# Concolic execution

**Seminar: Understanding of configurable software systems**

Bipin Oli

Advisors: Prof. Sven Apel,
Christian Hechtl

Saarland Informatics Campus,
Saarland University

## 1 Abstract

Configurable systems with many dials and knobs brings in a big testing challenge. In presence of many possible variants and configuration options, it is very important to automate the testing as much as possible. Directed automatic random testing, popularly known as concolic execution is a primary way how it is done. Concolic execution is a software verification technique that performs symbolic execution together with concrete input values. Concrete values are selected with the help of a constraint solver to guide a program flow in a particular direction. The selection of concrete values helps to scale the verification to a larger program as it makes the symbolic constraints smaller by selecting specific branches in the program. Compared to random execution, this allows us to guide the analysis in a direction likely to have bugs which makes this technique powerful. However, in doing so, we sacrifice the completeness of the analysis in favor of the depth of analysis. The sheer number of branches in a large program makes it difficult to perform a complete analysis, so we have to attack this problem in a different way. This is called a path-explosion problem. There have been many studies to deal with this problem. In this paper, I have categorically presented them.

## 2 Introduction

Software is an integral component in many aspects of modern life. Many critical systems are driven by software. Complex relationship between man and machine, and the variety of applications has produced complex software. Much of this complexity is inherent in the problem, however the act of implementation also brings in its own complexity. Furthermore, due to the dynamic nature of human needs the software is required to evolve and change all the time. It is expected to operate under various conditions and use cases. This has given rise to configurable softwares with lot of configurable options and variants. Many complex software systems are configurable today. Configurable options allow users to select the right options as per their need to tailor-made the system. This flexiblity allows users to tweak the system in myrid of ways to produce a varient that suits them. The possible number of variants a user can choose from can be exponentially

huge with a lot of options and features interdependent on each other. This also produces many complexity and chances of failure in case of misconfigurations. In practice in the industry [16, Holistic configuration management at faceboo], there have been many examples of major outages and security branches with it's origin at the misconfiguration of a configurable software system and buggy inter-option relationships. Due to this it is very important to verify the software to use it with confidence. However, the complexity, pace of change of the software, it's size and the sheer number of inter-dependencies and variants makes it very challenging to properly verify the software.

There are various functional and non-functional factors on which a software needs to be verified. This paper primarily focuses on the verification aspect which deals with the correct execution of a software under different input conditions going through possible execution paths in a program.

One way to test a software is by providing it with random inputs. It is a black-box testing method where we don't know the inner details of the program, so it is not possible to know if the input has covered all the states of the porgram. With random inputs, it is very unlikley to get the exact condition of the branch. So, we would be running different inputs without progressing any further in the program exploration. For the deeply nested state, the probability of reaching that state is practially zero. This makes it impractical to cover various states of the program. However, in abscence of other alternatives, this method is still used. Specially, a close cousin of random testing called fuzzing [14] has proven its usefulness. Fuzzing is a method of inputing gibbrish bytes of random input to a system to see if that crashes the system. It differs from random testing in a way that we are not inputing random valid values but invalid gibbrish to the program to see if the system handles that gracefully.

An alternative to random testing is symbolic execution. It is a white-box method of testing where we can see the instructions of the program, because of which, we can execute the program with symbolic inputs. We evaluate the expressions in terms of symbolic inputs and the constraints of symbols representing the conditional branches interms of those symbols. We can use constraint solver to find concrete values for the symbols satisfying the constraints. Compared to random testing, this allows us to cover the parts of program which is very hard to reach with random inputs. Unfortunately, symbolic execution doesn't scale. The number of constraints grows exponentially with the size of conditionals which makes it impossible to go deeper into the program. Also, the program can depend on external libraries and environmental factors which cannot be solved by a constraint solver. Constraint solver also struggles to solve for certain constraints such as finding an input that satisfies the output of hash. Furthermore, due to the path selection heuristic of the constraint solver the symbolic execution can get stuck, going through the same path again and again. Due to these things, the analysis gets stuck and struggles to reach a deeper state which makes symbolic execution insufficient in many practical uses.

Directed automated random testing [9, DART], more commonly known as concolic execution, is a technique that combines symbolic execution along with

concreate inputs to mitigate the challenges faced by symbolic execution. When the branch is reached, a constraint solver can be used to find a concreate values that satisfies the condition. Then, the condition of the branch is negated and fed to the constraint solver to see if the other branch from the conditional is reachable. Whenever the constraint solver struggles to solve the constraints, a random concreate value can be used to simplify the process. This way concolic execution doesn't get stuck in the middle of execution.

Concolic execution has been widely used in practice to find many security bugs. SAGE [10] tool from microsoft is a prime example of it. SAGE is a white-box fuzzing tool. It fuzzes the inputs, notably files, under the guidence of concolic execution which helps SAGE to perform fuzzing with good code coverage. Microsoft has been using SAGE since 2007 and it has found many bugs in big applications. Microsoft reported in the technical paper of SAGE [10] that SAGE helped them find one-third of all the security-related bugs in Windows 7 and they have been running SAGE 24/7 since 2008 on 100 plus machine/cores in the Microsoft security testing labs. It has processed over billions constraints. Other open-source tools such as KLEE [2] have also been widely used in industry.

A major issue while concolic execution is that it is not feasible to test all the execution paths of the program. The number of paths of execution grows exponentially with the number of branches in the program. This is callend a path-explosion problem. As, it is not possible to exhaustively test all the paths in the program, many approaches have been proposed to attack this problem in different ways, such as: by prioritizing the branches likely to lead to a bug, by decomposing the analysis into smaller components, etc. I have categorially presented various approaches.

## 3 Approaches

### 3.1 Analysis on compositional parts

Earliest ideas to deal with path-explosion were based on performing analysis compositionally. Chakrabarti et al. (2006) [6] presented an approach for identifying data and control inter-dependencies using the static program analysis. The idea is to group together highly-intertwined components together and separate the program into smaller pieces such that they can be individually tested. They used popularity of the component and the sharing of code to determine the split threshold. If the function is very popular and is being called from a lot of places then it is likely to be not closely linked to any component, whereas if two functions share many of the same functions then it is likely that the higher level operation they perform is close to each other. They evaluated the effectiveness of the idea by implenting it against the open source implementation of telephony protocol for call establishment called oSIP protocol (http://www.gnu.org/software/osip/osip.html), showing that the automatic program partitioning can be used to test software without generating many false alarams caused by unrealistic inputs being injected at interfaces between the units.

In 2007, Godefroid et al. [8] further proposed a complementary idea [8] to Chakrabarti et al. [6] in the context of performing concolic execution compositionally by adapting the known known techniques of static analysis. They proposed an algorithm called SMART standing for Systematic Modular Automated Random Testing, as a more efficient search method than DART [9] without compromising in the level of completeness provided by DART. SMART works by testing functions in isolation and by collecting the testing results as function summaries which are expressed using the function inputs are preconditions and outputs as post-conditions. The function summaries are re-used to test higher level functions. Starting from the top-level function of the call-flow graph the summaries are calculated in a top-down way dynamically. Using the initial random inputs the summaries are calculated from top-level function until the execution terminates. Using the DART then the analysis is backtracked computing summaries in each of those executions. When the analysis is run again the earlier computed summaries can be re-used without having to perform detailed analysis again. Due to this, SMART improves the exponential program execution of DART to linear, making it much more scalable.

### 3.2 Hybrid of random and concolic execution

Majumdar et al. (2007) [13] proposed an algorithm that combines random testing and concolic execution. Their idea is to perform random testing until saturation and switch to concolic execution from there to explore the neighbouring states exhaustively. Random testing helps to reach the deep state of the program quickly whereas concolic execution from there helps to widen the reach, thereby helping in deep and wide exploration of the program state space. First the algorithm starts with random testing keeping track of coverage points. When the test saturates i.e the coverage points doesn't improve any further, it switches to concolic execution from that program state. Using concolic execution the algorithm tries to find an uncovered point. When it founds one, the algorithm switches back the random testing mode from there. This swtiching back and forth between random testing and concolic execution brings in the benefits for both methods. Random testing inexpensively allows for a long program execution to a deep state and concolic execution helps to symbolically search in a exhaustive way for a new path.

Majumdar et al. [13] compared this algorithm to separate individual random testing and concolic testing on the VIM text editor (150K lines code in C) and their implementation of popular red-black tree data structure. They found that the hybrid testing consistently outperforms the two methods interms of the branch coverage. Furthermore, as the hybrid method relies on faster and cheaper random testing to explore the program state as much as possible which helps in avoiding the expensive constraint solving whenever possible. This avoidance of constraint sovling doesn't hamper the ability to explore the program state as the algorithm switches back to local exhaustive search whenever necessary. Concolic execution as proposed in DART [9] by itself can get stuck in exploring a huge number of possible paths which prevents it from reaching a particular state of

interest. Random testing can get saturated, never being able to push through a branch. Thus, the combination of them in this hybrid method helps to avoid both of those limitations.

Even though the hybrid method shows the clear benefits over the individual methods, it still suffers from the limitations of the concolic execution. The discovery of new path of coverage depends on the capacity of the constraint solver and the exhaustive local search which suffers from path-explosion. Thus, this method may not achieve 100 percent coverage but it can improve the coverage considerably.

### 3.3 Heuristic based approaches

A different approach in dealing with the path-explosion problem in concolic execution is by prioritizing the path likely to lead to the discovery of bug. There have been many studies in heuristically selecting the path.

In the 2008 paper [1] from Burnim et al, they proposed several heuristic search strategies. They proposed a search strategy guided by the control-flow graph of the program. The main idea was the greedy strategy to choose the branch based on the the distance to the uncovered branches in the CFG. This branch would be the one to be negated for, in the concolic execution process. They also proposed random search strategies. Compared to traditional random-input testing, in their strategies they proposed to randomly select the execution paths instead selecting sample form the uniform distribution of all possible program paths. They implemented the proposed strategies and open-sourced the implementation under the name CREST which is an implementation in the C programming langugae. Further, they tested their strategies on grep 2.2 (15K lines of code) and Vim5.7 (150K lines) and found a much better and faster branch coverage compared to the traditional depth-first search strategy of concolic execution.

Xie, Tao and Tillmann et al. proposed a fitness-guided approach for path exploration (2009) [18]. They proposed an approach called Fitnex that uses state-dependent fitness values to guide the exploration of paths. The fitness values are calculated through a fitness function. Fitness function calculates how close is the discovered candidate path to the test target i.e a branch that has not been covered yet. They combined this idea of fitness along with the known heuristics to handle the cases where the fitness would fail. A fitness value is calculated for the paths that have already been explored. Using the fitness values the paths are prioritized. During the path exploration a branch is selected based on this fitness gain.

In 2014, Seo, Hyunmin and Kim, et al. proposed an idea [15] which considers context information of how the execution reaches the branch called Context-guided search (CGS). This idea is different from the traditional search strategies in a sense that traditional search strategies primarily focus on the branch coverage information in the branch selection process. Context-guided search works by exploring the branches in the current execution tree where each visted branches are considered if they should be selected for the next input or should they be skipped. The context is calculated by looking at the way execution reached the

current branch. The context of preceding branches and dominator information is used in the calculation of this context. To select the branch for the next input, context-guided search compares the context with the contexts of previously selected branches which are stored in the context cache. The branch is selected for the next input if the context of it is new, otherwise it is skipped from the next input.

Wang, Xinyu and Sun, et al. (2018) proposed an idea [17] of defining optimal strategy based on the cost of constraint solving and the probability of the path in the program. Thus, the problem of determining the best course of action can be modeled as a model checking problem of Markov Decision Process (MDP) with costs. As a result, the exsiting algorithms and tools can be used in solving the problem. Using this, they tried to systematically decide when to perform symbolic execution and when to perform concrete execution along with the program path to apply symbolic execution to. The computation of the optimal strategy has a high complexity so they proposed a greedy algorithm to approximate the optimal strategy.

So far, many individual heuristics were proposed but the selection of them were mainly manual. In the 2018 paper titled "Automatically Generating Search Heuristics for Concolic Testing" [3] Cha, Sooyoung and Hong, et al. introduced a way to automatically select a suitable heuristics based on the parameters. They also proposed an optimization algorithm to find a good values for parameters in an efficient way. They created different classes of search heuristics and defined in the context of parameters. The selection of the heuristic is then determined by the value of parameters. The value of parameters is thus an important part in the choice of the right heuristic. However, the search space for the value of parameters is too large to search through in in a brute-force way. Due For this, they used the iterative process of iteratively refining the search space based on the feedbacks from the previous runs of the concolic execution.

Improving on the idea of the parametrized search of heuristic [3], in 2019, Cha, Sooyoung and Oh et al. proposed an idea [5] of adapting the search heuristics on fly by using the accumulated knowledge from the previous runs of concolic executions. The presented algorithm can learn and switch between search heuristics automatically during concolic execution. They define the space of possible search heuristics using the parametric search method [5] proposed earlier. The algorithm continuously switches the search heuristic from the space during the concolic execution. To do so, it uses the accumulated knowledge to learn the probability distribution of the effective and ineffective search heuristics. The algorithm then samples a new set of search heuristics from the distributions. This process continuously iteratively until we run out of the predefined testing time.

In the 2022 "Neural Computing and Applications" journal [12], Jeon, Seungho and Moon, et al. proposed an idea [12, Dr. PathFinder] of using reinforcement learning to prioritize the path in concolic execution. The idea is to combine reinforcement learning with a hybrid fuzzing [[14], [10]] solution for prioritizing the path. The aim is to search the deep paths first and to prune the shallow paths. They formally defined a learning algorithm for deep reinforcement

learning agent to guide concolic execution towards searching a deeper path first. They named this idea Dr. PathFinder. During concolic execution, when the reinforcement learning agent comes across a branch, the agent evaluates the state and decides which direction to continue the search. During this process, paths that are considered shallow are eliminated, and those that are deemed deep are prioritized in the search. This minimizes redundant exploration, enabling more efficient memory utilization and mitigating the issue of path-explosion. They experimented with the CB-multios dataset to find the effectiveness of Dr. PathFinder for deep bug cases. They found that this method discovered approximately two times more bugs than the Driller-AFL and approximately five times more bugs than AFL. Dr. PathFinder not only detected more bugs but also produced 19 times less test cases and consumed at least 2% less memory compared to Driller-AFL.

Dr. PathFinder assumes that deep paths are more interesting than shallow paths in finding bugs. This heuristic has shown its effectiveness during experiments. However, due to its bias in favor of the deep paths, this method struggles to find bugs located in the shallow paths.

### 3.4 Interpolation based approach

In 2013, Jaffar, Joxan and Murali, et al. suggested a new method [11] for concolic testing that uses interpolation to prevent path-explosion. Using the proposed method the paths that are not guaranteed to hit a bug are subsumed which helps to deal with the path-explosion problem. The idea works with a concept of annotation which is bascially the information of the form "if C then bug" i.e if the condtion C evaluates to true along the path then the path has a bug. When an unsatisfiable path-condition is fed to the solver, the interpolant is generated at each program point in the path. Interpolant is bascially a formula that describes why the path is infeasible i.e why it doesn't imply a bug. This means, if in the future the interpolant is implied again at the program point through a different path, that path can be subsumed as the path is guaranteed to not find a bug. This helps us to negate the branches that would be spawned by this new path and so on, thereby giving the exponential improvement.

However, there are challenges [11] in applying the idea of interpolation directly in concolic execution. This idea depends on the given annotations which can only be built after the full exploration of the tree. Before that we can only have half interpolants. Since, much of the exploration or path selection is driven by heuristics, we don't get the control of the search order. Due to these half interpolants the method can suffer with soundness. To address this issue, it is important to track the subtrees that have been explored completely. The node corresponding to the completely explored subtree will have a full-interpolant whereas the ones without complete exploration will have the half-interpolants. Due to this, only the nodes with full-interpolants can perform subsumption if the method has to be sound. This limits the number of subsumption that can be performed. To tackle this issue, Jaffar, Joxan and Murali, et al. [11] proposed a greedy technique called greedy confirmation that perfoms some limited path

exploration itself i.e execution of few extra paths without affecting the search order of the concolic execution. This extra limited exploration is guided by the subsumption. It helps to produce full-interpolants in the nodes having only half-interpolants, thereby improving the overall subsumption rate. Jaffar, Joxan and Murali, et al. [11] implemented this method and compared it with CREST [1] and found that there was a significant improvement, as a large part of the paths executed by the heuristics in CREST were simply pruned by this method as they were redundant.

### 3.5 Template-guided approach

In the theme of learning-algorithm based on the feedback from the past runs of concolic execution, Cha, Sooyoung and Lee et al. proposed an idea of using template-guided method in their 2018 paper titled "Template-Guided Concolic Testing via Online Learning" [4]. It is an idea that is orthogonal to the exsiting ideas such as search heuristics, so it can be conbined with them to produce a better result. In this method, a template is basically a partially symbolized input-vector. It's job is to reduce the search space. Using a set of templates in concolic execution, this method exploits common input patterns which effectively improves the coverage. These templates are generated automatically using the online learning with the feedbacks from the past execution runs. When the algorithm runs, it collects the data which will be used to automatically learn useful template. This way the algorithm adaptively reduces the search space for concolic execution. The key idea in this method is to guide the concolic execution with templates, which by selectively generating the symbolic variables, restricts the input-space. This approach differs from traditional concolic execution, which monitors all input values symbolically. Instead, it identifies a subset of input values to treat as symbolic and assign specific concrete inputs to the remaining values, effectively shrinking the initial search space. However, the task of determining which inputs to track symbolically and how to substitute the rest with suitable values is a challenge. To tackle this challenge, they proposed an algorithm that performs concolic execution, creates templates automatically, and enhances them. The algorithm is based on the two main ideas. First, generating the candidate templates from a set of effective test-cases using the idea of sequential pattern mining [7]. The test-cases are responsible for improving the code coverage and they are collected while performing the conventional concolic execution. Second, the use of algorithm to learn the effective templates from the set of candidates during concolic execution. The proposed method runs with a template and iteratively ranks the effectiveness of the template.

Cha, Sooyoung and Lee et al. implemented this approach [4] in CREST [1]. They tested the implementation on open-source C programs of medium size upto 165K lines of code. They compared the results with traditional concolic executions and found that this method significantly outperforms traditional concolic execution methods in term of bug-finding and branch coverage. They mentioned that on performing this method in vim-5.7 for 70 hours, this method exclusively covered 883 branches which the tranditional methods failed to reach. They also

mentioned that they used this method to find real bugs in several open-source C programs such as sed-4.4, gawk-4.21, and grep-3.1.

### 3.6 Optimization of symbolic execution

In a more practical note, path-explotion can also be attacked from a different angle by improving the performance of symbolic execution. In an attempt to make the concolic execution more practical in security testing, in the 27th USENIX Security Symposium, Insu Yun, et al. presented their design of a fast concolic execution engine called QSYM [19]. The design supports hybrid fuzzing with a faster concolic execution engine. They improved the speed of the concolic engine by tightly integrating the sysmbolic emulation with the native execution using dynamic binary translation. This makes it possible to have more fine-grained implementation thus producuing faster instruction-level symbolic emulation. Furthermore, they relaxed the strict requirement of soundness from the concolic execution engine to make it more performant, yet taking the advantage of a faster fuzzer for validation. This provided them a lot of opportunities to optimize the performance. For example, optimistically solving constraints and pruning uninteresting basic blocks.

QSYM [19] works in two main steps. First, it performs fast concolic execution through efficient emulation. Symbol generation emulation was found to be a major bottleneck in the performance of the concolic execution. They solved that with instruction-level selective symbolic execution, advanced constraints optimization techniques, and tied symbolic and concolic executions. This way by reducing the emulation usuage and by optimizing the emulation speed the engine performs faster.

Second, QSYM [19] works with a new heuristic for hybrid fuzzing. Insu Yun, et al. proposed a heuristic tailored for hybrid fuzzing to solve the solve unsatisfiable paths optimistically. This heuristic also prunes out resource-intensive back blocks.

Due to these optimizations, QSYM makes reexecution based repetitive testing practical. As the concolic execution engine is more performant, the concreate execution of external environments is also feasible. Due to this, QSYM helps to deal with incomplete environment models with incorrect symbolic execution. Because of these optimizations and the possible of repetitive testing, QSYM is also free from napshots incurring significant performance degradation.

Insu Yun, et al. compared QSYM [19] to the existing state-of-the-art fuzzers and found that it outperforms them. They noted that QSYM found 14x more bugs than VUzzer in the LAVA-M dataset. QSYM also outperformed Driller in 104 binaries out of 126 they tested. They also noted that QSYM managed to find 13 previously unknown security bugs in eight real-world programs which included Dropbox Lepton, OpenJPEG and ffmpeg. Considering those programs having been thoroughly tested by other state-of-the-art fuzzers such as AFL, OSS-Fzz over time, it gives a good indication of the effectiveness of QSYM.

## 4 Dicsussion

see ralated work section of https://dl.acm.org/doi/pdf/10.1145/2635868.2635872 i.e A Context-Guided Search Strategy inConcolic Testing paper

also of https://dl.acm.org/doi/pdf/10.1145/3180155.3180166 i.e 2018: Automatically Generating Search Heuristics for Concolic Testing paper

## 5 Conclusion

## References

1. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. pp. 443–446 (2008)
2. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
3. Cha, S., Hong, S., Lee, J., Oh, H.: Automatically generating search heuristics for concolic testing. In: Proceedings of the 40th International Conference on Software Engineering. pp. 1244–1254 (2018)
4. Cha, S., Lee, S., Oh, H.: Template-guided concolic testing via online learning. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 408–418 (2018)
5. Cha, S., Oh, H.: Concolic testing with adaptively changing search heuristics. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 235–245 (2019)
6. Chakrabarti, A., Godefroid, P.: Software partitioning for effective automated unit testing. In: Proceedings of the 6th ACM & IEEE International conference on Embedded software. pp. 262–271 (2006)
7. Fumarola, F., Lanotte, P.F., Ceci, M., Malerba, D.: Clofast: closed sequential pattern mining using sparse and vertical id-lists. Knowledge and Information Systems 48, 429–463 (2016)
8. Godefroid, P.: Compositional dynamic test generation. In: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 47–54 (2007)
9. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. pp. 213–223 (2005)
10. Godefroid, P., Levin, M.Y., Molnar, D.: Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft. Queue 10(1), 20–27 (2012)
11. Jaffar, J., Murali, V., Navas, J.A.: Boosting concolic testing via interpolation. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 48–58 (2013)
12. Jeon, S., Moon, J.: Dr. pathfinder: hybrid fuzzing with deep reinforcement concolic execution toward deeper path-first search. Neural Computing and Applications 34(13), 10731–10750 (2022)
13. Majumdar, R., Sen, K.: Hybrid concolic testing. In: 29th International Conference on Software Engineering (ICSE'07). pp. 416–426. IEEE (2007)

14. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of unix utilities. Communications of the ACM 33(12), 32–44 (1990)
15. Seo, H., Kim, S.: How we get there: A context-guided search strategy in concolic testing. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 413–424 (2014)
16. Tang, C., Kooburat, T., Venkatachalam, P., Chander, A., Wen, Z., Narayanan, A., Dowell, P., Karl, R.: Holistic configuration management at facebook. In: Proceedings of the 25th symposium on operating systems principles. pp. 328–343 (2015)
17. Wang, X., Sun, J., Chen, Z., Zhang, P., Wang, J., Lin, Y.: Towards optimal concolic testing. In: Proceedings of the 40th International Conference on Software Engineering. pp. 291–302 (2018)
18. Xie, T., Tillmann, N., De Halleux, J., Schulte, W.: Fitness-guided path exploration in dynamic symbolic execution. In: 2009 IEEE/IFIP International Conference on Dependable Systems & Networks. pp. 359–368. IEEE (2009)
19. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In: Proceedings of the 27th USENIX Security Symposium (Security). Baltimore, MD (Aug 2018)