LeanSym: Efficient Hybrid Fuzzing Through Conservative Constraint Debloating

Xianya Mi*
Artificial Intelligence Research Center(AIRC)
Beijing, China
mixianya@126.com

Sanjay Rawat
University of Bristol
Bristol, The United Kingdom
sanjay.rawat@bristol.ac.uk

Cristiano Giuffrida
Herbert Bos
giuffrida@cs.vu.nl
herbertb@cs.vu.nl
Vrije Universiteit
Amsterdam, The Netherlands

ABSTRACT

To improve code coverage and flip complex program branches, hybrid fuzzers couple fuzzing with concolic execution. Despite its benefits, this strategy inherits the inherent slowness and memory bloat of concolic execution, due to path explosion and constraint solving. While path explosion has received much attention, *constraint bloat* (having to solve complex and unnecessary constraints) is much less studied.

In this paper, we present LeanSym (LSym), an efficient hybrid fuzzer. LSym focuses on optimizing the core concolic component of hybrid fuzzing by conservatively eliminating constraint bloat without sacrificing concolic execution soundness. The key idea is to partially symbolize the input and the program in order to remove unnecessary constraints accumulated during execution and significantly speed up the fuzzing process. In particular, we use taint analysis to identify the bytes that may influence the branches that we want to flip and symbolize only those bytes to minimize the constraints to collect. Furthermore, we eliminate non-trivial constraints introduced by environment modelling for system I/O. This is done by targeting the concolic analysis solely to library function-level tracing.

We show this simple approach is effective and can be implemented in a modular fashion on top of off-the-shelf binary analysis tools. In particular, with only 1k LOC to implement simple branch/seed selection policies for hybrid fuzzing on top of unmodified Triton, libdft, and AFL, LSym outperforms state-of-the-art hybrid fuzzers with much less memory bloat, including those with advanced branch/seed selection policies or heavily optimized concolic execution engines such as QSYM and derivatives. On average, LSym outperforms QSYM by **7.61%** in coverage, while finding bugs **4.79x** faster in 18 applications of Google Fuzzer Test Suite. In real-world application testing, LSym reported **17** new bugs in **5** applications.

*At the time of work, the author was associated with VUSec and National University of Defense Technology, China

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RAID '21, October 6–8, 2021, San Sebastian, Spain
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9058-3/21/10…$15.00
https://doi.org/10.1145/3471621.3471852

CCS CONCEPTS

• **Security and privacy** → **Software and application security**; **Vulnerability scanners**.

KEYWORDS

hybrid fuzzing, concolic execution, taint analysis

ACM Reference Format:
Xianya Mi, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2021. LeanSym: Efficient Hybrid Fuzzing Through Conservative Constraint Debloating. In *24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21), October 6–8, 2021, San Sebastian, Spain.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3471621.3471852

1 INTRODUCTION

To improve code coverage, state-of-the-art hybrid fuzzers combine a fuzzing component with a concolic (symbolic+concrete) execution component. The hybrid fuzzer relies on the former to still generate the bulk of the inputs, but falls back to the latter to generate inputs that would be hard to otherwise consider due to the complex *constraints* that the input bytes must satisfy. In program terms, these constraints correspond to the branch conditions that govern which values in the input lead to new code coverage. The harder the constraints to be solved for a given target branch, the more difficult it is for a fuzzer to generate the right inputs using (semi-random) mutation. Nonetheless, solving complex constraints is possible with concolic execution. As a result, hybrid fuzzers rely on the speed of fuzzing for the simple cases and the slower, but more powerful, concolic execution for the difficult ones [19, 41, 45].

In theory, concolic execution can greatly improve the fuzzing process and perform sound and complete analysis, by simultaneously exploring multiple paths in the target program. However, in practice, concolic execution scales poorly to non-trivial applications and test cases, mainly due to path explosion and *constraint bloat* (having to solve many complicated and unnecessary constraints) [44]. While path explosion has been extensively studied in the past, this paper focuses on reducing the large number of constraints that overwhelm the solver's computational and memory capacities—a problem we refer to as *constraint bloat*.

The main, and unfortunately limited, answer to the scalability problems in symbolic/concolic execution has been to reduce the number of paths (and hence also constraints) through different variants of optimized concolic execution [12, 17, 18]. In particular, prior research has explored different ways to minimize the parts of the program to execute symbolically, by means of type dependence [4], neural networks [39], slicing and chopping [42], etc. However, after

deciding which part of the program is symbolic and which concrete, existing engines typically collect all the constraints during concolic execution for each target branch by making the entire input symbolic. However, in practice, many of the constraints thus accumulated are *not relevant for a given target branch* at all [20].

An alternative answer to deal with constraint bloat might be to write a tailored concolic execution engine for hybrid fuzzing from scratch, with the aim of symbolizing only the relevant instructions in the program. This is the approach explored in QSYM [45], where a custom and efficient concolic execution engine implemented in some 16,000 lines of code supports a hybrid fuzzer, using a dependency chain to determine the instructions that are relevant for each target branch and therefore should be executed symbolically. Although the optimized concolic component improves fuzzing performance significantly, the input is still fully symbolized, which may result in path explosion and constraint bloat. To work around these issues, QSYM uses a timeout to switch to an optimized mode which drops all the constraints and only attempts to solve those directly related to a given branch. Unfortunately, this strategy is unsound and may result in invalid inputs that do not help the fuzzer make forward progress.

In this paper, we present LeanSym (LSym), a new and efficient hybrid fuzzer that outperforms even tailored solutions with much less memory bloat and using off-the-shelf binary analysis tools. The key idea is to use partial symbolization (symbolizing only those parts of the inputs and program that are relevant for the target branch) to aggressively eliminate constraint bloat, while preserving the soundness of concolic execution. In particular, LSym removes constraints accumulated during execution that have no bearing on the target branch to speed up the fuzzing process significantly in two ways. First, it eliminates non-trivial constraints introduced by environment modeling for system I/O by targeting the symbolic analysis solely to library function-level tracing. Second, it uses dynamic taint analysis to identify the bytes that may influence the branches that we want to flip and symbolize only those bytes to minimize the constraints to collect.

We implemented LSym on top of unmodified Triton [38], libdft [3], and AFL [46] in 1k LOC to implement simple and modular branch/seed selection policies for hybrid fuzzing. We evaluated LSym on various benchmark datasets such as the Google Fuzzer Test Suite [2] as well as 5 real-world applications and compared its performance against state-of-the-art hybrid fuzzers, namely QSYM. On average, LSym outperforms QSYM by **7.61%** in coverage, while finding bugs **4.79x** faster. LSym also found **17** new bugs in **5** applications with the latest versions, while QSYM failed to find 9 of them.

Summarizing, we make the following contributions:

- We analyze the performance bottlenecks in concolic execution as used in hybrid fuzzers and identify constraint bloat as one of the main problems (§ 2).
- We show that we can significantly reduce the bloat and improve hybrid fuzzing performance by means of function-level training (§ 4) and taint-assisted input symbolization (§ 5). .
- Building on these insights, we implemented LSym, a hybrid fuzzer on top of common off-the-shelf tools (§ 6) that outperforms highly optimized state-of-the-art solutions (§ 8)

To foster follow-up research, we will open source our LSym prototype upon acceptance of the paper.

2 CONSTRAINT BLOAT IN CONCOLIC EXECUTION

Though path explosion is a well known and studied issue in classical symbolic execution and concolic execution based approaches, scalability still remains a challenge and constraint solving is possibly the next biggest bottleneck [5]. Constraint solving is especially expensive due to constraint bloat, which, in turn, has two main causes. First, the concolic execution engines by themselves do not really know what happens to the symbolic state when the program interacts with the environment, for instance through system calls. For this reason, today's concolic execution engines come equipped with *models* to emulate these effects on the symbolic state for every possible interaction. Unfortunately, as we shall see, symbolization at the system call level is complicated and quickly leads to the accumulation of many (not very interesting) constraints. Second, existing concolic execution engines apply (overly) wide symbolization of inputs and instructions with an eye on (broad) program exploration rather than flipping target branches.

2.1 Relevant branches

To highlight the problem, we use the example shown in Listing 1, where the vulnerability in Line 24 is hidden inside a number of complicated checks.

Listing 1: A bug in nested code. Branches marked 'R_n' are related to the bug in line 24. The 'U_n' branches are unrelated.

```
1
2  int main (int argc, char *argv[]) {
3    char *file = argv[1];
4    char B[2000]; // buffer for input bytes (to be tainted)
5    FILE *fp = fopen(file, "r");
6    size_t fsize = ftell (fp); // get file size
7    int check = 0;
8    fread(B, fsize, 1, (FILE*)fp);
9    for (i = 0; i < (fsize-6); i++)
10       if ((B[i]+B[i+1]) < (B[i+3]+B[i+4]+B[i+5])) // R0
11           check++;
12
13   if (check < 1) return -1;
14
15   if (B[18] + B[19] == 'b') { // R1
16       ...
17       if (B[2] + B[4] == 'X') // U1
18           ...
19       if (B[15] + B[18] == 'U') { // R2
20           ...
21           if (B[4] + B[8] == 'X') // U2
22               ...
23           if (B[15] + B[14] == 'g') //R3
24               ...VULNERABLE CODE...
25           else ... // R4
26       }
27       ...
28   }
29   return 0;
30 }
```

To trigger the vulnerability, the program should reach the `true` edge of the branch R3, after satisfying three prior branch conditions (R0, R1 and R2). Note that the other conditional statements preceding R3 (U1 -- U2) are not important for the outcome of R3. Specifically, a previously executed branch A is only relevant for a later branch B, if B is *control-* or *data-dependent* on A.

Listing 2: Constraints generated for the code in Listing 1 on negating the branch R3.

```
1
2 (B[0]+B[1])-(B[3]+B[4]+B[5] < 0) &&
3 ...
4 B[18] + B[19] - 'b' = 0 && B[2] + B[4] - 'X' = 0 &&
5 B[15] + B[18] - 'U' = 0 && B[4] + B[8] - 'X' = 0 &&
6 B[15] + B[14] - 'g' = 0
```

To explore such code, concolic execution maintains both symbolic and concrete states. Upon encountering a branch during execution, it will try to find a new path by solving the negation of the branch together with all the constraints previously collected. In this simple example, if an input reaches R3 (i.e. satisfies all the previous constraints), but sets the branch outcome to `false`, it will try to find an input that evaluates `R3` to `true`. To do so, popular concolic execution solutions such as KLEE, S2E, and Triton[1] will generate symbolic expressions similar to the ones shown in Listing 2, to be solved by a constraint solver.

Clearly, many of the constraints are not relevant for triggering the bug at all and satisfying them with a constraint solver needlessly consumes both time and memory.

2.2 Existing work on reducing symbolic code

Existing solutions attempt to symbolize only relevant statements build on a technique known as *chaining* [16], a form of slicing using program dependency graphs [21]. Specifically, by building a dependency tree, solutions such as QSYM [45], DGSE [43], and STIG [14] determine which statements in a program are relevant for the current target branch.

QSYM [45], a state-of-the-art hybrid fuzzer optimizes the process even further by dynamically switching between two modes. In *full* mode, QSYM builds for each branch a full dependency tree consisting of all relevant statements for that branch and tries to solve (only) the corresponding constraints. However, in case full mode takes too long (i.e., QSYM cannot solve the constraints within a few seconds), it falls back to *optimistic mode* which uses taint analysis to find which input bytes influence the target branch and solves *only* that constraint. We noticed that Triton [38] supports a similar mode, known as *last-branch-only*, although it does not make use of taint analysis to prune the statements that are not relevant.

The code in Listing 1 shows that solutions that leverage dependency trees to reduce constraints (such as QSYM and similar systems), may still accumulate many (useless) constraints. In this case, the loop even creates a chain of constraints so that each constraint c_{i+1} at loop iteration $i + 1$ is related to the constraint c_i at loop iteration i. As a result, we end up collecting all the constraint in the code except for the ones related to U1 and U2. In optimistic mode, QSYM does not use dependency relation and therefore, only focuses on the last constraint `B[15] + B[14] - 'g'= 0`.

2.3 Challenges in containing constraint bloat

For small examples, off-the-shelf constraint solvers may well find a solution still, but for real-world applications with large program sizes and numerous paths, the constraints to flip a target branch quickly become too complicated for the constraint solver to solve efficiently. In the remainder of this section, we revisit the two main challenges in handling constraint bloat.

Challenge C1 Reduce constraint bloat due to excessive symbolization by hooking at the level of system calls.

Concolic execution always starts with symbolizing the program input. In the absence of source code, it is often not obvious which exact function acts as the source of the tainted input. For this reason, binary-only solutions typically symbolize input bytes by hooking system calls for file operations, such as `open()`, `read()`, and `mmap`. While convenient, system call hooking introduces additional constraints which in turn lead to additional overhead for the solver.

The main sources of these extra constraints are wrapper functions such as the system call APIs in `glibc` that perform a variety of operations before performing the actual system call. As an example, consider the reading of a string of bytes from a file using the `fopen()` and `fread()` functions from the `glibc` library that eventually execute the `open()` and `read()` system calls, respectively.

To get access to the target string from the input file, `fopen()` needs to copy the file descriptor returned by the `open()` syscall to the address of the I/O buffer that implements the `FILE` struct. Next, `fread()` uses a pointer in this structure to track which offset in the input file buffer should be accessed and copied into an intermediate buffer. These are all complex operations. With symbolization at the system call level, such intermediate operations increase both the emulation time and the number of constraints. As the input file grows in size, these additional mass of (not very interesting) constraints influences the efficiency of concolic execution to certain extent.

Challenge C2 Reduce constraint bloat due to whole-input symbolization.

Current concolic execution solutions symbolize the entire input and emulate all instructions that operate on these symbolized bytes. As we saw in Listing 2, doing so results in many constraints that are not relevant. As we saw earlier, advanced solutions (in QSYM and similar systems) reduce constraint bloat by computing a precise dependency chain, allowing them to emulate only the relevant instructions. For the example in Listing 1, QSYM in full mode would keep all the constraints related to `R0 - R3`, but shed (only) those of U1 and U2. In optimistic mode, QSYM's fallback solution, it abandons the soundness of symbolic execution altogether and tracks only `R3`.

Both modes have problems. It is intuitively clear that optimisitic mode often leads to invalid solutions. For instance, when solving the constraint for `R3` using bytes 15 and 14, it may pick a value for byte 15 that interferes with `R2` and cause the program to not even reach `R3`. Clearly, a single constraint on the last branch is not sufficient. Unfortunately, the complete dependency chain in full mode *also* leads to problems—in the form of constraint bloat (and its accompanying high memory overhead). Specifically, given an input and the target branch `R3` that the fuzzer needs to flip, the relevant constraints are only those related to R2 and R3. Picking the right values for bytes 14 and 15, such that they satisfy both conditions is sufficient to flip the branch. In other words, using the full dependency graph may vastly overshoot the minimum number of constraints needed to flip the branch!

[1] default mode

Table 1: Constraints collected by different methods for the code in Listing 1, for the target branch RB3.

	time	mem (MB)	solving time	constraints	solved vars	success
QSYM*	14s	538	10s +284 ms	951	949	no
Triton*	28s	206	10s + 876 ms	951	949	no
LSym	8s	57	1 ms	9	2	yes

*QSYM and Triton both had to fall back on optimistic/last-branch-only mode as neither could solve the constraints otherwise, even in 20 minutes.

2.4 The cost of constraint bloat

In this section, we perform a preliminary evaluation of the effects of constraint bloat to motivate the remainder of this work. As a first example, consider Table 1, where the first two rows show results for the trivial example of Listing 1 with QSYM and an optimized version of Triton. Like QSYM, the Triton-solution drops to last-branch-only mode if it fails to find a solution with full constraints within a time-out period of ten seconds. Without the timeout, both QSYM and Triton can not solve the target branch even in twenty minutes. We then aborted these experiments, since spending this long on a single branch makes these solutions impractical. The experiments show that constraint bloat can easily becomes a bottleneck for constraint solvers. Indeed, even for this simple example, the number of constraints (almost one thousand) and the amount of memory (hundreds of megabytes) are very high for both QSYM- and Triton-based solutions, while neither of them even found an input to reach the target code. In this paper, we show that we can do much better. For instance, the last row shows that LSym reduces the number of constraints by two orders of magnitude, and easily finds an input that leads to the vulnerability in just 1 ms of constraint solving.

While Listing 1 is a toy example, the problem of constraint bloat also surfaces in real-world binaries, as shown in Table 2, where we evaluated `objdump-2.34` targeting one specific compare instruction using QSYM and the same optimized version of Triton, as well as LSym. Again, we see that the amount of memory and the number of constraints in the QSYM and Triton-based solutions are high, and because of their complexity, they take a long time to solve. Indeed, although our seed inputs hit the target branch six times, in every single case the constraints were too complex for Triton to solve within the time-out period. In other words, all branches go to optimistic/last-branch-only mode for a "solution" (that unfortunately often invalidates prior constraints). Similarly, QSYM resorted to optimistic mode in half of the cases, even though QSYM's concolic execution engine is much more efficient than Triton's. The bottom row in the table again shows that we can do better with only a fraction of the constraints, less memory overhead and a much lower solving time. Moreover, the reduction in constraints means that they could all be solved in less than a second without the need for the fallback mode. This is important since optimistic mode has strong bearings on the code that we cover. For instance, Triton generated even less coverage than the original seed input. The main reason is that optimistic/last-branch-only modes ignore previous data dependencies so that the solutions more often than not invalidate previous constraints. A quick investigation of newly covered paths via optimistic mode on applications in Google Fuzzer Test Suite confirms that many of them are error paths. In other words, the more inputs we generate optimistically, the less (meaningful) coverage we may get (see Section 8.1).

Table 2: Constraints collected by different methods for a single branch in objdump-2.34.

	mem (MB)	solving time*	constraints	solved vars	coverage
original					1384
QSYM	662	10s+0.491s	407	38	1582
Triton	673	10s+13ms	371	8	1225
LSym	206	0.289s	33	8	1592

*for the seed hitting the target branch last

3 LEANSYM

LSym addresses the issue of constraint bloat by augmenting concolic execution with two techniques.

Overcoming C1: Function level tracing. Instead of symbolizing at the syscall level, LSym performs symbolization at function level (i.e. it delays hooking until the I/O related execution reaches a library function). For instance, in the example of Listing 1, instead of hooking syscalls `open()` and `read()`, we start tracing from `fopen()`, `fread()` from `libc`. The similar approach has also been used in prior work in symbolic execution (e.g., by creating efficient memory models and concretely applying system and application arguments, e.g. S2E[12], Mayhem[8], angr [40] and KLEE [6] for external library wrapper) and unit-level testing work [36]. It should, however, be noted that while source-based solutions, like KLEE, also apply library wrappers, their motivation is to model the external library functions to continue with the execution. We, on the other hand, apply similar strategy for constraint debloating. As an addition, we simplify the automatic detection of functions calling I/O syscalls as such functions can be custom functions in the application rather than standard libc APIs. In Section 4, we explain how we identify the appropriate functions to start tracing.

Overcoming C2: Taint-assisted partial symbolization. LSym exploits the fact that concolic execution involves a concrete input and uses it to also compute the (dynamic) dataflow. Specifically, rather than constructing a full dataflow-based dependency chain, LSym uses dynamic taint analysis (DTA) to identify the tainted (input) bytes that affect each statement. For each target branch A and the corresponding set T_A of input bytes affecting it, it symbolically executes only those statements that are similarly affected by bytes in T_A. We achieve this by symbolizing (only) the input bytes in T_A and perform concolic execution as usual, without touching other bytes of the input.

Thus, for the example in Listing 1 we would not change the behavior of branch R1 and focus instead on R2 and R3, as well as the constraints generated within the loop that involve those same tainted bytes (14 and 15).

The bottom row of Table 1 shows that, for this example, doing so reduces the number of constraints by two orders of magnitude and easily finds an input that leads to the vulnerability. In Section 8, we evaluate LSym on real-world applications.

Fig. 1 illustrates LSym's high-level design. It consists of four main components: (A) the main fuzzer (e.g., AFL [46]), (B) the taint analysis engine (e.g., libdft [25]), (C) the branch selector, and

(D) a concolic execution engine (e.g., Triton [38]). The main fuzzer maintains a queue of inputs from which we randomly select one and run it under taint analysis to determine the tainted bytes for every branch. Next, the branch selection module selects branches to flip (see Section 6), while the concolic execution engine uses the taint information and function level tracing to generate a minimal set of constraints corresponding to the selected branches. Finally, a SAT solver returns a solution satisfying these constraints (if any), which is then used to generate new inputs.

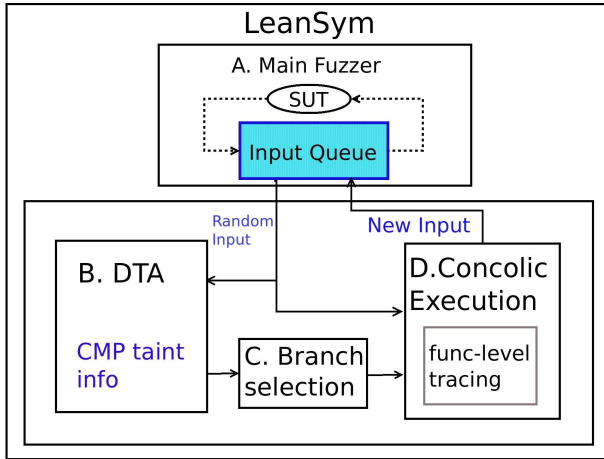


Figure 1: Architectural Overview of LSym.

In the next two sections, we describe function-level symbolization and taint-assisted partial symbolization in detail.

4 FUNCTION-LEVEL TRACING

Library function wrappers for system calls can introduce additional and unnecessary constraints while performing concolic execution. To address this problem, we propose a simple function-level tracing and symbolization design as an alternative to the traditional syscall-level approach. Our approach is inspired from source-code based solutions, like KLEE [6], that provide wrappers for external library functions and works directly on executables in automatic manner. For our purpose, we only focus on file processing syscalls and library functions. Achieving this goal requires two steps: (1) determining which file processing library functions to hook, and (2) summarizing such functions to perform symbolization.

From functions to syscalls. Without access to source code, our goal is to track syscalls like `open()`, `read()`, etc. and then determine which library functions wrapping these system calls we need to symbolize. For this purpose, we rely on the Intel PIN [30]'s dynamic binary instrumentation (DBI) framework.

We hook syscalls by using PIN's dynamic instrumentation. Next, we use PIN's routine tracing instrumentation to generate a *dynamic callgraph* where the deepest node (leaf node) contains a call to a syscall and upper most node corresponds to a known library function. In our current implementation, we focus on file processing syscalls and corresponding library functions from `libc`. Before starting the main analysis task, we run the application on given seed inputs to determine which function summaries are required for the application. For instance, when we run our pintool on the example in Listing 1, we find `fopen()` and `fread()` call the relevant syscalls and are thus selected for function-level symbolization. We also considered alternative designs, such as manual annotations at the library interface level (but this approach may miss relevant library functions across implementations) and static callgraph analysis (but this approach may be unnecessarily conservative for the simple task at hand), but discarded them in favor of a more practical approach.

Function summaries. After determining the target library functions, we provide wrappers that summarize such functions and symbolize the relevant data by doing a manual analysis. This is similar to the approach adopted in **angr** [40] for providing python wrappers for library functions. However, a distinction should be made in that angr provides summaries of known library functions (called as *SimProcedures*) to deal with the problem of path explosion, which we have not implemented in our current prototype. We believe incorporating such features will further enhance constraint debloating. Also, unlike angr, we do not model the whole effect of a syscall on the memory, but rather only focus on the taintflow propagation. For most of the functions, this is as simple as grabbing the address of a file buffer from argument registers and symbolizing all the bytes in the buffer. For instance, the `fread()` wrapper can, in principle, orderly symbolize the bytes of the read buffer upon library function return. Nonetheless, syscalls such as `rewind()`, `fstat()`, `lseek()`, and `fseek()` may change the current read offset and result in a non-linear input-to-buffer mapping.

To address this problem, we track information in the file structure stored in memory. In particular, we target variables (e.g., `_IO_read_ptr`) in the file structure to calculate the right offset at every read operation. In the wrapper, we maintain a map between each symbolic byte in the read buffer and the original input byte offset. This strategy allows us to map each byte of the input to each byte read by a given libray call after collecting and solving the branch constraints.

Thanks to this simple summarization strategy, we can aggressively prune all the unnecessary constraints normally collected inside complex library function implementations like `fread()` by existing concolic execution tools.

5 TAINT-ASSISTED PARTIAL SYMBOLIZATION

In this section, we describe the main LSym component, namely taint-assisted partial symbolization. As we observed earlier, the efficiency of concolic execution depends on two factors: the time to emulate instructions and collect constraints, and the time to solve the collected constraints for a given branch that we want to flip. While optimizations are possible [16, 45], it is difficult to minimize such times if we fully symbolize the input and need to collect the many resulting constraints during concolic execution.

LSym's key intuition is that, in a coverage-oriented hybrid fuzzing scenario, we are only interested in targeting conditional branches that depend on *some* bytes of the input. For example, as shown in Listing 1, for the branch at line 23 to be negated and reach the vulnerable code R3, we only need to focus on offsets 14 and 15 in the input affecting the branch (assuming previous constraints are satisfied). As a result, full input symbolization is unnecessary (and, in fact, detrimental due to constraint bloat).

Building on this intuition, LSym only symbolizes the input bytes that affect the target branches, pruning many complex and unnecessary constraints while retaining the same context (and soundness) as full symbolization. For this purpose, LSym relies on byte-granular dynamic taint analysis (DTA) to determine which input bytes affect which target branches in the program. With this information, LSym can symbolize only the relevant (or *tainting*) input bytes for partial and efficient symbolization.

To gather the required taint information, LSym simply runs the program with the given input using DTA. LSym instructs the DTA engine to use the entire (file) input as taint source and all the compare (i.e., `cmp`) instructions as taint sinks. With byte-granular taint propagation performed by the DTA engine, this strategy allows us to record tainted input byte offsets and operand values for each `cmp` instruction controlling a branch.

In the next step, LSym selects the target `cmp` instructions and symbolizes only the input bytes tainting the operands of such instructions. For example, in Listing 1, say we want to exclusively target (and flip) the branch at line 23. To this end, LSym collects taint information for the corresponding `cmp` instruction and reports bytes 14 and 15 as the only controlling input bytes.

In the final step, LSym symbolizes the input bytes reported by DTA and collects constraints during concolic execution. Due to taint-assisted partial symbolization, the latter only needs to emulate instructions operating on the relevant symbolic data. When the execution reaches the target branch instruction, we have gathered all the necessary constraints to negate the branch. In the example, the condition is negated as `B[15] + B[14] - 'g' == 0`.

At that point, the collected constraints are given to a constraint solver, which calculates the values of the relevant input bytes to guide the execution on the flipped edge of the target branch. In the example, the solver provides the following solution: `B[14] == 0x33 && B[15] == 0x34`. LSym uses such solution to *replace* the original input bytes with the provided values and generate a new test case.

Our DTA implementation does not consider *implicit (indirect) taintflow* when propagating taint[2]. As noted by Cavallaro *et al.* [7], capturing implicit flows result in taint explosion, thus more false positives, while not considering implicit flows introduces missing taintflows. Due to the later side-effect, the precision of constraints is negatively impacted. For us, this problem results in losing *relevant* constraints thereby making constraint solving incorrect. We, however, observe that for fuzzing on real-world applications, the effect of implicit flows is not very adverse (see Section 8.3.1 for empirical results).

6 HYBRID FUZZING POLICIES

Building on lean concolic execution for hybrid fuzzing, LSym can afford simple fuzzing policies to implement efficient and practical fuzzing. At the architectural level, we follow the standard approach used by prior hybrid fuzzing solutions such as QSYM. That is, we use AFL [46] as the main fuzzing engine with two instances (master and worker) to synchronize with the inputs generated by our concolic execution component. Within this architecture, LSym relies on basic seed selection, branch selection, and branch scheduling policies to isolate the benefits of constraint debloating and improve the comparability of our results.

Seed selection. The seed selection policy dictates how LSym selects test cases from the worker instance of AFL to perform concolic execution—feeding the descendant test cases back to AFL. LSym can flexibly support different seed selection policies. For a fair comparison against QSYM, the closest competing solution, we adopt its same seed selection policies based on four metrics: coverage (seed leading to new coverage), importance (original seed), size (the smaller the seed size, the better), age (the younger the seed, the better). Nonetheless, using more advanced seed selection policies—such as those explored in recent work [10, 11, 47]—to provide orthogonal improvements is possible.

Branch selection. The branch selection policy dictates how LSym selects the branches to flip within each seed considered for concolic execution. LSym can flexibly support different branch selection policies. For example, Burnim and Sen observed that in practice, a simple *random branch selection* strategy beats more sophisticated strategies, like CFG based or DFS [5]. However, in our current implementation, similar to QSYM we select all the branches in the execution trace of an input under consideration. Nonetheless, using more advanced branch selection policies is possible and has the potential to amplify the effectiveness of partial input symbolization (the fewer the branches selected, the fewer the constraints handled by concolic execution). We leave the exploration of such large design space to future work.

Branch scheduling. The branch scheduling policy dictates how LSym schedules the selected branches for concolic execution. A naive policy would simply schedule all the target branches for partial input symbolization in a single concolic execution run. However, this policy may trivially re-introduce constraint bloat. In fact, in the worst case, it might lead DTA to conclude we need to symbolize the entire input, falling back to an inefficient full input symbolization baseline. At the other extreme, scheduling one concolic execution run for each branch would introduce unnecessary computational redundancy (and overhead). To minimize constraint bloat and computational overhead, our current LSym prototype clusters the branches controlled (i.e., tainted) by the same input bytes together and schedules each cluster in a single concolic execution run. We found this policy to work well in practice and exploits the full benefit of partial input symbolization.

7 IMPLEMENTATION

We implemented our LSym prototype in 1,126 lines of code (LOC) overall. The concolic execution component is implemented in 641 LOC and based on the Triton [38] concolic execution engine. The DTA tool is implemented in 371 LOC and based on the libdft64 [3] DTA framework. The remaining 114 LOC are for the fuzzer runner.

Concolic execution. In theory, LSym can be implemented on top of any existing concolic execution engine, such as S2E [12], KLEE [6], angr [40], Triton [38], and QSYM [45]. We originally considered QSYM for comparability purposes, its support for program binaries, and its efficient concolic emulation. However, QSYM's custom-optimized concolic execution engine implements several

[2]Implicit flow arises when a variable is assigned within an *if-then-else* statement whose condition involves a sensitive (tainted) variable, e.g., `if(y=1) then x:= 1;else x:= 0;endif`. Clearly, the value of X is dependent on , even though there is no assignment of the latter to the former.

tightly coupled optimization techniques, making it difficult to extend, decouple the optimizations, and implement our modular LSym design. As a result, we decided to implement LSym on top of the less efficient but more flexible Triton engine.

Triton [38] is a dynamic binary analysis framework that provides a concolic execution engine, AST representations, DTA, and a SMT solver interface. Triton's concolic execution engine is implemented by using Intel PIN [30] API which also facilitates the integration of our function-level/syscall-level profiling tools. Our concolic execution component is implemented on top of the Triton API to easily support partial symbolization (i.e., specifying which bytes we want to symbolize) and function-level tracing (i.e., hooking into the target library functions and perform just-in-time symbolization based on our function summaries). In our implementation, we configured LSym with the same optimistic/last-branch-only mode that kicks after the 10 seconds timeout period (similar to QSYM).

Dynamic taint analysis. In theory, we could implement our LSym design on top of Triton's DTA engine. However, Triton only supports single-label DTA, that is we can only check if a sink value is tainted (or not) with no data lineage information. Our design instead requires one label for each input byte to map input offsets to affected (tainted) branches. A way to approximate this design with Triton is to run the program through DTA many times and, at each run, taint a different input byte offset. Since this is inefficient, we instead opted for libdft64 [3], an Intel PIN-based DTA framework that supports multiple labels at the byte granularity. Generating the branch instruction taintflow is based on the VUzzer's taintflow implementation [37].

8 EVALUATION

We evaluated our LSym prototype on a workstation running Ubuntu-16.04 on Intel(R) Xeon(R) CPU E5-4650 v4 with 64 GB RAM. We used *gcov* to collect coverage information. Hereafter, with *coverage* we mean number of *lines of source code* executed (unless otherwise mentioned). We also use the *opt* and *no-opt* tool name suffixes to denote *optimistic* and *non-optimistic* configurations of the tools considered (recall that in *optimistic* mode we solve only the target branch constraints as a fallback strategy upon timeout, rather than giving up on the branch).

Our evaluation focuses on answering the following questions:

- **RQ1:** Does LSym improve (valid) input generation when compared to full data dependency-based concolic strategies like QSYM's?
- **RQ2:** Does LSym incur more overhead in terms of memory consumption when compared to full data dependency-based concolic strategies like QSYM's?
- **RQ3:** Does LSym improve hybrid fuzzing with limited time and computing resources?

8.1 The cost of optimistic solving

In this section, we show that optimistic solving (dropping all the constraints except for the target branch), as is used in QSYM, is an overly aggressive constraint debloating strategy that may lead to generating *uninteresting* inputs. In particular, we empirically show that optimistic mode: (1) often invalidates the previous constraints, thereby affecting coverage in unpredictable ways; (2) may generate invalid inputs, thereby taking the program into an error state (i.e., trivial error-handling code). Moreover, we show that such error-handling code may contribute substantially to the overall reported coverage, but providing improvements that are not reflected in the ability to find more bugs. Previous works (e.g. TFuzz [32] and VUzzer [37]) have adopted strategies to specifically avoid error-handling blocks. For empirical findings, we use three real-world applications– libpng-1.2.56, pcre2-10.00 and objdump-2.34.

For `libpng-1.2.56`, we use a 218-byte size PNG seed input and test with QSYM, Triton, and LSym. We configure each tool to negate all the branches in the seed execution trace and measure the updated coverage.

As shown in Table 3, the total coverage of all inputs generated by QSYM is 1,005 and 871 for QSYM-no-opt. This means that optimistic mode accounts for 134 out of a total of 1,005 new inputs. An inspection of the source code reveals that 94 out of the 134 (~70%) newly covered code lines by optimistic solving actually belong to error-handling code. (The number -94 is calculated by substracting 88 from 182, indicating that 94 more error-handling code is introduced by optimistic mode.) The implication is that, while *no-opt* mode allows QSYM to never fail in generating inputs, the gain in code-coverage is often *accidental* in the sense that new input is not the *intended* one that flipped the target branch, but rather some random branch, leading the execution to error-handling blocks. We dub this as inferior input generation for executing error handling blocks was not intentional as opposed to existing work (e.g. [24]) that target error-handling blocks for bugs.

Table 3: Coverage of libpng-1.2.56 on a 218-byte seed input by QSYM, QSYM-no-opt and LSym.

	coverage	difference	error code	difference
QSYM	1005	\	182	\
QSYM-no-opt	871	-134	88	-94
LSym	991	-14(+109, -123)	95	-87

In fact, although the table suggests QSYM has more coverage than LSym (1005 vs. 991), manual code inspection reveals that the additional coverage is mainly due to executing error-handling code. Specifically, we manually analyze the line coverage and determine that QSYM has 14 more coverage in total than LSym and LSym generates 109 lines of coverage which QSYM fails to cover. Meanwhile, QSYM has 123 lines of coverage which LSym fails to cover, however, 87 out of 123 are new error handling code. (The number 87 is calculated by substracting 95 from 182.) This shows that LSym can generate more code coverage other than error-handling code than QSYM, and LSym can solve constraints that QSYM fails to solve correctly within timeout limit (thus falling back to optimistic mode).

We repeated the same experiment with pcre2-10.00 with a 36-byte input and the results are shown in Table 4. We observe that QSYM generates 45 more lines of error code than LSym, and 31 of which are caused by negating the branches incorrectly under optimistic mode (compared with the coverage generated by QSYM without optimistic solving, i.e. QSYM-no-opt). We also observe that LSym generates 28 lines of code regarding the target branches, however QSYM fails to achieve them. Instead, QSYM finds 289 extra

lines of code which includes error-handling code and code caused by negating non-target branches (a case of *accidental* code-coverage).

Table 4: Coverage of pcre2-10.00 on a 36-byte seed input by QSYM, QSYM-no-opt and LSym.

	coverage	difference	error code	difference
QSYM	1320	\	70	\
QSYM-no-opt	1209	-111	39	-31
LSym	1003	-317(+28, -289)	25	-45

We observed similar findings in the case of `objdump-2.34`. When we consider the `if-else` branch in the `bfd_elf_string_from_elf_section` function (*elf.c*), the branch is hit 6 times during the execution of the seed. When QSYM attempts to solve the 3 deepest runs of the branch, the solver resorts to optimistic solving. The 3 new inputs generated by optimistic solving never hit the target branch and branch off earlier in `bfd_elf_setup_sections` (again a case of *accidental* code-coverage). We also notice that the inputs generated by optimistic solving finally makes the program execution into error handling code, and in result, QSYM shows 17 more line coverage in `objdump.c` compared with LSym, which turns out to be error handling code. In other words, these extra coverage is not what we originally try to achieve and in result are useless.

Overall, our results answer **RQ1**, confirming LSym outperforms state-of-the-art solutions in valid input generation and produces higher-quality inputs overall.

8.2 The cost of constraint bloat

In this section, we show that inverse policy, collecting *all* constraints based on (data) dependency chaining, may severely impact performance on real-world applications. We do so by directly comparing LSym to QSYM, while noting that QSYM's concolic execution engine is much faster [33].

8.2.1 The cost of tracking all branches. Solutions such as QSYM try to solve *all* the branches found during the execution of the target application—including those from dynamic libraries (`libc.so.6`, `linux-vdso.so.1`, `/lib64/ld-linux-x86-64.so.2`, etc.). However, targetting such libraries has little or no *positive* effect on the coverage on the main application. LSym, on the other hand, is application-aware in the sense that it selects branches only from the main application[3].

In order to measure the effect on code-coverage and memory usage, we ran a series on exepriments on four applications– `libpng-1.2.56`, `libarchive-2017- 01-04`, `harfbuzz-1.3.2` and `boringssl-2016-01-12`. The results are summarised in Table 5. In the following, we elaborate our findings on `libpng-1.2.56`. We run QSYM and LSym on `libpng-1.2.56` with a single png seed file of 218 bytes. To cover each branch, QSYM takes **2 hours, 27 minutes and 12 seconds** on this single input, while LSym takes only **11 minutes and 14 seconds**. We also show the results when we continue running LSym to keep solving other branches from the newly generated inputs for the remaining time, completing 19 rounds on concolic execution and achieving better coverage than QSYM (Table 5).

QSYM produces 7031 inputs[4] and achieves 1005 lines of coverage. In the same time span, LSym generates inputs for the branches in 19 files, achieving 1145 line coverage, *despite a much slower concolic execution engine*. We also observe that out of 7031 inputs generated by QSYM, 1050 (14.93%) of them are generated in optimistic mode, which means QSYM could not solve the related branches in 10 seconds. In contrast, out of the 844 inputs generated by LSym, only 72 (9.68%) are solved by optimistic mode. These results show that the constraint debloating in LSym enables it to solve more complicated branches than QSYM. Another thing worth mentioning is that QSYM consumes much more memory than LSym, with 5437 megabytes at most in the concolic execution process for one input, while LSym consumes only 300 megabytes in one run, and at most 1005 megabytes for 19 files. We observe similar effects in three other applications, as shown in Table 5. In all of the cases, LSym is more efficient in solving constraints than QSYM as is evident from column *opt* in Table 5 (LSym does not fall to optimistic mode as often as QSYM).

Table 5: The result of running libpng-1.2.56, libarchive-2017-01-04, harfbuzz-1.3.2 and boringssl-2016-01-12 with LSym and QSYM for the same time duration, and also LSym for the same seed input.

prog	method	files	inputs	opt	time	mem(mb)	cov(loc)
libpng	**QSYM**	1	7031	1050	2:27:12	5438	1005
	LSym	19	844	72		1005	1145
	LSym	1	58	7	0:11:14	300	991
libarchive	**QSYM**	1	5178	764	0:23:21	15597	2244
	LSym	54	1065	0		440	2318
	LSym	1	431	0	0:07:03	440	1719
harfbuzz	**QSYM**	1	7819	849	0:22:12	2544	2891
	LSym	19	1917	8		469	3184
	LSym	1	745	2	0:01:43	469	2880
boringssl	**QSYM**	1	4737	353	1:28:33	6767	1321
	LSym	18	3572	235		1769	1345
	LSym	1	1165	38	0:27:38	1769	1295

8.2.2 The impact of bloat on input size . The practical implications of constraint bloating become even clearer if we relate it to input size. Intuitively, if the data-dependency chain used in QSYM and similar approaches gets bigger this will also affect the performance. As an illustration, we measure the memory overhead and execution time of QSYM, Triton and LSym on four programs from the Google Fuzzer Test Suite that have input intensive computation: *guetzli-2017-30*, *libarchive-2017-01-04*, *boringssl-2016-01-12* and *libpng-1.2.56*. As suggested by Klees *et al.* [27], we choose seed inputs of different sizes for this evaluation. However, our choice of maximum size is constrained by the fact that on inputs larger than 1 KB, QSYM could not finish within a three hour time limit. Figure 2 shows that the time and memory consumption for QSYM, Triton and LSym for growing input sizes diverge quickly and are only comparable for very short inputs. These are not isolated cases and we encountered many issues with input sizes during our experiments. For instance,

[3]For fuzzing a library, we statically compile it with the utility by using `--disable-shared` option.

[4]The inputs do not translate to branches, as QSYM frequently produces multiple inputs for a single branch.

Table 6: The number of UNSAT branches for LSym and QSYM on eight applications. This indicates that how many branches cannot be solved by the solver (Z3) based on the constraints collected by both methods.

libarchive	unsat	solved	percentage
LSym	7	390	1,76%
QSYM	81	169	32,40%
libxml2	unsat	solved	percentage
LSym	42	260	13,91%
QSYM	126	214	37,06%
pcre2	unsat	solved	percentage
LSym	48	664	6,74%
QSYM	894	520	63,22%
sqlite	unsat	solved	percentage
LSym	0	12	0,00%
QSYM	330	673	32,90%
boringssl	unsat	solved	percentage
LSym	1064	16984	5,90%
QSYM	4325	5737	42,98%
guetzli	unsat	solved	percentage
LSym	137	1800	7,07%
QSYM	2829	3734	43,11%
harfbuzz	unsat	solved	percentage
LSym	948	13343	6,63%
QSYM	5024	9207	35,30%
libpng	unsat	solved	percentage
LSym	90	1305	6,45%
QSYM	4411	7015	38,60%

when we ran `libpng-1.2.56`, with a png seed file of 1096 bytes, QSYM did not finish in 4.5 hours at which point it had already consumed 20GB of memory. In contrast, LSym finished in 3 hours with at most 3GB overhead. We conclude that with respect to **RQ2**, LSym incurs much less memory overhead compared to constraint-heavy solutions such as QSYM—even if it drops to optimistic mode after the ten seconds time-out.

8.2.3 The efficiency in solving complex branches. One of the main design criteria in QSYM is to tackle the problem of *over-constraints* which may result in unsatisfiable (UNSAT) constraints in generating new inputs [12]. LSym's taint based constraint collection also aims to address the problem over-constraints. In order to test the quality of collected constraints by LSym and QSYM, we analyse a set of applications to empirically observe the number of UNSAT branches in both of the approaches. We run these applications on seed inputs to record for how many branches, the solver returns UNSAT, thereby indicating the infeasibility of generating inputs that flip those branches. In case of UNSAT, QSYM falls back to optimistic mode to still generate inputs. We can see in Table 6 that LSym returns UNSAT branches at most 13.91% of the times, while QSYM returns UNSAT at least 32.40%. which shows that QSYM fails to solve more complex branches more often than LSym, which could be a side effect of over-constraints[5]. This number shows the ability of the concolic execution engine to solve complex constraints if the problem of over-constraints is addressed carefully and LSym is better at minimizing the problem of over-constraints.

To see the effect of these UNSAT behaviour on the input generation, we manually analyse a branch at `0x405af4` in `guetzli-2017-3-30` which is located in source code file `jpeg_data_reader.cc:965`. The branch is actually a switch command, and the target code lies in lines 1022 and 1023. In the original input, these two lines are not covered. Ideally, concolic execution should flip the branch and cover the lines 1022 and 1023. In QSYM normal mode, trying to flip this branch returned UNSAT by the solver, indicating that constraints were not solvable. As a result, QSYM falls back to optimistic mode, generating 7 optimistic new inputs. For the same branch, LSym generates 7 inputs without the solver returning UNSAT. By analysing the coverage introduced by both methods, we find out that LSym covers lines 1022 and 1023, while QSYM fails to do so in spite of generating inputs with optimistic mode. On further analysis, we found that while solving the branch `0x405af4` in optimistic mode, QSYM accidentally negate a previous branch on line 230. In the original input and also inputs generated by LSym, branch at line 130 is not taken and lines 231-235 (guarded code by the line 130) are not covered, however, inputs generated by QSYM flip branch at line 130 and cover lines 231-235. This further confirms our finding of section 8.1 that inputs generated by QSYM's optimistic mode may not always flip the target branch.

8.3 Concolic execution evaluation

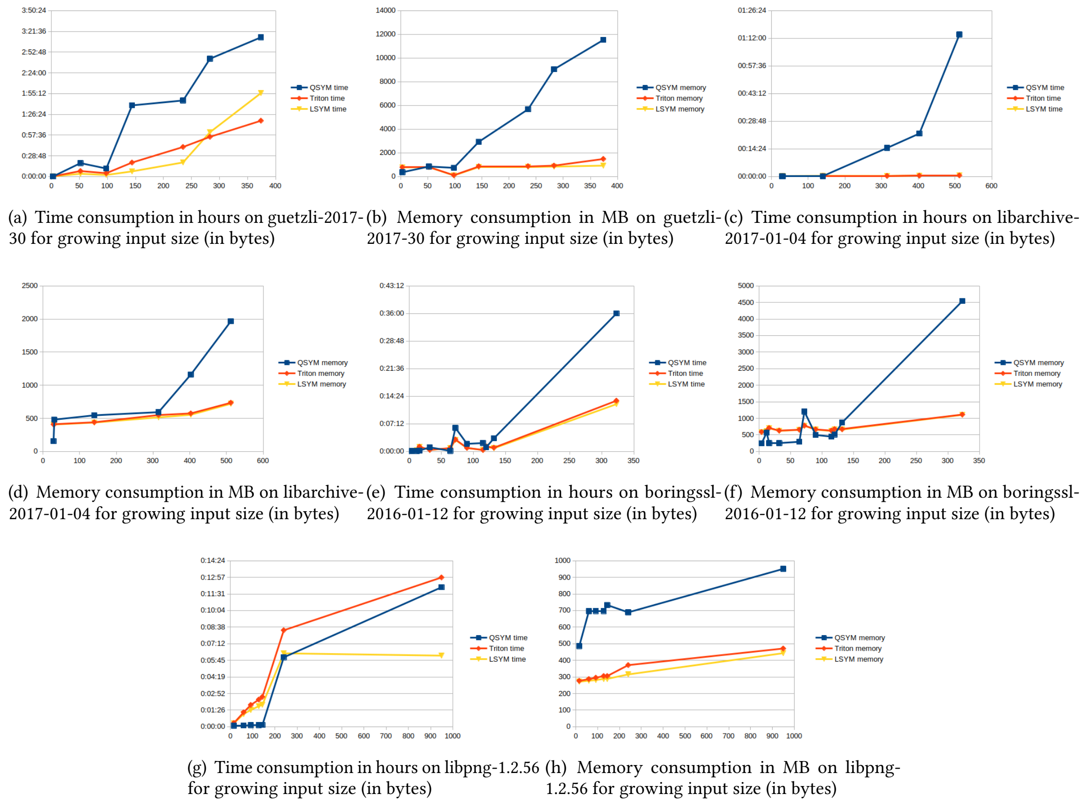
In this section, we compare LSym's concolic execution approach to that of its Triton baseline. In particular, we are interested in measuring the improvements on concolic exection time and coverage due to function-level tracing and taint-assisted partial symbolization.

For this purpose, we select two random input files with different sizes across four applications (3 representative applications from the Google Fuzzer test suite and the popular `objdump` used in much prior work in the area). Table 7 presents our results, also breaking down the improvements by the individual components (*function* for function-level tracing and *taint* for taint-assisted partial symbolization). On average, function-level tracing alone can reduce concolic execution time by 9.39% while taint-assisted symbolization alone yields a 30.65% reduction. Overall, LSym improves Triton's conolic execution time by 37.54% on average. On top of these improvements, LSym also improves Triton's concolic execution coverage by 3.96%, which stems from the fewer instances the less effective optimistic mode is needed by the underlying engine.

As shown in Table 7, the improvements vary across applications and input sizes and so do the speedups offered by function-level tracing and taint-assisted partial symbolization. For applications with a larger number file processing operations, function-level tracing has a more noticeable impact. For applications with many nested branches, partial symbolization is a more important contributor to the improvements.

A more detailed LSym vs. Triton comparison on emulation time, constraint solving time, and memory consumption is presented in Table **??** in the appendix. Summarizing here, across all test cases, we observe a considerable reduction in emulation and constraint solving time (26.23% and 29.41% on average). LSym also reduced memory consumption by 20.62% on average. All these improvements, while already interesting, become much more significant when LSym is used as an end-to-end hybrid fuzzer rather than a standalone concolic execution engine. In Section 8.3, we show

[5] The percentage is calculated `unsat*100 / ( unsat + solved )`.



(a) Time consumption in hours on guetzli-2017-30 for growing input size (in bytes)
(b) Memory consumption in MB on guetzli-2017-30 for growing input size (in bytes)
(c) Time consumption in hours on libarchive-2017-01-04 for growing input size (in bytes)
(d) Memory consumption in MB on libarchive-2017-01-04 for growing input size (in bytes)
(e) Time consumption in hours on boringssl-2016-01-12 for growing input size (in bytes)
(f) Memory consumption in MB on boringssl-2016-01-12 for growing input size (in bytes)
(g) Time consumption in hours on libpng-1.2.56 for growing input size (in bytes)
(h) Memory consumption in MB on libpng-1.2.56 for growing input size (in bytes)

Figure 2: Time and memory cost for QSYM, Triton and LSym

how our single-input improvements in concolic execution time and coverage in particular are crucial to improve the effectiveness of hybrid fuzzing.

8.3.1 Effect of not tainting implicit flows. We measure the effect of not considering implicit flows by following a simple strategy. We collect the constraints for a target branch and generate the corresponding input. If we are able to flip the branch, then we consider it a case where lack of implicit has not affected the result. In our experiment, we run libxml2-v2.9.2 and try to negate every branch of the binary with a xml seed file. Then we analyse the line coverage of source code to see if each branch is negated successfully. The results show that out of 62 branches in the binary, 45 are negated successfully, while 17 fail. The ratio of successfully negated branches is 72.6%. We consider this result good enough for the fuzzing application, in spite of loosing few constraints. The results presented in the previous section 8.1 on other applications also manifest the similar trends. For example, in the case of `objdump`, LSym reached and flipped branches showing that the lack of implicit taintflow does not affect the accuracy substantially. In the next subsection, we will show the results of hybrid fuzzing evaluation.

8.4 Hybrid fuzzing evaluation

So far we have evaluated the performance of LSym with respect to specific dimensions individually, this section evaluates LSym as a hybrid fuzzer in terms of code-coverage and bug detection. We use two different datasets for our evaluation: the Google Fuzzer Test Suite (GFTS) [2] and five real-world applications. We use the real-world applications to show that LSym is practical and finds

Table 7: Improvement on concolic execution time and coverage for LSym vs. its Triton baseline. function=function-level tracing only. taint=taint-assisted partial symbolization only. LSym =both taint and function combined.

application			Triton	function	taint	LSym
boringssl-2016-02-12	input 1	time	0:24:19	0:25:17	0:21:06	0:17:05
		coverage	1090	1087	1090	1090
	input 2	time	0:30:32	0:30:19	0:20:49	0:13:51
		coverage	1131	1146	1156	1151
guetzli-2017-3-30	input 1	time	0:01:42	0:01:21	0:01:16	0:01:22
		coverage	91	91	108	108
	input 2	time	0:20:37	0:18:48	0:09:33	0:07:55
		coverage	99	99	99	99
objdump-2.34	input 1	time	1:08:27	0:58:08	0:52:48	0:52:17
		coverage	1499	1499	1498	1499
	input 2	time	1:03:18	0:52:01	0:50:23	0:49:38
		coverage	784	784	784	784
libpng-1.2.56	input 1	time	0:15:56	0:15:15	0:14:25	0:13:51
		coverage	987	987	1087	1087
	input 2	time	0:14:22	0:12:40	0:04:27	0:04:04
		coverage	1164	1164	1177	1177
average improvement		time		-9,39%	-30,65%	-37,54%
		coverage		+0,12%	+4,01%	+3,96%

new bugs in real software.In all our experiments, we fuzz each application for 24-hours.

8.4.1 Experimental Results on Google Fuzzer Test Suite. The Google Fuzzer Test Suite contains a set of 24 applications with different functionality and complexity, and a clear description of their (known) vulnerabilities. Though the test suite has 24 applications, due to issues with Triton and libdft, we could only run LSym successfully on 18 of them. Table 8 shows the six applications that failed to run with a explanation for the failure. In our comparison, we include both QSYM and the widely used AFL [46] fuzzer. Recall that in our implementation, LSym uses AFL as the main (front-end) fuzzer and Triton as the backend concolic engine).

Table 8: Applications in Google Fuzzer Test Suite that fail to run by LSym.

applications	reason for failure
freetype2-2017	libdft64 fails to trace tainted bytes
lcms-2017-03-21	
libssh-2017-1272	
openthread-2018-02-27	
openssl-1.1.0c wpantund-2018-02-27	Triton fails to disassemble the code

We run a 24-hour test on each application with four methods (AFL, Triton, QSYM and LSym) for **six times** and compare the results on the geometric (geo) means over these runs. Table 9 shows the overall results of our fuzzing evaluation. Note that none of the fuzzers found any crash in the bottom 7 applications in the table. For all the 11 applications where we did find crashes, LSym found the target (known) vulnerability in each application much faster than QSYM. Besides, LSym found the target vulnerability in two applications (harfbuzz-1.3.2 and sqlite-2016-11-14) where AFL, QSYM and Triton fail to find any crash. However, while the coverage of LSym better on average, it is not always better than QSYM. For example, for applications like libxml2-v2.9.2, libarchive-2017-01-04, and libpng-1.2.56, the average coverage of LSym is not as good as QSYM. There are two reasons, as we have shown in the previous sections. First, QSYM generates more coverage in other (unrelated) libraries, because it analyzes all branches in the application. Second, QSYM and Triton more frequently perturb previous constraints in optimistic code and in doing so may lead to execution elswhere (often error handling code).

In general, for the 18 applications, the increase in percentage of coverage and time speed-up rate for LSym is shown in Table 9 and Table 10. On average, LSym achieves 7.13% more coverage and hits the target vulnerability 2.81× faster than AFL; 7.61% more coverage and hits the target vulnerability 4.79× faster than QSYM; 10.87% more coverage and hits the target vulnerability 2.67× faster than QSYM. Finally, it finds vulnerabilities in two applications where AFL, QSYM and Triton failed to find anything in 24 hours.

8.4.2 Finding new bugs in real-world applications. In addition to Google FTS, as a case study for bug finding, we run QSYM and LSym on the latest available versions of 5 real-world applications, each for 24 hours. Table 11 presents our results. As shown in Table 11, we found 17 **new** bugs across applications. For crash traging, we used crashwalk [1], useful to group crashes into clusters and simplify further manual analysis. We then used AddressSanitizer to run each crashing input and manually inspected the root cause. We reported all the new bugs to the developers with detailed information. Four bugs have already been fixed at the time of writing. As shown in Table 11, QSYM fails to find 9 of the new bugs found by LSym. We also notice that LSym finds the new bug much more faster than QSYM in most cases.

9 RELATED WORK

In this section, we survey the most relevant related work on concolic execution and hybrid fuzzing.

9.1 Concolic Execution

A number of solutions have previously suggested strategies to make symbolic or concolic execution more efficient. The chaining approach [16] has pioneered work on using data-flow information to accelerate concolic execution, using data dependency analysis to accelerate testing. SAGE [18] combines symbolic execution with coverage-maximizing heuristics to mitigate its scalability problems and find bugs quickly. STINGER [4] uses static type-dependency analysis to symbolically execute only the parts of the program that may interact with symbolic values. Triton [38] is an instrumentation-based dynamic binary analysis tool which implements concolic execution and optimizes the collected constraints. Triton supports emulation for all the Linux x86/x64 system calls, thus environment-related constraints can be introduced by tracing system calls and symbolizing the corresponding memory locations. Similar to QSYM [45], Triton also supports non-sound constraint debloating policies by only focusing on the constraints of a target branch. Chopper [42] is a source-based solution which allows users to exclude uninteresting parts of the code during symbolic execution, using static analysis to detect and resolve side effects. Other solutions focus on optimizing constraint solving itself. For instance, NEUEX [39] uses neural networks to check satisfiability of mixed constraints and solve them efficiently. Compared to all these solutions, LSym focuses on optimizing off-the-shelf concolic execution tools by means of conservative constraint debloating, which is a simple and effective way to improve the scalability of concolic execution without much compromising its soundness. Very recently Poeplau *et al.* proposed SymCC [34] and SymQEMU [35]–compilation-based symbolic execution technique for source and binary solutions respectively. We believe that the technique presented in LSym can further enhance symbolic execution performance by combining above mentioned techniques with taintflow analyses (e.g. LLVM DSan pass with SymCC and TaintBochs [13]/PANDA [15] with SymQemu).

9.2 Hybrid Fuzzing

Combining concolic execution with fuzzing in a hybrid fuzzing system has recently been gaining momentum in the research community. Driller [41] suggests triggering concolic execution only when fuzzing gets "stuck" to reduce the scalability impact. Other approaches suggest more sophisticated seed/branch selection policies. For instance, DigFuzz [47] proposes a Probabilistic Path Prioritization method which defines the complexity of each branch and only

Table 9: Fuzzing result on Google Fuzzer Test Suite with AFL, Triton, QSYM, and LSym. Due to space paucity, we divided the table into two and Table 10 shows results for the remaining applications. *geomean* column shows means that spans over coverage column of Table 10.

applications	method	code line coverage	time to find target bug	improvement on coverage	improvement (in times) on finding bugs
c-ares-CVE-2016-5180	AFL	40.65	00:22:46	4.85%	3.52
	QSYM	39.83	00:24:52	7.00%	3.85
	Triton	40.48	00:25:52	5.29%	4.00
	LSYM	42.62	00:06:28		
guetzli-2017-3-30	AFL	3950.16	04:27:02	16.74%	7.06
	QSYM	3721.57	06:00:08	23.91%	9.51
	Triton	3605.52	01:53:47	27.89%	3.01
	LSYM	4611.22	00:37:51		
json-2017-02-12	AFL	3473.19	00:01:08	1.03%	1.84
	QSYM	3392.09	00:04:16	3.44%	6.92
	Triton	3463.99	00:01:12	1.29%	1.95
	LSYM	3508.83	00:00:37		
libxml2-v2.9.2	AFL	6046.78	13:07:29	4.33%	5.19
	QSYM	6970.87	09:19:53	-9.50%	3.69
	Triton	5910.36	04:33:38	6.73%	1.80
	LSYM	6308.41	02:31:38		
llvm-libcxxabi-2017-01-27	AFL	11527.11	00:14:30	1.21%	1.07
	QSYM	10970.74	00:21:02	6.34%	1.56
	Triton	11066.52	00:23:07	5.42%	1.71
	LSYM	11666.78	00:13:30		
openssl-1.0.2d	AFL	3380.02	00:37:17	0.64%	4.34
	QSYM	3351.56	00:29:16	1.49%	3.41
	Triton	3354.88	00:43:07	1.39%	5.02
	LSYM	3401.59	00:08:35		
pcre2-10.00	AFL	23737.37	00:58:29	11.01%	1.34
	QSYM	25448.9	01:21:47	3.54%	1.87
	Triton	24879.05	01:03:36	5.91%	1.45
	LSYM	26350.23	00:43:45		
re2-2014-12-09	AFL	5414.22	01:30:51	3.32%	4.41
	QSYM	5394.16	10:17:47	3.70%	29.97
	Triton	5476.92	01:24:59	2.13%	4.12
	LSYM	5593.75	00:20:37		
vorbis-2017-12-11	AFL	3145.95	11:19:53	2.53%	1.67
	QSYM	3130.53	22:03:54	3.03%	3.26
	Triton	3040.56	21:20:50	6.08%	3.15
	LSYM	3225.49	06:46:06		
harfbuzz-1.3.2	AFL	12235.46		8.63%	**new**
	QSYM	12217.58		8.79%	**new**
	Triton	12535.6		6.03%	**new**
	LSYM	13291.75	16:06:20		
sqlite-2016-11-14	AFL	8485.51		19.03%	**new**
	QSYM	7393.5		36.61%	**new**
	Triton	5519.13		83.01%	**new**
	LSYM	10100.32	23:25:35		
geomean	AFL			**7.13%**	**2.81**
	QSYM			**7.61%**	**4.79**
	Triton			**10.87%**	**2.67**

uses concolic execution on these hardest-to-solve branches. Similarly, LEGION [29] applies *Monte Carlo tree search* algorithm to identify the most promising location to explore next and applies a form of *directed fuzzing* to reach there. In doing so, it applies symbolic execution to flip rare branch that is guarded by hard constraints. LSym can benefit from LEGIONn's branch selection policy to decide which node to target next for branch flipping. MEUZZ [10] and SAVIOR [11] suggest more sophisticated polices based on coverage- or bug-oriented predictions (respectively). Pangolin [22] proposes a incremental method called "polyhedral path abstraction", in order to reuse precious computation results. SDHF [28] suggests using hybrid fuzzing in a directed fuzzing scenario, by identifying the sequence of statements before the target. HFL [26] addresses kernel-specific hybrid fuzzing challenges. A very different and interesting hybrid fuzzing approach is implemented in TFuzz [32] wherein the program is transformed by disabling input checks in the program so that input generation is fast. Further, to reduce the false positives, symbolic execution is used to verify the validity of the detected bug. As symbolic execution is expensive, TFuzz applies it on selected inputs thereby reducing the overhead on such a heavy technique. However, as also noted in DigFuzz [47], main contribution of most of the hybrid-fuzzing solutions focuses on the deriving intelligent ways of optimizing the invocation of concolic execution due to its known performance overhead. LSym, on the other hand, focuses

Table 10: (Table 9 conti..) Fuzzing result continues on Google Fuzzer Test Suite with AFL, Triton, QSYM, and LSym. where none of the fuzzers found any bug.

applications	method	code line coverage	time to find target bug	improvement on coverage	improvement (in times) on finding bugs
boringssl-2016-01-12	AFL	1488.95		3.19%	
	QSYM	1443.25		6.46%	
	Triton	1388.49		10.66%	
	LSYM	1536.5			
libarchive-2017-01-04	AFL	5361.98		8.70%	
	QSYM	6580.91		-11.43%	
	Triton	5195.87		12.18%	
	LSYM	5828.69			
libjpeg-turbo-07-2017	AFL	3910.65		8.01%	
	QSYM	3803.15		11.06%	
	Triton	3002.17		40.69%	
	LSYM	4223.78			
libpng-1.2.56	AFL	1324.64		2.96%	
	QSYM	1477.36		-7.69%	
	Triton	1333.01		2.31%	
	LSYM	1363.79			
openssl-1.0.1f	AFL	631.94		10.54%	
	QSYM	660.21		5.81%	
	Triton	540.96		29.13%	
	LSYM	698.56			
proj4-2017-08-14	AFL	1281.31		5.25%	
	QSYM	1325.5		1.74%	
	Triton	1263.51		6.73%	
	LSYM	1348.59			
woff2-2016-05-06	AFL	14.56		14671.43%	
	QSYM	779.9		175.77%	
	Triton	42.04		5015.89%	
	LSYM	2150.72			

Table 11: New bugs found by LSym vs. QSYM. WA=write access, RA= read acces,LL=llvm-libcxxabi-ce3db12

		bug type	status	LSym				QSYM			
				found	coverage	crash	new bug found time	found	coverage	crash	new bug found time
1	size-2.34	null point dref	rep & fix	yes	4666	20	0:05:41	yes	5088	18	0:06:43
2	bento4-06c39d9	heap overflow	rep & fix	yes	8877	1055	0:00:05	no	9886	251	0:19:21
3		WA violation	rep & fix	yes				yes			
4		null point dref	rep & fix	yes				no			
5		null point dref	reported	yes				yes			
6		heap overflow	reported	yes				no			
7	yasm-c9db6d7	heap UaF	reported	yes	17258	964	0:21:42	yes	16525	188	0:09:35
8		RA violation	reported	yes				yes			
9		RA violation	reported	yes				no			
10		stack overflow	reported	yes				no			
11	binaryen-0c58de1	heap overflow	reported	yes	3619	610	0:00:05	no	7730	529	0:00:05
12		illegal inst.	reported	yes				yes			
13		assert fail.	reported	yes				no			
14		WA violation	reported	yes				no			
15		assert fail(16x)	reported	yes				yes			
16	LL	assertion failure	reported	yes	7147	147	0:12:29	yes	7438	71	0:52:02
17		stack overflow	reported	yes				no			

on improving the performance of concolic execution to improve the hybrid fuzzing. From this perspective, LSym can use several of these techniques to further improve the overall performance and vice-a-versa. From this perspective, the closest to our technique is QSYM [45], which we have discussed and compared thoroughly in this paper.

Other fuzzers have previously used dynamic taint analysis to find bugs more quickly. However, unlike LSym, existing taint-assisted fuzzers all rely on dynamic taint analysis to improve input mutation. For instance, VUzzer [37] uses taint analysis to identify which bytes can influence the target branch and only mutates those bytes. Angora [9] and derivatives (e.g., ParmeSan [31]) use a similar strategy but uses gradient descent-based mutation rather than random or magic number-based mutation like VUzzer. TIFF [23] implements bug-directed mutation by inferring the type of the input bytes using dynamic taint analysis.

10 CONCLUSION

In this paper, we presented LeanSym (LSym), an efficient hybrid fuzzer based on constraint debloating. Rather than custom-optimizing

the underlying concolic execution engine for hybrid fuzzing, LSym intelligently combines run-of-the-mill binary analysis tools to eliminate unnecessary constraints during concolic execution and speed up input generation in fuzzing (as an application). In particular, LSym focuses on two simple constraint debloating strategies, namely function-level tracing for system I/O emulation and taint-assisted partial input symbolization. These strategies result in more efficient and memory-conscious operations of the core concolic execution component. More importantly, despite the simple branch/seed selection policies and glue code for hybrid fuzzing implemented in only 1k LOC, LSym outperforms state-of-the-art custom-optimized hybrid fuzzers and finds more bugs in real-world applications.

While developing LSym, we also notice limitations of our current implementation that provide a room for further improvements. We have not empirically tested the effect of implementing implicit taint-flows on performance in terms of overhead and accuracy. As noted earlier, function-level tracing involved manually creating the summaries of relevant syscalls and library functions. This functionality can be enhanced further by adopting function summary approach of angr, for example. A further area of improvement comes from a more intelligent approach for input and branch selection. Overall, we believe LSym provides a new hybrid fuzzing baseline that is practical, modular, and easily extensible moving forward. To foster more research in the area, we plan to open source our LSym prototype in the near future.

ACKNOWLEDGMENTS

We thank our shepherd, Nathan Burow, and the anonymous reviewers for their feedback. This work was supported by the Netherlands Organisation for Scientific Research through grants NWO "TROPICS" (628.001.030), "INTERSECT", and "Theseus" and by the EKZ through grant "Memo" . Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of any of the sponsors.

REFERENCES

[1] [n.d.]. crashwalk. https://github.com/bnagy/crashwalk.
[2] [n.d.]. *Google Fuzzer Test Suite.* https://github.com/google/fuzzer-test-suite
[3] [n.d.]. libdft64. https://github.com/vusec/vuzzer64/.
[4] Saswat Anand, Alessandro Orso, and Mary Jean Harrold. 2007. Type-Dependence Analysis and Program Transformation for Symbolic Execution. In *Proc. TACAS'07*. Springer-Verlag, 117–133.
[5] Jacob Burnim and Koushik Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *Proc. ASE'08*. 443–446. https://doi.org/10.1109/ASE.2008.69
[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. OSDI'08*. 209–224.
[7] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. 2008. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *DIMVA*. Springer Berlin Heidelberg, Berlin, Heidelberg, 143–163.
[8] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proc.S&P'12*. IEEE Computer Society, 380–394.
[9] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *IEEE S&P'18*. San Francisco, CA, USA.
[10] Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. 2020. MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing. *arXiv preprint arXiv:2002.08568* (2020).
[11] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. [n.d.]. SAVIOR: Towards Bug-Driven Hybrid Testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. 15–31.
[12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Trans. Comput. Syst.* 30, 1 (2012).
[13] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. 2004. Understanding Data Lifetime via Whole System Simulation. In *Usenix Sec'04 (SSYM'04)*. 22.
[14] TheAnh Do, A. C. M. Fong, and Russel Pears. 2013. Dynamic Symbolic Execution Guided by Data Dependency Analysis for High Structural Coverage. In *Evaluation of Novel Approaches to Software Engineering*. Springer Berlin Heidelberg, 3–15.
[15] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable Reverse Engineering with PANDA. In *PPREW'15*. Article 4.
[16] Roger Ferguson and Bogdan Korel. 1996. The Chaining Approach for Software Test Data Generation. *ACM Trans. Softw. Eng. Methodol.* 5, 1 (Jan. 1996), 63–86. https://doi.org/10.1145/226155.226158
[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. *SIGPLAN Not.* 40, 6 (2005), 213–223.
[18] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS'08*. Internet Society.
[19] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1 (Jan. 2012), 20–27.
[20] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Sec'13)*. 49–64.
[21] S. Horwitz, T. Reps, and D. Binkley. 1988. Interprocedural Slicing Using Dependence Graphs. In *Proc. PLDI'88*. ACM, 35–46.
[22] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang. 2020. Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In *IEEE Symposium SP'20*. 1199–1213.
[23] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2018. TIFF: Using Input Type Inference To Improve Fuzzing. In *Proc. ACSAC'18*. ACM, 505–517.
[24] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2020. Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection. In *USENIX Security 20*. USENIX Association, 2595–2612.
[25] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *SIGPLAN/SIGOPS VEE '12*. ACM, 121–132.
[26] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *NDSS'20*.
[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *CCS18'18*. 2123–2138.
[28] H. Liang, L. Jiang, L. Ai, and J. Wei. 2020. Sequence Directed Hybrid Fuzzing. In *Proc. SANER'20*. 127–137.
[29] Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin I.P. Rubinstein. 2020. LEGION: Best-First Concolic Testing. In *Proc. ASE'20*. 54–65.
[30] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI'05* (Chicago, IL, USA). ACM, New York, NY, USA, 190–200.
[31] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX SEC'20*. USENIX Association.
[32] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *Proc. IEEE S & P'18*. 697–710. https://doi.org/10.1109/SP.2018.00056
[33] Sebastian Poeplau and Aurélien Francillon. 2019. Systematic Comparison of Symbolic Execution Systems: Intermediate Representation and Its Generation. In *Proc. ACSAC'19*. 163–176.
[34] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *USENIX Security 20*. 181–198.
[35] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *NDSS'21*.
[36] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing Nasa Software. In *Proc. ISSTA'08*. ACM, 15–26.
[37] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*.
[38] Florent Saudel and Jonathan Salwan. 2015. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*. SSTIC, 31–54.
[39] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. 2019. Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints. In *Proc. NDSS'20*.
[40] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, Grosen John, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE S&P'16* (SAN JOSE, CA, USA). IEEE CSP.
[41] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS'16*. Internet Society, 1–16.

[42] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped Symbolic Execution. In *Proc. ICSE'18*. ACM.
[43] Haijun Wang, Ting Liu, Xiaohong Guan, Chao Shen, Qinghua Zheng, and Zijiang Yang. 2017. Dependence Guided Symbolic Execution. *IEEE Trans. Softw. Eng.* 43, 3 (2017), 252–271.
[44] Xiaoyin Wang, Lingming Zhang, and Philip Tanofsky. 2015. Experience Report: How is Dynamic Symbolic Execution Different from Manual Testing? A Study on KLEE. In *Proc. ISSTA'15*. 199–210.
[45] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Sec'18)*.
[46] Michal Zalewski. [n.d.]. American Fuzzy Lop. At: http://lcamtuf.coredump.cx/afl/.
[47] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *NDSS'19*.

A APPENDICES

A.1 An Example of Function-level Tracing

We take the simple example from Listing 1 and show how we hook the functions to partially symbolize the input file. We use ltrace and strace to find the potential relationship between functions and syscalls, as shown in Listing 3 and Listing 4. We can see that `fopen()` opens the input file at a specific address (0x1b5a010) and the `fread()` function reads from the same address into a string buffer (located at 0x7fffb1cf81d0). Thus, we can hook `fopen()` and `fread()`.

Listing 3: Using ltrace to demonstrate libc functions called in the example program.

```
1  $ ltrace ./example input
2  __libc_start_main(0x400666, 2, 0x7fffb1cf86a8, 0x4008e0
3  <unfinished ...>
4  fopen("input", "r")                    = 0x1b5a010
5  fread(0x7fffb1cf81d0, 950, 1, 0x1b5a010)         = 1
6  puts("branch 2"branch 2
7  )                                = 9
8  puts("branch 3"branch 3
9  )                                = 9
10 puts("branch 5"branch 5
11 )                                = 9
12 puts("branch 0"branch 0
13 )                                = 9
14 +++ exited (status 0) +++
```

Listing 4: Using strace to demonstrate syscall functions called in the example program.

```
1  $ strace ./example input
2  execve("./symbex5", ["./example", "input"],
3  [/* 11 vars */]) = 0
4  ......
5  open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
6  ......
7  open("input", O_RDONLY)        = 3
8  ......
9  read(3, "BBBBBBBBBBBBBBB4BB!ABBBBBBBBBBBB"..., 4096) = 950
10 ......
11 +++ exited with 0 +++
```

We can also use the automated PIN tool to output the calling sequences of file processing syscalls. As shown in Listing 5, the `fopen()` function sequence falls into the `open()` syscall, while in Listing 6, `fread()` function sequence falls into the `read()` syscall. Thus, we know the functions related file processing to hook are `fopen()` and `fread()`.

Listing 5: Fopen calling sequences from function to syscall in the example program.

```
1  address: 0x400550,  function name: fopen@plt
2  address: 0x4004f0,  function name: .plt
3  address: 0x7f6dd2f91d70,  function name: fopen
4  address: 0x7f6dd2f438a0,  function name: .plt.got
5  ......
6  address: 0x7f6dd2f9db30,  function name: _IO_file_fopen
7  address: 0x7f6dd2f9da40,  function name: _IO_file_open
8  address: 0x7f6dd301b030,  function name: open
9  [*] Open syscall 0x7f6dd301b03e: 2(0x7ffec8ae76f0, 0x0,
10 0x1b6, 0x0, 0x8, 0x1)returns: 0x3
```

Listing 6: Fread calling sequences from function to syscall in the example program.

```
1  address: 0x400520,  function name: fread@plt
2  address: 0x4004f0,  function name: .plt
3  address: 0x7f6dd2f921a0,  function name: _IO_fread
4  ......
5  address: 0x7f6dd2f9d1a0,  function name: _IO_file_read
6  address: 0x7f6dd301b250,  function name: read
7  [*] Read syscall 0x7f6dd301b25e: 0(0x3, 0x9c7240, 0x1000,
8  0x7f6dd2ee8700, 0x9c70f0, 0x7ffec8ae50f0)returns: 0x3b6
```

The next step is to implement function summary. We can get the string buffer address which should be symbolized by analyzing the disassembly code of `fopen()` and `fread()` functions, as shown in Listing 7 and Listing 8. Another important information is the read offset of the input, which we should use to relate the symbolized bytes to the original input offset. As shown in Listing 9, we can get the location address where the value of `_IO_read_ptr` and `_IO_read_base`, then we can calculate the value of the file reading offset as described in Section 4.

Listing 7: Disassembly code of calling fopen() in the example program.

```
1  0x00000000004006c0 <+90>:   mov
2  0x00000000004006c3 <+93>:   callq  0x400550 <fopen@plt>
3  0x00000000004006c8 <+98>:   mov
4  0x00000000004006cf <+105>:  mov    -0x3f8(
```

Listing 8: Disassembly code of calling fread() in the example program.

```
1  0x00000000004006dd <+119>:  mov
2  0x00000000004006e0 <+122>:  mov    $0x1,
3  0x00000000004006e5 <+127>:  mov    $0x3b6,
4  0x00000000004006ea <+132>:  mov
5  0x00000000004006ed <+135>:  callq  0x400520 <fread@plt>
6  0x00000000004006f2 <+140>:  movl   $0x0,-0x404(
```

Listing 9: Information of file structure in the example program to demonstrate how to calculate the file reading offset.

```
(gdb) p *fp
$1 = {_flags = -72539000,
  _IO_read_ptr = 0x6025f6 "",
  _IO_read_end = 0x6025f6 "",
  _IO_read_base = 0x602240 'B' <repeats 15 times>,
  "4BB!A", 'B' <repeats 180 times>...,
  _IO_write_base = ......,
  _IO_write_ptr = ......,
  _IO_write_end = ......
  _IO_buf_base = 0x602240 'B' <repeats 15 times>, ,
  "4BB!A", 'B' <repeats 180 times>..., _IO_buf_end =
  0x603240 "", _IO_save_base = 0x0, _IO_backup_base = 0x0,
  _IO_save_end = 0x0, _markers = 0x0,
 ......
```

A.2 An Example of Taint-assisted Symbolization

We firstly use taint analysis to get the tainted bytes of each branch in the example shown in List 1. In Listing 10, we can see that the target branch is located at 0x40085, where the tainted offsets that will influence this branch is offset 14 and 15. Therefore, we will symbolize offset 14 and 15 in the input. After symbolizing the related offset by tracing function calls, we can collect constraints and solve them, as shown in Table 1 in Section 2.

Listing 10: Tainted information of target branches in the example program.

```
......
32 reg reg 0x0000000000400773
{0,1} {1} {1} {1} {} {} {} {}
{3,4,5} {4,5} {4,5} {4,5} {} {} {} {}
0x84 0xc6
32 reg reg 0x0000000000400773
{1,2} {2,4} {2,4} {2,4} {} {} {} {}
{4,5,6} {4,5,6} {4,5,6} {4,5,6} {} {} {} {}
0x84 0xc6
32 reg reg 0x0000000000400773
{2,3} {3,4,5} {3,4,5} {3,4,5} {} {} {} {}
{5,6,7} {4,5,6,7} {4,5,6,7} {4,5,6,7} {} {} {} {}
0x84 0xc6
32 reg imm 0x00000000004007f1
{2,4} {4} {4} {4} {} {} {} {}
{} {} {} {} {} {} {} {}
0x84 0x58
32 reg imm 0x0000000000400816
{4,8} {8} {8} {8} {} {} {} {}
{} {} {} {} {} {} {} {}
0x84 0x58
32 reg imm 0x000000000040083b
{15,18} {18} {18} {18} {} {} {} {}
{} {} {} {} {} {} {} {}
0x55 0x55
......
32 reg imm 0x0000000000400860
{5,9} {9} {9} {9} {} {} {} {}
{} {} {} {} {} {} {} {}
0x84 0x58
32 reg imm 0x0000000000400885
{14,15} {14} {14} {14} {} {} {} {}
{} {} {} {} {} {} {} {}
0x76 0x67
......
```

A.3 Detailed Experimental Results for Section 8.3

Table 12 shows the experimental results for about the improvement on execution time, solving time, emulation time and memory consumption for concolic execution by LSym compared with Triton.

Table 12: Time and memory overhead of Triton and LSym.

application		Triton	LSym
boringssl-2016-02-12	execution time	0:34:11	0:27:34
	solving time(s)	200.098	114.661
	emulation time	0:30:01	0:24:33
	memory	134	107
guetzli-2017-3-30	execution time	0:17:18	0:11:01
	solving time(s)	48.021	11.324
	emulation time	0:16:30	0:10:49
	memory	110	89
libarchive-2017-01-04	execution time	0:08:50	0:07:52
	solving time(s)	143.33	137.931
	emulation time	0:06:24	0:05:28
	memory	123	96
objdump-2.34	execution time	2:38:09	2:01:32
	solving time(s)	19.296	12.703
	emulation time	2:37:47	2:01:04
	memory	474	230
pcre2-10.00	execution time	0:02:25	0:01:55
	solving time(s)	40.741	39.162
	emulation time	0:01:24	0:01:03
	memory	144	137
llvm-libcxxabi-2017-01-27	execution time	0:01:30	0:01:08
	solving time(s)	4.920	4.156
	emulation time	0:01:26	0:00:50
	memory	81	76
average improvement	execution time		**22.81%**
	solving time		**29.41%**
	emulation time		**26.23%**
	memory		**20.62%**