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. There have been many studies to deal with this path-explosion 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. It is important to verify the correctness of software to use it with confidence but this complexity, pace of change and the size of software makes it challenging to do so.

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 [9] 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 exection. 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 exection insufficient in many practical uses.

Directed automated random testing [5, 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 [6] tool from microsoft is a prime example of it. SAGE is a whitebox 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 says [6] 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 having processing more than 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) [3] 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. [4] further proposed a complementary idea [4] to Chakrabarti et al. [3] 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 [5] 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) [8] proposed an algorithm that combines random testing and concolic exection. 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 exection 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 exection from that program state. Using concolic exection 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. [8] 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 [5] 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 exection. 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 exection.

Xie, Tao and Tillmann et al. proposed a fitness-guided approach for path exploration (2009) [10]. 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.

- 2014: context guided - 2018: automatic heuristics - 2018: towards optimal - 2019: adapting changing heuristics

3.4 Interpolation based approach

In 2013, Jaffar, Joxan and Murali, et al. suggested a new method [7] 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 [7] in applying the idea of interpolation directly in concolic exection. 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. [7] 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 exection. 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. [7] 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

- 2018: template guided

3.6 Optimizatin of a constraint solver

- 2018: QSYM

3.7 Using reinforcement learning to prioritize the path

- 2022: korean paper

4 papers

4.1 2014: A Context-Guided Search Strategy inConcolic Testing

Gist: While moststrategies focus on coverage information in the branch selection process, we introduce CGS which considers contextinformation, that is, how the execution reaches the branch.Our evaluation results show that CGS outperforms otherstrategies.

Methodlogy: CGS explores branches in the current execution tree. Foreach visited branch, CGSexaminesthe branch and decideswhether toselectthe branch for the next input orskipit. CGS looks athow the execution reaches the current branch by calculat-ingk-contextof the branch from its preceding branches anddominator information. Then, thek-contextis comparedwith the context of previously selected branches which isstored in thecontext cache. If thek-contextis new, thebranch is selected for the next input. Otherwise, CGS skipsthe branch.

Configurability part: We evaluate CGS on top of two publicly available concolictesting tools, CREST [13] and CarFastTool [29]

4.2 2018: Automatically Generating Search Heuristics for Concolic Testing

Gist: developed a parame-terized search heuristic for concolic testing with an optimizationalgorithm to efficiently search for good parameter values. We hopethat our technique can supplant the laborious and less rewardingtask of manually tuning search heuristics of concolic testing.

Methodlogy: this paper presents a new approachthat automatically generates search heuristics for concolic testing.To this end, we use two key ideas. First, we define aparameterizedsearch heuristic, which creates a large class of search heuristics.The parameterized heuristic reduces the problem of designing agood search heuristic into a problem of finding a good parametervalue. Second, we present a search algorithm specialized to concolictesting. The search space that the parameterized heuristic poses isintractably large. Our algorithm effectively guides the search byiteratively refining the search space based on the feedback fromprevious runs of concolic testing

Configurability part: We have implemented our techniquein CREST [3] and evaluated it on 10 C programs (0.5–150KLoC)

4.3 2018: Template-Guided Concolic Testing via Online Learning

Gist: a template is a partially symbolized inputvector whose job is to reduce the search space. However, choos-ing a right set of templates is nontrivial and significantly affectsthe final performance of our approach. We present an algorithmthat automatically learns useful templates online, based on datacollected from previous runs of concolic testing. The experimen-tal results with open-source programs show that our techniqueachieves greater branch coverage and finds bugs more effectivelythan conventional concolic testing

In our approach, concolictesting uses a set of templates to exploit common input patterns that improve coverage effectively, where the templates are automatically generated through online learning algorithm based on thefeedback from past runs of concolic testing.

Methodlogy: we present template-guided concolic testing, a newtechnique for adaptively reducing the search space of concolic test-ing. The key idea is to guide concolic testing with templates, whichrestrict the input space by selectively generating symbolic variables.Unlike conventional concolic testing that tracks all input valuessymbolically, our technique treats a set of selected input valuesas symbolic and fixes unselected inputs with particular concreteinputs, thereby reducing the original search space. A challenge,however, is choosing input values to track symbolically and replac-ing the remaining inputs with appropriate values. To address thischallenge, we develop an algorithm that performs concolic testingwhile automatically generating, using, and refining templates. Thealgorithm is based on two key ideas. First, by using the sequentialpattern mining [9], we generate the candidate templates from a setof effective test-cases, where the test-cases contribute to improvingcode coverage and are collected while conventional concolic test-ing is performed. Second, we use an algorithm that learns effectivetemplates from the candidates during concolic testing. Our algo-rithm iteratively ranks the candidates based on the effectivenessof templates that were evaluated in the previous runs. Our tech-nique is orthogonal to the existing techniques and can be fruitfullycombined with them, in particular with the state-of-the-art searchheuristics

Configurability part: Experimental results show that our approach outperforms con-ventional concolic testing in term of branch coverage and bug-finding. We have implemented our approach in CREST [7] andcompared our technique with conventional concolic testing foropen-source C programs of medium size (up to 165K LOC). For allbenchmarks, our technique achieves significantly higher branchcoverage compared to conventional concolic testing. For example,for vim-5.7, we have performed both techniques for 70 hours, whereour technique exclusively covered 883 branches that conventionalconcolic testing failed to reach. Our technique also succeeded infinding real bugs that can be triggered in the latest versions of threeopen-source C programs: sed-4.4, grep-3.1 and gawk-4.21.

4.4 2018: Towards Optimal Concolic Testing

Gist: show the optimal strategy can be defined based onthe probability of program paths and the cost of constraintsolving. The problem of identifying the optimal strategy isthen reduced to a model checking problem of Markov DecisionProcesses with Costs. Secondly, in view of the complexity inidentifying the optimal strategy, we design a greedy algorithmfor approximating the optimal strategy.

Methodlogy: aim to develop a framework which allowsus to define and compute the optimal concolic testing strate-gy. That is, we aim to systematically answer when to applyconcrete execution, when to apply symbolic execution andwhich program path to apply symbolic execution to. In par-ticular, we make the following technical contributions. Firstly,we show that the optimal concolic testing strategy can bedefined based on a probabilistic abstraction of program behaviors. Secondly, we show that the problem of identifyingthe optimal strategy can be reduced to a model checkingproblem of Markov Decision Processes with Costs. As a re-sult, we can reuse existing tools and algorithms to solve theproblem. Thirdly, we evaluate existing heuristics empiricallyusing a set of simulated experiments and show that theyhave much room to improve. Fourthly, in view of the highcomplexity in computing the optimal strategy, we propose agreedy algorithm which approximates the optimal one. Weempirically evaluate the greedy algorithm based on both sim-ulated experiments and experiments with C programs, andshow that it gains better performance than existing heuristicsin KLEE

4.5 2018: Qsym : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing

Gist: we design a fast concolicexecution engine, calledQSYM, to support hybrid fuzzing.The key idea is to tightly integrate the symbolic emulationwith the native execution using dynamic binary transla-tion, making it possible to implement more fine-grained,so faster, instruction-level symbolic emulation. Additionally,QSYMloosens the strict soundness requirements ofconventional concolic executors for better performance,yet takes advantage of a faster fuzzer for validation, pro-viding unprecedented opportunities for performance op-timizations, e.g., optimistically solving constraints andpruning uninteresting basic blocks

Methodlogy: •Fast concolic execution through efficient emula-tion: We improved the performance of concolicexecution by optimizing emulation speed and reduc-ing emulation usage. Our analysis identified thatsymbol generation emulation was the major perfor-mance bottleneck of concolic execution such that weresolved it with instruction-level selective symbolicexecution, advanced constraints optimization tech-niques, and tied symbolic and concolic executions.

•Efficient repetitive testing and concrete environ-ment.The efficiency ofQSYM-makes re-execution-based repetitive testing and the concrete executionof external environments practical. Because of this,QSYMis free from snapshots incurring significantperformance degradation and incomplete environ-ment models resulting in incorrect symbolic execu-tion due to its non-reusable nature.

•New heuristics for hybrid fuzzing.We proposednew heuristics tailored for hybrid fuzzing to solveunsatisfiable paths optimistically and to prune outcompute-intensive back blocks, thereby makingQSYMproceed.

Configurability part: Our evaluation shows thatQSYMdoes not just outperform state-of-the-art fuzzers (i.e., found 14×morebugs than VUzzer in the LAVA-M dataset, and outper-formed Driller in104binaries out of126), but also found13 previously unknown security bugsineightreal-worldprograms like Dropbox Lepton, ffmpeg, and OpenJPEG,which have already been intensively tested by the state-of-the-art fuzzers, AFL and OSS-Fuzz.

4.6 2019: Concolic testing with adaptively changing search heuristics

Gist: adapting search heuristics on the fly via an algorithm that learns new search heuristics based on the knowledge accumulated during concolic testing

Methodlogy: we present an algorithm that automaticallylearns and switches search heuristics during concolic testing. Thealgorithm maintains a set of search heuristics and continuouslychanges them during the testing process. To do so, we first definethe space of possible search heuristics using the idea of parametricsearch heuristic recently proposed in prior work [5]. A technicalchallenge is how to adaptively switch search heuristics in the pre-defined space. We address this challenge with a new concolic testingalgorithm that (1) accumulates the knowledge about the previouslyevaluated search heuristics, (2) learns the probabilistic distributionsof the effective and ineffective search heuristics from the accumu-lated knowledge, and (3) samples a new set of search heuristics from the distributions. The algorithm iteratively performs thesethree steps until it exhausts a given time budget.

4.7 2022: Dr.PathFinder: hybrid fuzzing with deep reinforcement concolic execution toward deeper path-first search

Gist: propose a concolic execution algorithm that combines deep reinforcement learning with a hybrid fuzzing solution, Dr.PathFinder. When the reinforcement learning agent encounters a branch during concolic execution, it evaluates the state and determines the search path. In this process,“shallow” paths are pruned, and “deep” paths are searched first. This reduces unnecessary exploration, allowing the efficient memory usage and alleviating the state explosion problem.

Methodlogy: We formally define a learning algorithm for a deep reinforcement learning agent that allows concolic execution to first search for a deeper path.

We present a deeper path-first search concolic execution algorithm using a reinforcement learning agent and a hybrid fuzzer called Dr.PathFinder.

Result: In experiments with the CB-multios dataset for deep bug cases, Dr.PathFinder discovered approximately five times more bugs than AFL and two times more than Driller-AFL. In addition to finding more bugs, Dr.PathFinder generated 19 times fewer test cases and used at least 2bugs located in deep paths, Dr.PathFinder had limitation to find bugs located at shallow paths, which we discussed.

5 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

6 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. 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)
4. 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)
5. 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)
6. 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)
7. 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)
8. Majumdar, R., Sen, K.: Hybrid concolic testing. In: 29th International Conference on Software Engineering (ICSE'07). pp. 416–426. IEEE (2007)
9. 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)
10. 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)