A Survey of Symbolic Execution for Software Security

Guowei Yang[1], Maple Hu[1]
[1]ShanghaiTech University

1 Introduction

Many security vulnerabilities are often found by the implicit software bugs. For this purpose, many security testing applications have been developed to checking certain properties of a program hold for any possible usage scenario. For instance, the black-box testing[1] can test target program without the accessibility of source code and internal states. Another testing approach that could exploit full information from the target program is called white-box testing[2]. There also exists a gray-box testing[3] in the mid of two mentioned ways, which could attain the internal states of target program but no available source code. The black-box is the most efficient but not effective, because it would test the program using probably random inputs. As practical software program, which typically designed to accept highly structured inputs that covered by a determined space. The black-box may not perform well in such situations that we expect find as many bugs as we can in certain time, hence we prefer the white-box testing or gray-box testing when source code or internal states are available.

However security testing is often more serious than other problems. When we find somewhere in the program may be vulnerable, we should verify whether there exists a input to reach that point. Static program analysis for white-box testing would lack the capability. Fortunately dynamic program analysis could mitigate this deficiency. Such as fuzzing with the instrumented target program that we can get more precise trace information to generating interesting input for achieve high coverage. AFL[4] is a one of best-known fuzzer, which is the prototype of many other brilliant fuzzers[5, 6, 7, 8, 9]. For reaching more structural coverage, symbolic execution has been incubated in dozens of fuzzing tools by using various heuristic strategies. Symbolic execution provides a elegant solution to input generation, by systematically exploring all possible paths without requiring concrete inputs[10]. Furthermore, symbolic execution could reduce the inputs space, and even it could generate the specific input straightly to reach expected point.

Symbolic execution is not only glowing at hunting vulnerabilities, but also appearing in other interesting security fields. For instance, reproducing the case from rare error report is obviously hard work. Backward symbolic execution is a excellent method for solving it [11]. Automatic exploit generation(AEG) is such right place for symbolic execution[12]. AEG requires more constrains than

```
int checkValves(int wait1, int wait2) {
    int count, i;
    while(wait1 > 0) wait1--;
    count = 0, i = 0;
    while(i < size){
        int status = getStatusOfValve(i);

        if(status == -1) {
            count++;
        }
        i++;
    }

    if(count > 2)
        printf ("ALARM");
    }
    while(wait2 > 0) wait2--;
    return count;
 }
```

(a) checkVales function

```
int getStatusOfValve(int i){
    if(i < 0 || i >= size){
        printf ("ERROR");
        exit(EXIT_FAILURE);
    }
    int status = valvesStatus[i];
    return status;
}
```

(b) getStatusOfValve function

Figure 1: A simple program

just searching vulnerabilities, we should construct a proof of concept automatically to trigger a real security event. Environment-triggered malware detecting also require generating proper environment input to identify the function of malware[13]. Symbolic execution is fast-growing in security testing, but its still hard to implement in the real-world industrial software because symbolic execution is too heavy to use[10]. Therefore there comes up with many solutions to optimize the performance of symbolic execution[14, 15, 16, 17, 18]. The goal of this survey is to provide a overview of the main ideas, challenges, and solutions that developed in the security area by using symbolic execution.

The remainder of the article is organized as follows. In Section 2, we introduce a example to show how normal symbolic execution works. In Section 3, we summarize several known challenges in the implementation of practical symbolic execution engines. In Section 4, we make an overview of symbolic execution techniques in software security. In Section 5, we discuss the future of symbolic execution.

2 A Motivated Example

As shown in Figure 1. Firstly, we briefly give the detail introduction to the listed program. The `valvesStatus` is fixed-size array bounded by `size`. Suppose the entry point is the beginning of function `checkValves` and all variables that not explicitly defined in the figure are underdetermined (that is we should

use a fresh symbol to represent these variable in symbolic execution). We are supposed to reach the point at line 15. Note that the program point we often focus on are the ones that always the places where may occur errors. We use the symbols to replace the value-unknown variables to continue running program. In the beginning of function `checkValves`, our variables table of current context should be

$$\sigma = \{(\texttt{wait1}, w_1), (\texttt{wait2}, w_2), (\texttt{size}, s_1), (\texttt{valvesStatus}, [v_1, \cdots, v_{s_1}])\}$$

we often call σ as a program state, or simply state. It also can be represented by a map $\sigma : \mathcal{V} \to \mathcal{D}$. Before we do symbolic execution on the program, we should give a path that we are going to explore. In the static symbolic execution, we should select the path by a static well-designed path exploring algorithm. In contrast, using the information in program running to guide path selection is called dynamic symbolic execution or concolic execution. We only explain how static symbolic execution works here. There are two main places that we have to make a choice in the path selection. The first one is the branch conditional statement i.e., if-else statements with two branch. The second one is the unbounded loop, we have to decide the times of loop repetitions. We select `while` at line 3 is taken false in first time. Then We get the first condition that $\varphi_1 = (\texttt{wait1} \leq 0)$ to characterize our selection. The φ_1 is a conjunct of selected path π such that $P_\pi = \varphi_1 \wedge \varphi_2 \wedge \cdots \varphi_n$ where $\varphi_2, \cdots, \varphi_n$ are remainder conjuncts of path condition. After line 4, add $(\texttt{count}, 0)$ and $(\texttt{i}, 0)$ to state. We go in `while` at line 5, then P_π should be $(\texttt{wait1} \leq 0) \wedge (\texttt{i} < \texttt{size})$. With the function call at line 6, we go in `getStatusOfValve`. We give the parameter `i` of `getStatusOfValve` a new name i_2 to avoid ambiguity with i in current state. In the beginning of `getStatusOfValve`, add $(i_2, *)$ (we use $*$ for a fresh symbol) to σ and add conjunct $i_2 = i$ to P_π. The branch condition at line 22 should be evaluated to false by P_π. So we don't have to make a choice for this one. In the end of `getStatusOfValve`, the v_i is returned to `status` at line 6 and add $(\texttt{status}, v_i)$ to σ. To keep our state clear, we remove the local variables in last function scope before return. At line 8, we take the true branch and add $(\texttt{count}, 1)$ to σ. Something subtle happened at line 11. There is conjunct $(\texttt{i} < \texttt{size})$ in P_π, the conjunct become unsound after line 11. This relate to the core of symbolic execution–dynamic logic, or more specifically strongest postcondition. Given a condition P and a command c, you may ask what condition Q can we get from P after c, the Q is called postcondition. They are often in a form $P\{c\}Q$. That is actually means that we have state satisfied P before running c, then we can get state satisfied Q after running c. From this point of view, the weakest postcondition is trivial as $true$, thus we only care the strongest one. So we need to construct a postcondition operator sp for every command c inside the target programs. For instance, the $\text{sp}(a := e)$ is

$$\overline{P[e/x]\{x := e\}P}$$

where $P[e/x]$ is condition that substitutes all x in P with e. Follow the above

rule, the post condition of line 11 should be

$$(\texttt{wait1} \leq 0) \wedge (\texttt{i} < \texttt{size})\{\texttt{i} = \texttt{i} + 1\}(\texttt{wait1} \leq 0) \wedge (\texttt{i} - 1 < \texttt{size}),$$

and update $(\texttt{i}, 1)$ in σ. Then we take false branch in line 5 for leaving the loop, add conjunct $\texttt{i} \geq \texttt{size}$ to P_π. Finally we reach the point at line 14, the condition $\texttt{count} > 2$ will be $false$. Thus $P_\pi = (\texttt{wait1} \leq 0) \wedge (\texttt{i} - 1 < \texttt{size}) \wedge \texttt{i} \geq \texttt{size} \wedge \texttt{count} > 2 \equiv false$, that means the selected path to line 15 is infeasible. We can choose another path to do the same work, there are some algorithms called path exploring to achieve these works systematically.

From the simple example, we showed how naive symbolic execution works. We mentioned two critical components state and path condition, the state could be integrated into the path condition sometime. However, we selected a unsuccessful path to the target point in the example, the reason is that we are going to leave a interesting problem here for that. That is how we can use the symbolic execution techniques to quickly solve our own question. In addition to this problem, we will also discuss some of the details in the symbolic execution process that we described earlier.

3 Challenges in Symbolic Execution

At present, symbolic execution mainly faces three challenges: path explosion, memory model, constraint solving. (1) **Path explosion**: When performing symbolic execution on a program, the current state may be forked off at each branch of the program, and often more than once at the loop, so the number of states in the entire program can reach exponential levels in the number of branches. The space cost of keeping track of the large number of branches that will be explored is very high, while the time cost of solving for exponential levels of states is even more unacceptable. The cost in time and space limits the scalability and accuracy of symbolic execution. (2) **memory model**: In mainstream symbolic execution tools, the memory is modeled using a linear address space, assigning a fixed size for each memory object, and mostly using forking model for multiple resolution of memory objects. This model basically realized the modeling of concrete memory, but the expressive is definitely limited and thus poses many problems. For example, it is difficult to handle objects that change size dynamically, and it may also cause problems when dealing with complex pointer-based data structures. (3) **constraint solving**: Although the efficiency of SMT solvers has improved significantly in recent years, SMT is still inherently an NP-hard problem. The vast majority of the time cost in symbolic execution is constraint solving. Recent works focus on how to use the characteristics of the analyzed program to improve the efficiency of the solving, or use other methods to bypass unnecessary SMT solving.

3.1 Path Explosion

Path explosion is one of the most common and long-standing challenges that has received the most attention. Much classical works has been proposed to address path explosion. The ability of a symbolic executor to handle a large number of states directly determines its scalability. When solving a constraint of a path, the solving process will be forced to terminate beyond a time threshold and then the branch is not explored. This decides the ability to explore the deeper part of the program and determines the accuracy of the analysis algorithm. In recent efforts to address branch explosion, some works are devoted to propose well-designed pruning strategy [15] to avoid exploring parts of the program that have no effect on the result. Others combine symbolic execution with extra dynamic and static analysis methods to perform symbolic execution [14] only when necessary, thus reducing the scope of analysis from the whole program to a certain part of the program.

Chopped Symbolic Execution [15] uses static analysis techniques such as pointer analysis and program slicing to aid symbolic execution for pruning. It automatically detects uninteresting parts of the code, and also allows the user to specify the uninteresting parts of the code himself to avoid redundant branches being analyzed during the symbolic execution phase. The side effects of the pruned code may also have an impact on the code of interest. To avoid adding false positive and false negative results, when the symbolic execution phase has side effects in the code of disinterest outside the state, Chopper uses pointer analysis to determine the location of the instruction where the side effects occur, then slices the relevant part of the code, and finally selects the approximately smallest code fragment with side effects is selected and added to the program to be symbolically executed.

Driller [14] combines fuzzing and symbolic execution, leveraging their respective strengths as much as possible. fuzzing is lightweight to run and does not need to store large amounts of state. But for branches with complex conditional judgments, it is difficult to generate specific input data. For example `if(x==(1<<31)-1)...` , fuzzer can hardly generate exactly such large number to pass this branch and therefore cannot explore deeper part of this path. While symbolic execution can use SMT solver to easily get the input data and pass the branch, but symbolic execution on the entire program request larger cost. Therefore, Driller runs fuzzing first, invokes dynamic symbolic execution only if it cannot continue to reach a new state. The dynamic symbolic execution stage pre-constrains the input of user to the data generated by fuzzer, and then traces the previous path of fuzzing, using constraint solving to obtain special input data for the unexplored path. Then fuzzing is restarted again.

3.2 Memory Model

In the mainstream symbolic execution tools, the memory is modeled using a linear address space where each memory object has a fixed concrete size. For input data or memory objects of variable size such as strings, arrays, etc. may

lose coverage and reduce accuracy, so there are some works dedicated to symbolizing the address and size of memory objects, e.g., a symbolic-sized model[19] is proposed. In addition, the forking model is a popular method used in multiple resolving, where each memory object that can be referenced by pointers is forked each time a branch is reached, and the pointers on a path are restricted to refer only to the objects in this fork. But the forking model has some drawbacks, it is weakly scalable, prone to path explosion, and it cannot handle complex heap data structures well and thus loss of precision. A segmentation-based memory model[19] is proposed to improve the scalability and better support of complex heap data structures.

A segmented memory model[16] is proposed to address the shortcomings of the forking model commonly used in multiple resolution. When a pointer refers to multiple memory objects, such as a two-dimensional array, if there is a loop that dereferences on different locations of this array each time, the forked object model of mainstream symbolic execution tools will perform a fork in each loop, which not only leads to the path explosion problem, but also a large number of forks of the same symbolic pointer may also run out of memory quickly at runtime, and in addition, it cannot reason about all possible values on a given control flow path. The segmented memory model first uses pointer alias analysis to store all possible memory objects aliased to each other in memory segments in groups. This approach avoids forking because conservative alias analysis ensures that symbolic pointers can only point to objects within a single memory segment. Each memory segment has an upper limit on its size, so the method is guaranteed to have good scalability.

Bounded symbolic-size model[19] provides a method to represent the size of memory objects with symbols. They attach a symbolic expression to each memory object to represent the size and set a concrete value as an upper bound on the size. To avoid memory conflicts between objects during allocation, they add non-overlapping property to the path constraints. This approach introduces more symbols and constraints that increase the risk of state explosion, so some state merging strategies are also used to mitigate state explosion.

3.3 Constraint Solving

The main performance bottleneck in the symbolic execution is constraint solving. SMT is essentially an NP-hard problem, so a lot of time has to be spent on SMT solving during the analysis, and worse, many of the formulas corresponding to the branches may be too complex to be solved, so a number of paths in the program may not be explored, thus reducing both scalability and accuracy. Some recent work has been devoted to improving the performance of symbolic execution in the relevant part of the solution. One idea is to transform the constraints corresponding to branches into other more appropriate constraint solving problems according to the properties of part of the program and solve them with the corresponding solvers[17] . Another idea is to use some dynamic trick in the constraint execution to bypass the constraints solving procedure[20].

Constraints associated with arrays are often difficult to solve on SMT, but

are likely to be solved more efficiently by LIP solvers as an integer linear programming problem (LIP). Therefore a LIP-based approach[17] is proposed to optimize symbolic execution on array-dependent solutions. During the symbolic execution, the constraint is divided into a part with array terms and a part without array terms. For the part with array items, the associated index variables are extracted from each array read statement, and then the set of constraints associated with the index variables is solved using ILP to obtain upper and lower bounds on the index variables. Type inference is then performed on the code to obtain the type information of the array elements to determine the upper and lower bounds of the array elements and the size of individual elements. Then, based on the constraints obtained in the type inference and the constraints on the array elements in the code, a solution is performed using ILP to obtain the upper and lower bounds information related to the array elements. After that, it is possible to judge whether the constraints can be satisfied based on the interval information obtained from these two solves. In addition, if the constraint cannot be satisfied during these two solving processes, the solving process can be exited in advance.

Based on the synergy of symbolic execution and fuzzing, fuzzy-sat[20] expects to obtain information in the constructed symbolic expressions and solve the query without resorting to the SMT solver as much as possible. Specifically, the expression of a branch condition is b and the condition set on the path before current path is π. Then the driving input i that arrives at the current branch during fuzzing already satisfies $b \wedge \pi$, so if a mutation is performed using input i, a new test case i_0 can be constructed if it satisfies the condition $\neg b$ of another branch and the previous condition set π, so that dynamic symbolic execution is able to explore the other branch directly without performing SMT solving.

4 Implement Symbolic Execution in Software Security

4.1 Testing and Vulnerability Detection

In general, the symbolic execution technique does not appear along in a vulnerability detection strategy. It comes with other dynamic analysis like fuzzing and dynamic tainted analysis. Fuzzing is the most efficient method to hunt vulnerabilities today. To archive higher code coverage, combining symbolic execution is good choice. Tainted analysis can tell the symbolic execution engine which variable is under control by attackers. Then we can use symbols to replace these variables to explore more deep path. The method that integrate symbolic execution into dynamic analysis is called concolic execution or hybrid analysis.

DART [21] is a pure dynamic symbolic execution based unit testing technique. With a instrumented program, the path condition can be generated on the fly. The target program start with a random input, then a execution path and path condition be generated by following this input. The path exploring module will systematically choose a conditionl statement and flip the condition

in path condition that token by the previous path. The path condition solver will try to solve the new path condition and generate a input. Note that the path condition in DART is always linear. When meet the non-linear branch conditions in the program running, it will resolve the branch condition to a concrete value by querying the program state. It is obvious that the solution is not complete. DART will restart the program with a new random input to avoid the situation. Other dynamic symbolic execution techniques treat the problem in a lazy mode. The path exploring algorithm allows this happened, and solve it until the solution generated by SMT-solver is not consistent with the excepted path. That is the new input does not make the program reach the specific point, called divergence.

SAGE [2] is the testing tool that combines fuzzing and symbolic execution. It uses fuzzing to quickly reach normal branches, and solve the path condition that generated by symbolic execution on the fly to reach deep branches. The more deep branch reaches, the more complex path condition is solved. The most amazing thing is that fuzzing and symbolic execution appear to be naturally complementary. For instance, there is no general SMT-solver for non-linear conditions. That means one symbolic execution engine may fail to a path condition that included some such subformulas. The random seed generation algorithm could solve it with certain possibility.

Chipounov et al. [22] generalizes the combination idea to selective symbolic execution, which running a program in both concrete and symbolic with switch strategies. This idea is based on a fact that only few codes that has to be executed symbolically. Thus it can minimize the amount of code that has to be executed symbolically. There are six execution consistency models in the paper, which describe the portion between concolic and symbolic execution. We say an execution is consistent, which means there is valid input corresponding the execution. When we relax some path condition, the unsound result may be generated by solve the weaker condition. That is will make our execution is not consistent with original programs. The trade-off between performance and precision can be clearly discussed in these models. Each model is actually a different scenario that we may use in practice.

4.2 Reproduction

Jin et al. [23] proposes a general algorithm based symbolic execution for generating inputs to reproduce crash reports by collected execution data in client side. They treat the execution data as a ordered goals, which one goal is a statement in the target program. Following these goals, the symbolic execution engine has to explore paths covering these goals by order, and it chooses the shortest path between two continuous goals. In this paper, there are four types of execution data: points of failure, stack traces, call sequences, and program traces. The more execution data we have, the less time cost on input construction. But more execution data mean high overhead in collecting process at client side. Moreover the execution data have to avoid include user privacy. Therefore, the trade-off is an interesting open-problem. This paper also mentioned two kind

of symbolic execution for different reasoning direction. The forward symbolic execution is used in this paper. Reasoning start at the failure point is also a good point, many infeasible path could be reduce in early.

Pham et al. [24] proposes a crash reproduction tool for binary without requiring source code. There are two challenges has to be addressed: (1) The input for the binary are also binary (files), which the structure of the binary is unknown. Even though you have pre-knowledge about the inputs, you still do not know where each part of the inputs will be used in the target binary; (2) a execution of crash often relates to some modules that were loaded in the binary running. The first challenge make us hard to precisely modify the inputs to reach other points. The second challenge required all related modules have to analyzed. This paper solve the first challenge by performing a concolic execution with some selected seeds that can be reach the last crashed module but not trigger crash. Every seed corresponds a path constraints, each symbolic variable in the path constraints corresponds a fragment of seed. Then the structure of input is clear. A second challenge can be naturally solved by strengthen the path constrains of seeds with a another conditions with respect to the paths in the last crashed module. The path selection in the second challenges is solved by a algorithm called targeted search strategy.

4.3 Automatic Exploit Generation

Avgerinos et al. [12] proposes AEG that could fully analyze and exploit the vulnerability on the target system. The AEG works in five steps: (1) analyze the source code of target system with static program analysis techniques; (2) if there is a vulnerability found in the pre-analysis, then provide a input to trigger the vulnerability by solving the path constraints; (3) perform a dynamic analysis on the binary of target system with the input generated in step (2); (4) according to the collected runtime trace information such that memory layout and other local environment dependencies, enhance the path constraints that constructed for trigger the vulnerability in the step (2) with extra condition that the user input must include shellcode and gadgets of control flow hijacking. (5) try to solve the new path constraints in the step (4), if we can get a input from that, then feed it to the binary to verify the exploit works. The automatic exploit generation techniques are definitely complex, it requires both source code analysis and binary analysis.

Xu et al. [25] focuses on bypassing the security defenses in the real world programs such that DEP(data execution prevention) and ASLR(address space layout randomization). They use the ret2libs to bypass DEP, which is one of well known technique of ROP(return oriented programming). For bypassing ASLR, they use the specific gadget `JMP ESP` in some fixed area to hijack the control flow without knowing some address in the stack.

The automatic exploit generation is a one of the most interesting area in the software security. However, only a few simple types of vulnerabilities can be exploited automatically by the existed related tools. Furthermore, the exploitation generated by AEG tools often lack the robustness, that is the exploitation

may only can be used in some certain operational systems. But then again, constructing a exploitation of a vulnerability is even not cheap for an experienced security expert sometime.

4.4 Malware Detection

Baldoni et al. [26] shows how to analyze a windows malware with a symbolic execution assistant performed on Angr. Firstly, they extend the binary analysis framework in Angr to support 57 Win32 API. Then the analyzer can model the path constraints in the system calls. Modeling external function is one of the challenges to implement symbolic execution in practice. Sometime we do not want to the engine modeling path condition over application self and underlying system in the same time. Since the cost is not cheap and the path condition will be large and complex that hard to be solved. So we often give summaries to model external environment such that third-party libraries, local configuration, and system calls. The summaries are basically formal conditions with some free variables depending on the context. For reducing the path explosion, they control the symbolic variables by identifying the precise taint source, and limit the buffer length of symbolic variables. The paper shows symbolic execution is an excellent tool to quickly identify malware's actions that controlled by remote attackers.

Brumley et al. [13] proposes a heavy environment-trigger malware detection system for called MineSweeper, it consists of a mixed execution engine for supporting concrete and symbolic execution on the fly and a binary runner or sandbox for exposing malware behaviors. The environment-trigger malware is highly dependent on the host environment such that system time, local files and network inputs. If the current environment does not satisfy the required conditions, the malware will behave as a normal software. The mixed execution engine accept a binary and trigger type specification for assigning symbols to related variables. Note that mixed execution engine does not perform two mode executions in parallel. If one of operand of instruction is symbolic, then use symbolic execution evaluate it. Otherwise do concrete evaluation. There is a path predicate will be generated after mixed execution. Then the solver provide a solution, which includes trigger inputs that satisfied the path predicate. The trigger inputs is not typical string that you can feed it at the beginning of program running. So the runner should has to feed these trigger inputs on demand for certain points.

5 A Short Discussion

In this paper, we briefly provide an overview of symbolic execution techniques that have developed in the field of software security. With the development of optimization methods related to symbolic execution techniques, these techniques are no longer just experimental tools, but are gradually being applied in practice. That being said, there are still some challenges in the real-world

application of these techniques. For example, strong post-condition operators are typically constructed by hand. Additionally, when a new feature is introduced to a programming language, the symbolic semantics of that feature must be added to the relevant symbolic execution engine. Developing a symbolic execution engine for a dynamic language like JavaScript or PHP can be a particularly difficult task, due to the expressive nature of these languages. There are also relatively few open-source symbolic execution engines available for specific domains, which can make it difficult for others to build upon existing work. Despite these challenges, symbolic execution remains a valuable and precise method for synthesizing specific executions, and we hope that future work in this area will continue to progress and be open to researchers.

References

[1] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *2010 IEEE symposium on security and privacy*. IEEE, 2010, pp. 332–345.

[2] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[3] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[4] M. Zalewski, "American fuzzy lop," 2017.

[5] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.

[6] K. Serebryany, "libfuzzer–a library for coverage-guided fuzz testing," *LLVM project*, 2015.

[7] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1032–1043.

[8] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "$AFL++$: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[9] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "$kAFL$:$Hardware-Assisted$ feedback fuzzing for OS kernels," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 167–182.

[10] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[11] N. Chen and S. Kim, "Star: Stack trace based automatic crash reproduction via symbolic execution," *IEEE transactions on software engineering*, vol. 41, no. 2, pp. 198–220, 2014.

[12] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[13] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*. Springer, 2008, pp. 65–88.

[14] 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, no. 2016, 2016, pp. 1–16.

[15] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 350–360.

[16] T. Kapus and C. Cadar, "A segmented memory model for symbolic execution," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 774–784.

[17] Z. Shuai, Z. Chen, Y. Zhang, J. Sun, and J. Wang, "Type and interval aware array constraint solving for symbolic execution," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 361–373.

[18] V. Sharma, S. Hussein, M. W. Whalen, S. McCamant, and W. Visser, "Java ranger: Statically summarizing regions for efficient symbolic execution of java," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 123–134.

[19] D. Trabish, S. Itzhaky, and N. Rinetzky, "A bounded symbolic-size model for symbolic execution," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1190–1201.

[20] L. Borzacchiello, E. Coppa, and C. Demetrescu, "Fuzzing symbolic expressions," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 711–722.

[21] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.

[22] V. Chipounov, V. Kuznetsov, and G. Candea, "The s2e platform: Design, implementation, and applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, pp. 1–49, 2012.

[23] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 474–484.

[24] V.-T. Pham, W. B. Ng, K. Rubinov, and A. Roychoudhury, "Hercules: Reproducing crashes in real-world application binaries," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 891–901.

[25] L. Xu, W. Jia, W. Dong, and Y. Li, "Automatic exploit generation for buffer overflow vulnerabilities," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2018, pp. 463–468.

[26] R. Baldoni, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Assisting malware analysis with symbolic execution: A case study," in *International conference on cyber security cryptography and machine learning*. Springer, 2017, pp. 171–188.