Expediting Binary Fuzzing with Symbolic Analysis

Luhang Xu¹

Wei Dong^{1*} Liangze Yin¹

Weixi Jia¹ Y

Yongjun Li²

¹ National University of Defense Technology, Changsha, China

² Northwestern Polytechnical University, Xi'an, China

* Corresponding author

E-mail: me@xuluhang.cn, {wdong, yinliangze}@nudt.edu.cn, wxjia92@163.com, lyj@nwpu.edu.cn

Abstract

Fuzzing is an important method for binary vulnerability mining. It can analyze binary programs without the source code of the program, which is not easy to do by other technologies. But due to the blindness of input generation, binary fuzzing often falls into traps for a long time when the new mutated inputs cannot generate unexplored paths. In this paper, we propose an efficient and flexible fuzzing framework named Tinker. It defines the Growth Rate of Path Coverage to measure the current state of fuzzing. If the fuzzing falls into low-speed or blocked states, a symbolic analysis procedure is invoked to generate a new input which can help the fuzzing jump out of the trap. In the symbolic analysis procedure, we employ dynamic execution to track the traversed nodes. The untraversed branches are then identified according to the recorded data of AFL. At last, we employ CFG to construct complete paths to these branches and a new input is generated using symbolic execution. Tinker has been implemented and the experiments on DARPA CGC benchmark show that Tinker is more efficient in vulnerability mining than state-of-the-art binary vulnerability mining tools.

1. Introduction

Fuzzing is a representative method for software vulnerability mining [1]. The basic idea of fuzzing is to provide a large amount of invalid, unexpected, or randomly generated data as inputs to a program. The program is then monitored for exceptions such as crashes. Given that fuzzing does not need the inside information of a program, it is one of the most important techniques for binary program analysis.

Existing fuzzing techniques can be classified into two classes: black box fuzzing and white box fuzzing. Black box fuzzing [2] does not require any source information of the program. Researches on this technique mainly focus on effective inputs generation. White box fuzzing [3][4] has been widely studied in recent years. It usually combines fuzzing with other analysis methods such as symbolic execution [5] and dynamic blot analysis [6]. To improve the capability of vulnerability mining, white box fuzzing usually explores the inside structures of a program to guide the fuzzing process.

Fuzzing has many advantages compared with other vulnerability mining techniques. However, due to the blindness of input generation, traditional fuzzing often falls into traps in which most of the executed paths are redundant [7][8]. To mitigate this problem, recently there has been some work using other analysis methods such as symbolic execution [9] to improve the efficiency of fuzzing. However, there are still two weaknesses for existing work: 1) Existing work is not sensitive enough to the current state of fuzzing. They usually invoke symbolic execution when fuzzing is in blocked states, without considering those low-speed states in which fuzzing may explore just several new paths for a long time. In practice, these states might be the most cases. 2) Existing work cannot handle system calls properly. As a result, they can only be applied to simple environments rather than real programs. For example, Driller [9] can run just on a simplified operating system (OS) with only seven system calls [10].

This paper proposed a new binary fuzzing approach Tinker. It first employs the defined Growth Rate of Path Coverage (GRPC) to evaluate the efficiency of fuzzing in current state. If fuzzing is trapped into a low-speed or blocked state which cannot efficiently find new paths, a symbolic analysis procedure will be invoked to generate a new valid input of the program, such that fuzzing can jump out of the trapped state and again run in a high-speed state. We employ a tool Angr for CFG generation, which has rewritten most of the system calls properly. Our method can be applied to real programs. Moreover, as our unexplored paths are analyzed from the CFG of the target binary program, the input generated in our method is often more effective.

The main contributions of this paper include:

- We proposed an efficient and flexible binary vulnerability mining approach Tinker. It employs GRPC to evaluate the current state of fuzzing. If fuzzing falls into a trap, a symbolic analysis procedure is invoked to generate an input which helps fuzzing jump out of the trap.
- We proposed an effective symbolic analysis based new input generation method. It supports real binary program analysis and can always return an effective input to an unexplored path. CFG is used in this method to construct the paths to those untraversed branches.
- We implemented Tinker and evaluated it on DARPA CGC benchmark. The experimental results demonstrate that, compared with state-of-the-art tools, Tinker can detect more binary vulnerabilities and is often more efficient for unexplored path exploration.

The rest of this paper is organized as follows. Section 2 outlines the framework of Tinker. Section 3 and 4 present our GRPC-based fuzzing evaluation and symbolic analysis based fuzzing intervention methods, respectively. Section 5 provides the experimental results. Section 6 reviews the related work, and Section 7 concludes our paper.

2. Method Overview

In this section, we first discuss the challenge of binary fuzzing, and then we outline the framework of Tinker.

2.1. The Challenge of Binary Fuzzing

Consider the example shown in Figure 1. The program contains two strings *order* and *magic_byte*. It runs the *fault()* statement only if both *order* and *magic_byte* equal "*vuln*". For this example, fuzzing will enter the true branch of Line 4 only if it succeeds in mutating the value of *magic_byte* to "*vuln*". Given that "*vuln*" has 4 * 8 bits, a maximum number of 2^{4*8} variations of inputs are required for fuzzing to reach Line 6. For some other long string, it will be more difficult for fuzzing to enter the true branch. A string like "*vuln*" here that hampers fuzzing to new paths in mutation is called "MAGIC-BYTE". The similar problem occurs in Line 7. In this paper, we use the growth rate of path coverage (GRPC) to measure the efficiency of fuzzing. For this example, the GRPC will keep zero until the true branchs of Line 4 and Line 7 are explored.

```
1
    int main(void){
     char order [20], magic_byte [20];
2
3
     scanf ( "%s \setminus n", magic_byte);
4
     if (strcmp(magic_byte, "vuln")==0)
5
     {
6
        gets (order);
7
        if (strcmp(order, "vuln")==0)
8
         fault();
9
     }
10
     return 0;
11
    }
```

Figure 1: A challenging example for traditional fuzzing.

To further investigate this problem, we have analyzed the efficiency of traditional fuzzing for three real binary programs, as shown in Figure 2. The vertical axis P represents the number of new explored paths for each time unit (15 minutes). According to this figure, fuzzing might perform differently for different programs. 1) For the *Multipass* program, the value of P always keeps 40 on average, and we say that fuzzing is in "high-speed" state. 2) For the *Grisword* program, the value of P always keeps in just 10 on average after 4 time units, and we say that fuzzing is in "low-speed" state. 3) For the *Monster_Game* program, the value of P reduces to zero after 3 time units, and we say that fuzzing falls into a "blocked" state.



Figure 2: Efficiency of fuzzing for three real programs.

2.2. Framework of Tinker

When fuzzing falls into a blocked or low-speed state, it is usually difficult for fuzzing to jump out of such a state itself. In other words, it might be trapped into such a state for a long time, which significantly limits the efficiency of fuzzing. To deal with this problem, we proposed a new binary fuzzing approach called Tinker. The idea is that we continuously compute the value of GRPC during the fuzzing process to evaluate the current state of fuzzing. If we find that fuzzing is trapped into some blocked or low-speed state, we employ a symbolic analysis method to generate a new input to guide fuzzing to search the other untraversed branches, such that fuzzing always explores new paths and keeps in high-speed states for vulnerability mining.

The framework of Tinker is outlined in Figure 3. Before the fuzzing starts, the target binary program will be preprocessed to generate an instrumented binary and its control flow graph (CFG). During the process of fuzzing, we iteratively employ a GRPC-based fuzzing evaluation procedure to measure the current state of fuzzing. If the fuzzing falls into a low-speed or blocked state, we then invoke the symbolic analysis based fuzzing intervention process to generate a new input to motivate the fuzzing to a high-speed state. To generate the new input, we first select an input from fuzzing which leads to a redundant path. The instrumented binary is then executed with this input to obtain the traversed nodes of the corresponding path. According to the recorded data of AFL, we can obtain those untraversed opposite branches of the path. CFG is then employed to obtain the complete paths of these untraversed branches. At last, we select one of these paths and employ symbolic execution to generate a new input. In such a manner, whenever the fuzzing falls into a low-speed or blocked state, we can detect it in time and generate a new input that helps fuzzing jump out of the trap. In particular, in Tinker, fuzzing and symbolic-based analysis work in parallel. The intervention of symbolic-based analysis does not interrupt Fuzzing.

3. GRPC-based Fuzzing Evaluation

If the fuzzing falls into a blocked or low-speed state, it may spent most of its effort on iteratively exploring those redundant paths. To quantitatively evaluate the efficiency of fuzzing, we proposed the notion of Growth Rate of Path Coverage (GRPC), which defines the average number of new ex-



Figure 3: Framework of Tinker.

plored paths that grows per time unit over the latest period. The smaller the value of GRPC is, the lower the efficiency of fuzzing will be. If GRPC equals 0, then the fuzzing is trapped into a blocked state.

To compute GRPC, we divide the fuzzing process into intervals. Then the average number of new explored paths per time unit for the last interval can be obtained as equation (1), where q_m represents the total number of different paths that have been explored in the last m intervals, and Δt represents the time length of each interval. The value of q_m can be obtained from AFL directly.

$$p_m = \frac{q_m - q_{m-1}}{\Delta t}, \qquad m \ge 1 \tag{1}$$

Given that there may exist some noises sometimes, one interval might not reflect the actual state of fuzzing in practice. For example, the fuzzing might fall into a low-speed state in some interval, and have a large probability of returning back to a high-speed state in the next interval. To avoid such cases, we filter out those noises by using the idea of moving average [11]. In this idea, the *m*th sample result is an average of the 2n + 1 sample values on time points m - n, m - n + 1, ...,m + n - 1 and m + n. Similarly, we can employ the p_m 's of the last 2n + 1 intervals to compute our GRPC P_m , as shown in equation (2).

$$P_m = \frac{1}{2n+1} \sum_{k=0}^{2n} p_{m-k}, \qquad m \ge 2n \tag{2}$$

Now the problem is how to decide the value of n. If n is too big, the fuzzing will stay in those blocked and low-speed states for a long time; if n is too small, many of those noises might not be filtered out in our method, which may also decrease the efficiency of fuzzing. According to the work of moving average, we usually set n = 2 in practice [11]. Our experiments also proved that fuzzing can usually obtain the best efficiency for n = 2. For m < 2n, we use the average of all p_k for $k \le m$ to compute P_m .

After obtaining GRPC, we need to decide the current state of fuzzing. For any program, the GRPC P_m have a peak value P_{max} , which can be obtained from its execution history. We say that fuzzing obtains a highest efficiency if P_m equals P_{max} . Given that the values of P_{max} for different programs differ significantly, the threshold that determines the state of fuzzing should also be different for different programs. Our idea is to decide the threshold according to the value of P_{max} for each program. To this end, we can decide the current state of fuzzing using the following definition.

Definition 1 Suppose that the currently observed peak value of GRPC is P_{max} , w is a threshold factor. Then the current state of fuzzing is determined as follows. Fuzzing is at

- 1) high-speed state, if $P_m \ge P_{max} * w$; 2) low-speed state, if $0 < P_m < P_{max} * w$;
- 3) blocked state, if $P_m = 0$.

At the beginning of fuzzing, P_{max} can be set to a small value that is greater than zero. After each interval, P_{max} will be updated to P_m if current P_m is larger. Threshold factor w can be customized by user according to the performance of fuzzing. In our experiments, we tried several values and found that setting w = 0.4 can usually obtain higher efficiency. If fuzzing is at low-speed or blocked state, a symbolic analysis procedure will be started to generate a new input that enables fuzzing to jump out of the trap.

4. Symbolic Analysis Based Fuzzing Intervention

The purpose of symbolic analysis based fuzzing intervention is to generate a test input which can help the fuzzing to explore new branches. The idea is that we first select an untraversed branch and then use symbolic execution to generate an input to this branch. However, given that we have no source information of the target binary program, a problem is how to obtain the path of an input and generate the path condition constraints for an untraversed branch. To address this problem, we propose to use existing disassemble techniques to generate the CFG of the target binary program first, and then we use existing binary instrumentation tools to generate an instrumented binary which can obtain the execution path of an input. With the instrumented binary, we use dynamic execution to identify a path explored by fuzzing. According to the recorded data of AFL, we can obtain the untraversed opposite branches of the path. Then CFG is employed to generate the condition constraints of these branches, which are then solved by a constraint solver to generate valid inputs to these branches. The main steps of our symbolic analysis method are as follows.

1) **Instrumented binary and CFG generation**. To perform our symbolic analysis, we should generate an instrumented binary and the CFG of the target binary program first. For instrumented binary, dyninst [12] is a binary instrumentation tool satisfying all our requirements. Hence, we select dyninst for instrumented binary generation in our work. For CFG generation, an issue here is how to generate those system call nodes which cannot be disassembled directly. We found that Angr [13] can rewrite most of the system calls and generate a CFG with system call nodes, which makes our method work well for most real programs. Hence, we select Angr to perform the CFG generation.



Figure 4: An example CFG used in symbolic analysis

2) Untraversed branches identification. Dynamic execution is a technique which can track an instrumented program under an input and record the traversed nodes. To obtain an untraversed branch, we select an input from fuzzing, and then use dynamic execution to obtain the traversed path of the input. Then for each branch of this path, we will check if the opposite branch has been explored yet. In AFL, it has recorded all the traversed nodes during the fuzzing process. If an opposite branch has not been explored yet, then an untraversed branch has been found. We will generate an input to this branch in the following steps. Given that some branch may be unreachable in an real execution, we will obtain all untraversed branches of the path in this step.

3) Unexplored paths construction. To generate the input of an untraversed branch, we should identify the path to it and then generate the condition constraint of the path. To achieve this, we map the path to the CFG, and then construct a subgraph of the CFG which contains just nodes of the path. For the example shown in Figure 4, the subgraphs for F and G in Figure 4 can be represented by $G_F = \{A, B, C, D, E, F\}$, and $G_G = \{A, B, C, D, E, G\}$, respectively. An issue here is that dynamic execution does not consider those system calls. In other words, there are some system call nodes which have been traversed by fuzzing but not marked as traversed in the CFG yet. We should detect these nodes and mark them as traversed in the CFG before the subgraph generation.

4) New test input generation. In this step, we will gen-

Algorithm 1 Symbolic analysis based test input generation

Input: The binary *bin*, selected input *fin* from fuzzing. **Output:** New test input *testcase*.

//Instrumented binary and CFG generation $ins_bin = Instrument(bin);$ cfg = CreateCFG(bin)//Untraversed branches identification; $path = DynamicExecution(ins_bin, fin);$ $n_list = UntraversedBranches(path);$ //Unexplored paths construction; PathMap(path, cfg);MissedNodesAdd(cfg); $q_list = \emptyset;$ for each untraversed branch $n \in n_list$ do $g_list = g_list \cup PathConstruct(n);$ end for //New test input generation; for each subgraph $g \in g_list$ do cons = ConstraintGeneration(g);input = Solve(cons);if *input* is VALID then **RETURN** testcase; end if end for **RETURN** NULL ;

erate a new test input which explores an unexplored path and helps the fuzzing jump out of a trapped state. The idea is that for each subgraph we have constructed, we first construct a constraint formula according to it. The constraint formula is then solved by a constraint solver. In this paper, Z3 is employed as our constraint solver. Given a constraint, it can generate an input of the program if the constraint is satisfiable. Setting this input as the input of the binary program will generate a path to the untraversed branch. If Z3 returns unsatisfiable, then the selected path is invalid and we will generate the new input according to another unexplored path.

The whole procedure of our test input generation process is demonstrated in Algorithm 1. The input is a target binary program *bin*, and an input *fin* selected from fuzzing. We first use dyninst and Angr to obtain the instrumented binary ins_bin and the CFG cfg, respectively. Then the function DynamicExecution is invoked to obtain the execution path of fin. The function UntraversedBranches identifies those untraversed opposite branches of the path according to the recorded data of AFL and stores them in n_list . To generate the condition constraints of these untraversed branches, the path is mapped to cfg. The function MissedNodesAdd detects those missed system call nodes and adds them to the path in the CFG. For each untraversed branch n, PathConstruct constructs a subgraph which contains all nodes of the path leading to n. Given a subgraph, the function ConstraintGeneration generates a constraint formula, which is then solved by Z3 using the function *Solve*. If a valid input is generated, then we have obtained a new test input and the procedure terminates. Otherwise, the selected untraversed branch is unreachable and we need to continue the loop and analyze another untraversed branch.

5. Experimental Evaluation

We implemented Tinker based on the open source tools AFL 2.33b [14], Angr [13], dyninst [12], and Z3 [15]. AFL is used as the fuzzer in Tinker and we extended it with an information collection interface to dynamically obtain the number of explored different paths to compute GRPC. Angr is a python-based binary analysis framework integrating a variety of existing analytic techniques. In Tinker, it is used to translate the target binary program into an intermediate representation, and generate its CFG. Dyninst is used to generate the instrumented binary and Z3 is used as the solver to generate a new input of an unexplored path.

Our experiments were performed on a machine with four Intel Core i7-6700 cores and 16GB memory. The operating system is 64-bit Ubuntu 16.04 LTS. Although AFL supports parallelism, it will produce deviations and influence the comparison results. Hence, we used only one fuzzing process in the experiments.

We use the DARPA CGC sample binaries [16, 17] as our benchmark to evaluate the vulnerability mining capability of our method. However, these binaries can only run under the DARPA Experimental Cyber Research Evaluation Environment (DECREE), which is a simplified OS with only seven system calls. Tinker aims to find vulnerabilities in real-world binaries. Therefore, we select the data set CB-multios provided by TrailofBits team [18], who migrated the DARPA CGC benchmark from DECREE to Linux.

For the 244 binary programs in CB-multios, we filtered some special types of programs that could not be used in fuzzy test analysis.

We performed our experiments on the remaining 211 programs. To evaluate the efficiency of our method, we compared the results of Tinker with that of Angr and AFL-qemu, which employ symbolic execution and pure fuzzing techniques for vulnerability mining, respectively. Both of them are state-of-the-art binary analyzers. We use two hours as the time limit. For the symbolic execution in Angr and Tinker, the depth of loop exploration is limited to 500 to avoid explosion. The more programs can be triggered to crash, the stronger a tool's vulnerability mining ability will be.

The experimental results are shown in Figure 5. Tinker detected 116 vulnerable programs, AFL found 96 vulnerable programs, and Angr found only 47 vulnerable programs. Angr found 10 vulnerable programs which were not detected by AFL. Tinker found all the 106 vulnerable programs that AFL and Angr have found, and it found 10 more vulnerable programs which were not detected by either Angr or AFL. These results suggest that under the same time limitation and conditions, Tinker can usually find more program crashes than existing binary fuzzing and symbol execution

techniques.



Figure 5: The crashed programs found by Tinker, AFL and Angr in DARPA CGC benchmark.

To further analyze the efficiency benefited from our method, we compare the number of explored paths within two hours between Tinker and the traditional fuzzing which contains no fuzzing intervention. All the 116 vulnerable programs that have been detected by Tinker are considered as the benchmark. The comparison is shown in Figure 6.

In this figure, the examples are divided into 3 categories based on the improvement rate. The Y-axis represents the number of examples for each category. From this figure, our method explores more paths than traditional fuzzing for all examples. Particularly, our method searched more than 20% of paths for 26 examples, searched 10%-20% more paths for 36 examples, and searched less than 10% more paths for 54 examples. We further studied the programs in which the efficiency is improved less than 10%. Most of these programs contain simple string comparisons which can be well handled by traditional fuzzing. For those programs with complex structures, our method can usually obtain more than 20% of the efficiency. In this sense, Tinker is more suitable to deep vulnerabilities in practice.



Figure 6: Efficiency benefited from fuzzing intervention.

6. Related Work

A variety of binary fuzzers have been proposed in recent years. Most of them focus on seed generation. Some fuzzers try to generate highly structured input, such as Skyfire [19]. The others mainly aim to improve the efficiency of fuzzing, such as AFL-lafintel [20], which converts a program into LLVM IR [21]. Moreover, most of the fuzzers need to work at the source level. Although there exist some fuzzers such as AFL-qemu, AFL-dyninst [22] and Aflpin [23], which support binary programs without source code, their efficiency for vulnerability mining still requires further improvement.

Symbolic execution is one of the most successful techniques for binary program analysis. Tools adopting this technique include Angr [13], Mayhem [24] and S2E [25]. The main problem for this technique is the path explosion problem. In order to mitigate this problem, various approaches have been proposed (e.g. veritesting [26]).However, existing techniques and tools are still not effective enough for real binary program analysis.

To address the problems faced by above methods, white box fuzzing has been proposed and widely studied in recent years. It explores analysis methods to guide fuzzing to generate more effective test cases (e.g. Steelix [27] and Vuzzer [28]), such that the path coverage of fuzzing can be improved. However, due to the high frequency of additional analysis invoking, this technique usually brings heavy load.

Driller [9] is the closest work to Tinker. If fuzzing falls into a blocked state, it uses concolic execution to guide fuzzing to a new path. Compared with Driller, Tinker has a more efficient and flexible efficiency evaluation method which makes it more sensitive to the current state of fuzzing. In addition, Tinker employs Angr for CFG generation which has rewritten most of the system calls and can generate a CFG with those system call nodes. Hence, Tinker works well for real binary programs, while Driller can only be applied to binaries running on DECREE.

7. Conclusions

This paper presents a new fuzzing framework named Tinker. Unlike traditional white box fuzzing techniques, Tinker uses GPRC to measure the current work state of fuzzing, and invokes a symbolic analysis approach when fuzzing is in blocked or low-speed states. In the symbolic analysis approach, new input is generated using symbolic execution to guide fuzzing jump out of the trap. Tinker is implemented based on open source tools AFL, Angr, dyninst and Z3. Experiments on DARPA CGC benchmark show that Tinker has higher efficiency in vulnerability mining than the other stateof-the-art fuzzing tools such as AFL and Angr. In future work, we will try to integrate existing efficient symbolic execution algorithms in to our approach so that the vulnerabilities in more complex binary programs can be detected.

8. Acknowledgement

We would like to thank the authors of ANGR and AFL for opening their source code. This work is funded by National Natural Science Foundation of China (No.61690203, No.61532007), and 973 National Program on Key Basic Research Project of China (No.2014CB340703).

References

[1] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery.* Pearson Education, 2007.

- [2] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling blackbox mutational fuzzing," in *Proceedings of the 2013 ACM SIGSAC* conference on Computer & communications security. ACM, 2013.
- [3] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in ACM Sigplan Notices, vol. 43, no. 6. ACM, 2008.
- [4] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009.
- [5] J. C. King, "Symbolic execution and program testing," *Communica*tions of the ACM, vol. 19, no. 7, pp. 385–394, 1976.
- [6] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.
- [7] U. Kargén and N. Shahmehri, "Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing," in *Proceedings of the 2015 10th Joint Meeting on Foundations* of Software Engineering. ACM, 2015, pp. 782–792.
- [8] "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.
- [9] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.
- [10] "Cyber Grand Challenge," http://archive.darpa.mil/cybergrandchallen ge/about.html.
- [11] S. L. Marple and S. L. Marple, *Digital spectral analysis: with applications*. Prentice-Hall Englewood Cliffs, NJ, 1987, vol. 5.
- [12] "Dyninst api." http://www.dyninst.org/dyninst., 2005.
- [13] "Angr," http://angr.io/index.html.
- [14] "AFL-qemu," http://lcamtuf.coredump.cx/afl/technical_details.txt.
- [15] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Inter*national conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008, pp. 337–340.
- [16] J. Song and J. Alves-Foss, "The darpa cyber grand challenge: A competitor's perspective," *IEEE Security & Privacy*, vol. 13, no. 6, 2015.
- [17] "CyberGrandChallenge samples," https://github.com/CyberGrandChallenge/samples.
- [18] "DARPA Challenge Binaries on linux and os x," https://github.com/ trailofbits/cb-multios/.
- [19] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Security and Privacy (SP), 2017 IEEE Symposium on.* IEEE, 2017, pp. 579–594.
- [20] "Circumventing Fuzzing Roadblocks with Compiler Transformations," https://lafintel.wordpress.com/2016/08/15/circumventingfuzzing-roadblocks-with-compiler-transformations/.
- [21] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedbackdirected and runtime optimization.* IEEE Computer Society, 2004.
- [22] "AFL-dyninst," https://github.com/vrtadmin/moflow/tree/master/afldyninst.
- [23] "AFLPIN," https://github.com/mothran/aflpin.
- [24] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Security and Privacy (SP), 2012 IEEE Symposium on.* IEEE, 2012, pp. 380–394.
- [25] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," ACM SIGPLAN Notices, vol. 46, no. 3, pp. 265–278, 2011.
- [26] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014.
- [27] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the* 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017, pp. 627–637.
- [28] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings* of the Network and Distributed System Security Symposium (NDSS), 2017.