A Hybrid Symbolic Execution Assisted Fuzzing Method

Li Zhang
Institute for Infocomm Research
A*STAR
Singapore
zhang-li@i2r.a-star.edu.sg

Vrizlynn L. L. Thing
Institute for Infocomm Research
A*STAR
Singapore
vriz@i2r.a-star.edu.sg

Abstract—We present a new automated method for efficient detection of security vulnerabilities in binary programs. This method starts with a bounded symbolic execution of the target program so as to explore as many paths as possible. Constraints of the explored paths are collected and solved for inputs. The inputs will then be fed to the following interleaved coverage-based fuzzing and concolic execution. As the paths explored by the bounded symbolic execution may cover some unique paths that can be rarely reached by random testing featured fuzzing and locality featured concolic execution, the efficiency and effectiveness of the overall exploration can be greatly enhanced. In particular, the bounded symbolic execution can effectively prevent the fuzzing guided exploration from converging to the less interesting but easy-to-fuzz branches.

Keywords—fuzz testing, symbolic execution, software testing, vulnerability

I. INTRODUCTION

Software vulnerabilities are ever present risks that may be exploited by an attacker to steal sensitive data or gain other strategic advantages. A very recent lesson is the Windows vulnerability based WannaCry ransomware attack, which hit more than 75,000 computers in 99 countries within a day [1]. With the rolling out of internet of things, an increasing number of devices running potentially vulnerable software are becoming reachable to cybercriminals. It is hence in dire need to uncover the exploitable vulnerabilities and fix them before they are utilized by the attackers.

In many scenarios, the source code of the program to be vetted is not available. For instance, the source code of third-party components or even components from different groups of the same organization [2]. As a result, the binary code is often the only target. In fact, since it is what gets executed, evaluation based on the binary code can provide ground truth. On the contrary, source code examination may result in wrong conclusions about the software due to compiler errors and post-build transformations [2, 3]. This paper presents an efficient method to detect the software bugs from the binary code, which is based on complementary use of coverage-based fuzzing and hybrid symbolic execution.

Fuzzing is an automated black-box testing method that uses a fuzzer to randomly mutate or generate input test cases and feeds them to the target program for possible security exceptions (in most cases, crashes) [4]. Since its emergence, fuzzing has shown to scale well for large programs and be remarkably effective in revealing software vulnerabilities [5]. Nonetheless, fuzzing shares a well-known limitation with other black-box testing methods, i.e., limited code coverage. The reasons are two-fold. Firstly, most programs have a massive input space, and fuzzing can only explore a small segment of the space. Secondly, the randomly generated test cases from fuzzing are very likely to fail to pass through complex checks in the program. This limitation means that a large portion of the program code may not be exercised and serious security bugs that may reside in the non-exercised code portion will not be uncovered by fuzzing.

Coverage-based fuzzing attempts to improve the coverage with the help of lightweight (binary) instrumentation, where test cases that exercises a path with new basic blocks are retained as seeds for further fuzzing and the others will be discarded [6]. To some extent, this type of method is a decent answer to the question of how to explore the huge input space more efficiently. A good representative is AFL [7], which has discovered hundreds of high-impact bugs. Nonetheless, it is still difficult for the fuzzing tools to pass through complex checks in the program, which prevents them from exploring the code regions behind such checks.

Symbolic execution [8], which executes a program using symbolic inputs in place of concrete ones, is a good candidate to extend the reach of fuzzing beyond the complex checks. In particular, symbolic execution maintains for each explored path a Boolean formula that describes the path constraints of the branches taken along the path. If the path constraints are satisfiable, an SMT solver [9] can be used to generate concrete inputs that are guaranteed to execute the path. It is obvious that the complex checks that traps the exploration of fuzzing is easy to solve for symbolic execution.

Nonetheless, symbolic execution has its own limitations. As symbolic execution may fork a new execution instance at each encountered branch, the number of paths being explored will grow exponentially and may become prohibitively large at the end, which leads to the path explosion problem and...
seriously limits its scalability. A variant of symbolic execution is concolic execution, where symbolic execution traces the single path traveled by a concrete execution. The collected symbolic path constraints of the branches along the path are then incrementally negated, which results in new path constraints that can be solved for new test inputs that cover other paths. Although concolic execution mitigates the path explosion problem to a degree by trading completeness of exploration for scalability and performance, it is still only able to explore a small part of the program state space (typically the states close to the initial state) in practice [10].

The idea of utilizing symbolic or concolic execution to improve the coverage of fuzzing was firstly proposed in [10]. In this method, random testing based fuzzing is firstly performed from the initial program state; when it saturates, i.e., cannot reach any new coverage point, the concolic execution will be invoked from the present state to perform an exhaustive bounded search. Once a new coverage point is identified, the exploration will switch from concolic execution back to fuzzing. This method is able to significantly improve the test coverage compared to fuzzing or concolic testing alone. Nonetheless, due to the limit from the underlying symbolic execution engine, i.e., CUTE [11], this method requires the availability of source code.

Driller [12] takes advantage of a similar idea as [10], where the main exploration is offloaded to fuzzing and concolic execution is only invoked if its input reasoning ability is needed to guide the fuzzing to pass through complex checks. Due to the use of angr [13], a very recent symbolic execution engine, Driller is able to work on binary programs without the source code. Besides, instead of purely random testing, Driller uses the state-of-the-art coverage-based fuzzing tool AFL, which further enhances the testing efficiency.

Driller does not require any input file and starts fuzzing with a simple string “fuzz” being the seed. This is both its advantage and drawback. On one hand, test case generation is one of the most expensive tasks in software testing [14], it is very attractive that no high quality test case is required. On the other hand, as the direction of exploration is essentially decided by fuzzing, the lack of good fuzzing seeds may limit the testing coverage and hinder efficient detection of bugs. Although the interleaved concolic execution can improve the coverage in a way (i.e., by guiding the fuzzing to pass through complex checks), such exploration suffers from locality, since it is the paths from fuzzing that determine the neighborhood of the state space that will be explored [10, 15]. In other words, concolic execution may not be always effective in improving the test coverage of fuzzing. For instance, the experimental results of Driller show that concolic execution can only generate new inputs for 13 out of 41 DARPA CGC binaries that AFL got stuck on.

In view of this limitation, besides the interleaved concolic execution, we propose to perform a bounded symbolic execution before fuzzing. Due to the exhaustive nature of symbolic execution, an excellent coverage for the front part of the binary program can be achieved, meaning the unique paths in this code section that rarely can be reached by fuzzing will also be identified. After collecting as many such unique paths as possible under a pre-defined resource limit, concrete inputs that satisfies the path constraints will be recovered and then fed to the fuzzing tool.

Benefits from the addition of resource-bounded symbolic execution before fuzzing are twofold. Firstly, the importance of high quality seed files to the efficiency of fuzzing is well-known. The outputs of the added symbolic execution virtually act as such seeds, which can greatly improve the exploration efficiency. Please note that this is achieved without any external test case. Secondly, the added symbolic execution can help identify unique paths that may not be reached by fuzzing or locality featured concolic execution, which in turn will result in a better overall coverage.

Our idea is inspired by the hybrid fuzz testing in [16]. Nonetheless, hybrid fuzz testing just consists of the symbolic execution at the beginning and the following fuzzing. Without the interleaved concolic execution, it cannot handle the complex checks in the deeper code.

The primary contributions of this work are as follows:

- We propose to use both symbolic and concolic execution, i.e., hybrid symbolic execution, to work with coverage-based fuzzing. The bounded symbolic execution run before the fuzzing can facilitate fuzzing to cover more unique paths earlier, while the concolic execution can guide fuzzing to pass through complex checks. This enhanced combination can help further improve both the test coverage and efficiency, which in turn results in more effective and efficient detection of software vulnerabilities.
- Our implementation is based on AFL and angr, both of which support multiple architectures (e.g., x86, AMD64, ARM) and multiple platforms (e.g., Linux, Windows, Android). This makes it possible for our implementation to be used in different testing scenarios. Besides, our implementation is optimized to work in a single-machine environment, which can further enhance the applicability in various resource-constrained environments.

II. A Motivating Example

We illustrate the benefit of an additional bounded symbolic execution using a code snippet presented in Listing 1. The code snippet is constructed with several real-life code constructs that may hinder the progress of coverage-based fuzzers, such as magic bytes, deeper execution, and nested conditions [17]. Besides, a large for loop, which is usually deemed hard to solve for symbolic execution, is also added. Please note that our method does not require source code. The C code used here is just for better illustration.

The example program receives inputs from stdin and then executes different paths based on the input values. At line 11, there is a check of the first input variable against a magic number, i.e., 0x33050617. If this check fails, the program will immediately stop. In order to reach the region with bugs at line 20, the input needs to pass an additional complex check
at line 14 as well as passing the for loop at line 15-18 with the value of counter being 15.

If the program is tested by Driller, the fuzzing will be started first, where it may get stuck by the check at line 11. In this case, the concolic execution will be invoked, which can easily reason about the magic bytes. With the uncovered magic bytes, the exploration switches back to the fuzzing. Again, the fuzzing will find it difficult to pass through the check at line 14. However, as the fuzzing tool can find new paths in the else branch at line 24, it will direct the exploration that way without invoking concolic execution. This may significantly delay the discovery of the bugs at line 20. In fact, some programs may even design the else branch in such a way that fuzzing will continue to find new paths during the assigned testing time. In this case, concolic execution will never be invoked as a result, the bugs will never be detected.

```c
1 int main ( void ) {
2    int magic_1;
3    char buf[20];
4    int magic_2;
5    int i, counter = 0;
6    read(0, &magic_1, 4);
7    read(0, buf, 20);
8    read(0, &magic_2, 4);
9    if (magic_1 != 0x33050617)
10       EXIT_ERROR();
11    if (magic_2 != 0x33050617) {
12       for (i = 0; i < 20; i++) {
13          if (buf[i] == 'B')
14             counter++;
15       }
16       if (counter == 15)
17          ...some bugs here...
18       else {
19          ...some other tasks...
20       }
21       else {
22          ...some other tasks...
23       }
24    return 0;
25 }
Listing 1. A motivating example
```

In contrast, the bounded symbolic execution added in our method will explore both the if branches at line 11 and 14 and the else branch at line 24. The correspondingly recovered inputs will then be fed to the fuzzing tool, which will make the exploration in the if branch at line 20 and in turn the discovery of bugs happen much earlier.

The large for loop at line 15-18 used to be a big problem for symbolic execution due to the large number of paths. However, techniques such as veritesting [18] which performs smart path merging has facilitated symbolic execution to work efficiently with such complex code constructs. We ran the example program on our desktop for experiments (see section IV for detailed setup) and found that symbolic execution run with veritesting can recover the inputs that reach the buggy state within one minute.

III. THE PROPOSED METHOD

An overview of the proposed method is shown in Figure 1, where the testing starts with a resource bounded symbolic execution from the initial state of the program. The symbolic execution will run until a pre-defined threshold is reached. Then the constraints of the explored paths will be collected and solved for inputs by an SMT solver. These inputs will be fed to the fuzzing tool afterwards.

The second phase in our method is the interleaved fuzzing and concolic execution. With the recovered inputs from the previous symbolic execution, the fuzzing tool can easily explore some unique paths that were originally extremely difficult to reach. The fuzzing will continue until it saturates, i.e., cannot find any new path. In this case, concolic execution, will be invoked.

The concolic execution will trace each explored path that extended the code coverage during fuzzing. As concolic execution can easily resolve some complex code constructs (such as checks against magic bytes) that impede the exploration of fuzzing, some new code segments may be reached. Similar to the symbolic execution at the beginning, for each path traced by concolic execution, the path constraints will be extracted and solved for inputs. These inputs will also be fed to the fuzzing tool.

The above process will continue until a pre-defined condition, e.g., a time limit or discovery of a crash input, is satisfied. As most of the high-impact bugs will cause crash of the program, our method looks for crash inputs. Crash triage will be performed thereafter to analyze whether the identified crash inputs correspond to exploitable vulnerabilities.

In the remaining part of this section, the fuzzing process as well as the symbolic and concolic execution process will be illustrated in more details.

A. Fuzzing

The fuzzing tool utilized in our method is AFL. It is a coverage-based fuzzing tool with the core idea of an instrumentation-guided genetic algorithm. For binary programs without the source code, the instrumentation can be

![Fig. 1. Overview of the proposed method](image-url)
As the fuzzing progresses, AFL may not be able to derive any new input that can further expand the test coverage within a pre-defined number of mutations. In other words, the fuzzing has saturated. This is the time when concolic execution should be invoked. We adopt the same stuck heuristic as Driller to signal the invocation. Particularly, the pending_favs attribute, which indicates the number of mutated test cases that triggered new state transitions and will be used in future rounds of mutation, in the fuzzer_stats file from AFL is monitored. When this attribute drops to 0, our method will start to trace each input in the queue folder of the AFL master instance with concolic execution. Please note that in our method fuzzing can be run in parallel with concolic execution.

B. Symbolic and Concolic Execution

The bounded symbolic execution in our method is performed with angr. In angr, after the binary program is loaded, it will be translated into a side-effect-free intermediate representation (i.e., Valgrind VEX [20]) first. The lifted binary program is represented with a Project object, and its main interface for symbolic execution is named as PathGroup. The PathGroup interface provides an easy way to trace the hierarchy of paths as they split or merge. The symbolic execution techniques implemented in angr are mainly based on those proposed in Mayhem [21], which includes the index-based memory model and path prioritization strategies. Veritesting is available as an option for the PathGroup objects, which helps mitigate the problem of state explosion in loops by performing smart state merging. In fact, we observed that huge loops such as the one in Listing 2, which was unsolvable for symbolic execution before, can now be solved very quickly with the help of veritesting. Hence, the bounded symbolic execution in our method will run with this optimization technique. This will enable the bounded symbolic execution to explore a larger part of the target program.

For concolic execution, we take advantage of Driller’s concolic execution engine, which is also based on angr. Concolic execution traces each input in the queue folder of AFL master instance by pre-constraining that the symbolic input is equal to the concrete value of the actual input. Then the pre-constraining will be removed and the constraints for all the missed branches along the traced path will be solved for inputs if they are satisfiable. The recovered inputs will then be synced to the fuzzing process in the same way as the inputs from the bounded symbolic execution.

Concolic execution will be invoked every time the fuzzing saturates. To prevent it from tracing the same input in different rounds, the traced inputs is recorded in our method and will be skipped in the future rounds of concolic execution.

IV. EVALUATION AND DISCUSSION

The proposed method is implemented by utilizing the code from open-sourced Driller. As both AFL and angr support multiple architectures (e.g., x86, AMD64, and ARM) and multiple...
platforms (e.g., Linux, Windows, Android), it is possible to use our implementation in different testing scenarios. Unlike Driller which is implemented mainly for a computer cluster, our implementation is optimized to work in a single-machine environment. This optimization enables our method to be more applicable in often resource-constrained scenarios. However, it also means that our implementation cannot parallelize tasks like Driller when evaluating DARPA CGC binaries which needs to be run in a customized OS named DECREASE (built with a network of Vagrant boxes) [22].

The underlying symbolic execution engine angr is a very recent one with various advanced optimization techniques. As it is integrated with Z3 SMT solver [23], the extracted path constraints can be seamlessly solved for concrete inputs in angr. This saved us lots of effort to develop an input generator as in [16]. However, angr has not modeled all the system calls yet, which prevents our implementation from correctly handling binaries with missing system calls. In this section, we present some preliminary results from experiments on synthetic programs to assess the efficiency of our method. The machine used for all of the experiments is a desktop with Intel Core i7-6700 CPU @ 3.4 GHz (8 cores) with 16 GB RAM, and 64-bit Ubuntu Linux 14.04 LTS. For each test, five instances of AFL fuzzers were spawned.

The synthetic programs were created on top of the example program in Listing 1. We firstly added some tasks in the _else_ branch at line 24 and observed that the more tasks that can be easily explored by fuzzing in this branch, the later the _pending_favs_ attribute will drop to 0 and in turn the later the concolic execution will be invoked. In one of our experiment, fuzzing kept on exploring the _else_ branch before the test reaches a pre-defined timeout of 24 hours. We believe that some real-life programs have a similar feature, which makes the method of just interleaving fuzzing and concolic execution incapable of discovering the bugs hidden in other branches in time.

It was also observed that even though concolic execution was invoked, only the magic bytes at line 14 was uncovered and the fuzzing tool still need to spend a large amount of time to resolve the _for_ loop at line 15-18 and in turn pass the check at line 19. In contrast, the bounded symbolic running with veritesting technique was able to efficiently reason about the program and uncover the inputs that reach the buggy state at line 20. To quantitatively examine the difference, we performed a list of experiments, where the _else_ branch at line 24 is removed and a simple buffer overflow as shown in Listing 3 is added at line 20. We tested with the _for_ loop iterating from 0 to 20, 100, 200, and 500, respectively. Our method reported the bug in 1 minute for the first three cases and in 3 minutes for the last case, while the desktop version of Driller re-implemented by us spent around 60 minutes for the first case and cannot recover the inputs for the other cases within 24 hours. This shows that the bounded symbolic execution in our method not only can guide fuzzing to unique paths that are very hard to be reached by fuzzing alone, but also may efficiently reveal some bugs.

```
1 void crash()
2 {
3   int buf[4];
4   memset(buf, 0, 0x100);
5 }
```

Listing 3. The inserted code for simple buffer overflow

As a measure to further improve the performance, Driller proposes to perform short runs of symbolic execution around the path traced by concolic execution. However, the short run of symbolic execution which is performed after concolic execution still faces the problem of being possibly never invoked. Besides, unlike the bounded symbolic execution in our method which starts from the initial state of the program, the short run of symbolic execution shares the locality problem of concolic execution and may not be able to discover as many unique paths. Moreover, the short run of symbolic execution in Driller does not run with veritesting technique, which makes it feeble to resolve complex code constructs like the big _for_ loop. With that said, such an optimization measure can also be adopted in our method to prevent frequent switches between fuzzing and concolic execution.

V. RELATED WORKS

Utilizing symbolic execution to enhance the coverage of software testing has a history of more than four decades. Especially in the past ten years, due to the availability of powerful computers and constraint solvers, numerous methods and tools based mainly on symbolic execution [24–26] have been proposed. Representative ones include earlier EXE [27] and DART [28] which assumes access to the source code and more recent SAGE [29] and Mayhem [21] which just need the binary code for analysis.

Although various optimization techniques such as pruning unrealizable paths, bounding computational resources with search heuristics, and merging different paths with static program analysis have been proposed, the typical path explosion problem of symbolic execution (and its variant concolic execution) is only mitigated but not resolved. This makes it infeasible to test a whole large real-life program with just symbolic or concolic execution.

With this insight, several techniques [30–32] propose to utilize static program analysis or dynamic taint analysis to firstly identify the possibly vulnerable regions of the program and then steer symbolic execution only to the small regions for exhaustive testing. These techniques have been able to detect some specific classes of bugs in real-life programs. However, as they can only test the regions covered by the used test cases, a set of high-quality test cases is prerequisite. Besides, all these techniques require the source code.

Our method follows the line of [10, 12], which expands the test coverage with complementary use of fuzzing and concolic execution. Nonetheless, our method moves one step further by performing a bounded symbolic execution before the fuzzing. We argue that although symbolic execution cannot completely explore a complex program, the various optimization techniques has enabled it to efficiently handle complex
code constructs and explore a considerable amount of paths under bounded resources. The explored paths may contain unique paths that can rarely be reached by random testing featured fuzzing or locality featured concolic execution. As a result, the bounded symbolic execution will impel the whole exploration to be more effective and efficient.

We note that there have been some orthogonal works that aim to improve the efficiency of the fuzzing tool. For instance, AFLFast [6] uses a markov chain model to identify the low-frequency paths and focuses most fuzzing effort on such paths. As a unique paths that can rarely be reached by random testing code constructs and explore a considerable amount of paths, we can use our method immediately.

AFLFast [6] uses a Markov chain model to identify the low-frequency paths and focuses most fuzzing effort on such paths. As a unique paths that can rarely be reached by random testing code constructs and explore a considerable amount of paths, we can use our method immediately. To some extent, the recovered inputs from the bounded symbolic execution act as high-quality fuzzing seeds, which prevent the exploration from converging to certain branches and at the same time enhance the overall exploration efficiency.

Due to the incomplete list of system calls modeled in the underlying symbolic execution engine, we have just been able to test our method with some synthetic programs. As part of our future work, we will develop symbolic summaries for some missing system calls so that we can use our method to test real-life programs.

VI. CONCLUSION

We proposed to use hybrid symbolic execution, i.e., a bounded symbolic execution at the beginning and an interleaved concolic execution, to assist the exploration of coverage-based fuzzing. Our method can be used to test binary programs without the source code, and does not need any external test case. The bounded symbolic execution in our method may identify some unique paths that can be rarely reached by the following interleaved fuzzing and concolic execution. To some extent, the recovered inputs from the bounded symbolic execution act as high-quality fuzzing seeds, which prevent the exploration from converging to certain branches and at the same time enhance the overall exploration efficiency.

REFERENCES