Characterizing and Improving Bug-Finders with Synthetic Bugs

Yu Hu
New York University
yh570@nyu.edu

Zekun Shen
New York University
zs1127@nyu.edu

Brendan Dolan-Gavitt
New York University
brendandg@nyu.edu

***Abstract*—Automated bug-finding tools such as KLEE have achieved mainstream success over the last decade, and have proved capable of finding deep bugs even in programs that have received significant manual testing. Some recent works have demonstrated techniques for finding bugs in these bug-finding tools themselves; however, it remains unclear whether these correctness issues have any practical impact on their ability to uncover serious bugs. In this paper, we study this issue by conducting experiments with KLEE 1.4 and 2.2 on several corpora of memory safety bugs. Using *automated bug injection*, we can automatically find false negatives (i.e., bugs missed by KLEE); moreover, because the bugs we inject come with triggering inputs, we can then use *concolic execution* to tell which bugs were missed due path explosion and which are caused by soundness issues in KLEE. Our evaluation uncovers several sources of unsoundness, including a limitation in how KLEE detects memory errors, mismatches in the modeling of the C standard library, lack of support for floating point and C++, and issues with calls to external functions. Our results suggest that bug injection and other synthetic corpora can help highlight implementation issues in current tools and illuminate directions for future research in automated software engineering.**

***Index Terms*—software security, bug-finding, symbolic execution**

I. Introduction

Thoroughly testing a program has traditionally required a large number of manually-generated test cases. Starting with the introduction of practical SMT solvers around 2002 [7] [6] [45], automated test-case generation tools based on symbolic execution have received significant interest from both research community and practitioners. The focus of these bugs-finding tools is mainly on finding unknown bugs in real-world programs; however, there has been less attention on the reliability and completeness of the tools themselves. Existing work mostly focuses on demonstrating improvements on easy-to-measure quantities such as code coverage [11], [14], [33], [37], [54], [65] or by finding previously unknown bugs [18], [31]. This makes it difficult to compare the effectiveness of different techniques, and leaves open the question of how to identify weaknesses and areas for improvement in current tools.

It is difficult to gauge bug-finding merit and improvement to guide the use and development of these tools. For example, American fuzzy lop (AFL) [1] uses fuzzing while KLEE [12] uses symbolic execution and satisfiability modulo theory (SMT), but we cannot determine which technique is more efficient at finding bugs without a large amount of ground truth. Most existing corpora contain few test cases or are designed for static analyzers without triggering inputs. In addition, if the bug finders were already trained with these historical corpora, the dearth of updates may cause the corpora to lose value. To mitigate the lack of ground-truth corpora, current approaches to test bug-finders include differential testing with multiple independent analyzers [28], or using an automatic program generator to generate random programs with known behavior [19]. However, differential testing requires multiple independent implementations, and randomly generated programs tend to be small and differ in important ways from real-world programs.

In this paper, we explore the use of *automated bug injection* as a way to generate a ground truth with a large number of testcases to evaluate and improve bug-finding tools. Automated bug injection [21], [46] is a technique for inserting large numbers of bugs into existing, real-world programs. Unlike mutation testing [20], [24], each bug comes with a triggering input, which ensures that bugs can be reproduced. The core of our technique is the idea that each injected bug *not* found by a bug-finding tool can serve as a test case that sheds light on the performance of the bug-finder and represents an opportunity for improvement. In addition, by adding large numbers of bugs, we can empirically test the effects of different tool configurations on the rate of bug discovery. Table I shows the differences between historical, synthetic and automated generated bug corpora.

Our target bug-finding tool is KLEE [12], which uses symbolic execution to identify bugs. We separate our testing into two major parts. First, we use KLEE to analyze small programs, including 159 buggy variants of a small (62 SLoC) program and programs from the Juliet Test Suite [3]. Because KLEE is able to cover all paths in these small programs, and each program contains a known bug, any missed bug in this experiment represents a potential soundness issue for KLEE.

Second, because small, synthetic programs may not be representative of the complexity of real-world software, we also inject bugs into the `coreutils` suite of programs. Because these programs are generally too large to exhaustively explore them using symbolic execution, we face a challenge in determining which bugs are missed due to soundness issues and which are missed because the search space was too large. To overcome this, we implement a *concolic execution* mode

TABLE I: Corpora Comparison.

Corpus Category	Corpus Name	Given Input?	Real Bug?	Real Prog?	Large?	Automated?	Customized?
Historical	Linux Flaw [4]	✓	✓	✓	✓	✗	✗
Manual Synthetic	DARPA CGC [2]	✓	✗	✗	✓	✗	✗
	Juliet Test Suite [3]	✗	✗	✗	✓	✗	✗
	LogicBomb [60]	✓	✗	✓	✗	✗	✗
Automatic Generation	Evil Coder [43]	✗	✗	✓	✗	✓	✓
	LAVA [21]	✓	✗	✓	✓	✓	✓
	Apocalyse [46]	✓	✗	✓	✗	✓	✓

for KLEE; because our injected bugs come with triggering inputs, we can then direct KLEE down the exact path needed to trigger the bug. Any bugs missed in this mode can therefore be attributed to soundness issues in KLEE.

Our testing uncovers a number of sources of unsoundness, including a limitation in how KLEE detects memory errors that prevents it from finding a use-after-free vulnerability, a bug in KLEE v1.4 which does not check pointers passed to external library calls (fixed in KLEE 2.0), missing feature support (environment variable, floating point, multithreading, some system calls, inline assembly, C++, etc.) and inconsistencies between `glibc` and the uClibc standard library used by KLEE.

We chose to focus on KLEE in this work, since it is widely used in the research community; however, our broader goal is to show how synthetic bugs can be used to characterize the strengths and weaknesses of bug-finding tools and find areas where they could be improved. We can apply the same methodology to other symbolic execution tools such as angr, CREST, or FuzzBALL using our dataset and bug injection tools.

In summary, this paper makes the following contributions:

- We demonstrate that automated bug injection and other synthetic bug corpora can be used to identify issues in mature bug-finding tools like KLEE by finding and characterizing both false positives and false negatives.
- We provide a technique for distinguishing between missed bugs caused by soundness issues and those caused by resource limitations in symbolic execution engines by using concolic execution with known triggering inputs.
- We find that most soundness issues in our evaluations are caused by unimplemented features or limitations of the standard library model.

II. Background

A. Symbolic Execution and KLEE

In this paper, we focus on KLEE [12], a well-known symbolic executor. KLEE uses symbols to represent arbitrary values instead of concrete inputs to a program. When KLEE analyzes a program, it maintains a set of states representing the state of a program, including the history of branches taken and the state of objects in memory. Each state contains path constraints that record the branch decisions the state has made and the state's symbolic memory space. KLEE symbolically executes each instruction, potentially creating new states (*forking*) as necessary when it encounters conditional branches. It invokes the solver to check the validity of new states for certain "dangerous" operations such as pointer dereferences. When KLEE reaches an error state or the end of the program, it invokes an SMT solver such as STP or Z3 to generate a concrete input that satisfies the path condition. KLEE operates on LLVM bitcode, and so testing a program with KLEE requires compiling it with `clang` to the LLVM intermediate representation.

Most C programs depend on functions in the C standard library; however, the standard library (e.g., `glibc`) is usually not available in bitcode form, meaning that KLEE cannot symbolically execute these library calls. To deal with the external environmental problem more accurately and efficiently, KLEE includes a modified version of `uClibc` called `klee-uClibc`. In addition, KLEE models a POSIX environment, including files, directories, etc.

In our experiments, we focus on KLEE because it is very commonly used in software engineering research [10], [11], [17], [33], [37], [39], [42], [58] and can support real-world software such as the GNU `coreutils`.

B. Concolic execution

Concolic execution runs a program with symbolic execution along a given concrete execution path. Concolic execution was introduced by Godefroid et al. in DART [49], and many tools now have concolic functionality, such as CUTE [51], KLEE, jCute [50], Driller [53] and Triton [48]. In this work, we implement a concolic execution mode for KLEE and use it to distinguish between bugs missed due to insufficient searching and those missed due to deficiencies in the tool itself.

C. Automated Bug Injection with LAVA

LAVA is an automated tool that can inject multiple bugs to programs through source-level instrumentation [21]. LAVA adds memory corruption bugs to C programs, and generates corresponding triggering inputs. LAVA can provide a significant number of possible bug injections. For example, it can create 1700 possible bug injections in the coreutils `base64` program. Although LAVA supports injecting several different types of bug, in this paper we only inject memory safety bugs (i.e., dereference of out-of-bound pointers).

LAVA's bug injection operates in three main steps: dynamic taint analysis, source-level bug injection, and payload verification. LAVA's bug injection uses dynamic taint analysis to find program points where input is available, and then performs source to source translation to inject bugs that use the input

to trigger a buffer overflow when some condition is met. In the injected source code, a bug consists of two functions. `lava_set` copies a 4-byte value from the program's input into a slot in a global `lava_val` array. `lava_get` fetches the element from the array and returns it; if the returned value is equal to the trigger, the value is added to a target pointer, forcing it out of bounds. In this way, standard inputs are unlikely to hit the vulnerability. The injected bugs produced by LAVA all can be triggered by an attacker-controlled input, and some are potentially exploitable [27].

D. Juliet Test Suite for C/C++

The Juliet Test Suite was created by the NSA Center for Assured Software. The test suite contains 64,099 test cases, categorized according to the Common Weakness Enumeration (CWE) [41] classification system. Most source code files from the Juliet Test Suite contain at least one flaw each, and most contain only one. In most cases, the error is located in a function named "Bad". Most programs in the test suite do not request input or input files, and have only one path that goes through the "Bad" function and the fixed "Good" function.

However, since the Juliet Test Suite was designed to test static analysis tools, we found many Juliet Test Suite programs require some manual tuning to work with KLEE. For example, many target programs use the `rand` function to trigger the vulnerable function, which requires manually modifying the test cases to introduce symbolic data using `klee_make_symbolic`.

III. Methodology and Implementation

We separate our testing into two major parts. In the first part, we test small programs using LAVA and the Juliet Test Suite. Because these small programs only contain a few lines and branches, we expect that KLEE can traverse all possible path and should be able to find any bug we might add; in other words, if it misses a bug, this indicates a soundness issue that we can investigate further. In our small program analysis with the LAVA corpus, to uncover soundness issues, we create 159 buggy versions of a small C program using LAVA. In our experiments with the Juliet Test Suite, we selected 29,723 validated programs with flaws that covered 91 CWE categories.[1]

Because real-world programs may have different properties than the small, synthetic programs we evaluate in the first part, we also examine larger programs using bug injection. We inject more than 200 bugs into programs from the well-known GNU `coreutils` suite.[2] To distinguish missed bugs from soundness issues from those that could not be found within the given time limit, we modified KLEE to perform concolic execution. When running in concolic mode, KLEE follows a path we provide which is known to trigger the injected bug. In other words, this tests whether KLEE would be able to find our injected bug *if* its search could find the correct path.

In our experiments, we tested both KLEE v2.2 and KLEE v1.4. KLEE v2.2 is the latest stable version of KLEE, and KLEE v1.4 is widely used in prior research such as S2E [15] and UC-KLEE [44]. Thus, limitations in KLEE v1.4 are important to understand since they many be inherited by KLEE forks used by the research community.

A. Small Program Analysis

To create the small-program LAVA corpus we wrote a small (62 lines) non-vulnerable C program, `toy.c`[3], which processes and prints information about a simple binary file format. We used LAVA to automatically generate 159 buggy variants of this program. Each buggy program has a single bug injected. Next, we ran KLEE to find bugs and generate test cases. We checked all missed bugs and investigated them in detail to find underlying soundness issues in KLEE.

For the Juliet Test Suite, we selected all the test cases that could be compiled using `wllvm`/`clang` and that successfully generated the LLVM bitcode. We filtered out some cases such as `socket` which are known to be unsupported by KLEE. In addition, some testcases which needed to be manually tuned, such as those that used `rand`, are also excluded. This reduced the size of test suite from 64,099 tests down to 29,723. We then used KLEE to detect bugs in the programs and checked for:

- **True positives:** KLEE reports an error with the correct location and correct type.
- **False positives:** KLEE reports an error at a location in the program that does not contain a flaw.
- **False negatives:** KLEE missed the error in the target program.

We note that not all the CWEs in the Juliet test suite are necessarily suitable for testing KLEE. We identified these categories by running all the Juliet tests under KLEE and then examining any CWE category where KLEE found no true positives.

Some categories are simply out of scope for a tool that operates on LLVM intermediate representation (IR); examples of these include CWE546 ("Suspicious Comment" and CWE398 ("Poor Code Quality"). We label these as "out of scope" in Table II. We also ruled out some categories that could, in principle, be supported by KLEE, but which would require modifications to either KLEE or to the test cases; these are labeled as "Unimplemented" in Table II. Examples in this category include:

- **CWE190 (Integer Overflow)** is not detected by default with KLEE; however, if the program is compiled using UBSan's signed or unsigned integer overflow options, it can be detected.
- **CWE690 (NULL Deref From Return)** includes test cases where malloc may return NULL, which KLEE's

[1]The modified Juliet Test Suite can be found at https://github.com/yh570/Juliet_test_suite.git.

[2]The injected toy program and `coreutils` can be found at https://github.com/yh570/LAVA_corpus.git.

[3]Due to space constraints, we omit the source code listing for this program. The interested reader can find it at https://moyix.net/toy.c.

TABLE II: Categorizing unsupported CWE entries with KLEE

CWE	Description	Classification	Possible Solution
CWE426	Untrusted Search Path	Unimplemented	Assertion
CWE511	Logic Time Bomb	Unimplemented	Assertions/Symbolic Input
CWE377	Insecure Temporary File	Unimplemented	Blacklist
CWE475	Undefined Behavior for Input to API	Unimplemented	Blacklist
CWE775	Missing Release of File Descriptor or Handle	Unimplemented	Check open fds at exit
CWE762	Mismatched Memory Management Routines	Unimplemented	C++ support [62]
CWE78	OS Command Injection	Unimplemented	C++/Symbolic-size Memory Allocation/Socket
CWE690	NULL Deref From Return	Unimplemented	Fault injection [34]
CWE188	Reliance on Data Memory Layout	Unimplemented	Field of Struct
CWE667	Improper Locking	Unimplemented	Multi-thread [8]
CWE190	Integer Overflow	Unimplemented	Sanitizer
CWE196	Unsigned to Signed Conversion Error	Unimplemented	Sanitizer
CWE191	Integer Underflow	Unimplemented	Sanitizer
CWE469	Use of Pointer Subtraction to Determine Size	Unimplemented	Sanitizer
CWE364	Signal Handler Race Condition	Unimplemented	Signal/Race condition [59]
CWE464	Addition of Data Structure Sentinel	Unimplemented	Struct
CWE252	Unchecked Return Value	Unimplemented	Symbolic-size Memory Allocation [56]
CWE253	Incorrect Check of Function Return Value	Unimplemented	Symbolic-size Memory Allocation [56]
CWE832	Unlock of Resource That is Not Locked	Unimplemented	Thread
CWE366	Race Condition Within Thread	Unimplemented	Thread/Race condition [59]
CWE479	Signal Handler Use of Non Reentrant Function	Unimplemented	Thread/Syscall [63]
CWE23	Relative Path Traversal	Unimplemented	Traversal [22]
CWE36	Absolute Path Traversal	Unimplemented	Traversal [22]
CWE197	Numeric Truncation Error	Unimplemented	Truncation
Out of Scope: CWE773 Missing Reference to Active File Descriptor or Handle, CWE391 Unchecked Error Condition, CWE440 Expected Behavior Violation, CWE570 Expression Always False, CWE367 TOC TOU, CWE390 Error Without Action, CWE398 Poor Code Quality, CWE400 Resource Exhaustion, CWE401 Memory Leak, CWE459 Incomplete Cleanup, CWE467 Use of sizeof on Pointer Type, CWE478 Missing Default Case in Switch, CWE480 Use of Incorrect Operator, CWE481 Assigning Instead of Comparing, CWE482 Comparing Instead of Assigning, CWE483 Incorrect Block Delimitation, CWE484 Omitted Break Statement in Switch, CWE500 Public Static Field Not Final, CWE506 Embedded Malicious Code, CWE510 Trapdoor, CWE526 Info Exposure Environment Variables, CWE546 Suspicious Comment, CWE561 Dead Code, CWE563 Unused Variable, CWE571 Expression Always True, CWE606 Unchecked Loop Condition, CWE666 Operation on Resource in Wrong Phase of Lifetime, CWE674 Uncontrolled Recursion, CWE681 Incorrect Conversion Between Numeric Types, CWE835 Infinite Loop		

allocator does not support unless KLEE itself runs out of memory. Some prior work [34] has extended KLEE to support this type of fault injection, however.

- **CWE511 (Logic Time Bomb)** requires manual changes to the test case to be detected with KLEE. The trigger source (e.g., the random number generator or a time source) must be symbolized using `klee_make_symbolic`.

There are 12,120 testcases in unsupported CWE categories, and 17,603 testcases in supported CWE categories. We did manual analysis of all missed bugs in the supported list. We will describe these in our evaluation section. We also make our dataset available so that other researchers can extend our work.

B. Real-World Programs

In these experiments, we injected bugs into larger programs that better reflect real-world software. We selected ten programs from `coreutils`, including `base64`, `who`, `uniq`, `cat`, `od`, `cut`, `pr`, `ptx`, `tail`, and `sum` and injected more than 2000 bugs. We selected 200 bugs with unique attack points from the generated bug corpus. Each bug comes with an input that triggers the bug; we additionally collect the concrete path taken by the program on the triggering input for use with concolic execution.

We note that KLEE has its own concolic mode, known as *seed mode*. In this mode, the user provides a `ktest` file with the appropriate concrete input. However, we found that in our testing, the input file alone was generally not sufficient to cause KLEE to follow the exact path taken by concrete execution. We instead implemented a concolic path tracing mode for KLEE, which we describe here.[4]

In our implementation of concolic execution, we first record the trace of running KLEE with concrete input, then direct the symbolic executor using the concrete path. In the first run, we record every branch encountered and log information such as the program counter, branch conditions, and source code line. We then re-run KLEE with symbolic input and follow the concrete trace to create constraint sets for each step. We modified the `Executor::fork` function so that instead of actually forking, KLEE picks the successor state that corresponds to the next entry in the trace. When KLEE reaches the memory bug, KLEE will send the constraint sets to the solver. If the solver raises an error state, it means that KLEE would be able to find the bug, if given sufficient time to search all paths. Otherwise, we mark the bug as missed, meaning that KLEE's environment model or execution were not sufficient to uncover the bug.

[4]Our Concolic KLEE can be found at https://github.com/yh570/Concolic_klee.git.

Since KLEE has a full concrete environment during the record run, some branches may have different results in an symbolic environment (e.g. open a symbolic or concrete file). In this case, we ignore paths in all libc functions during replay and assume that they behave the same, despite the difference between symbolic and concrete environment. An exception is the compare functions, e.g. `strcmp`, `memcmp`, etc. Concolic KLEE replay follows the branch conditions in such functions because they do not involve the environment and are essential for computing the correct path conditions.

IV. Evaluation

A. Identifying Soundness Issues with the LAVA Corpus

In our small program testing, we generated 159 `toy.c` programs with bugs. Each `toy.c` program contains one unique bug. We ran both KLEE v1.4 and KLEE v2.2 on each program with a timeout of 5 hours and the configuration parameters used in the original KLEE `coreutils` experiments, except that we increased the maximum instruction time to 30 seconds and the memory cap to 2 GB. Both KLEE were able to fully explore each program without reaching the 5 hour time limit. Using `gcov`, we verified that both KLEE were able to cover 97% of instructions,[5] including all lines containing our injected bugs. KLEE v1.4 detected the bug and generated a triggering test case for only 68 of the 159 programs (42.8%), and KLEE v2.2 detected 150 out of 159 programs (94.3%). Since we know that KLEE v1.4 could exhaustively cover the program, missed bugs here represent potential soundness issues.

To understand why KLEE v1.4 missed these bugs, we classified each bug based on its location in the program. All bugs were injected into the parameters of user-defined functions (i.e., those in the main source of the program) or in the arguments to an external function (such as a `libc` function).

72 bugs were injected into parameters of user-defined functions such as `consume_record` and `parse_record` in `toy.c`; KLEE v1.4 was able to find all but four of these. We examined these four in detail and found that they were missed because the trigger value was used as a floating-point value. When KLEE v1.4 encounters a symbolic value used as a floating-point value, it concretizes the symbolic data to the constant 0. As a result, KLEE v1.4 cannot reason about the constraints needed to set the input to the trigger value. We note that recent work [38] has extended KLEE v1.4 to support floating-point values, which should allow it to detect these bugs.

LAVA injected 87 bugs into the parameters of external functions—specifically, the `printf` function. KLEE v1.4 missed all of these. To exeucte symbolically in libc library, KLEE can symbolically execute some standard library functions using `klee-uClibc`, a version of `uClibc` that is compiled to LLVM bitcode and linked into the program under test. Although `klee-uClibc` does not include `printf` by default, it can be enabled at compile time. This allows KLEE v1.4 to detect an additional 31 bugs (bringing its overall accuracy up to 62%). These bugs appear similar to the bugs added to user-defined functions (they are also out-of-bounds errors due to the large offset added to the pointer by LAVA). As an example, Figure 1 shows a bug in the first parameter of `printf` function, which has a large offset added whenever the trigger condition is met.

```
printf("fdata = %f\n"+ \
 (lava_get(131))*(0x6c6175de==(lava_get(131)) \
 || 0xde75616c==(lava_get(131))), \
ent->data.fdata);
```

Fig. 1: A bug injected into the first argument to the `printf` function.

The cause of these misses is that KLEE v1.4 does not check pointers passed to external library calls for validity. There are good reasons for this: in some cases (for example, the `void *addr` argument to `mmap`), pointer values passed to functions are not expected to point to a valid object. As long as the called function does not dereference the pointer, this will not result in a runtime error, and raising an alarm would be considered a false positive. Nevertheless, in our case, it means that there are bugs that will cause crashes that KLEE v1.4 cannot detect.

To further improve bug-finding performance when dealing with external function calls, we added an additional check to KLEE v1.4 that validates each pointer argument to external function calls. We modified the function `Executor::executeCall` in KLEE, which is invoked whenever KLEE encounters an LLVM `call` instruction. As we noted previously, these checks can in some cases cause false positives; to avoid this, we implemented an exclude list that suppresses pointer checks on functions known to cause false positives such as `syscall` and `realloc`.

We compared KLEE v1.4 and our modified KLEE by running it on the same 159 buggy versions of `toy.c`. The result shows that the modified KLEE can detect significantly more bugs: the overall rate of discovered bugs increases from 62.3% to 91.8%. For the 87 bugs injected into the arguments of `printf`, the number of bugs found increases from 31 (35.6%) to 78 (89.6%); of the remaining 9 missed bugs, 5 are due to missing floating-point support.

KLEE v2.2 has already added support for checking pointers passed to external functions, which allows KLEE 2.2 to find 150 bugs out of 159 injected bugs. The 9 bugs missed by KLEE v2.2 are all caused by the lack of floating-point support.

B. Evaluation with Juliet Test Suite

A limitation of our use of LAVA in the previous section is that the bugs injected are all simple pointer errors. To understand how KLEE deals with a wider range of serious bugs (i.e., those that could cause security issues), we run both KLEE v1.4 and KLEE v2.2 with the Juliet Test Suite. For each buggy program, we allow KLEE to run for 5 minutes, with

[5]The missing 3% correspond to the call to `exit()`, which is not logged.

4/8/16/32/64/128 bytes of symbolic std input on `stdin` and 4/8/16/32/64/128 bytes symbolic file input. We also used use KLEE's uClibc model to handle libc functions, with `printf` enabled. Due to space limitations, the rest of this section will only focus on the KLEE v2.2, but full results can be found in our released data.

In these experiments, KLEE can produce error reports of different kinds.

- Abort Failure: The program called `abort()`.
- Assertion Failure: An assertion failed.
- Divide Zero Error: A division or modulus by zero was detected.
- Execution Error: There was a problem that prevented KLEE from executing the program.
- External Function Error: The state depends on an unsupported external function.
- Free Error: Double or invalid `free()`.
- Memory Error: Write to read-only data.
- Model Failure: KLEE has lost precision (typically through concretization).
- Pointer Error: Stores or loads of invalid memory locations.

29,723 testcases in Juliet Test Suite could be correctly compiled with LLVM. We ran KLEE with Juliet Test Suite after filtering out those testcases that used known-unsupported functionality such as `socket` or that needed manual changes (e.g., making the return value of `rand` symbolic). This resulted in 33,533 KLEE-generated test reports: 893 false positive error reports, 13,972 false negative error reports and 18,668 true positives. We note that we have more reports than test cases because we configured KLEE to log *all* errors encountered rather than stopping after the first one. This means some test cases may have both false positives and false negatives; for example, KLEE generated three error reports for the `fgets_45.c` test case for CWE123 (Write What Where Condition), which includes two true-positive reports and one false-positive report. We only count a test case as a false negative if KLEE fails to generate any error report.

As previously mentioned, we divided the Juliet Test Suite into unsupported CWEs, which contain 12,120 testcases, and supported CWEs, which include 17,603 testcases. After filtering out those testcases in unsupported CWE categories, KLEE reports 732 false positive errors, 2,013 false negative errors, and 18,668 true positives from 17,603 supported testcases.

For each testcase, our evaluation script uses the ground truth for the location and type of bug in the program, and classifies KLEE's result as:

1) If the found bug is not in the "Bad" function, the error report will be marked as a **false positive**.
2) If KLEE did not find a bug (no matter true-positive or false-positive) in a buggy program, it will be marked as a **false negative**.

In the remainder of this section we examine common causes for these false positives and negatives.

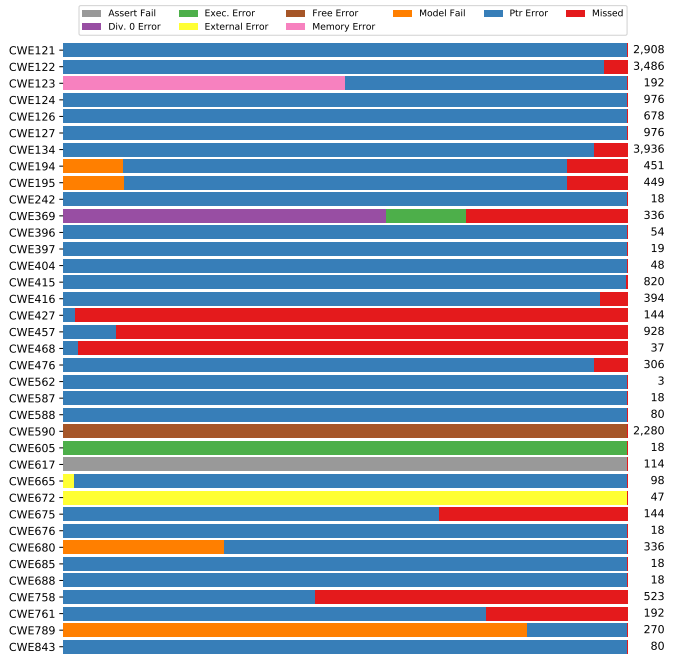

Fig. 2: KLEE v2.2 results for supported CWE entries in Juliet

1) False Positives: For many programs, we found that KLEE could not execute the program correctly. Based on the error reported by KLEE we classify these failures into execution errors (e.g., unknown instructions, unsupported inline assembly, etc.), symbolic model errors (KLEE concretizes some symbolic state, preventing it from exploring the full program), and external function errors (passing a symbolic argument to an external function which is not supported by KLEE). We report these separately in Figure 2. We note that these results indicate areas where KLEE could be improved to support a wider range of programs.

Ignoring false positives caused by execution errors, there are 204 false-positive error reports in our experiments: four in CWE123 (Write What Where Condition), 12 in CWE134 (Uncontrolled Format String), 54 in CWE396 (Catch Generic Exception), 19 in CWE397 (Throw Generic Exception), 96 in CWE675 (Duplicate Operations on Resource) and eighteen in

TABLE III: Summary of 159 toy.c bugs with different KLEE

Function Category	KLEE v1.4	KLEE v1.4 with printf enabled	Modified KLEE v1.4	KLEE v2.2	Total Bugs
user-defined function	68	68	68	68	72
printf	0	31	78	82	87
total	68	99	146	150	159

TABLE IV: False positives and negatives for CWE entries in Juliet

CWE ID	FP	TP	FN	Total	Cause of FP/FN	CWE ID	FP	TP	FN	Total	Cause of FP/FN
CWE122	0	3342	144	3486	64-bit system	CWE123	4	188	0	192	overwrite a pointer
CWE134	12	3688	236	3936	environment variable	CWE194	48	355	48	451	negative malloc
CWE195	48	353	48	449	negative malloc	CWE369	48	192	96	336	floating point
CWE396	54	0	0	54	C++ support	CWE397	19	0	0	19	C++ support
CWE415	0	818	2	820	only contains 'GOOD'	CWE416	0	375	19	394	regression bug
CWE427	0	3	141	144	environment variable	CWE457	0	88	840	928	out of scope
CWE468	0	1	36	37	out of scope	CWE476	0	288	18	306	out of scope
CWE605	18	0	0	18	execution errors	CWE665	2	96	0	98	external function call
CWE672	47	0	0	47	external function call	CWE675	96	0	48	144	double close
CWE676	18	0	0	18	C++ support	CWE680	96	240	0	336	model failure
CWE758	0	234	289	523	sanitizer	CWE761	0	144	48	192	environment variable
CWE789	222	48	0	270	model failure						

CWE676 (Use of Potentially Dangerous Function). We found that all target programs in CWE396, CWE397, CWE675, and CWE676 use C++, and although KLEE v2.2 has some initial support for C++ library calls using libc++, C++ programs that call external functions may still fail, resulting in a false positive. We note that some work has extended KLEE to support C++ features, but this has not yet been merged into mainline KLEE [62].

For the remaining false positives, four false positives in CWE123 turn out to be an artifact of allowing KLEE to continue after finding an error. For these test cases, KLEE also reports true positives (i.e., it finds the bug). However, because the vulnerability in the program (seen in lines 23–24 in Figure 3) allows writing an arbitrary value anywhere in memory, after KLEE continues past the pointer error it generates an input that uses the write-what-where vulnerability to overwrite a pointer in another data structure, which eventually causes KLEE to report a crash when the program exits.

The twelve false positives in CWE134 are caused by a bug in Juliet Test Suite. The affected test cases contain a memory safety error (effectively a `printf("%s")` call with no other argument, which causes printf to read out of bounds) in the "good" function, which was not intended to be buggy. KLEE correctly detects this error.

2) False Negatives: To better understand the false-negative error reports, we manually explored the missed bugs from each supported CWE category. We found that the main cause of missed bugs is because the testcases of the Juliet Test Suite were designed for static analyzers (marked as `out of scope` in Table IV). For example, the missed testcases in CWE476 (NULL Pointer Dereference) check for NULL *after* dereferencing a (valid) pointer. Juliet marks this NULL check as a flaw because it is unnecessary, but this kind of flaw obviously can't be detected by KLEE. Besides these missed bugs, we also find some genuine false negatives: unimplemented features such as `__isoc99_fscanf` (which might be considered as a bug since `fscanf` is supported with KLEE-uClibc), symbolic sizes to `malloc`, missing support for system calls, use of floating point, use of multithreading, and use of environment variables.

Some other missing cases are caused by the test case's design. The 144 missing cases from CWE122 (Heap Based Buffer Overflow) occur because the test cases allocate space for a pointer, and then try to store a double into that memory. On a 32-bit system, this will cause an overflow, but our test system is 64-bit, and hence no overflow occurs. For CWE194 (Unexpected Sign Extension) and CWE195 (Signed to Unsigned Conversion Error), KLEE does detect the negative malloc as a very large allocation, but simply reports it as a warning rather than an error.

We found one limitation in the way KLEE detects use-after-

```
1  #include <klee/klee.h>
2  typedef struct _linkedList
3  {
4      struct _linkedList *next;
5      struct _linkedList *prev;
6  } linkedList;
7  typedef struct _badStruct
8  {
9      linkedList list;
10 } badStruct;
11 static linkedList *linkedPrev, *linkedNext;
12 void CWE123_Write_What_Where_Condition_bad()
13 {
14     badStruct a;
15     linkedList head = { &head, &head };
16     a.list.next = head.next;
17     a.list.prev = head.prev;
18     head.next = &a.list;
19     head.prev = &a.list;
20     klee_make_symbolic(&a, sizeof(a), "a");
21     linkedPrev = a.list.prev;
22     linkedNext = a.list.next;
23     linkedPrev->next = linkedNext;
24     linkedNext->prev = linkedPrev;
25 }
26 int main(int argc, char * argv[])
27 {
28     CWE123_Write_What_Where_Condition_bad();
29     return 0;
30 }
```

Fig. 3: CWE123 minimal buggy example

```c
#include <stdio.h>
#include <stdlib.h>
char *a() {
  char b;
  char *c = malloc(2);
  if (c) {
    for (int d; 0;) ;
    free(c);
    return c;
  }
  return NULL;
}
int main() {
  char *c = a();
  puts(c);
}
```

Fig. 4: CWE416 minimal buggy example

free bugs in the false-negative reports of CWE416 (Use After Free). KLEE's memory error detector checks whether memory accesses refer to an object that is *currently* valid; this means that if the allocator places an object at the same address as a previously freed object, accesses to the freed object will be considered valid. A minimized version of of this test case can be seen in Figure 4; here, after the call to free, a stack allocation takes the same slot, so the invalid access in `puts` is missed. We reported this issue to the KLEE developers, who confirmed it as a known limitation, and suggested the use of `--allocate-determ`, which never re-uses memory, as a workaround.

A limitation of Juliet is that it is hard to determine whether false-negatives are caused by limited resources, experimental configuration that needs to be tuned, solver issues, or unimplemented features in KLEE. For example, CWE242 (Use of Inherently Dangerous Function) was missed with four or eight bytes of symbolic input, but correctly detected with 16 symbolic bytes.

Because the Juliet Test Suite is designed for static analyzers, triggering inputs are not provided for each testcase, which precludes us from using concolic execution to explore just the path that leads to the bug. This means that we lack the ability to reliably distinguish between false negatives caused by resource limitations and those caused by soundness issues. In the next section, we show that this problem can be overcome by using bugs injected with LAVA in combination with concolic execution.

C. Evaluation with Concolic Klee

We saw in the previous sections that synthetic bugs can be used to identify limitations and soundness issues in KLEE. However, as we saw with the Juliet Test Suite, it can be difficult to distinguish false negatives caused by resource limitations (e.g., timeouts and insufficient symbolic input) from actual soundness problems. In addition, the programs tested thus far are small and artificially constructed, and so may not be representative of real-world programs.

To address these issues, here we evaluate KLEE with real-world programs from the `coreutils` suite and use concolic execution with LAVA-injected bugs, allowing us to identify which missed bugs actually indicate soundness issues. We use our concolic execution mode to force KLEE to follow the path taken by an input known to trigger the bug, which factors out the difficulty of path search from the bug-finding task.

In our experiments, the KLEE v2.2 has a better result than KLEE v1.4, but the majority of results are overlapped. Since we can't determine the missing bugs are caused by path explosion or unimplementation problem, we focus on the experiment result from our concolic KLEE. Our concolic KLEE found almost all missed bugs in many of the utilities, including `base64`, `cat`, and `od`. However, some programs such as `pr`, `who`, `nl`, and `uniq` still have significant numbers of missed bugs. These results are summarized in Table V.

We found that the missed bugs here are caused by implementation differences between glibc and KLEE's uClibc. Although we injected our bugs and compiled our test programs using glibc, KLEE uses a different standard library. The two standard library implementations have the same API, but use different data structures to implement things like the libc `FILE` structure. For example, a buggy program compiled with glibc may corrupt a pointer member such as the base of the I/O write buffer within the `FILE` struct. However, in uClibc, the member at that offset is not a pointer, so the corruption goes undetected and the bug will not be triggered.

To confirm our findings, we compiled our buggy `coreutils` with uClibc. Because some of our buggy programs used struct `_IO_` which is not contained in uClibc, they cannot be compiled with uClibc. Table V shows that if the buggy program can be compiled with uClibc, the concolic KLEE can find all bugs using its uClibc environment.

Another interesting result is `uniq`, where concolic KLEE finds fewer bugs than plain (non-concolic) KLEE. We examined all detected and missed bugs in `uniq`, and found that the inputs generated by KLEE for the five bugs found by plain KLEE cannot be reproduced with the natively-compiled program. However, this turns out to again be caused by differences between uClibc and glibc: KLEE found an input that causes a crash when the program is built with uClibc, but does not trigger the bug under glibc.

In summary, when using KLEE with real-world programs, we find that its detection capabilities are robust, but that subtle differences can arise from implementation differences between glibc and uClibc. This may still have practical consequences, however, since it is not always easy or practical to build programs with uClibc, and glibc cannot, at present, be compiled to LLVM bitcode.

V. Discussion and Limitations

We believe our study offers several interesting conclusions for both practitioners as well as researchers in software engineering and developers of tools like KLEE.

First, we find that although there are many pitfalls that we encounter when trying to use KLEE with a diverse set of software, these limitations mostly correspond to well-known issues (lack of floating point support, problems with external function calls, unsupported language features, etc.). One exception we found to this is a regression bug in the latest KLEE, while the old version of KLEE can correctly detect the bug; and another exception is the mismatch between uClibc and glibc, which could be addressed by adding a recommendation to the documentation that users build their code with uClibc whenever possible. Overall, assuming one is aware of KLEE's current documented limitations, it is relatively straightforward to understand whether it will work with a given program under test. We take this as a sign of KLEE's maturity; end users are relatively unlikely to encounter breakage as long as they stay within the documented bounds of the tool.

Second, we found that the core functionality of KLEE is generally robust. The majority of issues we find are not bugs in KLEE itself, but instead arise from missing features or interactions with other libraries. This suggests that in terms of making KLEE more useful for testing real-world programs, development effort might better be spent on tasks like implementing more system calls, adding support for multithreaded programs, and so on. We also note that since some of this functionality (such as floating point, threading, and C++ support) is present in research-oriented forks of KLEE, upstreaming and maintaining this code in the main KLEE tree may be an attractive way to support a much wider range of programs.

Finally, we believe that our experience with the Juliet Test Suite demonstrates the need for new test suites tailored toward testing symbolic execution engines such as KLEE rather than static analysis tools. Such a test suite should:

- Come with triggering inputs to serve as ground truth.
- Represent a wide range of bugs. The current Juliet Test Suite generally does an excellent job here by including a large number of categories from the Common Weakness Enumeration (CWE).
- Focus on bug classes that can be detected using symbolic execution, perhaps in conjunction with sanitizers (i.e., avoid including bug classes like "dead code" or "suspicious comment")—or at least ensure test cases intended only for static analyzers are well-marked.
- Standardize the inputs and sources of nondeterminism used by the test cases, so that they can be easily symbolized by tools.

Most existing corpora are small, lack triggering inputs, and are typically not frequently updated. Automated bug synthesis tools [21], [29], [43], [46] may help with the creation of better test suites, since they can generate an endless stream of bugs, with known locations, triggering inputs, and representing the complexity of real programs.

We note that our current study still has some limitations. In particular, our "large" program corpus is derived from `coreutils`, which has been extensively used in prior work, including the original KLEE paper [12]. As a result, these programs may represent a best case scenario for KLEE: any functionality required to correctly execute them is likely to have been implemented already. We hope to extend our concolic execution approach to cover a more diverse set of real-world programs in future work.

As we said in Introduction section, our goal to is to show that both manually-made and automatic generated synthetic bugs can be used to characterize the strengths and weaknesses of bug-finding tools, and help to find areas where they could be improved. We plan to apply the same methodology and dataset to other bug finding tools.

VI. Related Work

Although symbolic execution was first developed in the 1970s [9], [16], [26], [32], it did not see wide usage until the development of practical SMT solvers in the 2000s, when systems such as EXE [13] and KLEE [12] were introduced. This has led to fifteen years of productive research into the use of symbolic execution for program testing and bug-finding. Despite its success, many challenges remain for symbolic execution, which are summarized in Baldoni et al.'s 2018 survey of symbolic execution [5].

To deal with real-world software, symbolic execution systems may sometimes choose to sacrifice soundness, as in

TABLE V: Concolic KLEE experiment results

	Injected Bugs	Found (v1.4)	Found (v2.2)	Found (Concolic)	uClibc Compiled?	Triggered (uClibc)
base64	28	22	20	28	yes	28
cat	8	6	6	7	yes	7
cut	3	1	1	2	no	-
od	24	9	13	22	yes	22
pr	2	0	0	0	yes	0
ptx	20	3	3	20	yes	20
sum	7	1	1	6	no	-
tail	8	3	6	8	yes	8
uniq	12	6	6	1	no	-
who	119	5	5	5	yes	5
unexpand	4	0	1	4	no	-
head	1	0	1	1	yes	1
nl	3	3	3	3	no	-
expand	4	0	0	4	no	-

Rutledge and Orso's PG-KLEE [47], or to test only small portions of the program, as in UC-KLEE [44]. In this work we focus on soundness issues in KLEE itself.

There are well-known gaps in KLEE's execution model, such as a lack of floating point support and missing support for C++ code. Some research forks of KLEE have sought to address these issues: Yoshida et al. introduced KLOVER [62], adds support for C++ programs, and Liew et al. [38] presented two independent implmentations that add floating point support to KLEE. As of October 2021, neither have been merged into mainline KLEE.

Testing the accuracy and correctness of bug-finding tools is difficult because ground truth is often scarce; the underlying distribution of bugs in real-world programs is not known. Prior work has attempted to overcome this by manually injecting bugs [66], using historical vulnerability corpora [30], evaluating the original program without any bugs injected [11], [14], [33], [37], [54], [65] and by finding previously unknown bugs [18], [31].

In recent years, some of test corpus with real bugs were developed. Hazimeh et al. developed Magma [25], which contains 118 validated CVE bugs. Su et al. [55] developed 52 real and reproducible crash bugs for testing. However, we found most existing corpora with real bugs contain few test cases. They also are typically not updated, meaning that bug-finders may overfit to the evaluation benchmark over time. For this situation, synthetic bug corpora can help to create ground truth corpora with frequently-updated testcases, as automatic bug injection tools can provide fresh corpora by automatically injecting bugs into real programs.

One well-known synthetic corpus is Juliet Test Suite [3]. In 2011, Juliet Test Suite v1.0 for C/C++ contained 45,309 testcases and covered 117 Common Weakness Enumeration (CWE) categories; in 2017, Juliet Test Suite v1.3 for C/C++ contained 64,099 testcases and covered 118 CWE categories. Using Juliet, Wagner et al. [57] tested five source code scanners. Juliet has also been used to test novel bug-finding systems: Li et al. [36], Zhang et al. [64] and Li et al. [35] each evaluated their systems using Juliet.

Aside from Juliet, Shiraishi et al. [52] presented a corpus of 1,276 simple synthetic programs aim to mimic common features of automotive software. Galea et al. [23] developed the `Hemiptera` corpus, which consists of over 130 bugs in real world software, and used it to evaluate KLEE. Their evaluation focuses on identifying roadblocks that prevent KLEE from finding the bugs in their corpus, and then providing manual assistance to KLEE that allow it to discover the bugs.

Kapus and Cadar [28] tested several symbolic execution tools, including KLEE, using CSmith [61], a random program generator. Using 700,000 generated programs they compare concrete and symbolic executions of the same program to identify bugs in the engines under test. The most differences between random program generator and automatic bug generation are the kinds of programs generated by CSmith do not reflect code patterns used by real software, while automatic bug generators can inject synthetic bugs in the real program. What's more, automatic bug generators always provide the triggering input while inject the bug into the real programs, which can help to distinguish whether misses are caused by resource limitations or soundness problems.

Finally, our work implements a concolic mode for KLEE in order to distinguish between bugs missed by resource limits. Other work has also added concolic execution to KLEE, including ZESTI [40] and S2E [15]. We implement our own concolic execution mode because, on the one hand, ZESTI is based on a relatively old version of KLEE, and on the other, S2E is made for whole-system execution.

VII. Conclusion

In this paper, we have explored the use of synthetic bugs to test and improve bug-finding tools, leveraging LAVA's ability to produce ground truth corpora as well as the manually-created Juliet Test Suite. In a small program for which we can achieve full coverage, we found some minor soundness issues related to floating point support and external function calls, and made changes to KLEE v1.4 that increase its bug-finding rate from 62.3% to 91.8% on our LAVA dataset. We also identified which cases in the Juliet Test Suite v1.3 were applicable to KLEE, and used it to identify and diagnose both false positives and false negatives. Finally, we used concolic execution to examine its performance on real world programs, finding that discrepancies between uClibc (KLEE's libc implementation) and glibc can cause KLEE to miss some bugs or report errors that are hard to reproduce. Documenting such discrepancies and making it easier to build programs against uClibc could make it easier to use KLEE.

Overall, our results provide promising evidence that although synthetic bugs are not identical to naturally occurring bugs in many ways, they can still provide useful insights into bug-finding tools, and allow empirical investigation into both bugs and bug-finders with large amounts of data. In addition, they serve a valuable diagnostic purpose: when a bug-finder misses an injected bug, the triggering input generated by LAVA can be used to help identify the precise reason the bug was missed, providing a clear path to improving bug-finding tools. We will release our full dataset in hopes that it can help guide further research and practice for symbolic execution-based bug-finders.

Acknowledgements

This research was supported in part by National Science Foundation (NSF) Award 1657199. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the NSF.

References

[1] afl, http://lcamtuf.coredump.cx/afl/.
[2] Darpa cgc chanllenges, https://github.com/lungetech/cgc-challenge-corpus.
[3] Juliet test suite v1.3, https://samate.nist.gov/sard/testsuite.php.
[4] Linux flaw, https://github.com/mudongliang/linuxflaw.
[5] Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *ACM Comput. Surv. 51*, 3 (2018).

[6] BARRETT, C., DE MOURA, L., AND STUMP, A. Design and results of the first satisfiability modulo theories competition (smt-comp 2005). *Journal of Automated Reasoning 35*, 4 (Nov 2005), 373–390.

[7] BARRETT, C., DE MOURA, L., AND STUMP, A. Smt-comp: Satisfiability modulo theories competition. In *Computer Aided Verification* (Berlin, Heidelberg, 2005), Springer Berlin Heidelberg, pp. 20–23.

[8] BERGAN, T., GROSSMAN, D., AND CEZE, L. Symbolic execution of multithreaded programs from arbitrary program contexts. *ACM SIGPLAN Notices 49* (12 2014), 491–506.

[9] BOYER, R. S., ELSPAS, B., AND LEVITT, K. N. Selecta formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software* (New York, NY, USA, 1975), Association for Computing Machinery, p. 234245.

[10] BUCUR, S., URECHE, V., ZAMFIR, C., AND CANDEA, G. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 183–198.

[11] BUGRARA, S., AND ENGLER, D. Redundant state detection for dynamic symbolic execution. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 199–211.

[12] CADAR, C., DUNBAR, D., AND ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 209–224.

[13] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur. 12*, 2 (Dec. 2008).

[14] CHIPOUNOV, V., GEORGESCU, V., ZAMFIR, C., AND CANDEA, G. Selective Symbolic Execution. In *5th Workshop on Hot Topics in System Dependability (HotDep)* (2009).

[15] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2e: a platform for in-vivo multi-path analysis of software systems. pp. 265–278.

[16] CLARKE, L. A. A program testing system. In *Proceedings of the 1976 Annual Conference* (New York, NY, USA, 1976), ACM '76, Association for Computing Machinery, p. 488491.

[17] CONVERSE, H. E. Non-semantics-preserving transformations for higher-coverage test generation using symbolic execution.

[18] CORIN, R., AND MANZANO, F. A. *Taint Analysis of Security Code in the KLEE Symbolic Execution Engine*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 264–275.

[19] CUOQ, P., MONATE, B., PACALET, A., PREVOSTO, V., REGEHR, J., YAKOBOWSKI, B., AND YANG, X. Testing static analyzers with randomly generated programs. In *Proceedings of the 4th International Conference on NASA Formal Methods* (Berlin, Heidelberg, 2012), NFM'12, Springer-Verlag, pp. 120–125.

[20] DEMILLO, R., LIPTON, R., AND SAYWARD, F. Hints on test data selection: Help for the practicing programmer. *IEEE Computer 11*, 4 (1978), 34–41.

[21] DOLAN-GAVITT, B., HULIN, P., KIRDA, E., LEEK, T., MAMBRETTI, A., ROBERTSON, W., ULRICH, F., AND WHELAN, R. LAVA: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)* (May 2016), pp. 110–121.

[22] FLANDERS, M. A simple and intuitive algorithm for preventing directory traversal attacks. *CoRR abs/1908.04502* (2019).

[23] GALEA, J., HEELAN, S., NEVILLE, D., AND KROENING, D. Evaluating manual intervention to address the challenges of bug finding with KLEE. *CoRR abs/1805.03450* (2018).

[24] HAMLET, R. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering SE-3*, 4 (1977), 279–290.

[25] HAZIMEH, A., HERRERA, A., AND PAYER, M. Magma. *Proceedings of the ACM on Measurement and Analysis of Computing Systems 4*, 3 (Nov 2020), 129.

[26] HOWDEN, W. E. Experiments with a symbolic evaluation system. In *Managing Requirements Knowledge, International Workshop on* (Los Alamitos, CA, USA, jun 1976), IEEE Computer Society, p. 899.

[27] HULIN, P., DAVIS, A., SRIDHAR, R., FASANO, A., GALLAGHER, C., SEDLACEK, A., LEEK, T., AND DOLAN-GAVITT, B. Autoctf: Creating diverse pwnables via automated bug injection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)* (Vancouver, BC, 2017), USENIX Association.

[28] KAPUS, T., AND CADAR, C. Automatic testing of symbolic execution engines via program generation and differential testing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2017), pp. 590–600.

[29] KASHYAP, V., RUCHTI, J., KOT, L., TURETSKY, E., SWORDS, R., PAN, S. A., HENRY, J., MELSKI, D., AND SCHULTE, E. Automated customized bug-benchmark generation. In *19th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2019), pp. 103–114.

[30] KASS, M. Nist software assurance metrics and tool evaluation (samate) project. In *In Workshop on the Evaluation of Software Defect Detection Tools* (2005).

[31] KIEZUN, A., GANESH, V., GUO, P. J., HOOIMEIJER, P., AND ERNST, M. D. Hampi: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (New York, NY, USA, 2009), ISSTA '09, ACM, pp. 105–116.

[32] KING, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (July 1976), 385–394.

[33] KUZNETSOV, V., KINDER, J., BUCUR, S., AND CANDEA, G. Efficient state merging in symbolic execution. *SIGPLAN Not. 47*, 6 (June 2012), 193–204.

[34] LE, H. M., HERDT, V., GROSSE, D., AND DRECHSLER, R. Resilience evaluation via symbolic fault injection on intermediate code. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)* (2018), pp. 845–850.

[35] LI, H., OH, J., OH, H., AND LEE, H. Automated source code instrumentation for verifying potential vulnerabilities. pp. 211–226.

[36] LI, Y., JI, S., LV, C., CHEN, Y., CHEN, J., GU, Q., AND WU, C. V-fuzz: Vulnerability-oriented evolutionary fuzzing.

[37] LI, Y., SU, Z., WANG, L., AND LI, X. Steering symbolic execution to less traveled paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (New York, NY, USA, 2013), OOPSLA '13, ACM, pp. 19–32.

[38] LIEW, D., SCHEMMEL, D., CADAR, C., DONALDSON, A., ZÄHL, R., AND WEHRLE, K. Floating-point symbolic execution: A case study in n-version programming. In *IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)* (11 2017), pp. 601–612.

[39] MARINESCU, P. D., AND CADAR, C. Make test-zesti: A symbolic execution solution for improving regression testing. In *Proceedings of the 34th International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE '12, IEEE Press, pp. 716–726.

[40] MARINESCU, P. D., AND CADAR, C. make test-zesti: A symbolic execution solution for improving regression testing. In *International Conference on Software Engineering (ICSE 2012)* (6 2012).

[41] MITRE CORP. CWE: Common weakness enumeration. https://cwe.mitre.org/.

[42] PALIKAREVA, H., AND CADAR, C. *Multi-solver Support in Symbolic Execution*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 53–68.

[43] PEWNY, J., AND HOLZ, T. Evilcoder: Automated bug insertion, 2020.

[44] RAMOS, D. A., AND ENGLER, D. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 49–64.

[45] RANISE, S., AND TINELLI, C. The smt-lib format: An initial proposal. In *In PDPAR* (2003), pp. 94–111.

[46] ROY, S., PANDEY, A., DOLAN-GAVITT, B., AND HU, Y. Bug synthesis: Challenging bug-finding tools with deep faults. ESEC/FSE 2018, ACM.

[47] RUTLEDGE, R., AND ORSO, A. Pg-klee: Trading soundness for coverage. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (2020), pp. 65–68.

[48] SAUDEL, F., AND SALWAN, J. 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* (2015), SSTIC, pp. 31–54.

[49] SEN, K. Dart: Directed automated random testing. In *Hardware and Software: Verification and Testing* (Berlin, Heidelberg, 2011), K. Namjoshi, A. Zeller, and A. Ziv, Eds., Springer Berlin Heidelberg, pp. 4–4.

[50] SEN, K., AND AGHA, G. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*

(Berlin, Heidelberg, 2006), T. Ball and R. B. Jones, Eds., Springer Berlin Heidelberg, pp. 419–423.

[51] SEN, K., MARINOV, D., AND AGHA, G. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE* (2005), M. Wermelinger and H. Gall, Eds., pp. 263–272.

[52] SHIRAISHI, S., MOHAN, V., AND MARIMUTHU, H. Test suites for benchmarks of static analysis tools. In *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (2015), pp. 12–15.

[53] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting fuzzing through selective symbolic execution.

[54] SU, T., PU, G., MIAO, W., HE, J., AND SU, Z. Automated coverage-driven testing: combining symbolic execution and model checking. *Science China Information Sciences 59*, 9 (2016), 98101.

[55] SU, T., WANG, J., AND SU, Z. Benchmarking automated gui testing for android against real-world bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2021), ESEC/FSE 2021, Association for Computing Machinery, p. 119130.

[56] TRTÍK, M., AND STREJČEK, J. Symbolic memory with pointers. In *Automated Technology for Verification and Analysis* (Cham, 2014), F. Cassez and J.-F. Raskin, Eds., Springer International Publishing, pp. 380–395.

[57] WAGNER, A., AND SAMETINGER, J. Using the juliet test suite to compare static security scanners. In *2014 11th International Conference on Security and Cryptography (SECRYPT)* (2014), pp. 1–9.

[58] WANG, X., ZHANG, L., AND TANOFSKY, P. Experience report: How is dynamic symbolic execution different from manual testing? a study on klee. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2015), ISSTA 2015, ACM, pp. 199–210.

[59] WANG, Y., GAO, F., WANG, L., YU, T., ZHAO, J., AND LI, X. Automatic detection, validation and repair of race conditions in interrupt-driven embedded software. *IEEE Transactions on Software Engineering* (2020), 1–1.

[60] XU, H., ZHAO, Z., ZHOU, Y., AND LYU, M. R. On benchmarking the capability of symbolic execution tools with logic bombs. *CoRR abs/1712.01674* (2017).

[61] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in c compilers. vol. 46, pp. 283–294.

[62] YOSHIDA, H., LI, G., KAMIYA, T., GHOSH, I., RAJAN, S., TOKUMOTO, S., MUNAKATA, K., AND UEHARA, T. Klover: Automatic test generation for c and c++ programs, using symbolic execution. *IEEE Software 34*, 5 (2017), 30–37.

[63] YU, B., YANG, Q., AND SONG, C. *State Consistency Checking for Non-reentrant Function Based on Taint Assisted Symbol Execution*. 06 2019, pp. 498–508.

[64] ZHANG, B., YE, J., BI, X., FENG, C., AND TANG, C. Ffuzz: Towards full system high coverage fuzz testing on binary executables. *PLoS One 13*, 5 (2018), e0196733.

[65] ZHANG, C., GROCE, A., AND ALIPOUR, M. A. Using test case reduction and prioritization to improve symbolic execution. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2014), ISSTA 2014, ACM, pp. 160–170.

[66] ZITSER, M., LIPPMANN, R., AND LEEK, T. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2004), SIGSOFT '04/FSE-12, ACM, pp. 97–106.