Comparison of Different Hybrid Fuzzing Methods

Abhishek Srivastava(861307778)
University of California, Riverside
asriv003@ucr.edu

Abstract

This paper compares some of the different hybrid fuzzers which are proposed in recent years for finding deeper bugs present in the binary without much human intervention. It combines two very well known methods for finding vulnerabilities i.e dynamic and symbolic execution since they complement each other very well. Random mutational fuzz testing (fuzzing) and symbolic executions are program testing techniques that have been gaining popularity in the security research community due to their simple understanding and very reliable performances as well.

Introduction

Finding and exploiting vulnerabilities in binary code is a challenging task. The lack of high-level, semantically rich information about data structures and control constructs makes the analysis of program properties harder to scale. There have been fair amount of work and research done in order to provide better ways to discover software vulnerabilities and exploits. Automated vulnerability analysis systems have been designed actively by researchers . Existing approaches mainly falls into three main categories: **static**, **dynamic**, and **concolic analysis systems**. Each approaches have their advantages and disadvantages.

Static program analyses are used by many developers to test their programs because they are effective in finding some trivial bugs that can be caught by the rules that define security violations with very small resource. Static code analysis is a method to detect potential coding errors in the program source code without actually executing the program. Usually the analysis is done by the automated tools that implement rules that are checked against the code to verify if any portion violates them. Depending on the modeling, static code analysis can find from trivial bugs (but easy to miss) to complex one, which requires understanding of cross file/class relationship.

Static code analysis is powerful because it is fast and cheap while generally effective. It can provide detection of flaws in the software that dynamic analysis cannot easily expose. However, the limitation of static code analysis soon becomes obvious. It usually does not support all programming languages, and introduce many false negatives and false positives [1]. Also, the static code analysis is only as helpful as the rules that are employed. More importantly, static code analysis cannot reason about the concrete values for which are generated dynamically. Therefore, when the analysis does not find any bug, it does not imply that the target program is bug-free.

In order to overcome the limitations of static analysis, another program analysis paradigm dynamic analysis can be used. Dynamic program analysis technique, due to its nature, tends to have a significantly higher overhead compared to static analysis technique. With the cost of performance and resource, however, dynamic analysis provides extremely useful information such as concrete values and dynamic address resolution.

The most common dynamic analysis technique used by any security researchers is a program testing method called “fuzzing”. Fuzzing finds bugs in a target program by natively executing it with randomly mutated or generated inputs while monitoring the execution for abnormal

crashes. Fuzzing is good at quickly exploring the program code in depth because it runs the target program natively with concrete inputs. However, due to its nature, fuzzing often suffers from low code coverage problem.

Symbolic execution is another technique that has recently gotten the attention of security researchers. In contrast to fuzzing, symbolic execution tests a program by treating the programs input as symbols and interpreting the program over these symbolic inputs. In theory, *symbolic execution* is guaranteed to be effective in achieving high code coverage, yet this generally requires exponential resource which is not practical for many real-world programs.

Our intuition on hybrid fuzzing comes from the obvious differences in the characteristics of symbolic execution and fuzzing. Symbolic execution is capable of discovering and exploring all possible paths in the program, but is not practically scalable since the number of the paths quickly becomes exponential. While fuzzing is much faster than symbolic execution and thus guarantee to explore deeper portion of the code, it has limited code coverage in breadth.

Background

Our primarily subject of focus is **Driller** which is a hybrid vulnerability excavation tool which leverages fuzzing and selective concolic execution in a complementary manner, to find deeper bugs. Driller functions by combining the speed of fuzzing with the input reasoning ability of concolic execution allows it to quickly explore portions of binaries that do not impose complex requirements on user input while also being able to handle, without the scalability issues of pure concolic execution, complex checks on specific input. "Complex" checks as the checks which are too specific to be satisfied by generated input from an seed input-mutating fuzzer[2].

American Fuzzy Lop (AFL) is being used in the Driller as the fuzzer. AFL relies on instrumentation to make informed decisions on which paths are interesting. These features allow AFL to rapidly discover unique paths through an application, performing path discovery within a given compartment/code block of the application. Driller overcomes the weakness of fuzzing by determining specific user input required to pass complex checks for fuzzing to proceed forward, by taking advantage of its concolic execution component.

Angr, an open-sourced symbolic execution engine is used for Drillers as a concolic execution engine. This engine is based on Mayhem and S2E. Angr translates binary code into Valgrinds VEX intermediate representation, which is interpreted to determine the effects of program code on a symbolic state. The analysis engine tracks all concrete and symbolic values in memory and registers throughout execution. At any point in the program that the engine reaches, a constraint resolution can be performed to determine a possible input that satisfies the constraints on all symbolic variables in the state. Such an input, when passed to a normal execution of the application, would drive the application to that point [10]. The advantage of concolic execution is that it can explore and find inputs for any path that the constraint solver can satisfy. This makes it useful for identifying solutions to complex comparisons that a fuzzer would be unlikely to ever brute force.

Problem Specification

The main components of the Driller are AFL and Angr even they have very good qualities but they have some problems as well which is related to how they behave during execution of a binary for finding vulnerability. AFL uses instrumentation guided mutation methods to generate next seeds whose randomness does not always guarantee good results. Angr also suffers

from a problem, which is using a heuristic method to pick up next path to be explored by the fuzzer from multiple possible options. Since the optimal path is not chosen by the symbolic execution, to obtain vulnerability we might have to wait for correct path exploration.

The main objective of driller is to obtain more number of vulnerability in as less time as possible. Not using a optimal heuristic already hurt its performance, another reason where Driller is not optimal is the case where nested Ifs exists. Since each If's involves a specific value check and fuzzer will get stuck, after symbolic execution solve's the check for the fuzzer, on the next instruction fuzzer is again stuck on different check and thus waste time to reach the actual vulnerability. These weak aspects of Driller or simple hybrid fuzzer can be fixed and are discussed in the next section.

Improving Fuzzer

First aspect of the driller we will see are the methods using which we can improve the fuzzer aspect of the Driller or Hybrid Fuzzer. There different kind of fuzzing mechanisms exists to exploit for different vulnerability. Each have their pros and cons with respect to each other. Lets go over them one by one.

Mutation Based Fuzzing

These fuzzer's generate next test seeds for the testing by applying mutations on the existing samples. AFL is one of the example which is based on a mutation based fuzzing. The different mechanisms used to generate seeds for the fuzzer are based on: *Walking bit flips*, *Walking byte flips*, *Simple arithmetics*, *Known integers*, *Stacked tweaks*, and *Test case splicing.* [9]. This is one of the most commonly used fuzzer and probably most well known. We will look into other kinds of fuzzers and how well differ from each other and how they fair against AFL efficiency.

Coverage Based Fuzzing

It is random testing approach that requires no program analysis. A new test is generated by slightly mutating a seed input. If the test exercises a new and interesting path, it is added to the set of seeds, otherwise, it is discarded. It uses lightweight (binary) instrumentation to determine a unique identifier for the path that is exercised by an input. The strategy is to focus most of the fuzzing effort on low-frequency paths so as to explore more paths with the same amount of fuzz. AFLFast is the example of such fuzzer. AFL chooses seeds in the order they are added. Once all seeds have been fuzzed, AFL resumes with the first. A new cycle begins. AFLFast uses a different search strategy. It chooses seeds in the order of their likely progressiveness. In the same cycle, AFLFast chooses seeds earlier which exercise lower frequency paths and which are chosen less often [4].

It is evident that AFLFast is more scalable because the time to generate a test does not increase with the program size and also is highly parallelizable because the retained seeds represent the only internal state at a time. AFLFast identifies same amount of vulnerability as AFL but faster and sometimes identifies more number of uniques crashes.

Grammar Based Fuzzing

Mutation-based fuzzing generate test inputs by modifying valid inputs in a random or heuristic way. This type of fuzzer can be built with little or no knowledge of the input format or grammar of the target program. Grammar-based fuzzing generate test inputs that are processed in deep part of the target program, because they incorporate knowledge about valid input structures. However, current grammar-based fuzzing methods cannot inspect the grammar related code of the target software, since they only generate grammatically correct test inputs.

Grammar-based fuzzing method using dynamic information extracted from a program execution. The proposed method traces program execution with an input, and captures dependency relations between conditional branches, and also find input fields that influence conditional branches by dynamic taint analysis. When generating test inputs to find new execution paths, the input space can be confined such that test inputs are generated through the dependency relationships and influencing input field information, together with input grammar. This improves the efficiency of test input generation and increases the opportunity for executing deep program code.This fuzzing method shows 10% higher code coverage than the mutation-based fuzzer [3].

Evolutionary Fuzzing

AFL is application-agnostic fuzzing and employs a blind mutation strategy. It simply relies on generating a huge amount of mutated inputs in the hope of discovering a new basic block. Unfortunately, this approach yields a slow fuzzing strategy, which can only discover deep execution paths by sheer luck. we can increase the efficiency of AFL-like fuzzers manifold by accounting for information by considering where to mutate and what value to use for the mutation. The key intuition is to enhance the efficiency of general-purpose fuzzers with a "smart" mutation feedback loop based on control and data-flow application features without having to resort to less scalable symbolic execution.

Evolutionary fuzzing strategy, that is a fuzzing strategy that relies on an evolutionary algorithm for input generation. When generating new inputs, VUzzer an example of evolutionary fuzzer considers features of the application based on its execution on the previous generation of inputs. By considering such features, we make the feedback loop "smart" and help the fuzzer find inputs with non-zero IG with high frequency. *Data-flow features* provide information about the relationship between input data and computations in the application. VUzzer extracts them using well known techniques such as taint analysis and uses them to infer the structure of the input in terms of the types of data at certain offsets in the input. *Control-flow features* allow VUzzer to infer the importance of certain execution paths. We use control-flow features to deprioritize and prioritize paths. Its performance with that of AFL, showing that, in almost every test case, VUzzer was able to find bugs within an order of magnitude fewer inputs compared to AFL [8].

After going over through all different fuzzer we can see that there are multiple possibilities which can be used for finding vulnerabilities and can be used in Angr to improve performance. AFLFast and Vuzzer definitely will improve efficiency and execution time of the Angr given their properties.

Improving Symbolic Execution

In this section we will look into the ways we can improve the symbolic execution present in the Hybrid fuzzer and can we replace **Angr** with something better or improve it in any way possible. The main aspect of the symbolic execution is the heuristic it chose to explore the underlying tree. Picking in optimal path result in catching vulnerability faster. Different heuristics are designed to pick the path in an optimal way and different variation of this exists. Since all variants are based on different heuristic and there is no evaluation done using same dataset over these heuristics therefore we can not be sure that which heuristics perform better than whom. Only a theoretical estimation can be made since common or new techniques used by them are also somewhat different.

DeepFuzz

The main idea is interleaving concolic execution with constrained fuzzing in a way that allows us to explore paths providing maximal input generation frequency. We achieve this by assigning weights (corresponding to fuzzing performance) to the explored paths after each concolic execution step in order to select the ones with highest probability. If there is no format specified or available we just generate random input seeds. The concolic execution step receives a set of concrete program inputs X seed X and outputs a set of symbolic constraints collected along the paths belonging to these inputs. For each element in set of constraints the SMT solver checks if the symbolic constraints are satisfiable and in that case computes a new input for each element. Next we assign probabilities to these paths. we have explored paths weighted with probabilities select the paths that provide us maximal model generation frequency [5].Such a set of paths will guarantee us efficient fuzzing and maximal degree of freedom for subsequent payload generation in case we detect a vulnerability. Now that we have selected the paths with highest probability, we continue with fuzzing deeper layers of the program.

Sword Deep Fuzzing

The key idea of this approach is to search as more execution paths of the target problem as possible using symbolic execution which result in improving the code coverage and then doing taint analysis to check each execution path, the path-dependent taint information can guide the fuzzer to generate pertinent test cases, thus improving the efficiency of fuzzing. This framework is divided into four major parts: Path searching layer, path checking layer, fuzzing layer and database.

Path searching layer contains different modules whose goals are to find all possible execution paths of the target program within a set range. Path Checking layer goal is to check the execution paths found in the path searching layer and generate path-dependent information to guide the fuzzer to generate good test cases. Fuzzing layer contains different module, user interface selects target program and its parameters and start fuzzing. Controlling and scheduling with input generation module is responsible for test program execution. Target monitor monitors execution of program, automatically detect and determines abnormalities. If bug is found Exploitability analysis must be performed on it to determine whether it can further exploited or not. And Database module is used to store all the data and results produced [6].

FuzzBOMB

FuzzBall is a flexible engine for symbolic execution and automatic program analysis, targeted specifically at binary software. FuzzBALL builds a decision tree data structure as it explores possible binary executions. decision tree is a binary tree in which each node represents the occurrence of a symbolic branch on a particular execution path, and a node has children labeled "false" and "true" representing the next symbolic branch that will occur in either case. decision tree is used to ensure that each path which is explored is different, and that exploration stops if no further paths are possible.

FuzzBOMB uses an improved FuzzBALL symbolic execution engine in an approach that combines ideas from symbolic execution and static analysis in order to find vulnerabilities in binary programs. Because the space of program executions is vast, even in the constraint-based representations of symbolic reasoning, heuristic guidance is essential. FuzzBOMB uses problem relaxation heuristics to reduce the search space of possible executions. solving the relaxed problem determines. Reachability analysis to a vulnerability. If the relaxation of the program indicates a vulnerability is unreachable from a particular program decision point, then exploring from that point is fruitless and distance estimate at each decision point that lets exploration proceed along an estimated shortest path [7]. To generate the relaxation heuristic, FuzzBOMB

uses the causal model present within data-flow and control-flow graph (CFG) structures used in binary program analysis.

Conclusion

From the different methods and techniques described above we can see that there are multiple possibilities in which we can improve existing hybrid fuzzer **Driller**. We can use any combination of fuzzer and symbolic execution techniques to create a hybrid fuzzer but the main issue with that is that even though every techiniques have their goods and bads, we are unsure how they will behave when combined with each other.

References

[1] Brian S. Pak. 2012. *Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution.* School of Computer Science, Carnegie Mellon University, Pittsburgh.

[2] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna. 2016. *Driller: Augmenting Fuzzing Through Selective Symbolic Execution.* Department of Computer Science, UC Santa Barbara.

[3] Hyunguk Yoo, Taeshik Shon. 2016. *Grammar-based Adaptive Fuzzing: Evaluation on SCADA Modbus Protocol.* Department of Computer Engineering, Ajou University, Suwon, Republic of Korea.

[4] Marcel Bhme, Van-Thuan Pham, Abhik Roychoudhury. 2016. *Coverage-based Greybox Fuzzing as Markov Chain.* School of Computing, National University of Singapore, Singapore.

[5] Konstantin Bttinger and Claudia Eckert. 2016. *DeepFuzz: Triggering Vulnerabilities Deeply Hidden in Binaries.* Fraunhofer Institute for Applied and Integrated Security, Germany.

[6] Jun Cai, Jinquan Men, Shangfei Yang and Jun He. 2014. *Automatic Software Vulnerability Detection Based on Guided Deep Fuzzing.* The Academy of Equipment, Beijing, China.

[7] David J. Musliner, Scott E. Friedman, Michael Boldt, J. Benton, Max Schuchard, Peter Keller and Stephen McCamant. 2015. *FUZZBOMB : Autonomous Cyber Vulnerability Detection and Repair.* University of Minnesota.

[8] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida and Herbert Bo. 2017. *VUzzer: Application-aware Evolutionary Fuzzing.* Computer Science Institute, Vrije Universiteit Amsterdam.

[9] Micha Zalewski. *https://lcamtuf.blogspot.com/*

[10] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, Giovanni Vigna. 2014. *(State of) The Art of War:Offensive Techniques in Binary Analysis* UC Santa Barbara.