

# Concolic execution

Seminar: Understanding of configurable software systems

Bipin Oli

Advisors: Prof. Sven Apel,  
Christian Hechtel

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 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. Specially, a close cousin of random testing called fuzzing [6] 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.

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 in terms 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 [4, DART], more commonly known as concolic execution, 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.

Concolic execution has been widely used in practice to find many security bugs. SAGE [5] 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 says [5] 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 [1] 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 called 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) [2] 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 alarms caused by unrealistic inputs being injected at interfaces between the units.

In 2007, Godefroid et al. [3] further proposed a complementary idea [3] to Chakrabarti et al. [2] 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 [4] 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 Switching between random and concolic execution

- 2007: hybrid

### 3.3 Heuristic based approaches

- 2008: heuristics - 2009: fitness function - 2014: context guided - 2018: automatic heuristics - 2018: towards optimal - 2019: adapting changing heuristics

### 3.4 Interpolation based approach

- 2013: interpolation

### 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 2007: Hybrid concolic testing (\*\*)

**Gist:** an algorithm that in-terleaves random testing with concolic execution to obtain both a deep and a wide exploration of program state space. Our algorithm generates test inputs automatically by inter-leaving random testing until saturation with bounded ex-haustive symbolic exploration of program points. It thus combines the ability of random search to reach deep program states quickly together with the ability of concolic test-ing to explore states in a neighborhood exhaustively.

**Methodology:** present hybrid concolic testing, a simple algorithm that interleaves the application of random tests with concolic testing to achieve deep and wide exploration of the program state space. From the initial program state, hybrid concolic testing starts by performing random testing to improve coverage. When random testing saturates, that is, does not produce any new coverage points after running some predetermined number of steps, the algorithm automatically switches to concolic execution from the current program state to perform an exhaustive bounded depth search for an uncovered coverage point. As soon as one is found, the algorithm reverts back to concrete mode. The interleaving of random testing and concolic execution thus uses both the capacity of random testing to inexpensively generate deep program states through long program executions and the capability of concolic testing to exhaustively and symbolically search for new paths with a limited look-ahead.

The interleaving of random and symbolic techniques is the crucial insight that distinguishes hybrid concolic testing from a naive approach that simply runs random and concolic tests in parallel on a program. This is because many programs show behaviors where the program must reach a particular state and then follow a precise sequence of input events 'alpha' order to get to a required coverage point. It is often easy to reach using random testing, but not then to generate the precise sequence of events 'alpha'. On the other hand, while it is usually easy for concolic testing to generate 'sigma', concolic testing gets stuck in exploring a huge number of program paths before even reaching the states.

In the end, hybrid concolic testing has the same limitations of symbolic execution based test generation: the discovery of uncovered points depends on the scalability and expressiveness of the constraint solver, and the exhaustive search for uncovered points is limited by the number of paths to be explored. Therefore, in general, hybrid concolic testing may not achieve 100 percent coverage, although it can improve random testing considerably. Further, the algorithm is not a panacea for all software quality issues. While we provide an automatic mechanism for test input generation, all the other effort required in testing, for example, test oracle generation, assertion based verification, and mock environment creation still have to be performed as with any other test input generation algorithm. Further, we look for code coverage, which may or may not be an indicator of code reliability.

**Configurability part:** compare random, concolic, and hybrid concolic testing on the VIM text editor (150K lines of C code) and on an implementation of the red-black tree data structure. Our experiments indicate that for a fixed testing budget, hybrid concolic testing technique outperforms both random and concolic in terms of branch coverage of the state space exhaustively. In contrast, hybrid concolic testing switches to inexpensive random testing as soon as it identifies some uncovered point, relying on fast random testing to explore as much of the state space as possible. In this way, it avoids expensive constraint solving to perform exhaustive search in some part of the state space. Moreover, if random testing does not hit a new coverage point, it can take advantage of the locally

exhaustive search provided by concolic testing to continue from a new coverage point

## 4.2 2008: Heuristics for Scalable Dynamic Test Generation

**Gist:** several such heuristic search strategies, including a novel strategy guided by the control flow graph of the program under test.

**Methodology:** We propose a search strategy that is guided by the static structure of the program under test, namely the control flow graph (CFG). In this strategy, we choose branches to negate for the purpose of test generation based on their distance in the CFG to currently uncovered branches. We experimentally show that this greedy approach to maximizing the branch coverage helps to improve such coverage faster, and to achieve greater final coverage, than the default depth-first search strategy of concolic testing. We further propose two random search strategies. While in traditional random testing a program is run on random inputs, these two strategies test a program along random execution paths. The second attempts to sample uniformly from the space of possible program paths, while the third is a variant we have found to be more effective in practice

have implemented these search strategies in CREST, an open-source prototype test generation tool for C

**Configurability part:** We have implemented these strategies in CREST, our open source concolic testing tool for C, and evaluated them on two widely-used software tools, `grep 2.2` (15K lines of code) and `Vim5.7` (150K lines). On these benchmarks, the presented heuristics achieve significantly greater branch coverage on the same testing budget than concolic testing with a traditional depth-first search strategy.

## 4.3 2009: Fitness-Guided Path Exploration in Dynamic Symbolic Execution

**Gist:** To address the space-explosion issue in path exploration, we propose a novel approach called *Fitnex*, a search strategy that uses state-dependent fitness values (computed through a fitness function) to guide path exploration. The fitness function measures how close an already discovered feasible path is to a particular test target (e.g., covering a not-yet-covered branch). Our new fitness-guided search strategy is integrated with other strategies that are effective for exploration problems where the fitness heuristic fails.

**Methodology:** The core of our approach is the *Fitnex* search strategy guided by fitness values computed with a fitness function (Section 4.1). To deal with program branches not amenable to a fitness function, our approach includes integration of the *Fitnex* strategy with other search strategies (Section 4.2)

A fitness function (Section 4.1.1) gives a measurement on how close an explored path is to achieving a test target (e.g., covering a not-yet-covered branch). We compute a fitness value for each already explored path and prioritize these known paths based on their fitness values (Section 4.1.2). We compute a fitness gain for each branch in the program under test and prioritize branching nodes based on their corresponding branches' fitness gains (Section 4.1.3). During path exploration, we give higher priority to flipping a branching node with a better (higher) fitness gain in a path with a better (lower) fitness value (Section 4.1.4).

#### 4.4 2013: Boosting Concolic Testing via Interpolation

**Gist:** propose a new and complementary method based on interpolation, that greatly mitigates path-explosion by subsuming paths that can be guaranteed to not hit a bug.

**Methodology:** first, assume that the program is annotated with certain bug conditions of the form “if  $C$  then bug”, where if the condition  $C$  evaluates to true along a path, the path is buggy. Then, whenever an unsatisfiable path condition is fed to the solver, an interpolant is generated at each program point along the path. The interpolant at a given program point can be seen as a formula that succinctly captures the reason of infeasibility of paths at the program point. In other words it succinctly captures the reason why paths through the program point are not buggy. As a result, if the program point is encountered again through a different path such that the interpolant is implied, the new path can be subsumed, because it can be guaranteed to not be buggy. The exponential savings are due to the fact that not only is the new path subsumed, but also the paths that this new path would spawn by negating its branches.

Unfortunately, methods such as [12, 14, 11] cannot be used directly for concolic testing due to several challenges. First, the soundness of these methods relies on the assumption that an interpolant at a node has been computed after exploring the entire “tree” of paths that arise from the node. In concolic testing, this assumption is invalid as the tester can impose an arbitrary search order. For example, concolic testers such as Crest [3] and KLEE [4] use often many heuristics that may follow a random walk through the search space, thus making this method unsound. To address this problem, we need to keep track of nodes whose trees have been explored fully (in which case we say the node is annotated with a full-interpolant) or partially (similarly, a half-interpolant). Under this new setting, only nodes with full-interpolants are capable of subsumption in a sound manner. As a result, the amount of subsumption depends on how often nodes get annotated with full-interpolants from the paths explored by the concolic tester. Unfortunately our benchmarks in Section 6 showed that the above method by itself results in very few nodes with full-interpolants, thereby providing poor benefit to the concolic tester, because the tester rarely explores the entire tree of paths arising from a node. Hence, an important challenge now is to “accelerate” the formation of full-interpolants in order to increase subsumption.

For this, we introduce a novel technique called greedy confirmation that performs limited path exploration (i.e., execution of a few extra paths) by itself, guided by subsumption, with an aim to produce a full-interpolant at nodes currently annotated with a half-interpolant. It is worth mentioning that this execution of few paths is done without interfering with the search order of the concolic tester. This technique ultimately resulted in a significant increase in subsumption for our benchmarks, and is vital for the effectiveness of our method. We implemented our method and compared it with a publicly available concolic tester, Crest [3]. We found that for the price of a reasonable overhead to compute interpolants, a large percentage of paths executed by those heuristics can be subsumed thereby increasing their coverage substantially.

**Result:** We attacked the path-explosion problem of concolic testing by pruning redundant paths using interpolation. The challenge for interpolation in concolic testing is the lack of control of search order. To solve this, we presented the concept of half and full interpolants that makes the use of interpolants sound, and greedy confirmation that accelerates the formation of full-interpolants thereby increasing the likelihood of subsuming paths.

#### 4.5 2014: A Context-Guided Search Strategy in Concolic Testing

**Gist:** While most strategies focus on coverage information in the branch selection process, we introduce CGS which considers context information, that is, how the execution reaches the branch. Our evaluation results show that CGS outperforms other strategies.

**Methodology:** CGS explores branches in the current execution tree. For each visited branch, CGS examines the branch and decides whether to select the branch for the next input or skip it. CGS looks at how the execution reaches the current branch by calculating the context of the branch from its preceding branches and dominator information. Then, the context is compared with the context of previously selected branches which is stored in the context cache. If the context is new, the branch is selected for the next input. Otherwise, CGS skips the branch.

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

#### 4.6 2018: Automatically Generating Search Heuristics for Concolic Testing

**Gist:** developed a parameterized search heuristic for concolic testing with an optimization algorithm to efficiently search for good parameter values. We hope that our technique can supplant the laborious and less rewarding task of manually tuning search heuristics of concolic testing.



**Methodology:** this paper presents a new approach that automatically generates search heuristics for concolic testing. To this end, we use two key ideas. First, we define a parameterized search heuristic, which creates a large class of search heuristics. The parameterized heuristic reduces the problem of designing a good search heuristic into a problem of finding a good parameter value. Second, we present a search algorithm specialized to concolic testing. The search space that the parameterized heuristic poses is intractably large. Our algorithm effectively guides the search by iteratively refining the search space based on the feedback from previous runs of concolic testing.

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

#### 4.7 2018: Template-Guided Concolic Testing via Online Learning

**Gist:** a template is a partially symbolized input vector whose job is to reduce the search space. However, choosing a right set of templates is nontrivial and significantly affects the final performance of our approach. We present an algorithm that automatically learns useful templates online, based on data collected from previous runs of concolic testing. The experimental results with open-source programs show that our technique achieves greater branch coverage and finds bugs more effectively than conventional concolic testing.

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

**Methodology:** we present template-guided concolic testing, a new technique for adaptively reducing the search space of concolic testing. The key idea is to guide concolic testing with templates, which restrict the input space by selectively generating symbolic variables. Unlike conventional concolic testing that tracks all input values symbolically, our technique treats a set of selected input values as symbolic and fixes unselected inputs with particular concrete inputs, thereby reducing the original search space. A challenge, however, is choosing input values to track symbolically and replacing the remaining inputs with appropriate values. To address this challenge, we develop an algorithm that performs concolic testing while automatically generating, using, and refining templates. The algorithm is based on two key ideas. First, by using the sequential pattern mining [9], we generate the candidate templates from a set of effective test-cases, where the test-cases contribute to improving code coverage and are collected while conventional concolic testing is performed. Second, we use an algorithm that learns effective templates from the candidates during concolic testing. Our algorithm iteratively ranks the candidates based on the effectiveness of templates that were evaluated in the previous runs. Our technique is orthogonal to the existing techniques and can be fruitfully combined with them, in particular with the state-of-the-art search heuristics.

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

#### 4.8 2018: Towards Optimal Concolic Testing

**Gist:** show the optimal strategy can be defined based on the probability of program paths and the cost of constraint solving. The problem of identifying the optimal strategy is then reduced to a model checking problem of Markov Decision Processes with Costs. Secondly, in view of the complexity in identifying the optimal strategy, we design a greedy algorithm for approximating the optimal strategy.

**Methodology:** aim to develop a framework which allows us to define and compute the optimal concolic testing strategy. That is, we aim to systematically answer when to apply concrete execution, when to apply symbolic execution and which program path to apply symbolic execution to. In particular, we make the following technical contributions. Firstly, we show that the optimal concolic testing strategy can be defined based on a probabilistic abstraction of program behaviors. Secondly, we show that the problem of identifying the optimal strategy can be reduced to a model checking problem of Markov Decision Processes with Costs. As a result, we can reuse existing tools and algorithms to solve the problem. Thirdly, we evaluate existing heuristics empirically using a set of simulated experiments and show that they have much room to improve. Fourthly, in view of the high complexity in computing the optimal strategy, we propose a greedy algorithm which approximates the optimal one. We empirically evaluate the greedy algorithm based on both simulated experiments and experiments with C programs, and show that it gains better performance than existing heuristics in KLEE.

#### 4.9 2018: Qsym : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing

**Gist:** we design a fast concolic execution engine, called QSYM, to support hybrid fuzzing. The key idea is to tightly integrate the symbolic emulation with the native execution using dynamic binary translation, making it possible to implement more fine-grained, so faster, instruction-level symbolic emulation. Additionally, QSYM loosens the strict soundness requirements of conventional concolic ex-

ecutors for better performance, yet takes advantage of a faster fuzzer for validation, providing unprecedented opportunities for performance optimizations, e.g., optimistically solving constraints and pruning uninteresting basic blocks

**Methodology:** •Fast concolic execution through efficient emulation: We improved the performance of concolic execution by optimizing emulation speed and reducing emulation usage. Our analysis identified that symbol generation emulation was the major performance bottleneck of concolic execution such that we resolved it with instruction-level selective symbolic execution, advanced constraints optimization techniques, and tied symbolic and concolic executions.

•Efficient repetitive testing and concrete environment. The efficiency of QSYM makes re-execution-based repetitive testing and the concrete execution of external environments practical. Because of this, QSYM is free from snapshots incurring significant performance degradation and incomplete environment models resulting in incorrect symbolic execution due to its non-reusable nature.

•New heuristics for hybrid fuzzing. We proposed new heuristics tailored for hybrid fuzzing to solve unsatisfiable paths optimistically and to prune out compute-intensive back blocks, thereby making QSYM proceed.

**Configurability part:** Our evaluation shows that QSYM does not just outperform state-of-the-art fuzzers (i.e., found 14× more bugs than VUzzer in the LAVA-M dataset, and outperformed Driller in 104 binaries out of 126), but also found 13 previously unknown security bugs in eight real-world programs like Dropbox Lepton, ffmpeg, and OpenJPEG, which have already been intensively tested by the state-of-the-art fuzzers, AFL and OSS-Fuzz.

#### 4.10 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

**Methodology:** we present an algorithm that automatically learns and switches search heuristics during concolic testing. The algorithm maintains a set of search heuristics and continuously changes them during the testing process. To do so, we first define the space of possible search heuristics using the idea of parametric search heuristic recently proposed in prior work [5]. A technical challenge is how to adaptively switch search heuristics in the pre-defined space. We address this challenge with a new concolic testing algorithm that (1) accumulates the knowledge about the previously evaluated search heuristics, (2) learns the probabilistic distributions of the effective and ineffective search heuristics from the accumulated knowledge, and (3) samples a new set of search heuristics from the distributions. The algorithm iteratively performs these three steps until it exhausts a given time budget.

#### 4.11 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.

**Methodology:** 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 2 bugs 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. 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)
2. 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)
3. 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)
4. 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)

5. 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)
6. 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)