Advancements In Concolic Execution To Attack Path Explosion Problem

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 bring in a big testing challenge. In the presence of many possible variants and configuration options, it is very important to automate the testing as much as possible. Concolic execution is the 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 that make 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. The complex relationship between man and machine, and the variety of applications have 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 software with a 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 flexibility allows users to tweak the system in myriad ways to produce a variant 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 [35, Holistic configuration management at Facebook], there have been many examples of major outages and security branches with its origin at the misconfiguration of a configurable software system and buggy interactions between options [[19], [22]]. 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, its 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 software needs to be verified. This paper primarily focuses on the verification aspect which deals with the correct execution of software under different input conditions going through possible execution paths in a program.

2.1 Random testing

One way to test 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 program. With random inputs, it is very unlikely 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 practically zero. This makes it impractical to cover various states of the program. However, in absence of other alternatives, this method is still used. Especially, a close cousin of random testing called fuzzing [[15], [27]] has proven its usefulness. Fuzzing is a method of inputting gibberish 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 inputting random valid values but invalid gibberish to the program to see if the system handles that gracefully.

2.2 Symbolic execution

An alternative to random testing is symbolic execution [[25], [1], [6]]. It is a white-box method of testing where we can see the instructions of the program, and because of this, 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 in terms of those symbols. We can use a constraint solver to find concrete values for the symbols satisfying the constraints. Compared to random testing, this allows us to cover the parts of the 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 the 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.

2.3 Concolic execution

Directed automated random testing [16, DART], more commonly known as concolic execution [31], is a technique that combines symbolic execution along with concrete inputs to mitigate the challenges faced by symbolic execution. When the branch is reached, a constraint solver can be used to find a concrete value 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 concrete value can be used to simplify the process. This way concolic execution doesn't get stuck in the middle of execution.

Advantages: Concolic execution has two important advantages. First, it allows us to alleviate the limitations of constraint solvers which limit the effectiveness of symbolic execution. To elaborate on it, when the symbolic condition in a branch is too difficult for the constraint solver to handle, the symbolic execution would have been completely stuck thus stopping the analysis. This means the potential bugs down the execution paths would have been missed. However, with a concolic execution, since it simplifies the constraints with concrete values, the analysis can continue further with both concrete and symbolic values thus resulting in the potential discovery of those bugs. Second advantage of the concolic execution is that the execution that leads to the error during it is trivially sound.

Historical developments: Since the early work from Godefroid et al. [16], there have been various implementations and expansions on the idea of concolic execution. In the 2005 paper titled "CUTE: A concolic unit testing engine for C" [33], Koushik Sen, et al. extended the idea of concolic execution to data structures. There has been a parallel work from Cadar, Cristian, and Ganesh, et al. following similar ideas which independently developed tools such as EXE [5]. EXE further improved and evolved into a tool called KLEE [4]. Koushik Sen, et al. further generalized the idea of concolic execution in a multithreaded setting called jCUTE [32] to test multithreaded Java programs. Concolic execution was later combined with fuzz testing [18] in Microsoft Research [17, SAGE] to test their applications for security vulnerabilities where it produced a significant impact.

Industrial impact: Concolic execution has been widely used in practice to find many security bugs [[18], [23], [24], [28]]. SAGE [17] tool from Microsoft is a prime example of it. SAGE is a whitebox fuzzing tool. It fuzzes the inputs, notably files, under the guidance 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 [17] 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 machines/cores in the Microsoft security testing labs. It has processed over billions of constraints. Other open-source tools such as KLEE [4] have also been widely used in the industry.

Limitation: A major issue in 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. Just visiting the top 30 branches in the execution tree using a breadth-first search (BFS) already requires billions (2^{30}) of concolic execution runs, whereas real-world programs typically have much more branches than that. This is called a path-explosion problem.

3 Approaches to tackle limitation of concolic execution

As it is not possible to exhaustively test all the paths in the program, many approaches have been proposed [29] to attack the path-explosion 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 categorically presented various approaches.

3.1 Analysis on compositional parts

Some earliest ideas to deal with path-explosion were based on performing analysis compositionally.

Automatic program partitioning: Chakrabarti et al. (2006) [10] 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 the 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 implementing 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 alarms caused by unrealistic inputs being injected at interfaces between the units.

Reusing summaries from compositional executions: In 2007, Godefroid et al. [14] further proposed a complementary idea [14] to Chakrabarti et al. [10] in the context of performing concolic execution compositionally by adapting the 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 [16] without compromising on the level of completeness provided by DART. SMART works by testing functions in isolation and by collecting the testing results as function summaries. Summaries of functions can be calculated by performing successive iterations, one path at a time. Function summaries are expressed using the function inputs as preconditions and outputs as post-conditions. A precondition defines a set of equivalent concrete executions. If all the constraints along a program path are in the given theory of constraints, then all concrete executions corresponding to inputs that satisfy the same precondition are ensured to follow the same program path. To avoid repeating identical function summaries in equivalent but different calling contexts, preconditions are stated as constraints on function inputs rather than program inputs. 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 the 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 a detailed analysis again. Due to this, SMART improves the exponential program execution of DART to linear, making it much more scalable.

This idea of compositional concolic execution is performed in such a way as to preserve the soundness of bugs. Any error path discovered through compositional symbolic execution is guaranteed to be sound because, by design, each execution is grounded in a concrete execution. However, one imprecision in this approach is its incompleteness when it comes to falsification, as it is possible to miss bugs by not exercising certain executable program paths and branches.

3.2 Hybrid of random and concolic execution

Majumdar et al. (2007) [26] 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 neighboring 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 don'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 finds one, the algorithm switches back to the random testing mode from there. This switching back and forth between random testing and concolic execution brings in the benefits of both methods. Random testing inexpensively allows for long program execution to a deep state and concolic execution helps to symbolically search in an exhaustive way for a new path.

Majumdar et al. [26] compared this algorithm with random testing and concolic testing by individually running them on the VIM text editor (150K lines code in C) and their implementation of popular red-black tree data structure. They found that hybrid testing consistently outperforms the two methods in terms of branch coverage. Furthermore, the hybrid method relies on faster and cheaper random testing to explore the program state as much as possible which helps in avoiding expensive constraint solving whenever possible. This avoidance of constraint solving 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 [16] 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 clear benefits over the individual methods, it still suffers from the limitations of the concolic execution. The discovery of a 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 can considerably improve the coverage.

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 bugs. There have been many studies on heuristically selecting the path.

Backtracking: In 2006, Yan, Jun and Zhang, et al. presented [38] backtracking ideas and search heuristics to deal with the path-explosion problem in concolic execution. They discussed the SAT-based approach and a backtracking algorithm to solve the problem. They presented a technique called SCEH that aims to improve the efficiency of existing methods by using several search heuristics and symmetry-breaking techniques in a backtracking search algorithm. All of their techniques were implemented in a tool called EXACT which was introduced by them in this paper showing the improvement over earlier methods.

Heuristic searches: In the 2008 paper [3] 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 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 of selecting samples from 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 language. 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.

Fitness-guided approach: Xie, Tao and Tillmann et al. proposed a fitness-guided approach for path exploration (2009) [37]. 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. The 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 fitness would fail. A fitness value is calculated for the paths that have already been explored. Using the fitness values the paths are prioritized. Fitnex prioritizes its search by minimizing fitness values that represent the closeness of a path to covering a test target. During the path exploration, a branch is selected based on this fitness gain.

Context-guided approach: In 2014, Seo, Hyunmin and Kim, et al. proposed an idea [34] 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 the 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 visited branch is considered if they should be selected for the next input or should 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, a 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.

Markov Decision Process with costs: Wang, Xinyu and Sun, et al. (2018) proposed an idea [36] 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 the Markov Decision Process (MDP) with costs. As a result, the existing 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.

Automatically choosing a heuristic: So far, many individual heuristics were proposed but the selection of them was mainly manual. In the 2018 paper titled "Automatically Generating Search Heuristics for Concolic Testing" [7] Cha, Sooyoung and Hong, et al. introduced a way to select a suitable heuristics based on the parameters automatically. They also proposed an optimization algorithm to find good values for parameters in an efficient way. They created different classes of search heuristics and defined them 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 of the choice of the right heuristic. However, the search space for the value of parameters is too large to search through in a brute-force way. For this, they used the iterative process of iteratively refining the search space based on the feedback from the previous runs of the concolic execution.

Adapting search heuristics on the fly: Improving on the idea of the parametrized search of heuristic [7], in 2019, Cha, Sooyoung and Oh et al. proposed an idea [9] of adapting the search heuristics on the 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 [9] 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 effective and ineffective search heuristics. The algorithm then samples a new set of search heuristics from the distributions. This process continues iteratively until we run out of the predefined testing time.

Prioritizing deep paths using reinforcement learning: In the 2022 "Neural Computing and Applications" journal [21], Jeon, Seungho and Moon, et al. proposed an idea [21, Dr. PathFinder] of using reinforcement learning to prioritize the path in concolic execution. The idea is to combine reinforcement learning with a hybrid fuzzing [[27], [17]] 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 the 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 fewer 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 [20] 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 basically the information of the form "if C then bug" i.e if the condition 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 basically 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 exponential improvement.

However, there are challenges [20] 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 control of the search order. Due to these half-interpolants, the method can struggle to be sound. 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. [20] proposed a greedy technique called greedy confirmation that performs some limited path exploration itself i.e execution of a 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. [20] implemented this method and compared it with CREST [3] 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 past runs of concolic execution, Cha, Sooyoung and Lee et al. proposed the idea of using the template-guided method in their 2018 paper titled "Template-Guided Concolic Testing via Online Learning" [8]. It is an idea that is orthogonal to the existing ideas such as search heuristics, so it can be combined with them to produce a better result. In this method, a template is basically a partially symbolized input-vector. Its 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 online learning with feedback from past execution runs. When the algorithm runs, it collects the data which will be used to automatically learn useful templates. 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 two main ideas. First, generating the candidate templates from a set of effective test cases using the idea of sequential pattern mining [13]. The test cases are responsible for improving the code coverage and they are collected while performing the conventional concolic execution. Second, the use of the 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 [8] in CREST [3]. They tested the implementation on open-source C programs of medium size up to 165K lines of code. They compared the results with traditional concolic executions and found that this method significantly outperforms traditional concolic execution methods in terms of bug-finding and branch coverage. They mentioned that by performing this method in vim-5.7 for 70 hours, this method exclusively covered 883 branches which the traditional 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

On a more practical note, path exploration 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, at the 27th USENIX Security Symposium, Insu Yun, et al. presented their design of a fast concolic execution engine called QSYM [39]. The design supports hybrid fuzzing with a faster concolic execution engine. They improved the speed of the concolic engine by tightly integrating the symbolic emulation with the native execution using dynamic binary translation. This makes it possible to have more fine-grained implementation thus producing 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 with a lot of opportunities to optimize their performance. For example, optimistically solving constraints and pruning uninteresting basic blocks.

QSYM [39] 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 usage and by optimizing the emulation speed the engine performs faster.

Second, QSYM [39] works with a new heuristic for hybrid fuzzing. Insu Yun, et al. proposed a heuristic tailored for hybrid fuzzing to solve unsatisfiable paths optimistically. This heuristic also prune 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 concrete 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 snapshots incurring significant performance degradation.

Insu Yun, et al. compared QSYM [39] 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 Discussion

Testing software using concolic execution augments the existing testing practices like unit-testing, integration testing, etc. in software development. Compared to unit-testing, concolic execution doesn't require any manual implementation from a developer. It is an automatic process. However, it is not a replacement over unit-testing. Unit-testing means testing a unit of a software component separately and asserting that it works correctly. The verified units of software can then be integrated together to create a software system. To test the correctness of the integration, integration testing can be performed. Such integrated components can as a whole be tested using end-to-end tests to check the correctness of software considering the business functionality it should provide. Using these methods, in theory, the software can be tested with 100% coverage, covering all kinds of edge cases. However, in practice, such testing is often not implemented due to business deadline or are added later. They often miss the edge cases and don't cover all the execution paths. Also, being a manual process these testing are time-consuming. They require creating mock objects and special environments which add additional time and effort. Since they are manually implemented the tests themselves can have bugs in them. Writing a test for a test would be a vicious infinite cycle. Furthermore, in practice, the push toward 100% coverage often creates brittle tests. When all the aspects of the software are asserted by tests, it makes it difficult to quickly change the codebase as changing the codebase often leads to many tests being failed. This limits the chances of design changes and rapid innovation. Over time, it just hinders the developers. They will be incentivized to write namesake tests for the sake of coverage and will be scared to perform the needed design changes in the codebase. Even with all this, the big coverage doesn't guarantee the absence of bugs.

Due to this, concolic execution can be an important method [[30], [12], [11]] of verifying the software in addition to normal manual tests. Since it is an automatic process it can continuously run in the background, continuously testing for bugs and crashes. Microsoft is a clear example of this. They have long benefitted [[18], [17], [2]] from integrating concolic execution in their software development cycle.

5 Conclusion

In this paper, I have presented concolic execution in the context of verifying complex configurable software systems. I explained how concolic execution works compared to other methods, explaining limitations that gave birth to it. I mentioned the historical context and advantages of concolic execution followed by an explanation of the major limitations faced by it. I categorically presented the works that have tried to mitigate the limitation (path-explosion) of concolic execution from various angles. Finally, I have discussed the role of concolic execution in the context of the existing manual testing practices prevalent in the industry.

References

1. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) 51(3), 1–39 (2018)

2. Bounimova, E., Godefroid, P., Molnar, D.: Billions and billions of constraints: Whitebox fuzz testing in production. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 122–131. IEEE (2013)
3. 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)
4. 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)
5. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: Automatically generating inputs of death. ACM Transactions on Information and System Security (TISSEC) 12(2), 1–38 (2008)
6. Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 1066–1071 (2011)
7. 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)
8. 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)
9. 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)
10. 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)
11. Christakis, M., Müller, P., Wüstholz, V.: Guiding dynamic symbolic execution toward unverified program executions. In: Proceedings of the 38th International Conference on Software Engineering. pp. 144–155 (2016)
12. Csallner, C., Tillmann, N., Smaragdakis, Y.: Dysy: Dynamic symbolic execution for invariant inference. In: Proceedings of the 30th international conference on Software engineering. pp. 281–290 (2008)
13. 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)
14. 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)
15. Godefroid, P.: Fuzzing: Hack, art, and science. Communications of the ACM 63(2), 70–76 (2020)
16. 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)
17. 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)
18. Godefroid, P., Levin, M.Y., Molnar, D.A., et al.: Automated whitebox fuzz testing. In: NDSS. vol. 8, pp. 151–166 (2008)

19. Han, X., Yu, T.: An empirical study on performance bugs for highly configurable software systems. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 1–10 (2016)
20. 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)
21. 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)
22. Kenner, A., May, R., Krüger, J., Saake, G., Leich, T.: Safety, security, and configurable software systems: a systematic mapping study. In: Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A. pp. 148–159 (2021)
23. Kim, K., Jeong, D.R., Kim, C.H., Jang, Y., Shin, I., Lee, B.: Hfl: Hybrid fuzzing on the linux kernel. In: NDSS (2020)
24. Kim, K., Kim, T., Warraich, E., Lee, B., Butler, K.R., Bianchi, A., Tian, D.J.: Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 2212–2229. IEEE (2022)
25. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)
26. Majumdar, R., Sen, K.: Hybrid concolic testing. In: 29th International Conference on Software Engineering (ICSE'07). pp. 416–426. IEEE (2007)
27. 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)
28. Pham, V.T., Böhme, M., Roychoudhury, A.: Model-based whitebox fuzzing for program binaries. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 543–553 (2016)
29. Sabbaghi, A., Keyvanpour, M.R.: A systematic review of search strategies in dynamic symbolic execution. Computer Standards & Interfaces 72, 103444 (2020)
30. Sabbaghi, A., Rashidy Kanan, H., Keyvanpour, M.R.: Fsct: A new fuzzy search strategy in concolic testing. Information and Software Technology 107, 137–158 (2019), `https://www.sciencedirect.com/science/article/pii/S0950584918302428`
31. Sen, K.: Concolic testing. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. pp. 571–572 (2007)
32. Sen, K., Agha, G.: Cute and jcute: Concolic unit testing and explicit path model-checking tools: (tool paper). In: Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18. pp. 419–423. Springer (2006)
33. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for c. ACM SIGSOFT Software Engineering Notes 30(5), 263–272 (2005)
34. 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)
35. 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)
36. 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)

37. 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)
38. Yan, J., Zhang, J.: Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. In: 30th Annual International Computer Software and Applications Conference (COMPSAC'06). vol. 1, pp. 385–394. IEEE (2006)
39. 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)