QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing

Insu Yun[†] Sangho Lee[†] Meng Xu[†] Yeongjin Jang[*] Taesoo Kim[†]

[†] *Georgia Institute of Technology*
[*] *Oregon State University*

Abstract

Recently, hybrid fuzzing has been proposed to address the limitations of fuzzing and concolic execution by combining both approaches. The hybrid approach has shown its effectiveness in various synthetic benchmarks such as DARPA Cyber Grand Challenge (CGC) binaries, but it still suffers from scaling to find bugs in complex, real-world software. We observed that the performance bottleneck of the existing concolic executor is the main limiting factor for its adoption beyond a small-scale study.

To overcome this problem, we design a fast concolic execution engine, called QSYM, to support hybrid fuzzing. The 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, so faster, instruction-level symbolic emulation. Additionally, QSYM loosens the strict soundness requirements of conventional concolic executors for better performance, yet 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 the state-of-the-art fuzzers, AFL and OSS-Fuzz.

1 Introduction

The computer science community has developed two notable technologies to automatically find vulnerabilities in software: namely, coverage-guided fuzzing [1–3] and concolic execution [4, 5]. 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, such as `x > 0`. On the contrary, concolic execution is good at finding inputs that drive the program into tight and complex branch conditions, such as `x == 0xdeadbeef`, but it is very expensive and slow to formulate these constraints and to solve them.

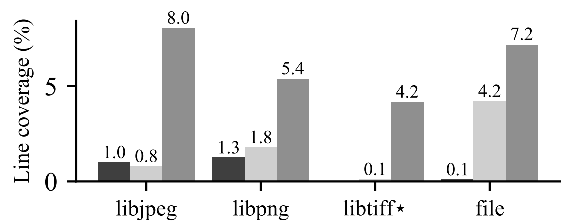

Figure 1: Newly found line coverage of popular open-source software by state-of-the-art concolic executors, Driller and S2E, and our system, QSYM, until they saturated. We used five test cases in each project that have the largest code coverage. Test cases generated by QSYM cover significantly more lines than both concolic executors. In libtiff, Driller could not generate any test case due to incomplete modeling for `mmap()`.

To take advantage of both worlds, a hybrid approach [6–8], 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). For example, Driller [8] demonstrates its effectiveness of the hybrid fuzzing in the 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 real bugs in non-trivial, real-world applications. We observed that the performance bottlenecks of their concolic executors are the main limiting factor that deters their adoption beyond the synthetic benchmarks.

Unlike the promise made by concolic executors, they fail to scale to real-world applications: the symbolic emulation is too slow in formulating path constraints (e.g., libjpeg and libpng in Figure 1) or it is often not even possible to generate these constraints (e.g., libtiff and file in Figure 1) due to the incomplete and erroneous environment models (Table 4).

In this paper, we systematically analyze the performance bottlenecks of concolic execution and then overcome the problem by tailoring the concolic executor to support hybrid fuzzing (§2). The key idea 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 (§3.1). 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 (§3.2), and to improve the performance, i.e., by pruning uninteresting basic blocks (§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 [9]. 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 [10], 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 [11] 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 paper 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 paper is organized as follows. §2 analyzes the performance bottleneck of current hybrid fuzzing. §3 and §4 depict the design and implementation of QSYM, respectively. §5 evaluates QSYM with benchmarks, test sets, and real-world test cases. §7 explains QSYM's limitations and possible solutions. §8 introduces related work. §9 concludes this paper.

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.

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 1 shows

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

Table 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)` [12] to test angr because angr does not support the `fadvise` syscall, which is used in the coreutils applications.

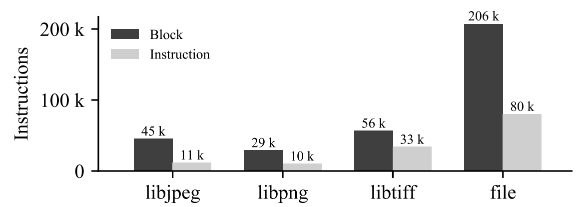

Figure 2: The number of instructions in symbolic basic blocks and the number of symbolic instructions in popular open-source software. More than half of the instructions in the basic blocks are not symbolic instructions, which can be executed natively.

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 3,000 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 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 1,795 instructions [13] described in a 2,000-page manual [14]. 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 [15]) 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., 1,795 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 machine instruction results in multiple IR instructions. For instance, based on our evaluation, the VEX IR [16], 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 2), 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.

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 [17]. 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 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.

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 shows a code snippet of the file program. If concolic execution visits `file_zmagic()`, it sticks there

```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: 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()`.

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, 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 Design

In this section, we explain our design decisions to realize QSYM. Figure 4 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

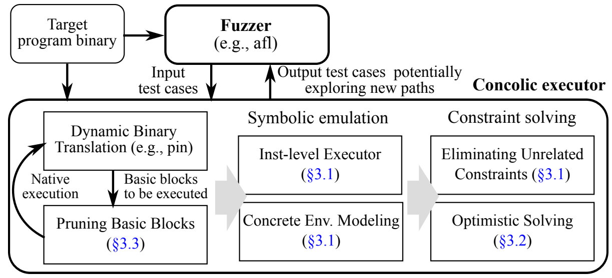

Figure 4: Overview of QSYM's architecture as a hybrid fuzzer. QSYM takes a test case and a target binary as inputs and attempts to generate new test cases that might explore new paths. It uses Dynamic Binary Translation (DBT) to natively execute the input binary as well as to select basic blocks for 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.

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 basic blocks. By doing this, QSYM reduced the number of symbolic emulations by a significant magnitude (5×, see Figure 10 in §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.

3.1 Taming Concolic Executor

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

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

```
// 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 5: 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.

Initial PoV	Qsym	Driller
1 # create user	1 # create user	1 # create user
userone	userone	\xfb\xfb\xfb\xfb\xf4\xf1\xf1
1 # create user	1 # create user	1 # create user
usertwo	usertwo	\xfb\xfb\xfb\xfb\x0b\xfb\xf1
2 # login	2 # login	2 # login
userone	userone	\xfb\xfb\xfb\xfb\xf4\xf1\xf1
1 # send message	4 # delete message	4 # delete message

Figure 6: 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.

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 5), 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 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 [18]. 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 6 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 6). Moreover, due to its environment support (§3.1) or various heuristics (§3.2, §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, 19], but it is difficult to apply to forking-based symbolic execution, which breaks process boundaries [20]. Since QSYM can achieve performance without introducing forking-based symbolic execution [21], 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.

3.2 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). 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 10). 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) 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 [22]. 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 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, 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 [23].

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*.

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

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

Table 2: QSYM's main components and their lines of code.

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.

4 Implementation

We implement the concolic executor from scratch. QSYM consists of 16K lines of code (LoC) in total, and Table 2 summarizes the complexity of each of its components. QSYM relies on Intel Pin [24] 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 [25] 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.

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? (§5.1, §5.2)
- **Justifying design decisions.** How effective are the design decisions made by QSYM in terms of bug finding? (§5.3, §5.4, §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? (§5.3)
  2. **Optimistic constraints solving.** How reasonable is QSYM's optimistic constraints solving in terms of finding bugs? (§5.4)
  3. **Pruning basic blocks.** How effective is our approach to prune basic blocks in terms of the overall performance and code coverage? (§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 (§5.1, §5.2, and §5.4) and one core for testing concolic execution only (§5.3 and §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.

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). It is worth noting that Google's OSS-Fuzz generated 10 trillion test inputs a day [28] 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. §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 4.

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: 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 [26, 27]. 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	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`

Table 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.

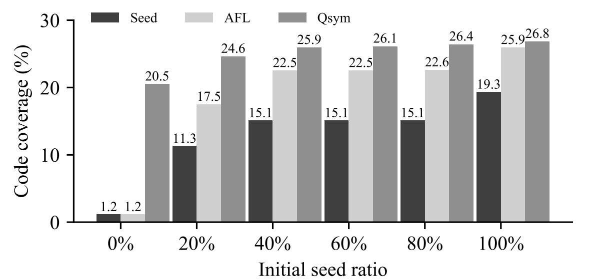

Figure 7: Code coverage of libpng after a six-hour run of QSYM and AFL (two AFL instances for a fair comparison) with an increasing number of seeding inputs. In the 0% case, we put an invalid PNG file consisting of 256 'A's as an initial input. The 100% case includes 141 sample PNG image files provided 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.

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 7). 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 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 [29]. This example also hints at the difficulty of constructing complete test cases that cover all the fea-

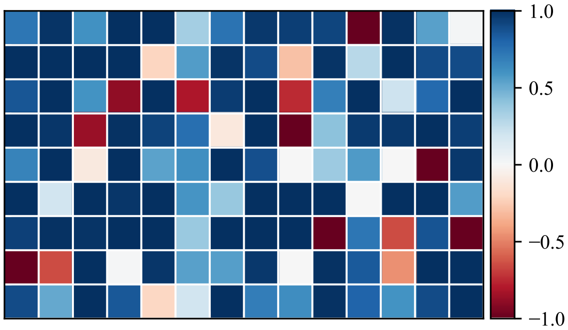

Figure 8: This color map depicts the relative code coverage for five minutes that compares QSYM's with Driller's: the blue color means that QSYM found more code than Driller, and the red color means the opposite (see §5.3 for the exact formula). Each cell represents each CGC challenge in alphabetical 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.

tures implemented in software, where we believe QSYM's approach can shed some light on.

5.3 Fast Symbolic Emulation

To show the performance benefits of QSYM's symbolic emulation, we used the DARPA CGC dataset [30] to compare QSYM with Driller, which placed third in the CGC competition [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 [31] 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 8 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 9). Even with the 30-min timeout of Driller, QSYM explored more paths in 98 out of 126 binaries, whereas Driller's

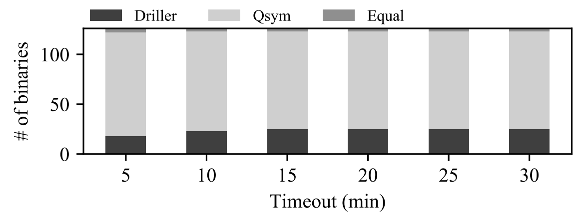

Figure 9: Comparing QSYM (5-min timeout) with Driller while increasing the time for constraints solving (from 5-min to 30-min). It shows that the reason Driller could not generate new test cases is not due to the limited time budget for solving the generated constraints.

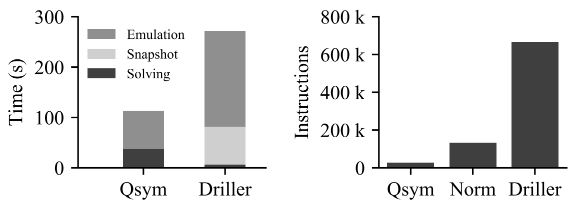

Figure 10: Average time breakdown of QSYM and Driller for 126 CGC binaries with initial PoVs as initial seed files, and the number of instructions that are executed symbolically. 'Norm' is the product of the number of instructions of QSYM and the average rate of increase of VEX IR, 4.69.

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	1,154 (0.01 %)	18,626,977
CROMU_00010	1,467 (0.18 %)	811,819
CROMU_00020	3,492 (11.15 %)	31,306
KPRCA_00013	4,589 (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

Table 5: The number of instructions in the CGC challenges that are not emulated due to the limitation of QSYM: no floating point operation supports.

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 10(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 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 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 this hypothesis, we measured the number of instructions that were not emulated by QSYM (Table 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

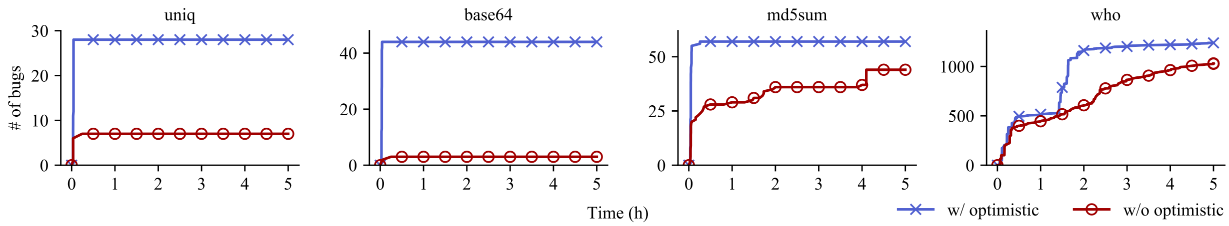

Figure 11: The cumulative number of bugs found in the LAVA dataset with or without optimistic solving by time.

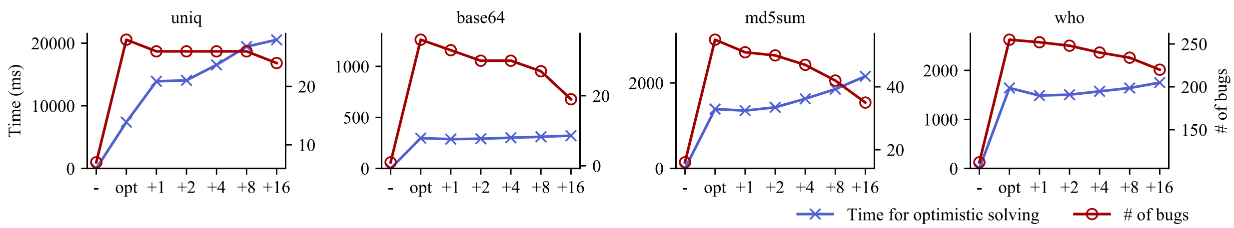

Figure 12: Time elapsed for optimistic solving and the number of unique bugs found in the LAVA dataset in a single execution of QSYM with an initial test case according to the number of constraints in optimistic solving. The minus symbol (–) represents the absence of optimistic solving; therefore, its 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.

	uniq	**base64**	**md5sum**	**who**
FUZZER	7 (25 %)	7 (16 %)	2 (4 %)	0 (0 %)
SES	0 (0 %)	9 (21 %)	0 (0 %)	18 (39 %)
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 %)	1,238 (58 %)
Total	28	44	57	2,136

Table 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.

symbolic execution.

5.4 Optimistic Solving

To evaluate the effect of optimistic solving, we compared QSYM with others using the LAVA dataset [10]. 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 [10]. 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 11 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 6). At first, we tested VUzzer [9] 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

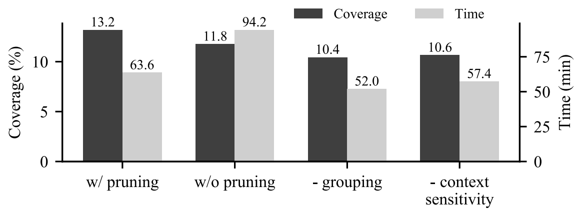

Figure 13: Total newly found coverage and elapsed time for libjpeg, libpng, libtiff, and file with five seed files, except for libjpeg, which has only four files, that have the largest code coverage in each project.

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 [9] due to its anonymized testing systems. In Table 6, FUZZER represents the results of a coverage-oriented fuzzer and SES represents the results of the symbolic execution. QSYM found 14× 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 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 12. 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.

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 13 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 13 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.

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.

6.1 ffmpeg

Figure 14 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.

6.2 file

Figure 15 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 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

```
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 14: 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 15: The file bug that QSYM found. The check for `descsz` is always true due to the incorrect use of logical OR operator.

that a concurrent bug report [27] detected this bug using a static analysis tool `cppcheck` [32].

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) 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 [33], 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 §3.2) 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 [9] and AFLFast [34], in this paper, 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 [13, 35] 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.

8 Related Work

8.1 Coverage-Guided Fuzzing

Coverage-guided fuzzing becomes 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 [36] provides accurate coverage information to mitigate path collisions.

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, VUzzer [9] develops application-aware mutation techniques by performing static and dynamic program analysis. Steelix [37] recovers correct magic values by collecting comparison progress information during program execution. FairFuzz [38] discovers magic values and prevents their mutations with program analysis and heuristics. Angora [39] adopts taint tracking, shape and type inference, and a gradient-descent-based search strategy to solve path constraints efficiently. These approaches, however, can only handle certain types of constraints. In contrast, QSYM relies on symbolic execution such that it has a chance to satisfy any kinds of constraints. In addition, a recent study, T-Fuzz [40], transforms a program itself to cover more interesting code paths, which could be combined with QSYM to remove unsolvable constraints from the program.

8.2 Concolic Execution

Concolic execution is a path-exploring technique that performs symbolic execution along a concrete execution path to direct the program to new execution paths. Concolic execution has been largely adopted for automatic vulnerability finding from source code [19, 41, 42] to binary [4, 5, 20, 21, 43].

However, concolic execution suffers from the path explosion problem in which the number of paths to explore grows exponentially with a program size. To mitigate this problem, SAGE [4, 44] proposes generational search to maximize the number of test cases in one execution and applies unrelated constraint solving [45]. Dowser [46] uses static analysis and taint analysis to guide concolic execution and minimizes the number of symbolic expressions to find buffer overflow vulnerabilities. Mayhem [21] combines forking-based symbolic execution and re-execution-based symbolic execution to balance performance and memory usage. In contrast, QSYM uses (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, and (3) instruction-level re-execution-based symbolic execution for better performance.

8.3 Hybrid Fuzzing

The concept of hybrid fuzzing is 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.

9 Conclusion

This paper presented QSYM, a fast concolic execution engine tailored to support hybrid fuzzers. QSYM makes hybrid fuzzing scalable enough to test complex, real-world applications. 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.

10 Acknowledgments

We thank the anonymous reviewers, and our shepherd, Mathias Payer, for their helpful feedback. This research was supported in part by NSF, under awards CNS-1563848, CRI-1629851, CNS-1704701, and CNS-1749711, ONR under grants N000141512162 and N000141712895, DARPA TC (No. DARPA FA8650-15-C-7556), NRF-2017R1A6A3A03002506, ETRI IITP/KEIT [2014-0-00035], and gifts from Facebook, Mozilla, and Intel.

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 soft-

ware 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] 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.–Mar. 2017.

[10] 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.

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

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

[13] 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.

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

[15] L. Project, "LLVM language reference manual," https://llvm.org/docs/LangRef.html#llvm-language-reference-manual, 2003.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] "CVE-2017-11543," https://cve.mitre.org/cgi-bin/

cvename.cgi?name=CVE-2017-11543.

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

[28] 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.

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

[30] DARPA, "Cyber Grand Challenge," https://www.cybergrandchallenge.com/, 2016.

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

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

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

[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] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: fuzzing by program transformation," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[41] 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.–Nov. 2006.

[42] 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.

[43] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[44] 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.

[45] 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.

[46] 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.