An Exploratory Study on Fuzzing and Concolic Execution Engines for Automated Vulnerability Discovery

Caleb Davenport, and Ahmad Y. Javaid
Electrical Engineering and Computer Science Department
University of Toledo, Toledo, OH 43606
caleb.davenport@rockets.utoledo.edu, ahmad.javaid@utoledo.edu

Abstract—With technology growth and popularity of agile development in the industry for faster application deployment times, testing software for vulnerabilities becomes a challenge. Several automated tools allow discovery of bugs and other issues feasible in a shorter time. Fuzzing and concolic execution engines are such tools that allow for automated vulnerability discovery in existing programs. Fuzzing engines are used to explore unexpected inputs and to ensure that these inputs do not lead to unexpected behavior. Concolic execution engines are used to traverse every possible branch in a program to ensure that the program as a whole does not contain any logic errors. Each of these methods has its strengths and weaknesses which depend on the specific content of the code that is being analyzed. These tools are important in the field of cybersecurity as they can help ensure that vulnerabilities are discovered before malicious attackers can discover and potentially exploit them. In this paper, we perform an exploratory study of these tools and discuss the effectiveness of these tools through a demonstration.

Keywords: fuzzing, concolic execution engine, automated vulnerability discovery

1. Introduction

Cybersecurity is a growing area with hundreds of new attacks launched using new and existing vulnerabilities while the number of vulnerabilities available to attackers continues to increase due to the introduction of faster software development practices. It is well known that security is not one of the top priorities even in 2019. According to a 2018 report, of the over 100,000 vulnerabilities published to the Common Vulnerabilities and Exposures (CVE) list, less than 6 percent were actually exploited in the wild [1]. This also indicates that a majority of vulnerabilities were not discovered accidentally and were exploited intentionally which also raises questions on the overall software testing aspect.

The increase in lines of codes in a typical application has also made the vulnerability testing process quite challenging. As an example, a 2017 report states that "the application attack surface is growing by 111 billion new lines of software code every year" while it is expected that 99% of all mission-critical applications will have some open source code [2]. Many vulnerabilities can be found in programs if given enough time and the proper tools. While human analysis of the code or manually validating inputs is valid approaches to finding bugs, these methods cannot be scaled easily. Thus, tools are necessary to automate this search and validation process. Keeping in mind the increase in lines of code for not only graphic-intensive games, but simple IoT based applications, automating the process of vulnerability analysis becomes the best choice. Two of the main types of tools that allow for this automation are fuzzers and concolic execution.

In this paper, we first take a look at the state-of-the-art in the area of automated vulnerability discovery in section 2. In sections 3 and 4, we briefly discuss fuzzing and concolic execution engines and examine their utility for an example application. Finally, in section 5, we conclude the paper with a few remarks on our findings.

2. Related Work

Among one of the recent noteworthy works, Stephens et al. proposed a new methodology to improve the effectiveness of fuzzing by leveraging selective concolic execution to reach deeper program code, while improving the scalability by using fuzzing to alleviate path explosion [3]. This tool, known as Driller, is a guided white box fuzzer which combines the two best techniques of vulnerability discovery. Several works have attempted to selectively choose optimal test cases to improve fuzzing by targeting regions of interest (ROI) in the target code. Another tool, Dowser [4], identifies ROI within code using static analysis and applying taint tracking to determine input bytes processed by these regions, and finally, symbolically explores the ROI. However, Dowser still faces the issue of path explosion.

BuzzFuzz [5] follows a similar approach to Dowser, except that it needs manual labeling of attack points from an auditor and relies on input test cases that reach vulnerable code, which makes it use limited. Flayer [6] is another attempt in improving fuzzers and allows the auditor to skip complex checks and dig deeper without the need for creating input in the required format. Another fuzzer, Taintscope, employs a checksum detection algorithm to patch branch
predicates which are difficult to satisfy [1]. Another recent popular approach is hybrid fuzz testing which applies limited symbolic exploration to find "frontier nodes" [2]. A recently proposed framework utilizing this approach, known as QSYM, is claimed to outperform several of the fuzzers discussed above [7].

Another recent work introduced Directed Greybox Fuzzing (DGF) [8] which generates inputs with the objective of reaching a given set of target program locations efficiently, as another tool called AFLGo. The authors also integrated AFLGo in Google’s OSS-Fuzz [9] and demonstrated its effectiveness. Another recent work (March 2019), proposes a novel open-source compositional fuzzing framework called Wildfire [10], which claims to reduce the analysis time to roughly 10% compared to popular fuzzers. The same authors previously proposed another hybrid fuzzing based tool called Munch, to solve the path explosion problem [11]. They demonstrate higher function coverage than symbolic execution or fuzzing alone.

Although we have discussed several state-of-the-art automated vulnerability discovery tools, interested readers may also refer to a recent survey from 2018 that details symbolic execution techniques and discusses the design, memory model, path explosion, and constraint solving at length [12].

### 3. Fuzzing Engines

The most common tool for discovering inputs in a program is a fuzzer. A fuzzer will generate random inputs into a program to ensure that all these unexpected inputs are handled correctly. More robust fuzzers will randomly modify known valid inputs in order to increase the chances of reaching novel sections of code. Most toolsets that are referred to as fuzzers contain only a single fuzzing engine, but toolsets could include multiple fuzzing engines, or employ the fuzzing engine only as a secondary function. A popular fuzzing engine is ClusterFuzz which is developed by Google [13] and is the engine used by Google’s OSS-Fuzz project [9]. The purpose of OSS-Fuzz is to exclusively test open-source software. OSS-Fuzz (and therefore ClusterFuzz by extension) has reported over 9000 bugs to developers over the course of nearly two years [14]. As of January 2019, Google made ClusterFuzz open-source allowing the fuzzing engine to be utilized by more than just open-source projects accepted into their OSS-Fuzz program [13].

The main advantage of a fuzzer is the ability to test many unexpected inputs that a programmer may have not intended or foreseen. For a trivial example, imagine a program that always expects there to be some input into the program. There should be some method to handle the case of no provided input. If this case is not handled, the program may crash or run unintended sections of code. However, the main disadvantage of a fuzzer is that it may not be able to access an entire section of a program if the inputs required to access such a section are extremely strict.

#### 3.1 Radamsa

One fuzzing engine is Radamsa, a fuzzing tool created by Aki Helin [16]. Radamsa is a tool that can work exclusively inside a UNIX environment. If given an input, Radamsa will apply random predefined mutations, the output the new text. The input can be stdin, a file, or none at all (create random data). Some of the mutations available are:

- **bd**: Drop a byte
- **bf**: Flip one bit
hashes are appended to the file fuzz_data visible in some sequence of bytes. Any exceptions will occur and the only mutation allowed is sr parameters used in this case specify that only one mutation output is then sent into the original Python script. The script can be found which runs one hundred times. It raise an exception on subsequent runs. In figure 4, a bash code which can be seen in figure 3. This python script takes a fuzzer

```python
#!/usr/bin/env python3
import hashlib
import sys

def quick_hash(s):
    num = hashlib.md5(s.encode()).hexdigest()
    num = int(num, 16)
    num = num % 100
    return num

def main():
    input_string = sys.argv[1]
    super_secret_hash = quick_hash(input_string)
    print(input_string)
    print(super_secret_hash)
    if super_secret_hash is 99:
        raise Exception("Crash")

if __name__ == "__main__":
    main()
```

Fig. 3: Python script demonstrating a problem detectable by a fuzzer

- bi: Insert a random byte
- br: Repeat a byte
- bp: Permute some bytes
- bei: Increment a byte by one
- bed: Decrement a byte by one
- ber: Swap a byte with a random one
- sr: Repeat a sequence of bytes
- sd: Delete a sequence of bytes
- uw: Try to make a code point too wide
- ui: Insert funny unicode
- num: try to modify a textual number

To demonstrate the capabilities of a fuzzer, I’ve created code which can be seen in figure 3. This python script takes in some input via stdin, performs a MD5 hash on the string, and outputs the hash modulo 100. If the output is equal to 99, an exception is raised. While these outputs are deterministic, they are essentially random. Thus, it is required to try several random inputs and until one of them causes a raised exception. Since the output is deterministic, the same input that raised an exception before will always raise an exception on subsequent runs. In figure 4, a bash script can be found which runs one hundred times. It generates a string which is then piped into radamsa whose output is then sent into the original Python script. The parameters used in this case specify that only one mutation will occur and the only mutation allowed is sr or repeating some sequence of bytes. Any exceptions will be plainly visible in stdout. All the inputs used and their modulo hashes are appended to the file fuzz_data.

After running the bash script, which runs the Python script one hundred times, there should be at least one crash visible. Searching the fuzz_data should reveal some inputs with the modulo hash of 99. If one of these inputs is inserted into the original Python script, the script will raise an exception. The original input used that was fuzzed was cybersecurity. Radamsa provided several inputs such as cybersecururururryrury and cybersecuresrry. An input that was able to cause the program to raise an exception was the string cybersecururururrurury. When used as an input into the original Python script, it will provide a modulo hash of 99 causing the raised exception.

4. Concolic Execution Engines

Another type of tool for automating vulnerability discoveries involves the use of concolic execution. Concolic comes from a portmanteau of concrete execution and symbolic execution; concolic execution performs both of these functions in its discovery process. Concrete execution finds where each possible branch of a program could keep track of these branches in a tree if proper conditionals at each node. Symbolic execution determines what inputs are required to meet a particular output. For example, symbolic execution will be able to find some values of var_a that matches var_a == 3 and some values that do not. Concrete execution creates the track of which inputs are used in a branch and which conditional needs to be solved next.

A key advantage of concolic execution engines is the ability to ensure that it can traverse all paths or at least know which paths exist in a program. It can also find very specific inputs required to traverse branches in some cases. However, there are many disadvantages to concolic execution. One is that it requires a lot of memory in order to function properly. The tree that contains all possible branches in a program could grow extremely quickly especially for larger programs. It is also not able to handle random values; concolic execution works best when the program is deterministic. It cannot handle one-way functions such as hashing or reversing cryptography. This means that concolic execution can get stuck at a branch that requires a specific hash and will be unable to traverse since reversing modern hashing and cryptography functions are currently unsolved problems. Most importantly, most implementations of concolic execution are much slower than that of fuzzing engines [15], [17]. A fuzzer can quickly generate a new input and immediately determine its value, while the value of new input from a concolic engine cannot be determined quickly. An example of code that performs

```bash
for run in {1..100}; do echo "cybersecurity" | radamsa -p od -m sr | xargs ./hash_crash.py; done >> fuzz_data
```

Fig. 4: Bash script that runs the Python script 100 times
#!/usr/bin/env python2

from triton import TritonContext, ARCH, Instruction

Triton = TritonContext()
Triton.setArchitecture(ARCH.X86)
astCtxt = Triton.getAstContext()
a = astCtxt.variable(
    Triton.newSymbolicVariable(32))
b = astCtxt.variable(
    Triton.newSymbolicVariable(32))

x = astCtxt.land([a > 0x20000000,
    a < 0x30000000,
    b == a + 0x44444444,])

print Triton.getModel(x)

Fig. 5: Symbolic Execution Example

well under concolic execution is figure 2 and code that runs poorly under concolic execution is figure 1.

4.1 Triton

Triton is a tool that allows for dynamic binary analysis [18]. One of its features is symbolic execution. If given several clauses, it can find a series of inputs that will either pass or fail the clauses depending on the requirement set by the concrete execution. Figure 5 is an example of symbolic execution. Normally, a script would be used to hook into a program and intercept current memory addresses. In this example, the variables are modified directly to demonstrate symbolic execution. The lines to focus on are in the assignment of variable x. Here, variable a must be greater than 0x20000000 and less than 0x30000000. Variable b must be equal to a + 0x44444444. Note that these variables are not be assigned these values; these are clauses that will either resolve to True or False.

Upon running this code, stdout will receive the following: 0L: SymVar_0 = 0x20000002, 1L: SymVar_1 = 0x644444446 SymVar_0 corresponds with variable a while SymVar_1 corresponds with variable b. As shown, the assigned values match all of the conditions that were set.

4.2 Angr

Angr is another binary analysis tool that includes some concolic execution features [19]. We demonstrate the concrete execution part of concolic execution here. One of the examples included in the documentation for angr was a program called fauxware [20]. This binary would ask for a username and password and reject incorrect username and password combinations. If a particular password was entered, the program would give access to the "admin console." This behavior is shown in figure 6. The username does not matter in this program at all, and this program does not do anything other than string matching; it does not have the ability to authenticate a user’s account. Since the string is hardcoded into the code, it can be analyzed, and the password can be discovered.

The provided example in the angr documentation also had a corresponding python program that would traverse the correct branches and find the conditional where the password that was entered was compared to the hardcoded password, as shown in figure 7. The script would then output what the hard-coded password was, or what would be sent to be analyzed via symbolic execution. In this case, the value was hunter22. When entering this code into fauxware, it allowed access into the "admin console."

Fig. 6: Concrete Execution Program Behavior

5. Conclusions

Finding vulnerabilities in programs is extremely useful in the field of cybersecurity. Being able to find vulnerabilities early prevents a malicious actor from exploiting these issues before they can be patched. In 2017, Google Project Zero was able to use fuzzers to discover 31 unique vulnerabilities in the DOM engines of various modern web browsers including Mozilla Firefox, Apple Safari, and their own Google Chrome [21]. The fuzzer AFL was able to find security bugs in major applications such as PHP, OpenSSH, PuTTY, and VLC [22]. The hybrid fuzzer and concolic execution engine QSYM was able to find 13 bugs across 8 different programs in 2018 [15]. All of these accomplishments, as well as the demonstration discussed in this paper, show that fuzzing engines and concolic executions engines are fantastic tools for analyzing programs to automatically discover vulnerabilities.

References

\$ ./angr_concolic.py

WARNING | 2018-12-05 14:53:38,775 |
cle.loader | The main binary is a position-independent executable. It is being loaded with a base address of 0x400000.

WARNING | 2018-12-05 14:53:39,934 |
angr.state_plugins.symbolic_memory | Register r15 has an unspecified value; Generating an unconstrained value of 8 bytes.

WARNING | 2018-12-05 14:53:39,935 |
angr.state_plugins.symbolic_memory | Register r14 has an unspecified value; Generating an unconstrained value of 8 bytes.

[...] 

WARNING | 2018-12-05 14:53:40,178 |
angr.state_plugins.symbolic_memory | Memory address 0x7ffffffffffff0000 has an unspecified value; Generating an unconstrained value of 79 bytes.

WARNING | 2018-12-05 14:53:40,178 |
angr.state_plugins.symbolic_memory | Memory address 0x7fffffffffeff70 has an unspecified value; Generating an unconstrained value of 8 bytes.

hunter22

Fig. 7: Concrete Execution Example


