, concolic execution will fail to interpret the intrinsics properly and cannot generate interesting test cases regarding them. Moreover, compilers try to transform arithmetic operations into equivalent bit-wise ones, which are more efficient in modern computers; however, algebraic structures of arithmetic operations can help solvers find solutions. Therefore, KLEE [9] selectively applies certain optimizations instead of compiler-defined ones to compile a program for symbolic execution.

HYBRIDRA also compiles Kernel without optimization for efficient concolic execution; however, it requires not only modifying a compiler flag but also patching code. Since Linux Kernel is full of undefined behaviors according to the C language standard (e.g., hardware features), Linux has to assume certain behaviors of compilers to handle these undefined behaviors. Thus, Linux builds on top of the pre-defined optimization level, `O2` in gcc, and utilizes several features, which are unavailable if we disable optimizations, namely

`O0`. One such feature is function inlining. Linux assumes that a certain function should be inlined. However, in fact, a compiler can choose how to inline a function depending on the optimization level. Thus, we have manually fixed Linux to be compiled even without optimizations.

4.5 Evaluation

To evaluate HYBRIDRA, this section attempts to answer the following questions:

- How effective is HYBRIDRA's approach in discovering new bugs in file systems? (§4.5.1)
- How effective is HYBRIDRA's compilation-based concolic execution in overall performance of hybrid fuzzing? (§4.5.2)
- How effective is HYBRIDRA's staged reduction in terms of code coverage? (§4.5.3)

Experimental Setup. We evaluate HYBRIDRA on a machine with the Intel Xeon CPU E7-4820 processor and 256GB RAM running Ubuntu 18.04. For the experiment in §4.5.2, we have used a dedicated machine with the Intel Xeon CPU E5620 processor 48GB RAM running Ubuntu 16.04 because QSYM cannot run on the latest kernel due to its kernel dependency.

4.5.1 Newly Discovered Bugs

We ran HYBRIDRA to fuzz four file systems for two weeks — `btrfs`, `ext4`, `ftfs`, and `xfs`— in Linux v5.3, which is the latest kernel version that LKL supports. In particular, we ran HYBRIDRA for 24 hours in every fuzzing campaign with 48 instances targeting a single type of file systems. Table 4.6 shows the bugs that are discovered by HYBRIDRA. In summary, HYBRIDRA has discovered 10 bugs, and four of which are new. Note that even though the latest kernel at the time of this writing is v5.8, HYBRIDRA uses v5.3 since LKL does not support the latest one. Therefore, some bugs were stale; their root causes have been patched in the latest kernel. These stale bugs show that many file system bugs are quickly

Table 4.6: Bugs in various file systems that HYBRIDRA discovered. **Concolic** represents that the bugs are directly discovered by concolic execution, and **New** represents that the bugs are newly discovered.

File system	**File**	**Function**	**Type**	**Concolic**	**New**
btrfs	fs/btrfs/extent_io.c	extent_io_tree_panic	Null pointer dereference	✓	✓
	fs/btrfs/free-space-cache.c	tree_insert_offset	BUG()	✓	
	fs/btrfs/extent-tree.c	btrfs_drop_snapshot	BUG()		
	fs/btrfs/extent-tree.c	walk_down_proc	BUG()	✓	
	fs/btrfs/relocation.c	merge_reloc_root	BUG()		
	fs/btrfs/root-tree.c	btrfs_find_root	BUG()	✓	✓
	fs/btrfs/ctree.c	setup_items_for_insert	BUG()		✓
	fs/btrfs/volumes.c	calc_stripe_length	Divide by zero		✓
ext4	fs/ext4/super.c	ext4_clear_journal_err	BUG()		
f2fs	fs/f2fs/segment.c	f2fs_build_segment_manager	Out-of-bounds read		

captured by developers thanks to continuous fuzzing efforts [29] and on-going research projects for file system fuzzing including Janus [68] and Hydra [15]. It makes discovering bugs in file systems extremely difficult; however, HYBRIDRA successfully discovered four bugs in `btrfs`, and we reported them to developers.

Concolic execution is helpful in discovering bugs in file systems. The **Concolic** column in Table 4.6 represents that HYBRIDRA found the bugs directly from concolic execution; the final mutation strategy for finding the bug is concolic execution. As a random process, fuzzing makes it difficult to understand the impacts of newly introduced strategies [114]. We believe that concolic execution is likely to help find other bugs by providing interesting test cases for fuzzing. However, this direct relationship can more clearly show the impacts of concolic execution. Linux Kernel's coding convention actually helps HYBRIDRA discover bugs in file systems. To mitigate the impacts from incorrect assumptions, Linux Kernel heavily uses the `BUG()` macro, which is equivalent to `assert()` in many programming languages. Unlike fuzzing, concolic execution can understand meanings of `BUG()` as constraints and produces test cases that violate the assertions. This helps HYBRIDRA discover many bugs regarding `BUG()`, as shown in the table.

4.5.2 Compilation-based Concolic Execution

To show the effectiveness of compilation-based concolic execution in HYBRIDRA, we have compared HYBRIDRA with QSYM and SymCC [10]. QSYM is an emulation-based concolic executor, and SymCC is a compilation-based one, but it uses page-table modeling instead of shadow memory, unlike HYBRIDRA. We have compiled HYBRIDRA's executor with a native compiler for QSYM and with SymCC's compiler for four file systems: `btrfs`, `ext4`, `ftfs`, and `xfs`. To measure the overhead for symbolic execution excluding solving, we disable symbolic inputs; we ran QSYM with an interactive mode without any standard input, and we ran SymCC by setting the environment variable, `SYMCC_NO_SYMBOLIC_INPUT`, following the author's guidelines. Similarly, we intentionally modified the metadata information for

Table 4.7: The average, ratio to HYBRIDRA, and standard deviation of symbolic execution for QSYM, SymCC, and HYBRIDRA for four file systems. It shows that HYBRIDRA is order-of-magnitude faster than QSYM and 56% faster than SymCC.

	btrfs			**ext4**			**f2fs**			**xfs**		
	QSYM	SymCC	HYBRIDRA	QSYM	SymCC	HYBRIDRA	QSYM	SymCC	HYBRIDRA	QSYM	SymCC	HYBRIDRA
Mean	218.50	1.14	1.07	168.73	1.07	0.63	177.44	1.48	0.93	173.18	1.28	0.68
Ratio	204.92	1.07	–	269.41	1.71	–	191.21	1.60	–	254.57	1.88	–
s.d.	2.94	0.01	0.00	1.57	0.05	0.00	2.67	0.02	0.00	1.52	0.10	0.00

a file system to make HYBRIDRA symbolize nothing. For seed inputs, we used default seed images (e.g., `btrfs-10.img`) and system call sequences (e.g., `open_link_fsync0`) from Hydra. We repeated this experiment three times and report its average and standard deviation.

As shown in Table 4.7, HYBRIDRA outperforms both QSYM and SymCC; HYBRIDRA is 230× faster than QSYM and 1.57× faster than SymCC on average. This is because QSYM requires expensive dynamic binary translation for its binary-only concolic execution. On the contrary, HYBRIDRA can eliminate this translation overhead by implanting functions for symbolic execution before compilation. Moreover, HYBRIDRA can outperform SymCC by reducing memory lookup cost using shadow memory, avoiding expensive page-table modeling. According to our profiling, the majority of overhead (i.e. 30%) comes from memory modeling in SymCC. It is worth noting that we failed to run concolic execution with SymCC for file systems due to its failures in supporting multi-threading (see, Table 4.4).

4.5.3 Staged Reduction

In this subsection, we evaluate the impacts of staged reduction, which is described in §4.3.3. In this evaluation, HYBRIDRA disables its system call mutation to compare our hybrid image mutation with random image mutation.

Concolic only. To understand the benefits of various reduction mechanisms, we first evaluate HYBRIDRA with concolic-only mode without fuzzing. In particular, we have used four variants of HYBRIDRA: HYBRIDRA with (1) staged reduction, (2) basic block reduction, (3) linear reduction, and (4) no reduction. To fairly compare, HYBRIDRA allows 9 minutes for one test case; for example, HYBRIDRA with staged reduction uses 3 minutes per stage, and one with other reduction uses 9 minutes for its single concolic execution. We have run three experiments for 24 hours and report their average values. It is worth noting that HYBRIDRA with concolic-only mode can be terminated earlier than a given timeout (i.e., 24 hours) if it fails to generate new test cases after consuming every case.

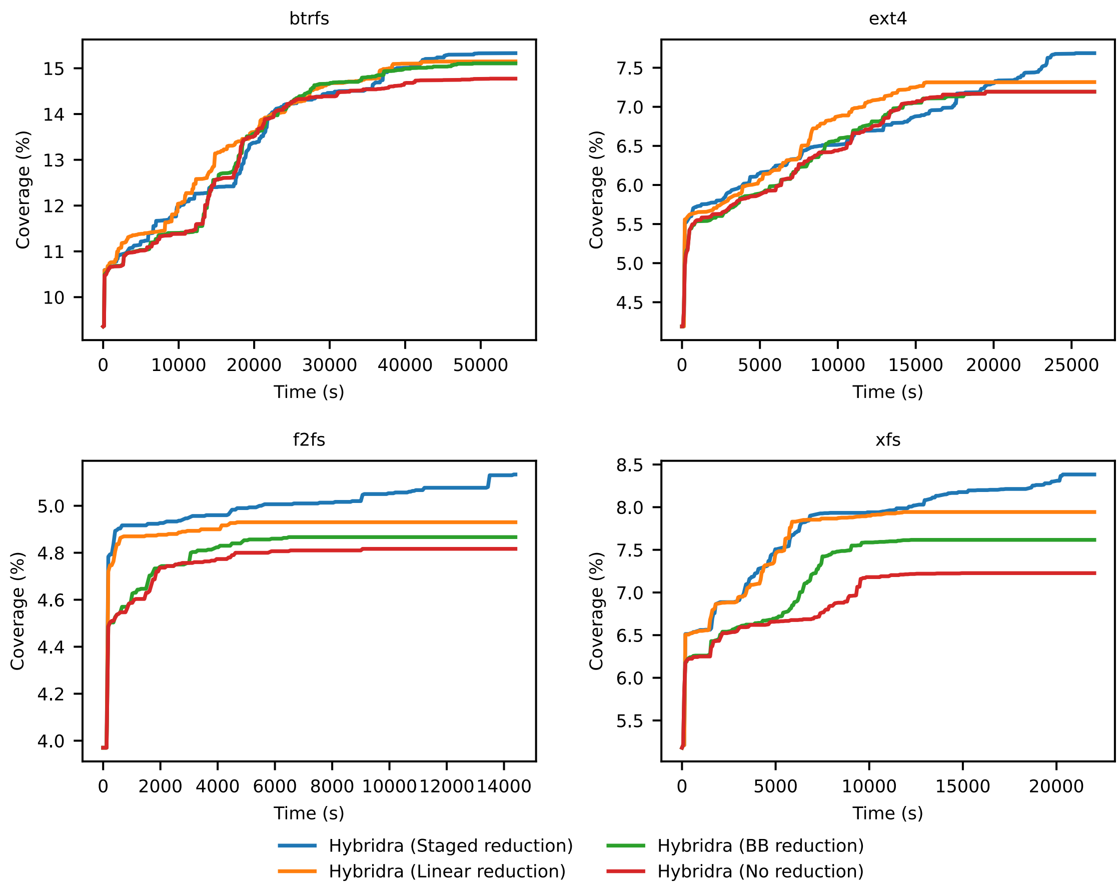


Figure 4.6: Code coverage of HYBRIDRA in concolic-only mode with various reduction mechanisms. By taking advantage of several reduction mechanisms, HYBRIDRA with staged reduction outperforms others even using the same timeout.

As shown in Figure 4.6, staged reduction has achieved more code coverage than other reductions, which shows its effectiveness. Interestingly, linear reduction works fairly well thanks to its efficient solving; however, its limited expressiveness makes it converge too early (see, flat lines in `ext4`, `ftfs`, and `xfs`). Compared to that, our staged reduction initially works similar to the linear reduction but still can make more complex test cases from other reductions (i.e., basic block reduction and no reduction); it eventually has achieved the highest code coverage.

Hybrid fuzzing. We also evaluate the end-to-end effectiveness of various reduction mechanisms with fuzzing. Compared to the previous one, this evaluation has two major differences: (1) HYBRIDRA re-enables random image mutation, and (2) it has no time limit

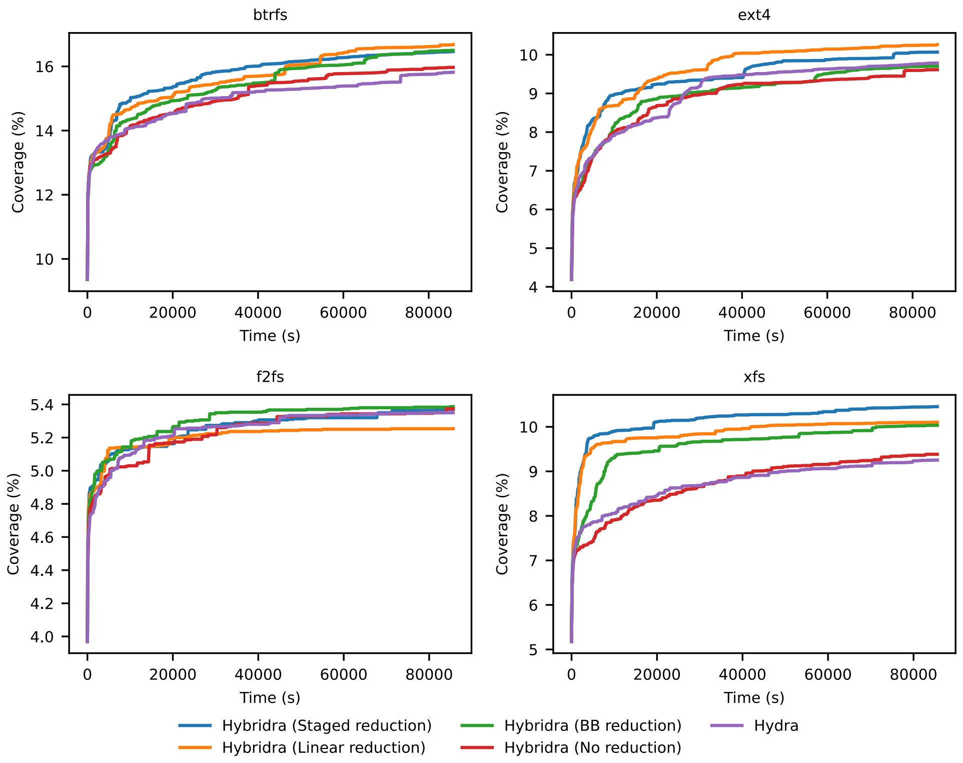


Figure 4.7: Code coverage of Hydra and HYBRIDRA in hybrid fuzzing with various reduction mechanisms.

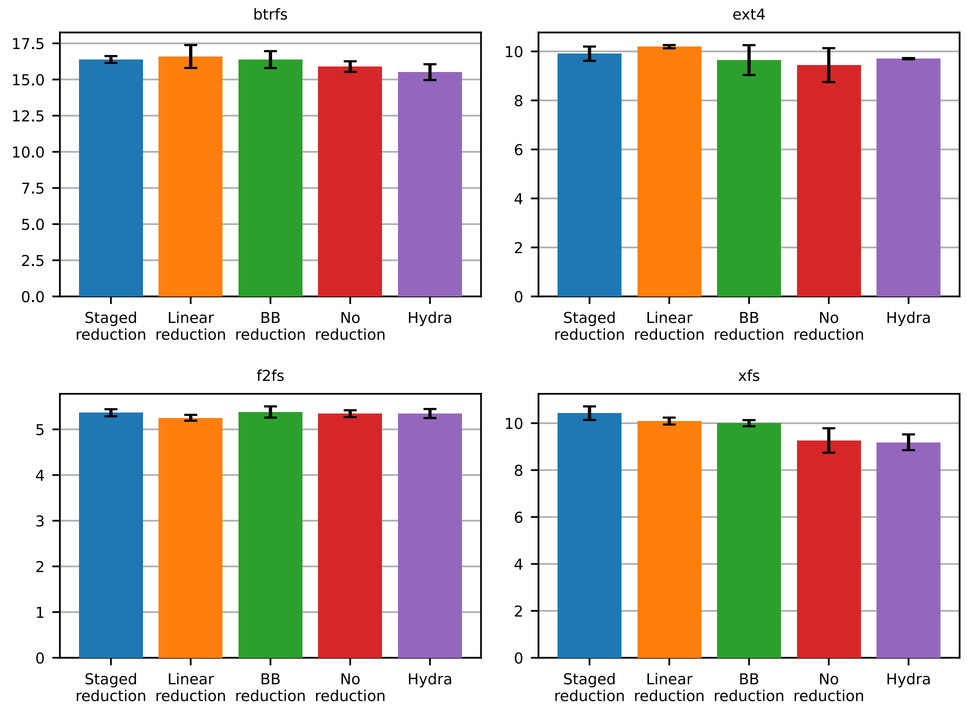


Figure 4.8: Eventual code coverage achieved by HYBRIDRA with its standard deviation. In summary, HYBRIDRA outperforms the fuzzing-only approach, Hydra.

for one concolic execution instance. In particular, we used the same scheduling algorithm for every reduction mechanism, as shown in Figure 4.5; HYBRIDRA retries concolic execution until it successfully terminates.

As shown in Figure 4.7 and Figure 4.8, hybrid fuzzing is helpful in exploring more code in file systems. In particular, HYBRIDRA with staged reduction has achieved the best code coverage on average among various reduction mechanisms. More importantly, HYBRIDRA outperforms fuzzing-only solution, Hydra, in all four file systems that we tested. This is because concolic execution has provided interesting test cases, which are difficult for fuzzing to discover.

4.6 Discussion and Limitation

Adoption beyond file systems. HYBRIDRA is based on the Linux Kernel Library (LKL) for efficient hybrid fuzzing. We only have focused on file systems; however, LKL also supports emulating network interfaces. Therefore, we believe HYBRIDRA can be extended to test the Linux's networking stack without substantial effort. Moreover, HYBRIDRA can work with other user mode Linux, including KUnit [115], a unit testing framework for Linux Kernel. Integration KUnit with HYBRIDRA would be interesting because it can diversify HYBRIDRA's scope to various features in Linux Kernel.

Limitations. Currently, HYBRIDRA limitedly supports thread safety under the assumption that only one thread modifies symbolic variables. We plan to overcome this limitation by re-designing thread-unsafe data structures in HYBRIDRA, including the symbol hash table. Moreover, it only supports symbolic integer values whose lengths are less than or equal to 64 bits. This is sufficient for testing major features in file systems; however, we plan to support floating-point and integers beyond 64 bits.

CHAPTER 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this dissertation, we discuss *concolic execution tailored for hybrid fuzzing* to overcome limited scalability of exiting hybrid fuzzers. Particularly, we propose two key approaches to make hybrid fuzzing scalable to real-world software and even kernel file systems: (1) systematic design for fast symbolic emulation and (2) efficient test case generation mechanisms by temporarily sacrificing soundness and completeness of concolic execution.

We first introduce QSYM, which is a binary-only concolic execution engine for hybrid fuzzing. QSYM redesigns existing binary-only concolic executors for performance to eliminate redundant symbolic emulations. QSYM also proposes two new techniques for efficient test case generation — optimistic solving and basic block pruning — based on the observation that coverage-guided fuzzing can act as an efficient validator to filter out incorrect test cases from concolic execution. Our evaluation results showed that QSYM outperformed Driller in the DARPA CGC binaries and VUzzer in the LAVA-M test set. More importantly, QSYM found 13 previously unknown bugs in the eight non-trivial programs, such as ffmpeg and OpenJPEG, which have heavily been tested by the state-of-the-art fuzzer, OSS-Fuzz, on Google's distributed fuzzing infrastructure.

We also propose HYBRIDRA to apply hybrid fuzzing to a more gigantic and convoluted target, a kernel file system. Since many issues for testing a file system are common for fuzzing and concolic execution, HYBRIDRA reformulates Hydra's approaches for concolic execution, which is the state-of-the-art file system fuzzer. Moreover, to fully utilize availability of source code, HYBRIDRA adopts compilation-based concolic execution with multi-threading support. HYBRIDRA also suggests staged reduction, which can combine multiple reduction mechanisms for efficient test case generation. As a result, HYBRIDRA outperforms Hydra by achieving higher code coverage in every file system that we tested (`btrfs`, `ext4`, `ftfs`, and `xfs`) and found four new bugs in `btrfs`.

5.2 Future work

This dissertation presents concepts and techniques for designing concolic execution for hybrid fuzzing. In this section, we discuss three research topics that are worth exploring.

Stateful Hybrid Fuzzing. Currently, concolic execution and its application, hybrid fuzzing, only focus on stateless software, mostly file parsers, but not stateful ones. Behaviors of a stateless program can be easily representable through symbolic constraints by emulating low-level execution such as machine instructions or IRs. However, a state of a certain program requires higher-level analysis, including data flows. In particular, two symbolic branches could be equivalent in control flow analysis, which have the same call stacks and addresses; however, they could have different data flows from different states. A data flow is difficult to handle because it is more sensitive and disorganized than a control flow. Since every execution has its own data flow, we need bucketization or summarization for grouping multiple data flows; however, there is no appropriate one for arbitrary programs. Therefore, existing concolic execution relies on domain-specific abstractions to convert state information into symbolic constraints [116]. Unfortunately, defining such an abstraction requires a lot of manual effort, which is not feasible in complex software such as Linux Kernel. HYBRIDRA also suffers from this limitation, and it only focuses on concolic execution for file system images, not system calls, which are stateful.

Verified Binary Concolic Execution. For instruction-level symbolic execution, QSYM manually embeds a symbolic representation for each instruction. Obviously, this is error-prone and fails to support many x86 instructions including floating point instructions. Moreover, it is also challenging to adopt QSYM's idea to other architectures such as ARM or MIPS due to a large amount of manual efforts.

One opportunity to resolve this issue is to use formal semantics of instruction set architectures (ISA) [117, 118]. A symbolic expression for a instruction is a different way of representing its semantics; we can auto-generate a symbolic expression from the formal semantics. We can still rely on high-performance DBT for concrete execution. Accordingly, concolic execution can support more instructions in a verified manner for multiple architectures without sacrificing performance.

Formalization. Concolic execution has been widely used for various goals, including sound analysis, whitebox fuzzing, and hybrid fuzzing. Even though this dissertation discusses the differences between classical concolic execution and one for hybrid fuzzing from observations in practice, it has no formal definition. Formally defining concolic execution for hybrid fuzzing can help further exploration of this space by clarifying underlying problems. It would also be helpful to organize existing work; a number of hybrid fuzzing works have been published after QSYM. Due to the absence of formalization, it is still ambiguous whether a specific work is for hybrid fuzzing or for generic concolic execution.

REFERENCES

[1] M. Zalewski, *American fuzzy lop*, http://lcamtuf.coredump.cx/afl/, 2015.

[2] Google, *Honggfuzz*, https://github.com/google/honggfuzz, 2010.

[3] ——, *OSS-Fuzz - continuous fuzzing of open source software*, https://github.com/google/oss-fuzz, 2016.

[4] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2008.

[5] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, Mar. 2011.

[6] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, Minneapolis, MN, May 2007.

[7] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," Master's thesis, Carnegie Mellon University Pittsburgh, PA, 2012.

[8] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[9] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[10] S. Poeplau and A. Francillon, "Symbolic execution with SYMCC: Don't interpret, compile!" In *Proceedings of the 29th USENIX Security Symposium (Security)*, Aug. 2020.

[11] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun. 2005.

[12] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference (ESEC) / 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Lisbon, Portugal, Sep. 2005.

[13] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Aug. 2020.

[14] C. Song, *Kirenenko: Super fast concolic execution engine based on source code taint tracing*, https://github.com/ChengyuSong/Kirenenko, 2020.

[15] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[16] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[17] OpenRCE, *Sulley*, https://github.com/OpenRCE/sulley.git, 2012.

[18] PeachTech, *Peach fuzzer*, https://www.peach.tech/products/peach-fuzzer/, 2007.

[19] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, Jun. 2011.

[20] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Beijing, China, Jun. 2012.

[21] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, Jun. 2014.

[22] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Seattle, WA, Jun. 2013.

[23] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, Montreal, QC, Canada, May 2019.

[24] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars.," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[25] W. Xu, S. Park, and T. Kim, "FREEDOM: Engineering a State-of-the-Art DOM Fuzzer," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Orlando, FL, Nov. 2020.

[26] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing javascript engines with aspect-preserving mutation," in *Proceedings of the 41th IEEE Symposium on Security and Privacy (Oakland)*, May 2020.

[27] S. Groß, "Fuzzil: Coverage guided fuzzing for javascript engines," Master's thesis, Karlsruhe Institute of Technology, 2018.

[28] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[29] Google, *syzkaller is an unsupervised, coverage-guided kernel fuzzer*, https://github.com/google/syzkaller, 2018.

[30] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing OS fuzzer seed selection with trace distillation," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Aug. 2020.

[31] H. Han and S. K. Cha, "IMF: Inferred model-based fuzzer," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[32] I. van Sprundel, LMH, S. Grubb, E. Sandeen, and J. Wilson, *Fsfuzzer*, http://people.redhat.com/sgrubb/files/fsfuzzer-0.7.tar.gz.

[33] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data Race Fuzzing for Kernel File Systems," in *Proceedings of the 41th IEEE Symposium on Security and Privacy (Oakland)*, May 2020.

[34] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[35] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[36] K. Serebryany, "Libfuzzer–a library for coverage-guided fuzz testing," *LLVM project*, 2015.

[37] M. Böhme, V. Manès, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," in *Proceedings of the European Software Engineering Conference (ESEC) / ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE) 2020*, Nov. 2020.

[38] V. Ganesh, T. Leek, and M. Rinard, "Taint-based Directed Whitebox Fuzzing," in *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, Vancouver, Canada, May 2009.

[39] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[40] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-State Based Binary Fuzzing," in *Proceedings of the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Paderborn, Germany, Sep. 2017.

[41] C. Lemieux and K. Sen, "FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage," *ArXiv e-prints*, Sep. 2017.

[42] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[43] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[44] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence.," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[45] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2010.

[46] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, Montreal, QC, Canada, May 2019.

[47] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[48] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar, "Generating tests from counterexamples," in *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, Scotland, UK, May 2004.

[49] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with java pathfinder," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Boston, MA, Jun. 2004.

[50] C. Csallner and Y. Smaragdakis, "Check'n'crash: Combining static checking and testing," in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, St. Louis, MO, May 2005.

[51] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *International Conference on Computer Aided Verification*, Springer, 2007, pp. 519–531.

[52] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct. 2006.

[53] F. Bellard, "QEMU, a fast and portable dynamic translator.," in *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, Anaheim, CA, Apr. 2005.

[54] Y. Liu, H.-W. Hung, and A. A. Sani, "Mousse: A system for selective symbolic execution of programs with untamed environments," in *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, Heraklion, Crete, Greece, Apr. 2020.

[55] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013.

[56] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Beijing, China, Jun. 2012.

[57] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, “Enhancing symbolic execution with veritesting,” in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, Hyderabad, India, May 2014.

[58] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, “Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations,” in *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

[59] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan, “DASE: Document-assisted symbolic execution for improving automated software testing,” in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, Florence, Italy, May 2015.

[60] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing mayhem on binary code,” in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[61] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A fast address sanity checker,” in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jun. 2012.

[62] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, “HFL: Hybrid fuzzing on the linux kernel,” in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[63] L. Zhao, Y. Duan, H. Yin, and J. Xuan, “Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing.,” in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[64] Y. Chen, M. Ahmadi, B. Wang, L. Lu, *et al.*, “Meuzz: Smart seed scheduling for hybrid fuzzing,” in *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Oct. 2020.

[65] D. Jones, *Linux system call fuzzer*, https://github.com/kernelslacker/trinity, 2018.

[66] MWR Labs, *Cross Platform Kernel Fuzzer Framework*, https://github.com/mwrlabs/KernelFuzzer, 2016.

[67] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, “kafl: Hardware-assisted feedback fuzzing for OS kernels,” in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[68] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, “Fuzzing File Systems via Two-Dimensional Input Space Exploration,” in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[69] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-scale automated vulnerability addition," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[70] Google, *Fuzzing for Security*, https://blog.chromium.org/2012/04/fuzzing-for-security.html, 2012.

[71] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, "Stratified synthesis: Automatically learning the x86-64 instruction set," in *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Santa Barbara, CA, Jun. 2016.

[72] Intel, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 2: Instruction Set Reference, A–Z*, 2016.

[73] L. Project, *LLVM language reference manual*, https://llvm.org/docs/LangRef.html, 2003.

[74] X. Leroy and D. Doligez, *Mosml/md5sum.c at master*, https://github.com/kfl/mosml/blob/master/src/runtime/md5sum.c, 2014.

[75] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, Jun. 2007.

[76] R. David, S. Bardin, J. Feist, L. Mounier, M.-L. Potet, T. D. Ta, and J.-Y. Marion, "Specification of concretization and symbolization policies in symbolic execution.," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Saarbrücken, Germany, Jul. 2016.

[77] T. Liu, M. Araújo, M. d'Amorim, and M. Taghdiri, "A comparative study of incremental constraint solving approaches in symbolic execution," in *Proceedings of the Haifa Verification Conference(HVC'14)*, Haifa, Israel, Nov. 2014.

[78] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[79] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, "Your exploit is mine: Automatic shellcode transplant for remote exploits," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[80] J. Hendrix and B. F. Jones, “Bounded integer linear constraint solving via lattice search,” in *Proceedings of the International Workshop on Satisfiability Modulo Theories*, 2015.

[81] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun. 2005.

[82] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis, “A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware.,” in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2012.

[83] *CVE-2017-11543*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-11543.

[84] *CVE-2017-1000249*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000249.

[85] O. Chang, A. Arya, K. Serebryany, and J. Armour, *OSS-Fuzz: Five months later, and rewarding projects*, https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html, 2017.

[86] *PNG specification: Chunk specifications*, https://www.w3.org/TR/PNG-Chunks.html, 1996.

[87] DARPA, *Cyber Grand Challenge*, https://www.cybergrandchallenge.com/, 2016.

[88] Shellphish, *Shellphish AFL package*, https://github.com/shellphish/shellphish-afl, 2016.

[89] *Cppcheck: A tool for static C/C++ code analysis*, http://cppcheck.sourceforge.net/.

[90] M. Rajpal, W. Blum, and R. Singh, “Not all bytes are equal: Neural byte sieve for fuzzing,” *arXiv preprint arXiv:1711.04596*, 2017.

[91] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, “End-to-end verification of processors with isa-formal,” in *Proceedings of the 28th International Conference on Computer Aided Verification (CAV)*, Toronto, Canada, Jul. 2016.

[92] M. Cai, H. Huang, and J. Huang, “Understanding security vulnerabilities in file systems,” in *Proceedings of the 10th Asia-Pacific Workshop on Systems (APSys)*, Hangzhou, China, Aug. 2019.

[93] *CVE-2015-8660*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8660, 2015.

[94] *Bobfuzzer*, https://github.com/bobfuzzer/CVE, 2019.

[95] M. Larabel, *The Linux kernel enters 2020 at 27.8 million lines in git but with less developers for 2019*, https://www.phoronix.com/scan.php?page=news_item&px=Linux-Git-Stats-EOY2019.

[96] O. Purdila, L. A. Grijincu, and N. Tapus, “Lkl: The linux kernel library,” in *9th RoEduNet IEEE International Conference*, IEEE, 2010.

[97] M. Cho, S. Kim, and T. Kwon, “Intriguer: Field-level constraint solving for hybrid fuzzing,” in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[98] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, “SAVIOR: Towards bug-driven hybrid testing,” in *Proceedings of the 41th IEEE Symposium on Security and Privacy (Oakland)*, May 2020.

[99] G. B. Dantzig, “Origins of the simplex method,” in *A history of scientific computing*, 1990, pp. 141–151.

[100] B. V. Dasarathy and B. V. Sheela, “A composite classifier system design: Concepts and methodology,” *Proceedings of the IEEE*, vol. 67, no. 5, pp. 708–713, 1979.

[101] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, “Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers,” in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[102] V. Nossum and Q. Casasnovas, “Filesystem Fuzzing with American Fuzzy Lop,” in *Vault Linux Storage and Filesystems Conference*, 2016.

[103] W. Han, M. L. Rahman, Y. Chen, C. Song, B. Lee, and I. Shin, “SynFuzz: Efficient concolic execution via branch condition synthesis,” *arXiv preprint arXiv:1905.09532*, 2019.

[104] The Clang Team, “DataFlowSanitizer design document,” 2007.

[105] G. E. Blelloch, D. Anderson, and L. Dhulipala, “ParlayLib – a toolkit for parallel algorithms on shared-memory multicore machines,” in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, Jul. 2020.

[106] C. Song, *Merge load/store patch from @jakkdu*, https://github.com/ChengyuSong/Kirenenko/commit/3e0f02cb, 2020.

[107] ——, *Add partial concrete test case from @jakkdu*, https://github.com/ChengyuSong/Kirenenko/commit/6e8d8bd0, 2020.

[108] ——, *Add partial concrete test case from @jakkdu*, https://github.com/ChengyuSong/Kirenenko/commit/28997fca, 2020.

[109] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, “Efficient sampling of sat solutions for testing,” in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, Gothenburg, Sweden, May 2014.

[110] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, “Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction,” in *Proceedings of the 41th IEEE Symposium on Security and Privacy (Oakland)*, May 2020.

[111] Y. Xie, A. Chou, and D. Engler, “Archer: Using symbolic, path-sensitive analysis to detect memory access errors,” in *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, Portland, OR, May 2003.

[112] W. Le, “Segmented symbolic analysis,” in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013.

[113] LLVM Project, “LLVM language reference manual,” 2003.

[114] L. S. J. Metzman, A. Arya, and L. Szekeres, “Fuzzbench: Fuzzer benchmarking as a service,” *Google Security Blog*, 2020.

[115] Google, *KUnit*, https://kunit.dev/, 2019.

[116] L. Simon, S. Liang, A. Rahmati, and M. Grace, “Caterpillar: Iterative concolic execution for seed generation,” in *1st International KLEE Workshop on Symbolic Execution (KLEE)*, 2018.

[117] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, “A complete formal semantics of x86-64 user-level instruction set architecture,” in *Proceedings of the 2019 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Phoenix, AZ, Jun. 2019.

[118] A. Armstrong, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, P. Sewell, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, *et al.*, “ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS,” in *Proceedings of the 46th ACM Symposium on Principles of Programming Languages (POPL)*, Cascais, Portugal, Jan. 2012.