Search-Based Concolic Execution for SW Vulnerability Discovery

Rustamov FAYOZBEK(a), Minjun CHOI†, Nonmembers, and Joobeom YUN(b), Member

SUMMARY Huge amounts of software appear nowadays. The more the number of software increases, the more increased software vulnerabilities are. Although some automatic methods have been proposed in order to detect and remove software vulnerabilities, they still require a lot of time so they have a limitation in the real world. To solve this problem, we propose BugHunter which automatically tests a binary file compiled with a C++ compiler. It searches for unsafe API calls and automatically executes to the program block that have an unsafe API call. Also, we showed that BugHunter is more efficient than angr through experiments. As a result, BugHunter is very helpful to find a software vulnerability in a short time.

key words: search-based, concolic execution, vulnerability

1. Introduction

Software vulnerabilities are crucial weaknesses that allow attackers to intrude into a system[1]. Before an attacker abuses the software vulnerability, it is necessary for us to test the software first. Until now, many security researchers have been implementing a wide range of software testing technologies. Although every technology has the same goal, they have different ways and have pros and cons. The most canonical examples are fuzzing[2] and symbolic execution[3]. However, these classical methods have a critical limitation which is it takes a long time to check the software binary.

In order to overcome this limitation, some advanced methods have been proposed in recent years. AEG[4] automatically finds software vulnerabilities by analyzing a compiled source code, but it always needs source codes of software. Therefore, its usage is restricted. Mayhem[5] utilizes the hybrid symbolic execution technique which combines online and offline execution. However, this system has a disadvantage. It needs high level system resources and a context-switch overhead. Another example, angr[6], is a multi-architecture binary analysis toolkit with the capability to perform fuzzing and dynamic symbolic execution alternately. However, this also takes a long time. As a result, it is inevitable to test and execute software efficiently.

In order to solve this problem and shorten the test time, we developed a search-based software test system named BugHunter. The key idea of our software testing program is that it reads the binary file of the C++, executes to the suspicious program block, and tests there exhaustively. BugHunter uses the search-based concolic execution to avoid unnecessary execution paths. First, it constructs a Control Flow Graph (CFG) by parsing the program binary. Second, it searches for suspicious program blocks having unsafe function calls to use a predefined unsafe function list. Third, it checks all other paths as avoid paths except for a path to the suspicious blocks. Fourth, it runs the concolic execution[9] to the suspicious blocks. Finally, it tests the program block exhaustively. As a result, our method greatly saves the execution time.

The advantage of BugHunter is described in detail as follows.

- BugHunter executes the program more efficiently than angr[6], which is the next-generation binary analysis framework. Using search-based concolic execution, it can explore only suspicious execution flows.
- BugHunter does not work on program source code but program executables. This means it can be widely applicable.
- BugHunter is light-weight. It does not require lots of resources such as a context switching overhead. In our experiments, BugHunter needed only 7.8% CPU and 1.3% memory usage on average.

2. Background

2.1 Static Analysis

Static analysis of software examines the text of a program statically without attempting to execute it. It examines either a program’s source code or a compiled form of the program, although there is a decoding problem[7]. Thus, using static analysis makes us examine all possible program flows rapidly rather than just the test-case execution. From a security viewpoint, this is a significant advantage[8].

BugHunter also uses static analysis on a program binary. First, it disassembles an executable to an assembly form. And then, it searches for unsafe function calls in the disassembled file. Once it finds unsafe function calls, it records its address and the block start address having an unsafe function call. This is accompanied with parsing CFG. In the Control Flow Graph (CFG), each node represents a program block and the edges depict the control flow of the instructions[12]. Unsafe functions are APIs which have
been used to attack buffer overflow and format string vulnerabilities so far. We have gathered these types of unsafe function information in Table 1. For example, `gets()` and `strcpy()` functions are unsafe because they do not check the target buffer length so the target buffer can be overflowed [8].

2.2 Dynamic Analysis

Dynamic program analysis is an analysis of computer software that is performed by executing programs on a real or virtual processor. Dynamic analysis can give more accurate results because it tests the program under the execution environment.

Concrete execution is one type of dynamic program analysis technique to generate a concrete input for a branch condition. Symbolic execution is program testing to assign a symbol for an input and build constraints for a branch condition. A constraint solver finds out a concrete value to solve these branch constraints. The advantage of symbolic execution is a wide code coverage because it uses abstraction for a variable.

Concolic execution [9] combines concrete execution and symbolic execution. It executes a program both concretely and symbolically. This concolic execution is used to remove the limitations of random (or fuzz) testing and symbolic execution. Concrete values from random testing are used to overcome the limitations of symbolic execution and symbolic execution is used to generate concrete test inputs that give better coverage than random testing [9]–[11].

BugHunter uses concolic execution. During program execution, it combines concrete testing and symbolic testing. It can execute directly to the unsafe function block by using a path pruner. The path pruner finds out unnecessary execution paths. As a result, BugHunter can save execution time and find a crash point well.

3. BugHunter

3.1 Search-Based Concolic Execution

There are several ways to test a software program. It is important to find out a vulnerable area quickly and test it efficiently because the classical test methods have limitations such as path explosion, excessive resource usage, etc. For this, BugHunter uses search-based concolic execution. First, it searches for unsafe function calls. If it finds at least one, it records the start address of that block. Second, it searches for all program paths in the CFG. BugHunter prunes all other paths except for the path to the unsafe function block. Finally, BugHunter executes concolic testing to that block.
mainly consists of two parts. Static analysis consists of three modules. The disassembler disassembles a program binary to an assembly code. The CFG drawer makes a CFG from the program binary. The unsafe function finder searches the whole program binary and finds out the address of unsafe function calls. Dynamic analysis also consists of three modules. The path pruner finds the shortest path from an entry node to target node and prunes unnecessary paths by using an avoid list. The avoid list is a table having no need to execute block addresses. The concolic execution engine executes concrete execution and symbolic execution alternatively. If it meets a symbol constraint, the constraint solver solves the constraint.

Algorithm 1 shows the pseudo code of the path pruning algorithm. The program binary was converted to CFG because we need this graph when we find the shortest path. After the CFG is created, all the nodes are examined until the unsafe function calls are found. Except for the path from the beginning of the program to the address where the unsafe function calls are located, all node addresses are assumed to be put into the avoid list. This is a crucial function in BugHunter.

Figure 3 shows the execution information of a sample program. It is mainly the selected path node and an avoid list after path pruning. First, BugHunter searches k shortest paths (KSP) in the CFG. Second, it identifies necessary nodes while exploring the k shortest paths in Fig. 3. Finally, it makes an avoid list except for necessary nodes. The numbers in Fig. 3 are start addresses of program blocks and Avoid List is a start addresses list of pruned program blocks.

4. Evaluation

We evaluated our system. We show a comparison result of two methods which are angr and BugHunter. We conducted our experiments on a machine with a 2.4 GHz Intel(R) Core i5 CPU and 4GB of RAM. We used the Kali Linux 2016.1 version. We conducted experiments using EasyCrack 101[14], which is 101 binary files of linux programs having software vulnerabilities. It was distributed by CodeGate conference, which is well known security conference.

Figure 4 shows the execution time of angr and BugHunter. This execution time is an elapsed time between the program starting and arriving time to the suspicious program block. As shown in Fig. 4, BugHunter runs to the suspicious block faster than angr. The average execution time of BugHunter is 7.99 seconds and the average time of angr is 16.82 seconds. BugHunter runs the concolic execution 2.1 times faster than angr.

Figure 5 shows the preprocessing and execution time of BugHunter. The preprocessing step includes the disassembling, the CFG drawing, the unsafe function finding, and the path pruning. The execution time is only the concolic execution time of a binary program. Average preprocessing time is 6.26 seconds and the average concolic execution time is 1.73 seconds. This fact shows that most of the execution time is spent in the preprocessing step. However, this preprocessing step makes the concolic execution very fast.
5. Conclusion

Automatic software testing has become essential nowadays. However, previous techniques have performance limitations such as a long execution time, excessive resources, etc. In this paper, we presented the BugHunter system that uses static analysis and dynamic analysis alternatively. It searches for unsafe API calls quickly by using static analysis and runs to the suspicious program block by using concolic execution. This technology achieves time effectiveness apparently. To show the effectiveness of BugHunter, we tested vulnerable 101 binaries with two systems, BugHunter and angr. BugHunter ran 2.1 times faster than angr. We believe that our method is very helpful to develop the automatic software testing technology.

Future work will focus on various software vulnerability case studies and enhancing system performances.

Acknowledgements

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (ITIP-2018-0-01423) supervised by the IITP (Institute for Information & communications Technology Promotion). Also, this research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 2018R1D1A1B07047323).

References