

# Dynamic symbolic execution (concolic execution)

Seminar: Understanding of configurable software systems

Bipin Oli

Advisors: Prof. Sven Apel,  
Christian Hecht

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 prioritize the branches likely to contribute to finding a bug. 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 [12] 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. 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 struggling to reach a deeper state. Due to these limitations symbolic execution is insufficient in many practical use.

Directed automated random testing [7, DART], more commonly known as concolic execution / dynamic-symbolic 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.

- SAGE and other examples - limitations

### 3 Introduction

The main idea of concolic testing is to execute the programs simultaneously with concrete values and symbolic values. When the program is executed, symbolic constraints along the executed path are collected in a formula called path con-

dition. Then, a branch is picked and negated from the path condition resulting in a new formula which is then fed to a constraint solver to check for satisfiability. If it is satisfiable, concrete test inputs are generated to follow the new feasible path. If it is unsatisfiable, the new path is infeasible and another branch has to be picked to be negated. This way, concolic testing attempts to improve the poor code coverage of random testing. A key characteristic of concolic testing is that path conditions can be simplified using concrete values whenever the decidability of their symbolic constraints goes beyond the capabilities of the underlying constraint solver. One major problem with concolic testing is that there are in general an exponential number of paths in the program to explore, resulting in the so-called path-explosion problem. Recently, several methods have been proposed to attack this problem from various angles: using heuristics focused on branch coverage [3], function summaries [8], using static/dynamic program analysis [2] and so on. We propose a new method based on interpolation, largely complementary to existing approaches, that significantly mitigates path-explosion by pruning a potentially exponential number of paths that can be guaranteed to not encounter a bug.

- What is it?
  - Where did it start?
  - How does it work?
  - Give an example
  - Why is it important? (give example of use in configurable systems)
  - It's contributions
  - It's limitations (give context to configurable system with many branches)
- One of the biggest challenges in concolic testing is that there are often too many branches to select for the next input. This is referred to as the path explosion problem [10, 11, 3]. The number of paths in the execution tree increases exponentially with the number of branches in the program. Visiting only the top twenty branches in the execution tree in a breadth first search (BFS) order requires more than one million concolic runs (220). However, programs usually have far more than twenty branches, for example, an execution path of `grep`, a 15K line of code program, contains more than 8,000 branches. Therefore, exploring all paths in an execution tree in a reasonable amount of time is not feasible.
- Example of the use of this technique in finding bugs in configurable system eg result of SAGE, EXE, etc.

## 4 Body

### 4.1 2007: Performing dynamic test generation compositionally

[6, paper]

**Gist:** The general idea behind this new search algorithm is to perform dynamic test generation compositionally, by adapting (dualizing) known techniques for interprocedural static analysis to the context of automated dynamic test generation.

**Methodology:** new search algorithm called SMART which stands for *duce a new algorithm, dubbed SMART for Systematic Modular Automated Random Testing*, a more efficient search method than DART without compromising completeness. It tests functions in isolation, collects testing results as function summaries expressed using preconditions on function inputs and postconditions on function outputs, and then re-use those summaries when testing higher-level functions.

A SMART search performs dynamic test generation compositionally, using function summaries as defined previously. Those summaries are dynamically computed in a top-down manner through the call-flow graph *GP*. Starting from the top-level function, one executes the program (initially on some random inputs) until one reaches a first function whose execution terminates on a return or halt statement. One then backtracks insofar as possible using DART, computing summaries for that function and each of those DART-triggered executions. When this search (backtracking) is over, one then resumes the original execution where it was called, this time treating it essentially as a black-box, i.e., without analyzing it and re-using its previously computed summary instead.

**Result:** SMART can perform dynamic test generation compositionally without any reduction in program path coverage. We also show that, given a bound on the maximum number of feasible paths in individual program functions, the number of program executions explored by SMART is linear in that bound, while the number of program executions explored by DART can be exponential in that bound. SMART = scalable DART

#### 4.2 2006: Software Partitioning for Effective Automated Unit Testing

[5]

**Gist:** present an approach that identifies control and data inter-dependencies between software components using static program analysis, and divides the source code into units where highly-intertwined components are grouped together. Those units can then be tested in isolation using automated test generation techniques and tools, such as dynamic software model checkers

**Methodology:** group together functions or components that share interfaces of complexity higher than a particular threshold. Complexity is determined by the popularity and sharing of code. The idea is that if the function is very popular and is being called from a lot of places then it is likely that it is not closely liked

to any component. And, the sharing of code meaning 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.

**Configurability part:** evaluated the effectiveness by applying the algorithm to the open source implementation of oSIP protocol (<http://www.gnu.org/software/osip/osip.html>) which is a telephony protocol for call establishment.

**Result:** showing that auto-matic software partitioning can significantly increase testcoverage without generating too many false alarms causedby unrealistic inputs being injected at interfaces betweenunits

### 4.3 2007: Hybrid concolic testing (\*\*)

**Gist:** an algorithm that in-terleaves random testing with concolic execution to obtainboth 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 thuscombines the ability of random search to reachdeeppro-gram states quickly together with the ability of concolic test-ing to explore states in a neighborhood exhaustively.

**Methodlogy:** presenthybrid concolic testing, a simple algorithmthat interleaves the application of random tests with con-colic testing to achieve deep and wide exploration of theprogram state space. From the initial program state, hy-brid concolic testing starts by performing random testingto improve coverage. When random testingsaturates, thatis, does not produce any new coverage points after run-ning some predetermined number of steps, the algorithmautomatically switches to concolic executionfrom the cur-rent program stateto perform an ex-haustive bounded depthsearch for an uncovered coverage point. As soon as one isfound, the algorithm reverts back to concrete mode. Theinterleaving of random testing and concolic execution thususes both the capacity of random testing to inexpensivelygenerate deep program states through long program execu-tions and the capability of concolic testing to exhaustivelyand symbolically search for new paths with a limited looka-head.

The interleaving of random and symbolic techniques isthe crucial insight that distinguishes hybrid concolic testingfrom a naive approach that simply runs random and con-colic tests in parallel on a program. This is because manyprograms show behaviors where the program must reach aparticular statesand then follow a precise sequence of in-put events 'alpha' order to get to a required coverage point. It is often easy to reachusing random testing, but notthen to generate the precise sequence of events 'alpha'. On theother hand, while it is usually easy for concolic testing togenerate 'sigma', concolic testing gets stuck in exploring a hugenumber 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

#### 4.4 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 `Vim` 5.7 (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.

#### 4.5 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).

#### 4.6 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 a 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.

#### 4.7 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]

#### 4.8 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 (0.5–150K LoC)

#### 4.9 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 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.

#### 4.10 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.

#### 4.11 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 executors 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.

#### 4.12 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.

#### 4.13 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.

#### 4.14 `paper.cc`

**Gist:**

**Methodology:**

**Configurability part:**

**Result:**

## 5 Different catagorical bodies

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: Auto-  
matically Generating Search Heuristics for Concolic Testing paper

- summerize, give an overview of the main points of each source and combine them into a coherent whole
- Analyze and interpret: don't just paraphrase other researchers—add your own interpretations where possible, discussing the significance of findings in relation to the literature as a whole
- Critically evaluate: mention the strengths and weaknesses of your sources
- Write in well-structured paragraphs: use transition words and topic sentences to draw connections, comparisons and contrasts

## 6 Conclusion

## 7 Papers

- 2006: they worked on backtracking algorithms for search heuristics [18]
- 2007: they combined the fuzzing techniques to improve the coverage [11] [6] [8]
- 2008: Heuristic based approach to select the branches [1]
- 2009: they worked on the fitness guided approach to improve the coverage [17]
- 2013: they boosted concolic testing by subsuming paths that are guaranteed to not hit a bug with their interpolation technique [9]
- 2014: they introduced a concept of context guided search strategy [14]
- 2018: automatic selection of suitable heuristic [2]
- 2018: template guided approach [3]
- 2018: based on probability of program paths and the cost of constraint solving [15]

- 2018: they improved the speed of SMT solver by removing the IR layer making it more practical to keep bigger constraints [19]
- 2019: fuzzy search strategy [13]
- 2019: adaptably changing search heuristics [4]
- 2021: Pathcrawler: proposed different strategies to improve the performance of concolic execution on exhaustive branch coverage [16]
- 2022: Dr. Pathfinder [10] combined concolic execution with deep reinforcement learning to prioritize deep paths over shallow ones for hybrid fuzzing

## 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. 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)
3. 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)
4. 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)
5. 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)
6. 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)
7. 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)
8. 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)
9. 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)
10. 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)
11. Majumdar, R., Sen, K.: Hybrid concolic testing. In: 29th International Conference on Software Engineering (ICSE’07). pp. 416–426. IEEE (2007)
12. 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)
13. 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>

14. 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)
15. 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)
16. Williams, N.: Towards exhaustive branch coverage with pathcrawler. In: 2021 IEEE/ACM International Conference on Automation of Software Test (AST). pp. 117–120 (2021)
17. 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)
18. 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)
19. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In: Proceedings of the 27th USENIX Security Symposium (Security). Baltimore, MD (Aug 2018)