Utrecht University
Department of Information and Computing Science

Computing Science master thesis

Evaluating Dynamic Symbolic Execution in the OOX based Symbolic Execution Engine

First examiner:
Wishnu Prasetya

Second examiner:
Gabrielle Keller

Candidate:
Tristan Albers

July 3, 2024

Abstract

Symbolic Execution is a technique which enables the complete verification of a piece of code. It can search for violations of predefined assertions and can either report a specific input which would trigger a violation or a valid verdict. However, these verification attempts costs large amounts of memory and computation power, especially for larger and more complex pieces of code. In an attempt to still verify these more complex programs we investigated Dynamic Symbolic Execution inside the OOX Ecosystem. We showed that the technique is also applicable for intermediate verification languages, is able to correctly validate and invalidate programs and on average does this faster and more reliable than previously implemented techniques. We also compared our implementation to state-of-the-art tools like JDart, JavaRanger and JBMC and showed that our proof of concept is competitive with their performance. The foundations in this work can function as a basis for more in-depth research about Dynamic Symbolic Execution.

Contents

1 Introduction **5**

1.1 Research Questions 7

1.2 Contributions 8

1.3 Thesis Structure 8

2 Related Work **9**

2.1 OOX 9

2.2 Heuristics for Complex Heap Programs 9

2.3 KLEE 10

2.4 Java Ranger 11

2.5 Incomplete Execution Techniques 11

2.6 JBMC: Bounded Model Checking for Java Bytecode 12

2.7 JDart 12

2.8 Genetic Algorithms and Dynamic Symbolic Execution 13

2.9 DSE Guidance with Partial Evaluation 13

2.10 SAGE 14

2.11 Mixed Concrete-Symbolic Solving 15

2.12 Automatic Exploit Generation 16

2.13 Mayhem 18

2.14 Driller 19

3 The OOX Ecosystem **21**

3.1 The language 21

3.2 Symbolic Execution 21

3.3 Limitations 24

4 Dynamic Symbolic Execution **25**
4.1 General Technique 25
4.2 DSE in the OOX ecosystem 27
4.3 Fuzzing the concrete execution 32

5 Evaluation **38**
5.1 Data Sets 38
5.2 Fuzzer Comparison 39
5.3 Concolic to Symbolic Comparison 43
5.4 Concolic to State-Of-The-Art Comparison 47

6 Conclusion **53**

Appendix

A MinDist Results **57**

B Depth-First Results **59**

C Concolic- Genetic Results **61**

D Concolic-Random Results **63**

E Fuzzer Comparisons **65**

F State-Of-The-Art Results **68**

Bibliography **73**

Acronyms

AST Abstract Syntax Tree. 21, 22, 31

CFG Control Flow Graph. 12, 21, 22

DSE Dynamic Symbolic Execution. 6, 11–13, 24–27, 31, 38, 43–54

GA Genetic Algorithm. 33, 34

IVL Intermediate Verification Language. 21, 25, 27, 31

MD2U Minimal Distance To Uncovered. 43–46, 50, 53

OOX IVL of Utrecht University. 21, 22, 24, 27, 28, 31, 32

PC Path Constraint. 22, 23, 26

RNG Random Number Generator. 32, 33, 40

SEE Symbolic Execution Engine. 21–23, 26, 29, 31, 54

SMT Satisfiability Modulo Theories. 23, 24, 26, 29, 31, 32, 34

1. Introduction

Computer programs are essential for many industries in our society. Applications made for the healthcare industry, banking and aviation run procedures and algorithms which are vital. It is essential that these programs behave the way the stakeholders intended otherwise there could be some dire consequences. An example of these consequences would be the incidents with the Therac-25. Because of an erroneous computer program, patients received an overdose of radiation [1].

Developers try to prevent incidents like this with the use of acceptance, integration and unit tests. Unit testing, the practice of writing tests for each component of the program to ensure no erroneous behavior takes place, is of particular interest because of its resulting metrics. Given a test suite consisting of these unit tests, tools such as Sonarqube can be used to gather the branch- and line coverage of the suite. These metrics are then used by software developers to substantiate claims about the correct behavior of the programs. However, this process is labor intensive and inherently incomplete since consistently maintaining a branch/line coverage of 100% is not realistic.

Another approach is formal verification which is a form of verification where mathematical techniques are used to guarantee the correct behavior of a piece of code. A form of formal verification is model checking. In model checking the program is first modeled as an automata. After these automata are constructed we can use techniques such as Linear Temporal Logic (LTL) to create specifications and check if the automata behave as the stakeholders intended [2]. This has certain benefits, such as the fact that verification can occur early in the design process. The drawbacks consist of the fact that the models are an abstraction of the program under test and their consistency must be checked. Another important fact about model checking is that there occurs a state explosion during the modeling of complex programs, making it more difficult to use the technique for industrial code.

Researchers in 1976 proposed a symbolic execution method for programs. This type of execution would be performed by a custom symbolic executor that treated the input variables as symbols instead of concrete values. The authors stated that if we treat the inputs as symbols it would be possible to reason about all possible execution paths. This is in contrast to concrete execution which can only reason about a single execution path. This means that symbolic execution is a natural extension of concrete execution. In this early work, the framework for symbolic execution was given for simple programs consisting of only signed integers, if-statements with "then" and "else" clauses and "go-to" statements. The authors also described how to convert the symbolic input

variables back to concrete variables using theorem proving. Using these features the authors were able to reason about all possible executions, but also retrieve concrete inputs to executions one might deem interesting. Finally, the authors concluded that using symbolic execution to represent a large or infinite class of normal executions can be useful in program testing [3].

The conclusions made by the researchers would prove to be true. The representation of all normal executions was used by other researchers to prove the absence of bugs in a piece of code, as well as reporting the bugs. However, the original representation of a program was quite limiting and thus researchers expanded the framework so it more accurately described the programs. Support for other variables like characters and strings were added, arrays were introduced and the introduction of polymorphism was researched. With the addition of these features, certain limitations were found. Researchers observed exponential growth in the state space, resulting in the explosion of execution paths. Memory limitations were also encountered, mainly due to the abstract modeling of complex data structures and their initialization. These observations resulted in a lot of different symbolic execution tools, each focussing on a specific language and introducing novel techniques to combat the limitations [4].

Likewise, researchers at Utrecht University made a symbolic execution engine. Their focus was the verification of an intermediate verification language called OOX. Currently, this system is able to verify concurrent programs and complex heap programs. However, The programs this system can verify are not complex or deep. All the previous tests have been run with a maximum statement depth of 150, which is not even deep enough to fully verify certain snippets of test code. In this work, we want to accomplish the verification of more complex programs with possibly deeper bugs. Previous research has shown that when trying to verify more complex programs which possibly contain deeper bugs the current model checking and symbolic verification techniques are not sufficient because memory, state and path explosions occur [4].

Because of the limitations of symbolic verification, a technique was created that attempted to circumvent these limitations. In previous research, this technique is called different names like Dynamic Symbolic Execution and Hybrid Symbolic Execution but for consistency, we will refer to it as DSE (Dynamic Symbolic Execution). This technique uses concrete executions to quickly explore the state space of a program until it is unable to progress any further. When the concrete part gets stuck DSE leverages the complete nature of symbolic execution to retrieve inputs which leads to previously uncovered paths. Thus, this approach tries to leverage the quick concrete execution times and selective complete analysis of symbolic execution to circumvent the limitations of symbolic verification.

To achieve the verification of more complex programs we will implement and research the Dynamic Symbolic Execution technique in the existing OOX Ecosystem. As described, this technique is used to circumvent the limitations of symbolic execution. Examples of tools that implement the technique are SAGE, JDART, DSE, AEG, Mayhem and Driller. However, these tools all use Dynamic Symbolic Execution in a specific way to achieve different goals. SAGE is more focused on larger-scale verification, while Mayhem, Driller and AEG all use the technique as an extra heuristic to guide the search towards specific security-related bugs. In this work, we will apply Dynamic Symbolic Execution to achieve the verification of complex programs with deeper bugs but we will not focus on specific bugs or errors that would decrease the state space. This is because we would preferably remain as complete as possible. This brings us to the research questions this work will try to answer.

1.1 Research Questions

Our goal is to verify programs and find bugs that the current OOX ecosystem cannot find. The technique under investigation to achieve this goal is Dynamic Symbolic Execution. The bugs we try to find are categorized as deep since we will go beyond the 150-depth limit of the previous research. The research questions we will answer are:

RQ 1) How does the introduction of Dynamic Symbolic Execution impact the speed, correctness and completeness of the Symbolic Execution Engine?

RQ 2) What is the impact of random and genetic fuzzing algorithms on the speed and correctness of Dynamic Symbolic Execution?

RQ 3) How does the OOX implementation of Dynamic Symbolic Execution compare to other state-of-the-art tools?

The first question focuses on the impact of the Dynamic Symbolic Execution technique and will compare its performance to the existing implementation. The second question tries to answer a question that is more focused on the technique. A large portion of the verification in Dynamic Symbolic Execution is performed with the help of a fuzzer, but the technique does not specify anything about how the fuzzer should preferably be implemented. We will take a closer look at this fuzzing element, comparing two implementations to see what impact each of them has on the performance of the overall technique.

1.2 Contributions

The contributions of this work are the following:

1. Introducing a Genetic Algorithm with test suite evolution into the Dynamic Symbolic Execution technique.
2. A comparison between Dynamic Symbolic Execution and symbolic verification-based techniques
3. Description on how to interpret the Dynamic Symbolic Execution technique for an intermediate verification language.
4. Extension of the OOX Ecosystem with Dynamic Symbolic Execution.
5. Providing an overview of techniques/tools that enable deeper (symbolical) exploration of programs.

1.3 Thesis Structure

This work is structured as follows. First, a related work section is provided which extensively covers all existing solutions pertaining to our work and details their workings. We then provide a background of the existing OOX ecosystem to establish the basis upon which this research is built. After this we will dive into the Dynamic Symbolic Execution technique, describing its general workings after which we explain how this generic template is fitted within the ecosystem. Then an evaluation of the technique is made, measuring the performance against the previous OOX ecosystem and state-of-the-art tools. We will then conclude with a discussion and future work section.

2. Related Work

In the following sections, the background of this work will be discussed. Since this research uses the work of D. van Vliet and S. Koppier as a basis, we will highlight elements of their work. Thereafter the sections will contain information about related work in the field. These sections will discuss influential tools that added novel contributions and techniques that are used to answer the research questions.

2.1 OOX

The basis of this research, and that of D. van Vliet, is the OOX language and its symbolic execution engine. OOX was proposed by Koppier as an intermediate verification language (IVL) which can be used to model concurrency of object-oriented languages like Java. All the semantics and formalizations of the IVL can be found in his thesis [5]. To accompany the IVL he also built an initial symbolic execution engine that is able to execute the program and verify it. This analysis is complete, meaning that if given unlimited resources the engine will find every bug. Besides the implementation, he also formalized the algorithms. Symbolic execution can incur a long execution time and memory explosions which is why Koppier also introduced several optimizations which were Formula Caching and Expression Evaluation. Even given these optimizations, the complete exploration of large programs and thus the detection of deep bugs is not possible.

2.2 Heuristics for Complex Heap Programs

A feature of object-oriented languages is inheritance and subtyping. While OOX was made to describe OO languages support for inheritance was not available yet which is why van Vliet added it to OOX. With this contribution, he also introduced a couple of new search heuristics. While verifying the inheritance implementation van Vliet found a state explosion. This occurred on a linked list which was declared with a super-type whereafter sub-types were added. The research also found that the explosion scaled exponentially after three sub-types where more than five sub-types caused a memory overflow. This occurred even with the optimized implementation of the aliasmap, meaning that without the aliasmap this would've occurred earlier.

The research also looked into four different heuristics: MD2U, Depth-First, Random-Path, and Round-Robin. The last one, Round-Robin, is a combination of Random-Path and MD2U. The experiments proved that Depth-First or Random-Path were faster than

the others. A remark that was made by the authors is that MD2U might prove more effective in larger-scale applications. This is due to the fact that it utilizes caches whose performance only starts to pay off in larger programs. These observations inspired our work to look at these larger programs with deeper bugs.

2.3 KLEE

KLEE is a symbolic execution tool that can verify programs and generate tests for the violated properties. KLEE is an improved version of EXE which was a similar tool previously created by the authors [6]. KLEE works directly with LLVM code and does not use an IVL like OOX to verify properties.

KLEE as a tool has many features and is for example able to simulate a complete OS environment. While it is not able to verify programs of industrial size, it works well on medium-sized programs. Because of its optimizations and modular implementation, many researchers have used this as a basis, especially if their research focussed on verifying properties of C++ [7].

The optimization techniques of KLEE are Compact State Representation, Query Optimization, State Scheduling, and Environment Modelling. The most interesting to look into in this thesis are the first three since OOX is not capable of interacting with the environment and environment interaction is not in scope for this research [5].

The first optimization is the "Compact State Representation". The authors state that since they track all memory objects, they can implement copy-on-write at the object level. With this implementation, they saw a decrease in per-state memory requirements. Because of their implementation of the heap as an immutable map, portions of the heap structure can be shared amongst multiple states.

Another optimization is query optimization. The authors state that the fastest query is no query. To reduce the strain on the constraint solver as much as possible they do a couple of things: Expression rewriting, Constraint set simplification, Implied Value Concretization, Constraint Independence, and Counter-Example Cache. The previous work of van Vliet has already implemented these optimizations besides Constraint Independence and Counter-Example caching.

The state scheduling optimization is essentially a heuristic that chooses which execution path should be prioritized. KLEE implements a Random-Path heuristic and a Coverage-Optimized Search heuristic where the latter aims to prioritize paths with program statements that are not visited yet.

2.4 Java Ranger

Java Ranger is an extension of the Symbolic PathFinder tool that employs the novel techniques: *Dynamic Method Region Inlining*, *Single-Path Cases* and *Early-Returns Summarization* [8][9]. To achieve *Dynamic Method Region Inlining* Java Ranger uses DSE (Dynamic Symbolic Execution) to merge multiple paths into a singular "Region Constraint". This region constraint is then used in conjunction with the DSE path constraint to describe the particular code region. After the creation of a region constraint the DSE jumps to the next exit point, which is either the branch's immediate post-dominator, a return statement of a method, or a set of program locations that requires DSE exploration, whereafter a new region constraint is generated. This process is repeated until the program under test is fully explored.

The construction of a region constraint starts when an if-statement is encountered. The "then" and "else" statement structure are encoded in Java Ranger's Intermediate Representation as a "Static Statement". The authors call this process statement recovery. After the statement is recovered information from DSE is used to concretize the statement further. The concretization of a static statement is a nine-step process explained in detail in the paper. For the sake of summarization, we will omit the specific steps but will mention that the steps eventually result in an instantiated statement. This instantiated statement has been transformed in such a way that it no longer contains any branching structure. Finally from this statement, a region constraint is generated.

In the experiment section, the authors compare the performance of Java Ranger to the Symbolic PathFinder tool upon which it was built and observed a reduction of 38% in execution time and a reduction of 71% in total execution paths. Furthermore, they participated in the SVComp software verification competition and won in the category "JavaOverall", which is a benchmarking set only consisting of Java programs. Lastly, the authors believe Java Ranger is complete and sound but defer the proofs to future work.

2.5 Incomplete Execution Techniques

Symbolic execution suffers from limitations like a path- and memory-explosion [4][10]. One way to mitigate these explosions is to relax the requirement of being fully complete. An incomplete technique tries to find bugs whilst being performant enough to verify larger programs. This is achieved by inspecting only the relevant execution paths or leveraging information gathered from concrete executions. This approach is often called Dynamic Symbolic Execution (DSE) but some researchers also refer to it as "Concolic Execution" or "Hybrid Symbolic Execution". The following sections will discuss incomplete tools, some using this approach, and their novel techniques.

2.6 JBMC: Bounded Model Checking for Java Bytecode

JBMC is a verification tool that uses bounded model checking to find either violations of a user-defined property, such as an assertion, or runtime exceptions. Its verification approach consists of parsing both the Java bytecode and the Java operational model of a program into a CFG (Control Flow Graph). This CFG consists of static single assignments meaning that a single node can be considered a statement whereas an edge can be viewed as a transition between two statements. From the CFG, verification properties and a specified bounding parameter a verification condition is built. If this verification condition is satisfiable a counterexample will be generated for the program, otherwise the program is regarded as being valid. Because of the bounding parameter, the verification of JBMC is inherently incomplete and can only be used to prove the absence of violations up to the specified bounding parameter. Furthermore, the verification condition is built as a quantifier-free formula.

Before the Java bytecode is converted to a CFG an extra translation step takes place. The bytecode is converted into a GOTO program. The authors state that this translation simplifies the representation of the program by replacing switch- and while-statements with if- and goto-statements. Furthermore, a symbolic simulation is done of this GOTO program. This simulation is responsible for unrolling loops and translating recursive functions into simple verification conditions.

JBMC has several verification functionalities. It is able to: track security vulnerabilities via taint-tracking, do equivalence checking on methods to further simplify the code and able to handle polymorphism, strings, arrays and exceptions. In theory, it is also able to verify concurrent programs but the authors state that this has proven to be quite difficult due to an occuring memory explosion [11].

2.7 JDart

JDart is a verification tool that uses DSE (Dynamic Symbolic Execution) to find violations. It is an extension of the Java Pathfinder tool and its main goal is to verify larger programs by allowing the verification of larger paths than symbolic verification. During a single DSE execution, JDart keeps track of both the symbolic values of the variables and the possible concrete values. The concrete values are gathered via an unspecified small meta-constraint solver which strives to find small concrete values.

JDarts implementation of dynamic symbolic execution is done via an executor and an explorer. The executor symbolically executes the code recording both the concrete and symbolic values of the inputs. When a decision point is reached this point is stored by the executor so back-tracking can take place. The explorer receives the recorded constraints of the executor and decides on its exploration strategy [12]. For the competition edition under review, this is a breath-first strategy [13]. The termination of JDart

happens either when a certain number of paths have been fully explored or when a time limit is reached. This choice is configurable by the user of the tool, but for the competition edition, the termination strategy is set to continue verification until the time budget has been spent.

The completeness of the tool is neither discussed in the competition contribution nor in the full paper of JDart. However, since the tool employs DSE and has no termination strategy which focuses on the complete exploration of a program we can gather that JDart is also an incomplete verification tool and thus cannot guarantee the absence of bugs in a program.

2.8 Genetic Algorithms and Dynamic Symbolic Execution

Researchers investigating the detection of cross-site scripting exploits, XSS, investigated a Dynamic Symbolic Execution approach for automated detection of these exploits [14]. This approach generates a test suite for the PHP code which could potentially trigger the XSS exploits. They generated these test suites with a random fuzzer, genetic fuzzer and DSE configured with a genetic fuzzer. They observed that there was a similar performance between the test suites generated with the random and genetic fuzzer, both approaches only found superficial bugs. When inspecting the DSE technique configured with a genetic fuzzer, they found that the generated test suite discovered more bugs than the other approaches.

Compared to our implementation of the DSE technique there are a couple of differences. Firstly, we implemented the technique for an IVL, meaning that our representation of the input is different than the representation of Avancini et al. since their input is modeled after the input of a PHP program. Secondly, their implementation of a Genetic Algorithm differs from ours. Where we are able to evolve multiple test suites with each other which results in a number of new test suites, Avancini et al. only evolve inputs with each other which results in a singular test suite.

2.9 DSE Guidance with Partial Evaluation

Researchers observed that static evaluation techniques, such as linters, can identify parts of the code that may exhibit erroneous behavior. One example given in the paper is an uncaught possibility of an overflow on an addition operator [15]. Since these evaluations are often an over-approximation the authors note that DSE can be used to investigate these evaluations further. An annotation system is proposed that explicitly states every evaluation made by the static tools. These annotations are then added to the code to be verified with the dynamic symbolic execution tool Pex. Besides verification, Pex is also able to generate test cases about the verified properties.

To prune the search space the authors created a method that removes verified paths. The observation made here is that since the tools provide an over-approximation each path that does not lead to an annotation can be pruned. This leads to more concise test suites generated by Pex. Drawbacks to be considered are the effectiveness of the static evaluation and the complete reliance on their annotations. The authors also state that these annotations can be done manually but this is labor intensive.

2.10 SAGE

SAGE is a tool whose main contribution is a generational search heuristic for DSE. The authors define Dynamic Symbolic Execution as a subset of an overall approach named Whitebox fuzzing. This stems from black box fuzzing where random inputs are used to test the program. These inputs are random or chosen arbitrarily and are thus referred to as a black box. Whitebox fuzzing also uses arbitrarily chosen inputs ranging from seeded inputs to inputs derived from a previously erroneous run. Where the two techniques differ is in that whitebox fuzzing generates inputs for subsequent runs based upon negated path constraints and are thus known or derived.

2.10.1 Generational Search

SAGE uses concrete inputs to drive the DSE. Where it differs from other tools using this approach is in its path search heuristics [16]. The authors made the observation that constraint solving can be costly and thus want to maximize the number of paths discovered by a single dynamic execution. To achieve this they propose a generational search algorithm. In this algorithm, the first generation concretely executes a path using random concrete inputs gathered from a seed. This results in a collection of path constraints. These constraints are then individually and systematically negated in a dynamic execution to create new concrete inputs which lead to uncovered paths. Using this approach a single dynamic execution with four conditional statements can create concrete inputs which lead to four undiscovered execution paths. These four concrete inputs are then collectively called the second generation. This cycle is repeated until a violation is found or every path is covered.

An important aspect of this search algorithm is the fact that each newly discovered input is scored by a heuristic meant to optimize block coverage as quickly as possible. Since the input with the highest score is picked as the next execution, it is possible for an input discovered by generation three to be executed before an input from generation two. This scoring system also solves the path divergence problem experienced in other research. Path divergence occurs when a path constraint is negated and inputs are generated which are supposed to follow path `P` but diverges to another previously covered path `P'` [17] [18] [16] [19]. Since the scoring system optimizes block coverage diverging inputs are scored lower and thus not picked to be executed. The authors

state that this technique is sufficient to deal with the divergence problem. This is because the diverging paths are never explored because of their low score. Furthermore, this technique has the benefit that no computation time is wasted in pruning them.

2.10.2 Experiments

The programs verified with SAGE discovered previously unknown bugs and security exploits in existing code bases. However, it must be noted that the bugs found were shallow. Furthermore, the number of bugs found depended on the initial seed given to SAGE where one seed would find only one bug a more optimal one could find six. Another interesting observation was that the authors found no correlation between block coverage and the amount of bugs found even though they hypothesized that there could be one. Lastly, the authors mentioned that the generational search algorithm would be fit for parallelization but did not implement it or elaborate on it any further.

2.10.3 SAGAN

In following work the authors introduced a logging tool for SAGE called SAGAN [17]. They added SAGAN to SAGE and ran this new tool for over 400 machine hours whereafter they reported their findings. The most significant reduction in time could be found in optimizing the constraint solving with techniques such as the "unsat-core" technique, caching, and constraint simplifications. These optimizations resulted in the fact that 90% of all constraints were solved within 0.1 seconds and 99% in under 1 second. Furthermore, while they did find path divergences and found that they negatively impacted the verification, they used the logging tool to find and fix these for a more complete result. This concluded in 33% of all security violations found.

2.11 Mixed Concrete-Symbolic Solving

The paper titled "Symbolic Execution with Mixed Concrete-Symbolic Solving" builds forth on the DART system [19]. The authors note the benefits of DSE, mainly its ability to handle third-party calls, but also observe a shortcoming with this feature [18]. It is possible that a library call is made whose argument is a symbolic value. This symbolic value should be substituted with a concrete value to make the execution of the library call possible. The way DART handles this is to first solve all the constraints regarding the symbolic value, ascertain a valid range, and then randomly assign a concrete value within this valid range, e.g. for the valid range `x >= 0` the assignment `x = 0` is chosen and inputted in the library call `library_call(x)` whose result can then be stored in a value such as `y`. When a conditional statement is then made on the variable `y` things can go awry. When trying to target one of the branches of the conditional statement path divergences can occur due to side effects in function `library_call()`.

To target one of the branches of a conditional statement more directly the authors proposed a system called concrete-symbolic, which splits up the path conditions (PC) into three distinct parts. The first part is called the `simplePC` which refers to everything easily solvable by off-the-shelf solvers. These can be checked for satisfiability and when they return unsat the authors state that per definition the whole PC is unsat. The second part is called the `complexPC` which refers to PC's where concrete variables are necessary, such as the patch condition `Y==0 && Y=foo(x)` where a concrete value for `x` is needed. The third and last part is called the `extraPC` and consists of extra equalities needed to link some complex variables to simple ones, safeguarding the soundness of the formulas. With this approach it is possible to construct PC of the form: `x>3 & y>10 & y=foo(x)` where first the range of `x` is determined whereafter it is assigned a concrete value. This value is then inputted into the library call `foo` to retrieve a value for `y` whereafter the whole constraint can be solved.

The authors state that their initial approach can still produce unsound results, but does verify more paths than other approaches such as DART. To further combat the unsoundness two heuristics are added, both of which are user-controlled. The first, more automatic, approach is "Incremental Solving". Referring to the previous example, not only `x=0` is chosen as a concrete value, but subsequent runs are also done with `x=1` and `x=2`, etc. up to a user-defined bound. The other heuristic is called the "Partitioning Heuristic". This heuristic is an annotation-based method where the user can define a partition at the top of the function to drive the symbolic value used in a library call toward a more correct concrete value. An example of an annotation partition could be `x>3 && x<5`. Applied to the previous example, the first concrete value filled in by the system would be three instead of zero.

2.11.1 Results

The authors state the following observations about their contribution. Firstly, their contribution is able to find more execution paths than DART but still remains inherently incomplete. Secondly, while partitioning based on user-provided input can be viewed as beneficial it can also be detrimental since it possibly restricts viable execution paths.

2.12 Automatic Exploit Generation

Automatic Exploit Generation (AEG) is a system consisting of several components working together to find exploitable security bugs and generate exploits for them. The exploit the authors focused on is buffer- or stack-overflows. With this exploit control-flow hijacking is possible which can be used to execute third-party code or spawn a shell with certain aspects. The proposed system is end-to-end and fully automatic, requiring only user input to drive heuristics if the user wishes to do so [20]. The system consists of six phases; pre-processing, source-analysis, bug-finding, dynamic binary analysis (DBA), exploit generation, and exploit verification. We will focus on the novel

techniques in this paper which appear in the source-analysis, bug-finding, and DBA phases while lightly covering the other phases.

2.12.1 Source Analysis

The AEG system takes as input the binary and the LLVM bytecode of the program which can be generated from the source code of the program. Given these inputs, lightweight static analysis is performed to gather information such as the max buffer length used throughout the program. Given this information, AEG assigns a maximum value to the symbolic variables used for buffer allocation as the largest found buffer plus 10%. This can later be used to drive DSM towards paths where an overflow is more likely to occur.

2.12.2 Finding Bugs

Symbolic verification is used to find bugs in a target program. The authors acknowledge the known limits of the technique and propose several optimizations to combat them. They hypothesize that this should be possible since classic verification is trying to prove the absence of bugs in a program and AEG doesn't try to give this guarantee. They relax this notion and state that every bug found with AEG is always exploitable. They do state that they believe AEG is able to find all buffer overflow exploits but don't provide proof. The first technique they describe is called "Pre-conditioned Verification". They state that since they are only interested in one certain bug they can prune non-interesting paths by adding an initial path-constraint. This constraint is called Π_{prec}. Examples for Π_{prec} are the known maximum length of inputs or a previously known prefix.

Heuristics are also used to prune the state space further. Two heuristics the authors propose are "buggy-path-first" and "loop exhaustion". Starting with the first, this is a novel technique that is only able to be used in a case where finding a general bug is insufficient. Classic verification would stop by the first bug found, but AEG is only interested in exploitable bugs. When a non-exploitable bug is found the path where it was found is given the highest priority to verify further. This heuristic was first based on anecdotal evidence but proved true in the evaluation. The intuition behind this is that if a buffer is misused once, the developer is probably prone to making more errors later.

The loop exhaustion heuristic can be seen as a depth-first heuristic in the sense that when a loop is encountered it is fully unrolled first before trying to exit it. Paths deeper in the loop have a higher priority in this heuristic. To avoid getting stuck in deep loops the authors propose to use preconditioned symbolic execution along with pruning to reduce the number of interpreters or give the highest priority to the deepest path.

2.13 Mayhem

Mayhem is a tool that continues the research done with AEG by taking the core techniques, optimizing them, and splitting the computation between two different engines where one is responsible for the concrete execution and the other is responsible for the symbolic execution. Some core assumptions have also changed, Mayhem works solely on the binary and does not assume it has the source code. It does also assume that the pre-conditions, discussed with AEG, are fully provided by the developer [21]. The static analysis was completely cut from the system with no clarifications from the authors other than the statement that domain knowledge can help drive execution more efficiently. We will discuss the different engines and how they interact and briefly describe all the optimizations the authors made.

2.13.1 Hybrid Symbolic Execution

The authors state that symbolic execution tools can be split into two categories called offline and online. Offline tools are described as classical symbolic executors that model everything fully symbolic whereas online tools are described as the concolic executors that combine concrete values into their (partially) symbolic execution. They state that Mayhem is a hybrid system, using the best features of online and offline tools. These features are that it should be able to run indefinitely, not repeat work or throw away previous work, and be able to reason about symbolic memory. To achieve this Mayhem is split into a Concrete Executor Client (CEC) capable of executing the binary on the CPU and a Symbolic Executor Server (SES) which drives execution, makes the calls to the SMT solver, and chooses which path is most interesting.

The concrete part of Mayhem takes place in the CEC. It is capable of executing a binary with variables that are considered symbolic. It does taint analyses and whenever a block is marked as tainted it is sent to the SES which can analyze the block and tell the CEC how to proceed. The CEC is also capable of running multiple concrete runs in tandem. In these runs everything up to the OS state is copied to ensure no side-effects occur during context switches. When a context switch should occur is calculated in the SES which in turn notifies the CEC.

Another part of Mayhem is checkpoints. Everything is stored in memory until a preset cap is reached. After this cap is reached path splits are recorded as checkpoints and put inside a queue. The queue is ordered based on which path is most interesting per certain heuristics later discussed. The checkpoints however are not executed until available memory comes free. This occurs when a path has been fully explored and the symbolic execution of that path has ended. Since multiple of these copies can occur CEC shares state across execution states. The modification to these states is stored to reduce the amount of storage needed.

The SES in Mayhem is an environment in which different symbolic executors may live. The number of concurrent executors is set as per user definition and each computes a different path. When the cap is reached checkpoints come into play again. They function the same as with the memory cap, as soon as one executor is freed it symbolically executes the most interesting checkpoint. Preconditioned symbolic execution is still part of Mayhem and functions the same as in AEG only differing in how the precondition is acquired. The path selection however is quite different from AEG. The buggy-path-first is disregarded. Instead, the authors mainly focus on the existence of symbolic pointers. The highest priority is given to paths where symbolic instruction pointers are found, thereafter priority is given to paths that identify symbolic memory access. The lowest amount of prioritization is given to the path that promises to cover previously uncovered code. With these techniques, Mayhem symbolically executes its programs.

2.14 Driller

Driller is a hybrid vulnerability excavation tool that uses whitebox fuzzing in combination with concolic execution to find different types of vulnerabilities that are deep in the code [22]. The novelty of this tool is the fact that it combines fuzzing and concolic execution to reach deep paths without limiting the types of vulnerabilities that can be detected. We will describe this technique and its performance in finding bugs.

2.14.1 Hybrid approach

The authors note that fuzzing and concolic execution each have their limitations. Fuzzing is particularly bad in reaching paths with specific guards like $input == "0x0123ABCD"$ since the probability of guessing this input with a randomized approach is so low it can be regarded as infinitely small. However, fuzzing does enjoy the benefit of being able to quickly execute a path in contrast to the concolic approach. Concolic execution creates a fork at every decision point which causes memory constraints the deeper a path travels thus is not scalable to reach deep points. However, it can solve the specific inputs needed to reach a certain instruction in the code. The authors observe that if they are able to combine these techniques they can use fuzzing for "generic inputs" which are defined as inputs that are not checked against specific guards. For these specific guards, they can use concolic execution to generate an input that is able to satisfy or negate the guard.

To achieve this the authors made whitebox-fuzzing the driving force behind path exploration. The fuzzer the authors used was AFL which uses a genetic approach to generate new inputs. The fitness is decided by how many new statements the input has reached with higher coverage of uncovered statements equalling a higher fitness. To handle loops buckets are created where one bucket groups all the inputs which reach the same depth. Important is that only the first input that reaches a certain depth

is rewarded for covering the statements. This coverage check is also used to decide whether the fuzzing part is "stuck" or not (Stuck being defined as unable to cover new statements with the new generation). If the fuzzer is stuck the input is given to the concolic executor.

The concolic executor uses the same index-memory representation as MAYHEM [21]. It uses the input received from the fuzzer as a precondition to ensure it walks the same path as the fuzzer. Whenever the execution encounters a conditional branch the precondition is lifted to solve it and check if it results in an uncovered statement. If this is the case the new input is recorded and execution is continued. When the execution hits the point where the fuzzer was stuck it solves the problematic branch and preferably a bit more. The authors state that concolic execution should resume a bit longer to prevent the case where the fuzzer and concolic executor would go back and forth too much, with the intuition being that one conditional branch may be followed by a couple more.

2.14.2 Results

The authors tested Driller on the DARPA dataset used for their Cybersecurity competition. They found that Driller scored as high as the winner of the competition and that it thus was competitive too. They do make the remark that this is not a definitive benchmark and that more research would be needed to see which tool would perform better. There are still some limitations to Driller. Since it uses fuzzing to combat the path explosion problem it relies on the fact that the fuzzer can make meaningful progress. If this is not the case Driller is reduced to a concolic execution tool, which is its worse-performing case.

3. The OOX Ecosystem

Several researchers at Utrecht University have studied symbolic verification. Their combined effort resulted in the OOX Ecosystem which is a collection of tools implementing novel techniques related to symbolic verification. In this chapter, we will describe the previous work done by these researchers to ascertain the context wherein our research takes place. The outline of this chapter is the following: First, we describe the work done on the OOX language, then we take a closer look at the current state of the symbolic execution engine. We will conclude with the current limitations of the ecosystem and place our research in that context.

3.1 The language

The OOX language is an IVL (Intermediate Verification Language) proposed by S. Koppier [5]. An IVL is a language into which the program under test is transpiled, meaning that a Java program under test first needs to be transpiled to OOX before verification can take place. OOX is designed to work on object-oriented programs, treating objects as first-class citizens, and has a strong type system. Its main feature was its capability to model concurrency via a fork-join model. In the work of S. Koppier, it was stated that while OOX was capable of modeling object-oriented languages its capabilities were limited to declaring classes and using objects.

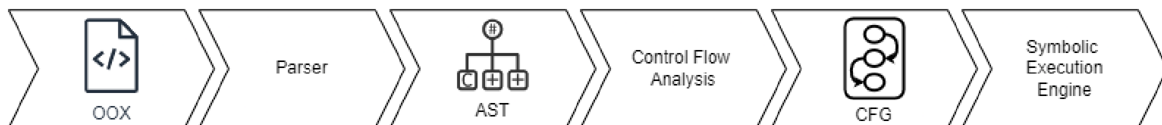
The introduction of inheritance and polymorphism in OOX came later with the work of D. van Vliet which focussed on complex heap programs [23]. To be able to verify complex heap programs with OOX the language was expanded to explicitly model inheritance and polymorphism. Besides this expansion, the verification engine, which consumes an OOX program and produces a verdict, was also improved to lazily handle object initialization.

3.2 Symbolic Execution

In the previous research symbolic execution is used to ascertain a verdict about the program under test. This verdict communicates whether the program contains any predefined violations or not. The tool that reasons about these violations and performs the symbolic execution is called the SEE (Symbolic Execution Engine). Before the SEE can reason about the program under test a couple of transformations need to take place. First, the OOX program is parsed into an AST (Abstract Syntax Tree). This AST is then passed through a control flow analysis which produces a CFG (Control Flow Graph).

Control flow analysis can be either interprocedural or intraprocedural. During the initial work on OOX by S. Koppier the design choice for intraprocedural analysis was made and remained unchanged throughout the development of the ecosystem. This means that one AST produces several disjoint CFGs, where each graph describes the flow of a single function in the program. This results in the SEE taking the following inputs: the AST of the program, the CFGs, a symbol table describing classes and fields in the program and the entry point of the program. A visual representation of this architecture is shown in Figure 3.1.

Figure 3.1: Overview of SEE architecture



The inputs given to the SEE provide the starting point and context for symbolic verification. During the symbolic execution important data such as the stack, heap and PC (Path Constraint) are tracked and stored in a state. The stack and heap are used to store the concrete variables and objects respectively. The PCs are formulas collected while executing a specific path. The formula of the PCs in a state describes the execution path taken by that specific state. Let us use the code snippet from Figure 3.2 as an example to illustrate the PCs and states.

Definition 1 (Path Constraint). Let $\sigma_1, ..., \sigma_n$ denote all the possible symbolic executions and $I_1, ..., I_n$ denote the input variables. The path constraint, denoted by PC, of a path σ_n is a formula that contains constraints upon the input variables $I_1, ..., I_n$ to follow the respective path.

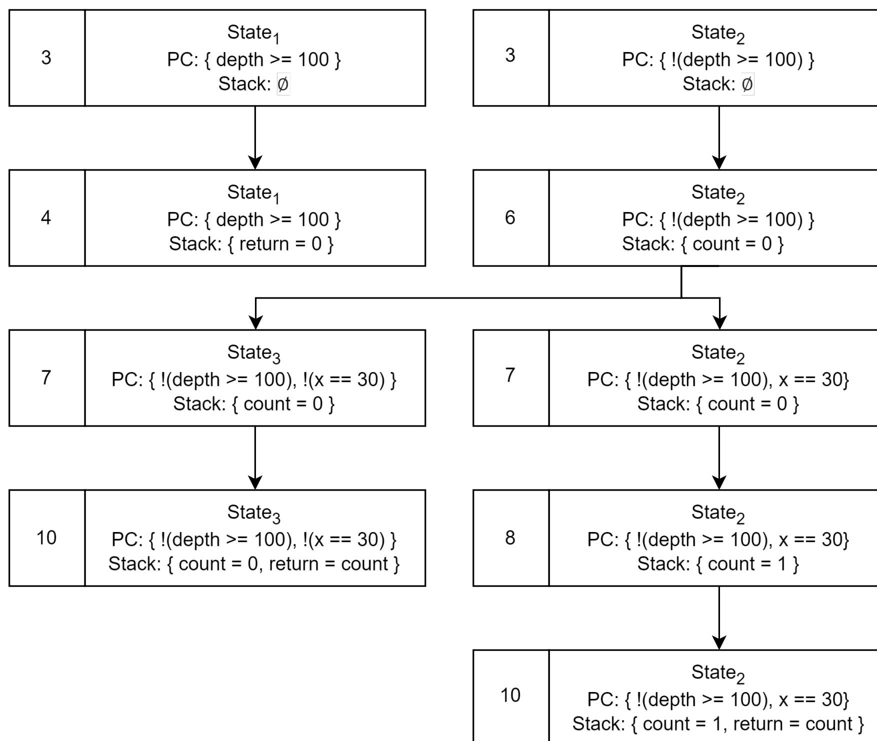
Figure 3.2: A code snippet of an OOX program

```
1   static int check(int x, int depth)
2   {
3       if (depth >= 100){
4           return 0;
5       } else {
6           int count := 0;
7           if (x == 30) {
8               count := 1;
9           }
10          return count;
11      }
12  }
```

The function body starts at line 3 with an if-statement. This means that at the start of the execution we already need to split the state. Each state follows a unique execution

path, one where the guard of the if-statement, `depth >= 100`, holds and one where it does not hold. To achieve this two states with identical data for the stack and heap are created. They only differ in the PC which for $state_1$ is "`depth >= 100`" and for $state_2$ is "`!(depth >= 100)`". After this separation, each state can independently be further executed. The process of choosing which state should be executed first is often delegated to a heuristic of some kind. Continuing our example, $state_1$ would immediately return 0. $State_2$ can still be further executed, at line 6 we add the variable "count" to our stack with the value 0. Thereafter a new split needs to be made. The newly created $state_3$ will receive the new constraint of `!(x == 100)`, which is the negated guard of the if-statement. After this addition, the count value is returned leaving the final PCs of $state_3$ at $\{!(depth >= 100),\ !(x == 100)\}$. Taking $state_3$ as an example, the stack and heap give us the values the program would have if the conjunction of the PCs holds. Visual representations of these states with respect to the program location is shown in Figure 3.3.

Figure 3.3: Representation of states at program location in the OOX program



The verdict is generated with the help of a SMT (Satisfiability Modulo Theories) solver. The SMT solver used in this research is Z3 which was also a design choice made by S. Koppier that remained unchanged. This solver consumes a formula and returns if it is satisfiable or not. The formula the SEE passes to the solver is constructed from the PCs and other state information. Because the SMT solver checks satisfiability, which is to say it checks if there is an assignment to the values of the data that upholds all the constraints, we cannot use the formula generated from the data directly. Instead,

we take the formula generated from the collected data during the run and negate it. Then we pass this negation to the SMT solver. If it returns the verdict as "satisfiable" we know that a counterexample was found and the non-negated formula does not hold. Vice versa, if it returns "unsatisfiable" we know that no counterexample was found and thus the original formula holds. The holding of the original formula directly corresponds to the verification verdict which is to say that if the SMT solver returns "unsatisfiable" the program is correct and vice versa.

3.3 Limitations

The work of S. Koppier and D. van Vliet both focussed on the symbolic verification of a program and used symbolic execution to achieve this. Koppier proposed this system to verify concurrent programs and van Vliet expanded the system to work on complex heap programs. However, symbolic execution has some inherent limitations. These come forth from explosions that occur if the program size is too large. These explosions mainly regard memory consumption and path size and are problematic enough that they prevent symbolic execution from verifying larger code bases or unearthing deeper bugs [4] [10]. This research will focus on that limitation, discussing a technique called DSE (Dynamic Symbolic Execution) that extends the OOX ecosystem with the capability to verify larger programs and find deeper bugs than symbolic execution.

4. Dynamic Symbolic Execution

A promising technique that is able to verify larger programs with deeper bugs is DSE (Dynamic Symbolic Execution). This technique leverages the speed of concrete execution and completeness of symbolic execution to quickly and thoroughly inspect the state space in search of violations. Different tools like SAGE [16], Mayhem [21], Driller [22], AEG [20] all have used this technique or its architecture to successfully cover larger programs and find deeper bugs. Based on their previous successes with this technique this research will evaluate the impact of DSE on a custom IVL.

The outline of this chapter is as follows. Firstly, the general architecture of DSE is discussed. The individual components and their functions will be examined. Secondly, the integration of DSE in the OOX ecosystem will be discussed. Lastly, the random fuzzing of concrete inputs will be discussed, along with our own Genetic Fuzzer.

4.1 General Technique

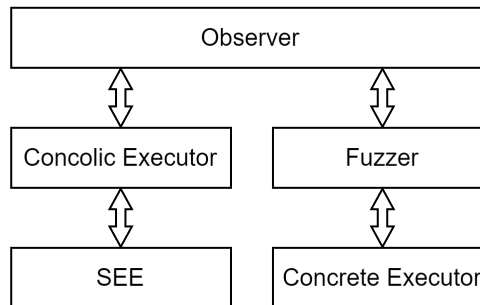
The DSE technique consists of a concolic executor and a concrete executor working in tandem to cover the program under test as thoroughly as possible. The executors are controlled by a controlled by an "Observer" class. The observer is the most top-level facing component and has a couple of functionalities. It should collect meta-data about the run and pass information between the two executors. The information passed between the executors contains interim verdicts and information about the progression of the search. If this verdict is "Invalid" a violation is found and should be propagated upwards. When no violation is found the observer is responsible for making the decision to continue, time out, or prematurely abort a run. This decision is based on how much of the program under test is covered and if any progress on the coverage of the program can be made. When looking at the different implementations of DSE it has to be noted that this class is not necessarily needed, and can be implemented as part of another main or parent class.

Concrete Executor

The concrete executor is responsible for exploring and verifying the program under test by executing code. Since this should be an actual execution of the program a crash of any sort should be handled as a violation of a safety property. Furthermore, this should be handled in such a way that the execution of the observer can continue. This way meta-data about the found violation can be reported and propagated up. If no violation is found the concrete executor should report that finding to the observer.

From this point, the implementation of the "concrete executor" starts to differ between implementations. Most of them create a wrapper around the concrete executor in the form of a fuzzer or other sort of algorithm. This wrapper can generate input values for the program and continuously call the concrete executor to explore the state-space. This is often done up to a pre-defined limit. An example of a wrapper like this is the generational algorithm in SAGE and the AFL fuzzer of Driller.

Figure 4.1: Information flow of DSE components



Concolic Executor

The concolic executor is used to explore the program under test by finding input values that lead to previously uncovered code. While the exploration of the program during the concrete phase is fast, it often gets stuck whereafter it is unable to achieve any meaningful progression. When this occurs concolic execution is invoked to find input values leading to uncovered paths. This process is what enables DSE to explore deeper than for example fuzzing techniques.
An example of a program location that is hard to reach with a fuzzer can be seen in Figure 3.2 at line 7. The transition to the "true" branch can only be taken with the satisfied constraint that the value of x exactly equals 30. However, it is difficult for our concrete executor to guess that exact value, even with a fuzzer in place. While this value is small and thus could eventually be guessed it is an example of the limitations of concrete execution, which are exacerbated with more specific constraints like an equal constraint on two strings.

Symbolic execution does not guess the value of x but treats it symbolically. The SEE can reason about the value that x should be to reach line 8. This is done by looking at the PC gathered up until line 7 in conjunction with the guard. Figure 3.3 shows an example of this at the transition from program location 6 to 7. The state that copies the guard follows the "true" branch, whereas the negation of the guard follows the false branch. If, from this example, one would need to infer the value of x at the true branch we get a formula that we can solve, namely "`x == 30 && !(depth >= 100)`". To get the answer to this formula it can be passed to an SMT solver like Z3. When retrieved, the values Z3 found for "x" and "depth" can then be inputted into the program to follow that specific path.

Concolic Execution

Concolic execution uses the information gathered by the concrete execution in combination with symbolic execution to discover new paths. Firstly, when the concrete executor cannot progress in finding new paths, the last program input which covers new statements is passed to the concolic executor. After receiving these input values the concolic executor walks the same path as the concrete exector. This is achieved via the pre-constraining of the path condition with the received values from the concrete executor. For example, let us take the program from Figure 3.2 and state that the concrete executor returned the set $\{x = 802,\ depth = 55\}$. The concolic executor, before execution begins, treats the received set as a constraint named "PC_{pre}" on the path condition. This pre-constraining of the path condition ensures that the concolic executor walks the same path as the concrete executor. When we now encounter decision points like if-statements we can calculate which branch is already covered by the concrete executor, and for which branch we would need to calculate the input values as to cover the previously uncovered statements. We will deliberate further on this process in the next section where we explain how the previously discussed methods work in the OOX ecosystem.

Definition 2 (Pre-Constraint). Let γ denote a program input and σ_γ denote the execution path of input γ. We can ensure that our concolic executor walks path σ_γ by adding a pre-constraint to the path constraint. This pre-constraint, denoted by PC_{pre}, constrains the input variables with the concrete values as described in γ. E.g. if we take the previously mentioned input γ of $\{x = 802,\ depth = 55\}$ our pre-constraint is $PC_{pre} = \{x == 802,\ depth == 55\}$.

4.2 DSE in the OOX ecosystem

Earlier work using the DSE technique only verified programs written in C [16][22]. To make the technique compatible with a custom IVL changes need to be made to the concolic and concrete executor. In this section, these changes to the technique will be discussed alongside any optimizations or improvements to the DSE technique.

4.2.1 Observer

The observer used in this work contains a list of every transition made between two program locations, alongside a count of how many times that specific transition has been encountered. The structure of this progression metric is shown in formula 4.1 and used throughout this work. The formula denotes the transition from a statement "one" to a statement "two", which is denoted by l_1 to l_2 and a which count is denoted by c. The count can be referred to as an iteration metric, containing how many times we have iterated over, or visited, a certain transition. This structure is a more coarse version of the concept named "bucketization", which enables deeper exploration by monitoring

the number of times the exploration iterated over a transition [24]. Using this concept the observer is able to target paths that iterate multiple times over a certain transition and treat them uniquely based on their iteration count. We state that a program cannot be explored further if *a)* no new transitions, l_k to l_m, can be found and *b)* the count associated with each transition does not increase. This check is run every time after the concrete and concolic executor have reported their findings. This constraint on progression can also be formulated the following way: If the concolic executor cannot increase the transition set any further we conclude our verification effort. In practice, this means that the technique returns a "Valid" verdict about the program under test.

$$\{ ((l_1, l_2), c)_1 , \dots , ((l_k, l_m), c)_n \} \tag{4.1}$$

Besides this addition to the tracking of progression, the observer behaves the same as in the standard technique, which is to say that it invokes the concrete and concolic executor in a loop and passes information between them while collecting meta-data about the run. The described workflow of the observer is illustrated in algorithm 1. Here we mention the requirement of a fuzzer and a concolic execution function. This is done to highlight the fact that these components are decoupled from the observer algorithm. Lastly, it is important to mention that this function only returns a verdict that is either "Valid" or "Invalid".

Algorithm 1 Observer algorithm that controls DSE

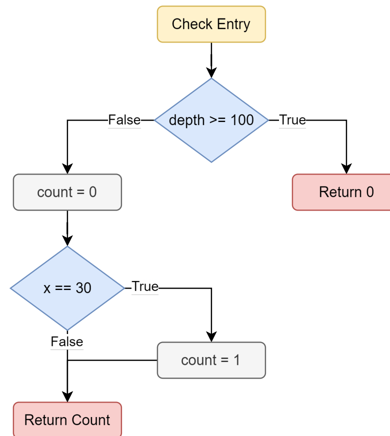
```
Require: fuzzer, concolic_execution
  total_coverage ← ∅
  new_inputs ← ∅
  while True do
    (coverage, last_input, validity) ← fuzzer(new_inputs)
    prev_total_coverage ← total_coverage
    total_coverage.update(coverage)
    if validity == False then
      return(Invalid)                                  ▷ Found bug
    else if prev_total_coverage == total_coverage then
      return(Valid)                                    ▷ Cannot progress
    end if
    new_inputs ← concolic_execution(last_input)
    collect_metadata()
  end while
```

4.2.2 Concolic Executor

The concolic executor receives the input values from the last execution of the concrete executor which covered new statements. It is then tasked with finding new program inputs that can cover previously uncovered statements of the program. In the OOX

ecosystem this functionality is built directly on the SEE and functions as a heuristic, deciding which path to take based on a pre-constraint. This results in a total path constraint during concolic execution of $PC_{total} = PC_{pre} \cup PC_{exec}$. When a decision point is reached, denoted in the SEE as an assume statement, there exists an opportunity for the executor to find input values leading to new paths. When an assumption is encountered it is checked for feasibility. This is done by converting the assumption to an assertion, negating the whole formula and passing it to the SMT solver Z3. If the solver returns that the formula is feasible we know that the path under investigation is the path previously walked by the concrete executor. This is true since we checked the feasibility with PC_{total} which includes PC_{pre}. If it returns unfeasible there can be two causes for this result. Either the specific branch this path tries to take can never be taken, or the PC_{pre} constrains the input variables in such a way that that path cannot be taken. This is checked by temporarily lifting the PC_{pre} constraints, effectively checking the satisfiability of the formula with only PC_{exec}. If this returns unsatisfiable we conclude the statements are unreachable. However, if the solver does return that the formula is satisfiable we have found a path that can be taken by the program, but not with the input values from the PC_{pre} constraint. To get the inputs that lead to the newly discovered path we request the model from Z3 and retrieve the values of the input variables leading to this new path. These values are stored and concolic execution continues until the end of the path which PC_{total} follows is reached.

Figure 4.2: Flowchart of the check function.



The collection of new inputs is calculated for every assume statement encountered by the concolic executor. For example, given the input set "$\{x = 10, \mathit{depth} = 10\}$" the concolic executor calculates new inputs twice. A flowchart of the example program is given in Figure 4.2. Each if-statement, denoted by the blue color, causes the executor to do this calculation when encountered. The collection algorithm can be viewed in Algorithm 2.

To help the algorithm perform better some optimizations were introduced. The optimizations are focused on reducing the amount of times the SMT solver will be invoked, since this proved to be the most computationally heavy part of the technique.

Algorithm 2 Collection of new inputs

Require: $stmt,\ PC_{pre},\ PC$
  procedure $collect_inputs(stmt)$
    if $stmt$ is $assumption$ **then**
      $expr \leftarrow get_expression(stmt)$
      $res_1 \leftarrow satisfiable_check(expr \wedge PC_{pre} \wedge PC)$
      if $res_1 = false$ **then**
        $res_2 \leftarrow satisfiable_check(expr \wedge PC)$ ▷ Lifting PC_{pre}
        if $res_2 = true$ **then**
          $input \leftarrow solve(expr \wedge PC)$
          $collect_input(input)$
        end if
      end if
    end if
  end procedure

Recalling the progression of covering new statements, as shown in 4.1, each transition consists of two statement locations and an iteration count. The concolic executor only tries to generate new inputs if the statement after the assume *1)* results in a previously unseen transition between l_k and l_m or *2)* covers an already existing transition but increases the iteration count c. Some work also went into trying to track which transitions between statements are always unsatisfiable, but this proved to be rather unstable causing an overly eager estimation that would skip over feasible program paths. This is due to the fact that a transition being unfeasible in one execution would not always mean it is unfeasible in another.

Algorithm 3 Concolic Execution

Require: $program_counter,\ start_statement,\ coverage_set,\ input_variables$
  procedure $concolic_execution(input_variables, coverage_set)$
    $new_inputs \leftarrow \varnothing$
    $\sigma \leftarrow preconstrain_program(start_statement,\ input_variables)$
    $tree \leftarrow single_execution_step(\sigma,\ program_counter)$
    while $tree$ is not $null$ **do**
      $current_root \leftarrow tree.get_root()$
      $current_pc \leftarrow current_root.pc$
      for $child$ in $tree$ **do**
        $next_pc \leftarrow child.pc$
        if $(current_pc, next_pc)$ not in $coverage_set$ **then**
          $new_inputs \leftarrow collect_inputs(child.statement)$
        end if
      end for
      $program_counter \leftarrow program_counter + 1$
      $tree \leftarrow single_execution_step(\sigma, program_counter)$
    end while
  end procedure

In algorithm 3 we denoted the previously described workflow of concolic execution. We use the σ notation to denote a program, which has information such as the current program counter and a starting statement. From this starting statement the SEE constructs a tree that continuously grows with each execution step. A single execution step essentially furthers the program by one atomic action, allowing us the inspect the program statement by statement. After each of these execution steps the possible transitions the program can take, denoted by the children, are checked against the existing coverage set to determine if there are any new transitions. If this is the case then we invoke the input collection in algorithm 2, otherwise the tree is extended and the root is set to the new statement resulting from the single execution step.

4.2.3 Concrete Executor

DSE relies on the performance of concrete execution to quickly discover the search space, covering statements that were previously uncovered. However, since OOX is an IVL it does not have an interpreter or compiler which can be leveraged. To circumvent this limitation the SEE is repurposed, enabling it to concretely execute the program. As demonstrated with concolic execution, by pre-constraining the path condition we can force the SEE to follow a certain path, essentially mimicking execution. However, the problem with this solution is the fact that the inputs are still marked as symbolic by the engine. To resolve these symbolic elements the SMT solver is invoked, which incurs a large computational cost. Given that these elements need to be resolved on every decision point to steer the executor towards the right path, the overall execution speed suffers greatly.

Since pre-constraining the path does not reduce the symbolic reasoning of the SEE we considered the use of stack insertion. With stack insertion the meaning of the AST is altered in a significant way, essentially inserting new statements initializing and declaring the input variables. Since only the inputs of a program are symbolical directly inserting their values in the stack completely removes the symbolical reasoning. This in combination with the already existing optimizations like expression evaluation results in fast execution speeds. From our observations, these execution speeds approximate the execution speed of interpreters like that of Python, but more importantly were about 100 times faster than the pre-constraining method.

Besides the repurposing of the SEE, a wrapper around the concrete executor in the form of a fuzzer was created. To gauge the impact of a fuzzer on DSE two types of fuzzers were implemented and tested. The fuzzing techniques and their implementation will be discussed next.

4.3 Fuzzing the concrete execution

A fuzzer is a tool that generates values for inputs. The generation of these values is often informed guesses, meaning that some algorithm is basing the generation of the value on a certain heuristic. In the context of the concrete executor, the fuzzer is responsible for invoking concrete execution with its generated values for the inputs. The metrics upon which it can base its decision are: the result of the run, the encountered transitions of the last run, the total found transitions and a list of found inputs from the concolic executor in the form of formulas. In this section we will discuss a fuzzer that uses a RNG (Random Number Generator) to generate its values, called the "Random Fuzzer", and a fuzzer that uses a form of genetic algorithms to generate its values called the "Genetic Fuzzer".

4.3.1 Random Fuzzer

The Random Fuzzer was initially created to assist the concrete execution with the generation of values. The first type of generation is based on the program inputs. A list of the inputs and their respective type is provided whereafter the Random Fuzzer invokes the RNG for each input to get a value. Depending on the input type the value is converted to their respective type. For integers and floats the RNG is called with the respective ranges, as defined per OOX language. For booleans, the RNG has a probability of 0.5 to return true. Lastly, for chars we defined a character set, generate a value between 0 and the length of the set and return the character at the index of the generated value.

The second type of generation is based upon the formulas received from the concolic executor. To extract a value from these formulas we pass them to the SMT Z3, which responds with a model. This model contains a value for each input that was present in the formula. In some cases, it is possible that the formula doesn't contain every input variable. This occurs because the gathering of input values is invoked at each decision point. When looking at Figure 4.2 we can see that the first decision point does not mention the input variable "x" at all. Thus it follows that if an alternative input is generated from this decision point no constraints exist on the variable "x". For this reason, when value generation is done based upon the formula an extra check is performed to ensure all inputs have concrete values. If this is not the case the RNG is invoked to generate the value.

When the Random Fuzzer is enabled it controls the concrete exploration of the program. It achieves this by first executing the program with all the inputs found by the concolic executor, if any. These inputs are initially reported as formulas but transformed to concrete variables to enable concrete execution. During each execution, it keeps track if there is progress in the exploration of the program. If the input values progress the exploration they are temporarily stored and the set of total found tran-

sitions is updated with the found transitions of the run. After all the input values of the concolic executor, if any, have been executed exploration with the RNG starts. Input values are generated and concretely executed, its values stored if it progressed exploration. The Random Fuzzer keeps running until it determines it is stuck. This is tracked via a counter, which increases if the concrete execution doesn't progress exploration. If the counter reaches five the fuzzer stops and gives the last progressing input values to the concolic executor to see if it can provide input values that cover new parts of the code. This process is also illustrated in Algorithm 4.

Algorithm 4 Random Fuzzer algorithm

```
Require: formula
  procedure Random_Fuzz(formula)
      for formula in concolic_inputs
          fuzz_execution(formula)
      end for

      while count < 5 do
          progression ← fuzz_execution(∅)
          if progression is false then
              count = count + 1
          else
              count = 0
          end if
      end while
      return(last_covering_input)
  end procedure

  function fuzz_execution(formula)            ▷ Either formula containing input or ∅
      input ← fuzz_input(formula)             ▷ Returns random input if formula = ∅
      (verdict, progression) ← execute(input)
      if verdict is false then
          throw verdict
      end if
      if progression is true then
          last_covering_input = input
      end if
      return progression
  end function
```

4.3.2 Genetic Fuzzer

As an alternative to the Random Fuzzer a Genetic Fuzzer was created. The motivation behind the use of a GA (Genetic Algorithm) lies in its evolutionary properties. GAs are a sort of algorithm inspired by the biological process of genetic evolution which mutates genotypes and genes to increase their fitness [25]. This fitness increases until either a maximum is found or a specific boundary, such as execution time, is exceeded.

If we define the fitness as the total coverage of a program we can use a GA to increase the capability of finding new transitions without invoking the computationally heavy SMT solver. That is if we are able to convert the data that needs to be evolved into the architecture of an GA.

Problem Definition

The first step in the creation of the Genetic Fuzzer was the problem definition. The goal is to increase the total coverage of a program, but the data that needs to be evolved has to fit the genetic mold, which is to say that it should be converted to genes that can be mutated. Towards this goal, we could translate each singular input of a program to a gene and assign it a fitness. However, this definition is not coarse enough since we cannot assign the coverage of a program to a singular input. To alleviate this we took inspiration from EvoSuite which is a tool that uses GA on the level of test-suites. It does not only mutate the finely detailed genes but also the coarsely defined test-suites [26][27]. There are three levels of detail, the finest being the gene. A singular gene denotes a singular variable, describing its type and value. A bit more coarse on the second level we have the genotype. The genotype denotes a collection of genes and can be seen as the actual input of the program, that is to say, we can convert a genotype into values for the input of the program. Since a genotype can be seen as a single run of the program, each genotype tracks its personal coverage. The coarsest level is the suite, which is a collection of genotypes and corresponds to the test-suite of EvoSuite. This suite essentially consists of test-cases, denoted by genotypes, and has its own coverage which is the cumulative coverage of all its genotypes. Using this definition it is now possible to 1) mutate specific inputs in the form of genes *2)* mutate specific test-cases in the form of genotypes and *3)* mutate whole suites where the goal is to find a suite with the highest coverage of the program.

Algorithm

A genetic algorithm mainly uses two types of modification, inspired by biology [25], which are mutations and crossovers. A mutation can be seen as modifying the value of a gene. A modification on a more coarse level is the crossover, which creates a new genotype by combining the first half of a genotype with the latter half of another genotype. These modifications are left to chance, meaning that a mutation or crossover will take place with a certain probability. From these basic algorithmic principles and some inspiration from EvoSuite our Genetic Fuzzer was created.

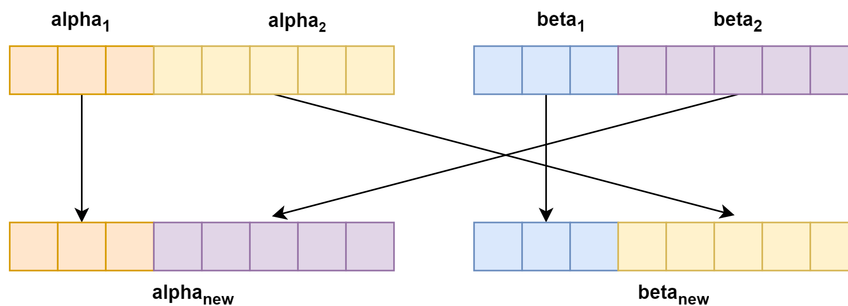
From a cold start, the Genetic Fuzzer creates six suites, each with ten randomly generated test cases, "genotypes". We then evolve this initial population of six suites for six generations. This process of evolving to the next generation starts with calculating the fitness of each suite. The fitness should correctly describe the best suite. Since we describe progress in the form of transitions and their count, the fitness function should reflect this. We define an ordering where firstly the highest amount of branches, or

unique transitions, is considered the most fit. However, if two suites have an identical amount of transitions, the number of times transitions have been encountered, also called iterations, are summed. The suite with the highest value is deemed the most fit. To calculate this fitness we need the transition set, as defined by 4.1, for each suite. The set for each suite is the union of the sets generated by the genotypes. The transition set of the genotypes is calculated by converting the genotype to the program input and concretely executing it. Now that the fitness has been calculated we can evolve our population to the next generation.

Definition 3 (Fitness Function). Let the coverage set be $\{ ((l_1, l_2), c)_1, \ldots, ((l_k, l_m), c)_n \}$, let n denote the amount of unique transitions, (l_k, l_m), in the coverage set and $\alpha = n$. Let $\beta = \sum_{i=1}^{n} c_i$ be the sum over all the iteration counts in the coverage set. Then our fitness function takes a coverage set of a suite and returns a tuple (α, β). The fittest suite has the highest α value. When suites have the same α value, the fittest suite has the highest β value. Otherwise, the fittest suite is chosen undeterministicly.

When evolving we first sort the suites based on their fitness. The suite with the highest fitness is then copied straight to the new generation. The reason for this is that we want to keep our best solution between generations. This is because evolving a population may produce fitter suites, but this is not guaranteed. From our observations, the evolved suites could even be worse than the initial population. Thus to prevent losing our best solution we do not edit it and copy it into the next generation. We then start with the crossover modification.

Figure 4.3: The crossover mutation between a suite "alpha" and a suite "beta"



The crossover modification has a probability of 0.8 and "breeds" two suites into two new suites. The suite that is targeted for breeding is either bred with our fittest suite or our second fittest suite with a probability of 0.5. The breeding of a suite with one of the fittest suites is done to try to improve the best suites as often as possible. Breeding a suite alpha with a suite beta goes as follows. First a split point between 1 and the length of the genotype list minus 1 is randomly chosen. We then split both suites and combine the split parts into new suites. For example, suite $aplha_{new}$ would consist of $alpha_1$ and $beta_2$ whereas suite $beta_{new}$ would consist of the splitted lists $beta_1$ and

$alpha_2$. During these modifications, it is ensured that the total amount of genotypes does not exceed the length of the amount of genotypes in the parents. If this is not done the genotypes in the suits could grow indefinitely, which is not preferred.

After the crossover modification, we mutate the newly created suites. Similar to EvoSuite we have two types of mutation on the suite level. The first type of mutation is the mutation of an individual test case with the probability of $1/T$ where T denotes the number of test cases in the suite. This mutation essentially triggers the mutation of every individual gene of the genotype, resulting in the negation of a boolean value, or an increase in the value of an integer. The other mutation regards the size of the suite. If the amount of genotypes in the suite is below a set maximum, there is a probability of 0.1 that a new genotype will be added. This is actually a loop, where there is a 0.1 chance that a single test case will be added. If that chance is taken there is a 0.1^2 chance that a second test case will be added to that suite and so forth. After these mutations, the suites are added to the new generation and the cycle starts again.

Algorithm 5 Genetic Fuzzer algorithm

```
Require: suites, max_generations
  procedure Gentic_Fuzz(suites, max_generations)
      calculate_fitness(suites)
      for generation in max_generations
          suites ← evolve(suites)
      end for
      fittest ← get_fittest_suite(suites)
      return(fittest)
  end procedure

  function evolve(suites)
      suite_list ← rank_by_fittness(suites)
      new_gen ← empty_list
      top1_suite ← pop(suite_list)
      top2_suite ← pop(suite_list)
      new_gen.add(top1_suite)
      target_suite ← top2_suite
      while target_suite is not null do
          (s1, s2) ← crossover(target_suite, top1_suite)
          (s1, s2) ← mutate(s1, s2)
          new_gen.add(s1)
          new_gen.add(s2)
          target_suite ← pop(suite_list)
      end while
      return(new_gen)
  end function
```

The population of suites is evolved six times whereafter the Genetic Fuzzer starts collecting data for the concolic executor. The choice for six generations was made based on observations that more generations, test suites, or test cases did not greatly increase the coverage but did have a noticeable effect on the execution time. The collection starts with the ordering of the suites based on rank. We then call the fittest suite and ask which input most recently covered new statements. This input is the input later used by the concolic executor to gather new paths. The genetic fuzzer also resets its complete population, except for the fittest suite. It then waits for the result of the concolic executor. When it receives the formulas containing new inputs it solves them all and adds all these inputs as new genotypes into the previously most fit suite. The Genetic Fuzzer then starts generating new suites to create the population for the first generation, adds the fittest suites with all the genotypes containing the inputs from the concolic executor and starts the process again.

5. Evaluation

The research questions we formulated in Section 1.1 were:

RQ 1) How does the introduction of Dynamic Symbolic Execution impact the speed, correctness and completeness of the Symbolic Execution Engine?

RQ 2) What is the impact of random and genetic fuzzing algorithms on the speed and correctness of Dynamic Symbolic Execution?

RQ 3) How does the OOX implementation of Dynamic Symbolic Execution compare to other state-of-the-art tools?

To answer the research questions we have performed several experiments. In these experiments, the tools and configurations under investigation are tasked to verify programs in three different data sets. These data sets are called: Deep-Set, MinePump and Jayhorn-Recursive. The MinePump and Jayhorn-Recursive sets originate from the software verification competition named SVComp. The programs in these sets were originally written in Java and were transpiled to OOX so our tool can verify them. The Deep-Set contains custom-made programs, some containing artificial bugs. We will further elaborate on the nature of the programs in these sets in a later section.

The output of the experiments will be used to draw comparisons. The first comparison will ascertain the influence of the fuzzer on DSE technique. We will investigate runs with the genetic fuzzer enabled and runs with the random fuzzer enabled. Using this comparison we can answer research question **RQ 2**. From these two configurations, the best-performing one will be used in a comparison with the symbolic execution approach of van Vliet [23]. This is done to ascertain the effect of DSE on the performance of the OOX ecosystem which will answer research question **RQ 1**. Lastly, a comparison will be made between the DSE technique and the top-performing tools of the SVComp 2022 competition, which are JDART, JBMC and Java-Ranger. This is done to ascertain the performance of DSE in relation to the state-of-the-art and will research question **RQ 3**. In this chapter, we will first elaborate on the programs in the data sets, after which we will analyze the proposed comparisons.

5.1 Data Sets

The data sets used for the experiments are chosen because of the nature of their bugs, which should preferably be deep, and the complexity of the programs. This follows from the hypothesis that the DSE technique should be able to find deeper bugs than the classical symbolic approach, which we can verify using these data sets. The choice

for the Jayhorn-Recursive and MinePump data sets follows from the observation that they are used by the software verification competition SVComp, which has stated that they provide data sets that can be used for comparisons of verification tools in an academic setting. Given this fact in addition to the fact that their programs are either complex, with several layers of recursion, or have deep bugs, in the MinePump set the bugs appear at the conclusion of the program, they are fit for our research purposes.

The custom data set, named Deep-Set, consists of an example program from the Driller paper [22] and two custom programs we created. All of these programs have a valid version, free of bugs and an invalid version with bugs. The bug in the Driller program only triggers if more than 80% of the elements in the array are a specific number. For these programs, the array itself is the input. This is a hard problem since it consists of a specific if-guard and requires a program to loop multiple times over the elements in the array. The custom programs we created follow the same principle but relax the difficulty of the guard and requires the program to loop three more times over the elements in the array which increases the overall depth. For all the programs in this set, it is required to walk the maximum depth of the program before a verdict can take place, which is the main difficulty of this set.

5.2 Fuzzer Comparison

In this first comparison, we aim to answer **RQ 2** by means of looking at the performance of the concolic execution with either the random fuzzer enabled or the genetic fuzzer enabled. In this experiment, we verify all the programs of the data set ten times for each configuration. The machine upon which the following experiments were run has an AMD Ryzen 7 2700X Eight-Core processor and 32 gigabytes of RAM. The metrics upon which we will base our comparisons are the branch coverage, the amount of concolic invocations, the amount of iterations and the percentage of how much of the program is discovered by the fuzzing algorithm. The coverage percentage and iteration count indicate how much of the program has been explored and how many iterations have been found. Using this metric we can gauge the completeness of the configuration. The concolic invocations metric indicates how many times the concolic executor has been called. This metric should preferably be as low as possible, a lower count indicating that our fuzzer can explore the program efficiently. This is an important performance metric since invoking the concolic executor costs a lot of computation time. In the same spirit, the metric of fuzzer percentage is used, which indicates how much of our discovered progression is due to the fuzzing algorithm. Preferably this percentage should be as high as possible, indicating that our fuzzing algorithm can discover unique program transitions without invoking the costly concolic executor. Lastly, the iteration count can be used to gauge the number of times the configuration iterated over previously discovered statements. Given two runs with similar coverage, the iteration count can be used to see which program iterated more over statements, with a higher count indicating a deeper exploration. It has to be noted that this compar-

ison is only relevant if two results have similar branch coverage since branch coverage is more important for progression than iterating over previously found transitions.

5.2.1 The Jayhorn Recursive Set

The overall average results over the Jayhorn-Recursive set are given in Table 5.1. We can see that the average time to verify a program is similar between the configurations, as is the branch coverage and percentage of transitions solely found by the fuzzing algorithm. Interestingly to note is the fact that while the percentage of transitions found by the fuzzing algorithm, denoted by the fuzzer% metric, is higher for the Genetic fuzzer, it also invoked the concolic executor more often. This can be seen while inspecting the c_invocs metric. We would expect this result to have a negative impact on both the execution time and fuzzer% metrics, but this is not the case for this set. This means that the genetic algorithm uses the inputs received from the concolic executor more efficiently. This efficient use stems from the genetic behaviour which mixes the provided inputs with random inputs to discover more transitions. In contrast, the random algorithm doesn't use the provided information to guide its search in any way but rather uses it as a checklist of paths that should be executed once. The metrics that show more significant differences are the completed paths, explored paths and deepest layer. While the explored paths metric is higher, these are not necessarily unique paths. This in combination with the fact that the genetic algorithm uses multiple test suites with different, possibly duplicate inputs explains the large difference. Lastly, the deepest layer metric denotes the deepest statement depth the verification has reached. We can see that the genetic algorithm covers more statements, in combination with the higher iteration count indicates that the Genetic fuzzer explores the programs under tests more thoroughly than the Random fuzzer does, albeit slightly.

Table 5.1: Average over Jayhorn Recursive Set

fuzzer	time	coverage	completed paths	paths explored	deepest_layer	c_invocs	iterations	fuzzer%
Genetic	38.17	89.64%	414.45	22706.15	1029.39	5.87	2201.49	58.75
Random	38.31	89.03%	53.45	3745.42	1018.10	4.21	2067.49	55.88

We can inspect the fuzzer% metric more in detail in figures 5.1 and 5.2. These figures describe the percentage of transition tuples found solely by the fuzzing algorithm after each concolic execution. The red line denotes the percentage of the random algorithm whereas the blue line denotes the percentage of the genetic algorithm. Figure 5.1 describes programs where the genetic algorithm found more transitions and Figure 5.2 programs where the random fuzzer found more transitions. In total, there are 10 program unique programs of which six clearly prefer the genetic algorithm and four prefer the random algorithm. The reason that these four programs prefer a random algorithm is because they are more favorable towards large varying inputs, which is what our RNG for the random fuzzer generates while the genetic fuzzer focuses more on incremental steps. The random fuzzer does have some downsides. While it does explore quickly in large steps it does not learn from the inputs it received from the

concolic executor whereas genetic does learn from this. This results in the four programs from Figure 5.2 having a lower overall branch coverage when using the random algorithm.

Figure 5.1: Comparison of what percentage of the transition coverage is caused by the fuzzing algorithm after each concolic invocation, showing the genetic fuzzer performance

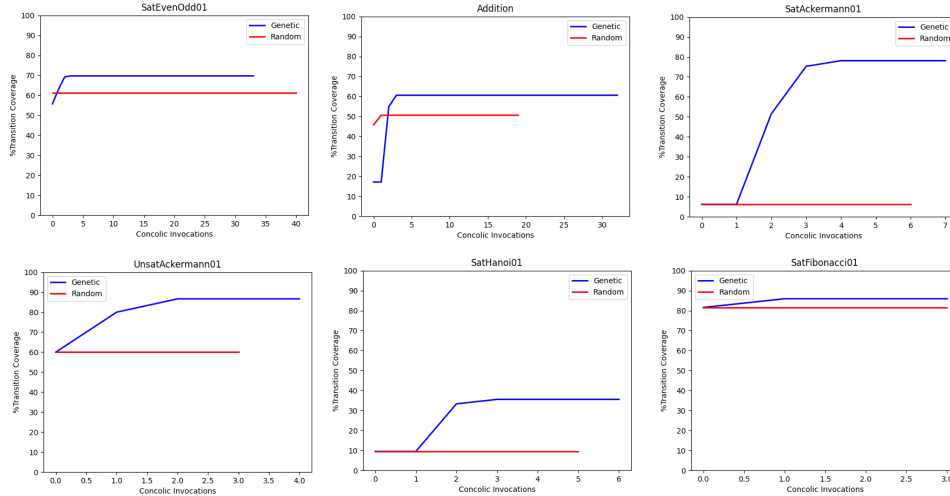
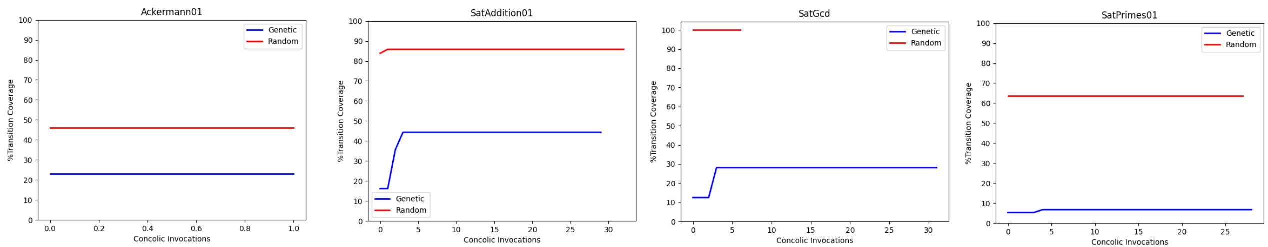


Figure 5.2: Comparison of what percentage of the transition coverage is caused by the fuzzing algorithm after each concolic invocation, showing the genetic random performance



5.2.2 The Deep Set

The overall average results over the Deep-Set are given in Table 5.2. Compared to the other sets it consists of fewer programs and the results of both configurations are very similar. The metrics that differ the most between the configurations are time, deepest statement layer reached and iteration count. While on average the genetic fuzzer is about three seconds slower, it does cover more statements (denoted by the deepest_-layer metric) and has significantly more iterations, indicating that while it is slower it is more thorough than the random fuzzer. This is consistent with the observations made in the Jayhorn-Recursive set. The reason the performance of the fuzzers is similar follows from the fact that most bugs in this set are discovered by the concolic executor. This fact is supported by the concolic invocation count, which is similar between the

two configurations. The reason that the genetic fuzzer does cover more statements and has more iterations is due to the test suites it generates, giving it a slight edge over the random fuzzer.

Table 5.2: Average over Deep Set

fuzzer	time	coverage	completed paths	paths explored	deepest_layer	c_invocs	iterations	fuzzer%
Genetic	66.62	97.27%	13667.44	59413.66	413.82	8.10	931.80	77.96
Random	63.71	97.29%	1799.32	8644.04	412.40	8.04	855.86	78.61

To further support this observation the ten-run average of each program using each configuration is given in Table 5.3. Here we can see the similar performance between the configurations and notice that for the programs that take longer with the genetic fuzzer, like SatDeep01, the genetic fuzzer does cover deeper statements and more iterations. The only outlier is the program "UnsatDriller". The reason the random fuzzer performed better on average is because it had ten similar seeded runs. The genetic fuzzer had one run that started with a bad seed, resulting in a worse average performance compared to the random fuzzer. From this observation, we can remark that the initial seed of the fuzzer can play a crucial role in its performance.

Table 5.3: Concolic Random vs Concolic Genetic Deep Set, 10 run average with a depth of 1500

program	fuzzer	time	verdict	coverage	completed paths	paths explored	error_layer	deepest_layer	c_invocs	iterations	fuzzer%
SatDeep01	Genetic	73.85	VALID	98.40%	13890.40	38674.60	0.00	399.00	3.70	602.80	99.08
SatDeep01	Random	63.76	VALID	98.40%	1961.00	6061.20	0.00	396.60	3.10	587.80	98.46
SatDriller	Genetic	120.08	VALID	94.70%	20849.30	101893.30	0.00	448.00	14.80	631.80	97.30
SatDriller	Random	120.09	VALID	94.70%	2640.20	14471.70	0.00	448.00	15.10	632.10	97.30
UnsatDeep01	Genetic	0.08	INVALID	97.82%	76.20	272.70	375.80	377.90	0.00	1031.60	0.00
UnsatDeep01	Random	0.07	INVALID	97.66%	57.30	215.10	375.40	376.60	0.00	639.40	0.00
UnsatDeep02	Genetic	20.54	INVALID	98.30%	3901.90	10907.90	390.20	396.20	1.00	1177.00	98.44
UnsatDeep02	Random	20.00	INVALID	98.30%	655.50	2083.20	385.80	392.80	1.00	1144.00	100.00
UnsatDriller	Genetic	118.53	INVALID	97.13%	29619.40	145319.80	400.50	448.00	21.00	1215.80	94.99
UnsatDriller	Random	114.62	INVALID	97.40%	3682.60	20389.00	445.00	448.00	21.00	1276.00	97.30

5.2.3 The MinePump Set

The overall average results over the MinePump set are given in Table 5.4. Interestingly, all of the bugs in the invalid programs have been found with zero concolic invocations. This means that any type of fuzzer is sufficient for finding the bugs in this set. Despite this fact, the same trend between the two configurations observed in the Deep-Set and Jayhorn-Recursive set continues. That is, the performance between the two configurations is similar, where the genetic fuzzer is slightly slower but does cover deeper statements and more iterations.

Table 5.4: Average over MinePump Set

fuzzer	time	coverage	completed paths	paths explored	deepest_layer	c_invocs	iterations	fuzzer%
Genetic	7.80	76.28%	700.45	8257.16	872.28	0.63	2278.63	19.35
Random	6.76	75.99%	161.51	2000.76	837.37	0.55	1777.78	19.33

5.2.4 Fuzzer Comparison Conclusion

RQ 2 states that we need to look at the impact of the random and genetic fuzzer on the speed and correctness of DSE. The two proposed DSE configurations, concolic execution with a genetic fuzzer and concolic execution with a random fuzzer, have similar results when looking at the averages over each data set. However, this does not necessarily mean that one configuration is always better than the other. To achieve an optimal performance regarding speed and correctness one would need to examine which properties are preferential and choose the configuration accordingly. If the program under test benefits from large random numbers, either positive or negative, the random fuzzer would be ideal. If the program under test benefits from incremental search the genetic fuzzer would be the better option. Having said that, it must be stated that while on average the genetic fuzzer is slightly slower it also covers more statements, more iterations and has a slightly larger branch coverage which increases its correctness when the program under test is valid. Because of this observation, the use of concolic execution with the genetic fuzzer is preferred as the default configuration. Additional comparison tables containing per-program comparisons can be found in Appendix E.

To answer **RQ 2**, on average both configurations have a similar execution speed but the genetic fuzzer can be regarded as more correct since it has the higher coverage regarding branches, found transitions and iteration count.

5.3 Concolic to Symbolic Comparison

In this section, we will try to answer **RQ 1** by making a comparison between the DSE technique configured with a genetic fuzzer and two existing heuristics in the OOX ecosystem. One of the existing heuristics is MD2U (Minimal Distance To Uncovered), proposed and implemented by van Vliet. This heuristic computes a distance for all statements in each method. This distance either denotes the distance to the first uncovered statement or the closest distance to the end of the method. It first prioritizes exploration of the statements within the method, after which it prioritizes leaving the method as soon as possible. In his work, he compared this heuristic to the classic Depth-First heuristic, which prioritizes the verification of deeper paths. This heuristic uses the amount of consecutive covered statements as a depth measurement. In his work, he concluded that the MD2U heuristic performed worse than Depth-First and hypothesized that this is due to the performance cost of the distance calculations. He stated that in larger programs it might perform better since the exploration would benefit from the distance calculations and the cache optimizations would outperform a Depth-First heuristic. Because this work focuses on larger programs with deeper bugs than the work of van Vliet we will compare our DSE implementation with both the MD2U and Depth-First heuristic.

5.3.1 The Jayhorn Recursive Set

The hypothesis that MD2U would perform better in larger or more complex programs can be substantiated by its results on the Jayhorn-Recursive set. From the seven invalid programs it correctly invalidates five of them, whereas the Depth-First heuristic only correctly invalidates three. Both heuristics have difficulties correctly validating the valid programs since they were able to only correctly validate two of the thirteen valid programs.

The DSE technique is able to correctly validate and invalidate every program of the Jayhorn-Recursive set. It must be noted that for the valid programs, the DSE technique has more similarities to testing than symbolical verification. While the DSE technique correctly validates programs without bugs, this verdict is the result of its inability to progress the exploration of the program. When the technique is not able to progress any further it assumes that it inspected every reachable statement and returns a valid verdict, though this assumption is not always true. This is in contrast with symbolical verification whose verdict holds more weight since it guarantees the exploration of the whole program. That being said, the valid verdict both symbolic approaches give occurs due to a time limit being reached. If within this time budget no violations have been found the symbolic approach reports a valid verdict, meaning that these verdicts do not correspond with a complete exploration of the program. This in turn makes the verdicts more comparable to the verdicts of the DSE technique.

Table 5.5 describes the average result of each heuristic on the Jayhorn-Recursive set. Besides correctly validating and invalidating programs we can see that the DSE technique also does this quicker and with higher branch coverage than the other techniques. While DSE does not reach statements as deep as the Depth-First heuristic, it does achieve a higher branch coverage. Tables containing per-program results for MD2U, Depth-First and Concolic Execution with a genetic fuzzer can be found in Appendix A, B and C respectively.

Table 5.5: Comparision of average result of heuristics on the Jayhorn Recursive set

heuristic	time	coverage	deepest_layer
ConcolicExecution	38.17	89.64%	1029.39
DepthFirstSearch	90.81	86.34%	1311.90
MinDist2Uncovered	99.49	89.23%	686.02

Lastly, a remark about the execution speed of the symbolic heuristics in the Jayhorn-Recursive set. In Table 5.6 we averaged the performance of each heuristic on the programs they correctly invalidated, leaving out all the programs they incorrectly validated. This table shows that if the symbolic approaches correctly invalidated a program they did so very fast while the DSE technique takes more time, but finds all the bugs. A probable reason for these low execution times can be extracted from the average deepest statement reached. The fastest heuristic, MD2U, shows an average

deepest layer of ten. This means that the bugs in these programs were shallow, which also explains why the Depth-First has on average a higher execution time.

Table 5.6: Comparision of heuristics averages on their correctly invalidated Jayhorn Recursive programs

heuristic	time	coverage	deepest_layer
ConcolicExecution	23.89	83.88%	946.37
DepthFirstSearch	0.07	88.70%	503.67
MinDist2Uncovered	0.03	78.71%	10.14

5.3.2 The Deep Set

The Deep-Set contains two valid programs and three invalid programs. Furthermore, the bugs in this set can be found around a statement depth of 400, given the right constraints on the inputs. The MD2U heuristic was not able to correctly validate or invalidate any program in this set. This can be explained by the statement depth the heuristic reached, which is only 334 and thus too low to encounter the bugs. The Depth-First heuristic is better able to verify these types of programs since it prioritizes the exploration of the deepest path. While Depth-First is not able to validate the correct programs within the time budget of two minutes it does find the bugs in two out of the three programs. The average results per heuristic on this data set, shown in Table 5.7, support the observations previously made in the Jayhorn-Recursive set. The DSE technique correctly validates and invalidates all the programs using less execution time and achieves a higher branch coverage and statement depth.

Table 5.7: Comparision of heuristics on the Deep set

heuristic	time	coverage	deepest_layer
ConcolicExecution	75.58	90.42%	594.85
DepthFirstSearch	97.27	83.19%	570.83
MinDist2Uncovered	138.66	67.22%	237.15

5.3.3 The MinePump Set

The programs in the MinePump set are derived from a singular program of which there are 25 invalid variations and six valid variations. The bugs in these programs appear around a statement depth of roughly 350. The MD2U heuristic achieves a maximum statement depth of around 300, meaning it is unable to find any bugs. This is similar to its performance on the Deep-Set. In addition to this, the heuristic is also not able to completely verify the valid programs, reaching the time limit of 2 minutes on every single run for every program in this set. For this particular set the MD2U heuristic is not fit to symbolically verify any program.

The Depth-First heuristic does perform well. It is able to correctly validate and invalidate the programs within the allotted time budget of two minutes, meaning that the reportedly valid programs are symbolically verified in a complete sense. This is an interesting observation because one would not expect such a large difference between two symbolic approaches using different heuristics. This does point towards the calculation performed by the MD2U heuristic to be very costly, not bringing any benefit in either the exploration or computation time of these programs.

The DSE technique is able to correctly validate and invalidate the programs in this set. Its execution time for the verification of invalid programs is about 60 times faster than the Depth-First heuristic, as can be seen in Table 5.8. The reason for this lies in the nature of the bugs in the MinePump set. All of these bugs are able to be found with any kind of fuzzer. This results in the concolic executor not being invoked during the verification of these invalid programs. For valid programs, the DSE technique takes more than twice as long compared to the Depth-First technique. This is shown in Table 5.9. As previously noted, the valid verdict of the DSE technique is not as strong as a valid verdict from symbolic verification. However, when looking at the branch coverage in Table 5.8 we can see that the DSE technique does achieve the exact same branch coverage as the symbolic technique. This shows that the valid verdict of the DSE technique does have some merit, but it must be noted that its completed paths and explored paths are much lower than the Depth-First heuristic. This indicates that the statement coverage of the Depth-First heuristic is much higher and thus its valid verdict holds more weight.

Table 5.8: Comparision of heuristics averages on the valid MinePump programs

heuristic	time	coverage	completed paths	paths explored	deepest_layer
ConcolicExecution	39.52	89.17%	3513.58	41486.30	1493.18
DepthFirstSearch	15.90	89.17%	9822.33	626429.67	1500.00
MinDist2Uncovered	122.25	72.63%	2097.83	14740.43	304.30

Table 5.9: Comparision of heuristics averages on the invalid MinePump programs

heuristic	time	coverage	completed paths	paths explored	deepest_layer
ConcolicExecution	0.19	73.19%	25.30	282.16	723.27
DepthFirstSearch	6.89	85.89%	517.44	34701.40	1394.72
MinDist2Uncovered	122.05	74.90%	1999.90	15510.85	306.64

5.3.4 Concolic vs Symbolic Conclusion

RQ 1 states that we need to look at the speed, correctness and completeness of all the tools we compared in this section. When looking at the results of MD2U we can state that this heuristic is not preferable for these data sets. While it did perform well on the complex programs in the Jayhorn-Recursive set it was unable to verify any deeper bugs. Even in the Jayhorn-Recursive set, it discovered mainly shallow bugs.

The Depth-First heuristic performed well overall but could not verify the valid programs of the Jayhorn-Recursive and Deep sets completely, even showing less branch coverage than the DSE technique. In this regard the DSE technique is more reliable for finding bugs and its valid verdicts arguably hold the same or more weight for this particular set. However, this is not the case for the MinePump set where Depth-First was able to completely verify all the valid programs and find all the bugs. That being said the performance for actually finding bugs, especially in terms of execution time, was worse compared to the DSE technique.

To answer **RQ 1**, for finding deep bugs in complex programs the DSE technique was shown to be a reliable alternative to the symbolic approaches. Even when returning a valid verdict it often had a comparable or higher branch coverage than the symbolic techniques. While it cannot provide the same guarantees as symbolic verification regarding completeness it has shown to be a valuable alternative to this technique especially regarding execution time and correctness.

5.4 Concolic to State-Of-The-Art Comparison

To answer **RQ 3** we will make one last comparison between the DSE technique and the state-of-the-art tools JDart, JBMC and Java-Ranger. These tools were chosen for comparison since they scored first, second and third respectively in the "JavaOverall" category of the SVComp software verification competition held in 2022. The datasets used in this work are a subset of the datasets used during this competition. The verification tools executed these data sets using the BenchExec benchmarking tool. To keep the experiments as similar as possible we created a BenchExec configuration for our DSE verification tool with the genetic fuzzer configuration. We then executed all the data sets in this section using the BenchExec tool. To spare computation time we will compare these results to the reported results of the state-of-the-art tools posted by SV-Comp. Our custom data set was also added to BenchExec and all the state-of-the-art tools executed this custom data via BenchExec. In this section, we will first briefly describe the tools and the novel concepts they employ for verification after which we will inspect the results of each data set in more detail.

5.4.1 SVComp Tools

JDart is a tool which is implemented as an extension of the Java Pathfinder framework and employs dynamic symbolic verification to reach its verdict. It uses an executor that executes the program while tracking constraints and an explorer that determines the search strategy [12]. Of the three tools, JDart is the most similar to our DSE implementation, the difference being that DSE has a separate concrete and concolic executor while the executor of JDart performs both tasks simultaneously. The explorer of JDart is comparable with our fuzzer. In DSE the fuzzer is responsible for the exploration while JDart uses undefined configurations which steer the path in directions these con-

figurations deems interesting.

JBMC stands for Java Bounded Model Checking and is based on CMBC. As the name implies it uses a bounded model checker to verify the program. After analyzing the Java bytecode the program under test is parsed to a large bounded formula and passed to a solver. From this result, it determines if the program is valid or not [28].

Java Ranger is a symbolic execution tool whose novelty lies in a technique called veritesting. This technique enables the tool to merge multiple paths, mitigating the path explosion problem. It further recognizes the small dynamically dispatched methods inside Java programs and employs a technique called dynamic inlining to ensure these methods are handled correctly while merging paths [8].

5.4.2 The Jayhorn Recursive Set

The programs in the Jayhorn Recursive Set proved to be difficult to verify for the tools. Of the 19 programs, the JBMC tool reported an Out Of Memory (OOM) verdict eight times and an Error or Unknown verdict another five times resulting in the verification of only 6 of the 19 programs. Of the 7 invalid programs in this set JBMC correctly reported an Invalid verdict on six of them, reporting an error on the seventh. This indicates that for these types of programs, the JBMC struggles to completely verify valid programs, either running out of memory or reporting an error.

The JDart tool reported a TimeOut verdict for 10 of the 19 programs in this set. On top of this, it also took more than 120 seconds on three other programs. Of the seven invalid programs it correctly found the bugs in six of them. This finding supports the previous observation that verifying the valid programs in this set is a difficult task.

The JavaRanger tool reported an Unknown verdict for 10 of the 19 programs in this set. Furthermore, on four occasions it reported a Valid verdict when it was close to completely spending its time budget. It must be stated that these programs were indeed valid and thus the verdict was correct. However, a more skeptical view could be that the tool reports a valid finding when it is close to reaching its time budget. Since we have no access to the source code we cannot state this as a fact but more research could be done to ascertain if this is actually the case or not. Of the seven invalid programs it correctly found the violations in only four of them, which is less than JDart and JBMC.

Running our DSE implementation with BenchExec didn't change much in the results. We were still able to correctly validate and invalidate all the programs in the set. Table 5.10 shows the average execution time for each tool on the set. Here we can see that DSE is much faster than the other tools. This is due to the reported time-outs and unknown verdicts of the other verification tools.

Table 5.10: Average Comparison Concolic, JDart, JBMC, Java-Ranger on the Jayhorn Recursive set

heuristic	time
DSE	33.20
JBMC	470.54
JDart	478.07
JavaRanger	572.44

While DSE is more reliable than the other techniques, in individual cases, it can be significantly slower. In Table 5.11 we show two invalid programs of the Jahorn Recursive set, namely UnsatEvenOdd01 and UnsatAckermann01. In the case of UnsatEvenOdd01 we can see that DSE takes much longer than the other tools, while in the case of UnsatAckermann01 it is only surpassed by JBMC since the other tools are unable to correctly invalidate the program.

Table 5.11: Comparison DSE, JDart, JBMC, Java-Ranger on two programs of the Jayhorn Recursive set

program	tool	time	verdict
UnsatEvenOdd01	DSE	35.78	INVALID
UnsatEvenOdd01	JBMC	1.10	INVALID
UnsatEvenOdd01	JDart	3.70	INVALID
UnsatEvenOdd01	JavaRanger	8.20	INVALID
UnsatAckermann01	DSE	23.90	INVALID
UnsatAckermann01	JBMC	4.80	INVALID
UnsatAckermann01	JDart	900.00	TIMEOUT
UnsatAckermann01	JavaRanger	800.00	UNKOWN

5.4.3 The Deep Set

The per-program results of the verification tools on the deep set are given in Table 5.12. For these programs, the verification tools need to reach a statement depth of c.a. 400 with the correct constraints to find the bugs. It consists of multiple while loops iteration over an array of 20 elements, pushing the verification tools to their limit. Starting with the Java Ranger tool we can see that it can only correctly verify one program, SatDriller. In the comparison of the Jayhorn Recursive set we speculated that this verification tool could report a valid verdict when its execution time nears the time limit. We can see a similar effect occurring here. While the tool reported a correct "Valid" verdict on the SatDriller program, it reported an incorrect "Valid" verdict on the UnsatDriller program. Both executions exceeded the 800 seconds mark, indicating that our speculation may be right and Java Ranger indeed reports a "Valid" verdict when nearing its time limit.

Of the 5 programs, the JDart tool reported a "TimeOut" verdict on three programs,

of which two were valid and one was invalid. Noticeably the other two invalid programs were correctly reported as "Invalid" within 5 seconds. The JBMC tool was the best performing tool on this data set, correctly validating and invalidating every program of this set within 4 seconds. This is quite surprising since all other symbolic tools under investigation, (Java Ranger and OOX with the depth-first and MD2U heuristic) found this task difficult. Its performance is the result of the bounded model checking technique and proves, in this set, to be the best performing technique. Regarding our DSE tool, it remains reliable, able to correctly validate and invalidate all the programs in this set albeit with a larger computation time. This is due to the fact that the computationally costly concolic executor is invoked quite often during the verification of these programs.

Table 5.12: JDart, Java-Ranger and JBMC results on Deep Set, 5 run average

program	tool	time	verdict
SatDeep01	DSE	56.51	VALID
SatDeep01	JDart	901.70	TIMEOUT
SatDeep01	JBMC	2.76	VALID
SatDeep01	JavaRanger	8.60	UNKOWN
SatDriller	DSE	91.91	VALID
SatDriller	JDart	902.19	TIMEOUT
SatDriller	JBMC	3.60	VALID
SatDriller	JavaRanger	829.43	VALID
UnsatDeep01	DSE	0.05	INVALID
UnsatDeep01	JDart	4.01	INVALID
UnsatDeep01	JBMC	2.50	INVALID
UnsatDeep01	JavaRanger	7.90	UNKOWN
UnsatDeep02	DSE	9.56	INVALID
UnsatDeep02	JDart	4.48	INVALID
UnsatDeep02	JBMC	2.50	INVALID
UnsatDeep02	JavaRanger	7.84	UNKOWN
UnsatDriller	DSE	86.10	INVALID
UnsatDriller	JDart	902.10	TIMEOUT
UnsatDriller	JBMC	3.34	INVALID
UnsatDriller	JavaRanger	829.06	VALID

5.4.4 The MinePump Set

The MinePump set was chosen because the violations inside the programs occur at a statement depth of c.a. 500. We also previously stated that while this depth might be considered deep the programs are not complex. This claim is substantiated by the performance of all the verification tools on this set. Every tool was able to correctly validate and invalidate every program in this set. Since this is the case we split this analysis based on the validity of the programs, first discussing the performance of the verification tools on the invalid programs in the set after which we discuss the performance of the tools on the valid programs of the set.

The average performance of the verification tools over the invalid programs in the set is given in Table 5.13. Here we can clearly see that the DSE technique is the quickest

in finding all the bugs of the program. As previously mentioned this is because every bug is found solely by the fuzzer, with no invocation to the costly concolic executor. This finding indicates that first invoking a fuzzer when validating a program before using symbolic verification results in large computational gains.

Table 5.13: Average Comparison Concolic, JDart, JBMC, Java-Ranger on Invalid Minepump Programs

tool	time
DSE	0.15
JBMC	2.41
JDart	4.13
JavaRanger	10.06

The average performance of the verification tools over the invalid programs in the set is given in Table 5.14. Here we can see that JBMC completely verifies the program the quickest. This in conjunction with the fact that it was the best performing state-of-the-art tool in the invalid set makes this tool the overall best performing verification tool of this set. JDart comes in second, taking on average 24 seconds to completely verify a program. Regarding the Java Ranger tool, while it was quick to find the bugs in the invalid programs, it takes on average more than 10 times longer than JDart to verify valid programs. Comparing the results of these tools to the DSE technique we can see that two of them take less computation time, making DSE not needed for these programs. One could argue that the DSE technique outperforms the Java Ranger tool, but Java Ranger uses symbolical verification. As discussed previously, a valid verdict from a symbolic verification tool is more complete than a valid verdict from the DSE technique since DSE is not able to verify a program in a complete sense. Because of this fact, compared to the state-of-the-art tools the DSE technique falls short on this specific set and one should preferably use one of the other verification tools.

Table 5.14: Average Comparison Concolic, JDart, JBMC, Java-Ranger on Valid Minepump Programs

tool	time
DSE	33.05
JBMC	3.17
JDart	24.33
JavaRanger	325.00

5.4.5 State-Of-The-Art Conclusion

In this comparison, we have highlighted the strengths and weaknesses of the tools under investigation. To answer **RQ 3** we will first summarize their strengths, then continue with their limitations while comparing their results to DSE. During the analysis,

it became evident that for less complex programs and shallow bugs, these tools are superior to DSE. Their guarantees are much stronger and their execution times are much lower. A clear example of this was the MinePump set, in which these tools excelled. When these tools were tasked with the verification of the deep set certain problems arose. Only the JBMC tool confidently reported its findings and averaged a lower execution time than DSE. At this point, a case could still be made that optimizations to these techniques could push the performance of these tools to verify more complex programs with deeper bugs, albeit with the Bounded Model Checking approach rather than the techniques used in JDart or Java Ranger. Lastly, when the tools were tasked with the verification of the Jayhorn Recursive set we encountered the limits of these techniques. The complete verification of valid programs was no guarantee, each tool at different programs either spent its entire time budget, reported an error, or reported an unknown verdict. While these problems occurred less for the invalid programs in this set, the tools were still not able to verify around 30% of the programs. The DSE technique outperformed them not only on execution time but maybe more importantly on reliability. A special case should be made for JBMC. While it was not able to verify a large portion of the valid programs, it correctly invalidated all programs except for one, with a far smaller execution time. Concluding, in comparison with the state-of-the-art the DSE technique certainly has shown to be reliable and correct enough to compete with them on complex programs and programs with deeper bugs. Its average execution time is lower than the state-of-the-art tools, but higher if we only average the results of the correctly invalidated programs.

6. Conclusion

In this work, we have discussed several techniques that can be used to mitigate the explosions that take place in symbolic verification. The related work section describes these techniques and the tools that implement them in detail. We continued with a description of the current status of the OOX ecosystem, describing which optimizations and techniques were added by the researchers who previously worked on this system. After describing the basis upon which we would continue our research we gave an overview of the Dynamic Symbolic Execution technique. We started by describing the general principles of this technique after which we explained our interpretation of this technique. This included the algorithms we have implemented for the concolic executor as well as the two different fuzzers that control the concrete executor.

We evaluated our DSE implementation with several analyses. First, we looked at the impact of the fuzzer on DSE to ascertain which configuration performed best. The results showed a similar performance when looking at the average execution time of both configurations. Furthermore, there was a slight difference in the explored depth between the configurations. The genetic fuzzer reached a deeper statement depth, had more iterations and found more progress with solely the fuzzer. It was also observed that the branch coverage was slightly higher with the genetic fuzzer enabled. Because of the similar execution speed and similar correctness, we chose to continue our experiments with the genetic fuzzer since its higher branch coverage indicated it was more complete than the random fuzzer.

After this analysis, we took the DSE configuration with the genetic fuzzer and compared its performance against the performance of the MD2U and Depth-First heuristic to ascertain the impact of DSE on the OOX ecosystem in accordance with research question **R1**. Here we observed that the MD2U heuristic performed best on complex programs with shallow bugs and the Depth-First heuristic performed best on programs where the bug occurs in deeper sections. However, compared to DSE both these heuristics performed worse. Neither heuristic was able to verify all the programs, reaching the set time limit in several cases. Furthermore, for two data sets, we showed that the heuristics were not able to completely verify valid programs. Regarding these valid programs, DSE showed a higher branch coverage using less execution time than both heuristics. From this, we can conclude that while the DSE is not able to symbolically verify programs, its valid verdicts are strong considering the branch coverage. In the MinePump set we saw that Depth-First was able to correctly validate and invalidate every program. For this specific data set, we must conclude that DSE is not preferable. While it was significantly quicker in finding violations, it was slower in validating cor-

rect programs. It also had a similar branch coverage to the symbolic approaches, but less path coverage, making the valid verdict of DSE holding less weight. In conclusion DSE was superior regarding execution speed and correctness, but is not as complete as symbolic approaches.

Lastly, we compared our DSE implementation to three state-of-the-art tools whose performance placed them in the top three of the SVComp software verification competition of 2022. Using this comparison we could ascertain how our DSE implementation compared against other tools. We showed that our DSE implementation was competitive with these tools, being more reliable and on average faster than two of them. Besides the average performances we also highlighted individual cases, continuing the trend that while a program may perform the best on average, this does not mean it performs the best on every single program. We also discussed the surprising performance of the JBMC tool, which on occasion was leagues ahead of the competition regarding execution times. However, we could not find a satisfying explanation for this performance, ultimately regarding this performance gain as an inherent benefit of the novel technique employed by the JBMC tool.

Future Work

This work gives a lot of opportunities for future work since it consists of several concepts which individually can be explored further. We will list these concepts and research ideas for future work below.

Optimizations for the concolic executor. During the implementation of the concolic executor we found a possibility to optimize the search for inputs. Currently, every assumption the executor encounters is checked by the SMT solver for interesting inputs. However, not every assume statement leads to an interesting input. Stripping all of the unimportant invocations of the SMT solver would optimize the DSE implementation. This also applies to any other optimizations of the SEE.

Investigating and optimizing fuzzer techniques. As shown in our evaluation the fuzzer used by DSE is essential for the efficient discovery of statement transitions. Besides our proposed random and genetic fuzzer other options might work even better. The genetic fuzzer itself could also be researched further. Research regarding the mutation and optimization of the test suite may improve its performance further.

Adding symbolic inputs to the fuzzer. Currently, the inputs received from the concolic executor are in the form of a formula like "$x > 10$". This is solved with an SMT solver to get a single satisfiable input. However, the fact that the inputs are reported in formula form can be exploited further, generating a range of inputs that satisfy the formula instead of a single one.

Investigating the JBMC tool. During the evaluation of the state-of-the-art tools the JBMC tool showed impressive performance compared to the competition. Further research into this phenomenon is needed to asses if the OOX Ecosystem could benefit from the introduction of this kind of bounded model checking.

Appendices

A. MinDist Results

Table A.1: MinDist Heuristic on Deep Set, 10 run average with a depth of 1500

name	time	result	coverage	completed paths	paths explored	error_layer	deepest_layer
SatDeep01	147.25	VALID	68.90	12860.50	21040.70	0.00	189.10
SatDriller	125.56	VALID	92.10	7792.30	21485.90	0.00	337.80
UnsatDeep01	144.14	VALID	70.00	12394.70	21520.90	0.00	185.30
UnsatDeep02	146.02	VALID	70.00	14962.90	24886.50	0.00	185.80
UnsatDeep03	143.04	VALID	10.20	10862.50	18881.80	0.00	190.80
UnsatDriller	125.98	VALID	92.10	6941.10	19152.30	0.00	334.10

Table A.2: MinDist Heuristic on Jayhorn Set, 10 run average with a depth of 1500

name	time	result	coverage	completed paths	paths explored	error_layer	deepest_layer
Addition	152.57	VALID	94.10	10229.40	10228.40	0.00	141.70
SatAckermann01	141.82	VALID	85.70	6256.70	12241.30	0.00	1114.90
SatAckermann02	142.63	VALID	85.70	7135.90	13976.80	0.00	1189.50
SatAckermann03	143.03	VALID	85.70	7385.60	14471.60	0.00	1202.00
SatAddition01	156.26	VALID	94.60	13265.90	13264.90	0.00	117.10
SatEvenOdd01	31.19	VALID	97.90	925.00	1350.00	0.00	1500.00
SatFibonacci01	164.31	VALID	97.03	10937.80	10936.80	0.00	1500.00
SatFibonacci02	0.03	VALID	96.20	1.00	198.00	0.00	727.00
SatFibonacci03	162.50	VALID	98.35	9048.00	9047.00	0.00	1500.00
SatGcd	145.31	VALID	95.70	8506.80	8505.80	0.00	198.30
SatHanoi01	129.01	VALID	96.60	6995.30	13766.00	0.00	1500.00
SatMccarthy91	175.76	VALID	95.66	9748.10	9747.10	0.00	1500.00
SatPrimes01	146.52	VALID	91.56	12367.30	12367.80	0.00	127.50
Ackermann01	0.04	INVALID	71.91	1.00	4.40	16.00	16.90
InfiniteLoop	0.03	INVALID	88.90	0.00	1.00	6.00	6.00
UnsatAckermann01	143.66	VALID	84.60	7757.20	15205.90	0.00	1241.20
UnsatAddition01	154.98	VALID	91.70	13414.50	13413.50	0.00	110.60
UnsatAddition02	0.01	INVALID	85.70	0.00	0.00	5.00	5.00
UnsatEvenOdd01	0.03	INVALID	77.65	1.00	6.30	16.20	16.80
UnsatMccarthy91	0.02	INVALID	69.37	0.00	1.00	6.00	6.00

Table A.3: MinDist Heuristic on MinePump Set, 10 run average with a depth of 1500

name	time	result	coverage	completed paths	paths explored	error_layer	deepest_layer
spec1-5_product59	122.82	VALID	68.16	2227.90	13310.40	0.00	291.50
spec1-5_product60	122.51	VALID	71.57	2465.30	16053.20	0.00	302.20
spec1-5_product61	121.77	VALID	75.04	1517.40	13648.50	0.00	316.90
spec1-5_product62	121.73	VALID	75.17	1526.30	13766.50	0.00	311.50
spec1-5_product63	122.35	VALID	71.74	2398.80	15523.70	0.00	296.50
spec1-5_product64	122.32	VALID	74.12	2451.30	16140.30	0.00	307.20
spec1-5_product1	121.66	VALID	76.52	1558.20	15013.80	0.00	307.80
spec1-5_product10	121.37	VALID	78.60	1610.50	17296.60	0.00	327.50
spec1-5_product11	122.25	VALID	76.19	2616.00	17717.00	0.00	299.10
spec1-5_product12	123.11	VALID	75.48	2231.80	13620.90	0.00	293.30
spec1-5_product13	121.87	VALID	73.97	1482.40	13137.00	0.00	298.40
spec1-5_product14	121.87	VALID	74.38	1505.80	13702.70	0.00	301.50
spec1-5_product15	122.53	VALID	72.48	2531.80	16217.90	0.00	297.10
spec1-5_product16	122.49	VALID	74.22	2501.30	16303.20	0.00	297.90
spec1-5_product2	121.49	VALID	78.27	1603.90	16530.60	0.00	316.20
spec1-5_product3	122.80	VALID	75.07	2427.20	15288.50	0.00	294.10
spec1-5_product4	122.83	VALID	74.06	2376.80	15028.20	0.00	298.60
spec1-5_product48	121.93	VALID	72.11	1478.50	12251.90	0.00	302.10
spec1-5_product49	121.99	VALID	71.45	1492.20	12814.90	0.00	310.50
spec1-5_product5	121.39	VALID	75.25	1594.80	16463.80	0.00	312.10
spec1-5_product50	121.43	VALID	74.32	1591.60	15486.10	0.00	317.10
spec1-5_product51	122.35	VALID	70.11	2557.50	16614.60	0.00	297.20
spec1-5_product52	122.33	VALID	72.73	2505.50	16459.50	0.00	305.60
spec1-5_product53	121.42	VALID	76.09	1569.10	15208.10	0.00	316.20
spec1-5_product54	121.50	VALID	76.10	1559.70	14984.30	0.00	320.40
spec1-5_product55	122.32	VALID	74.40	2547.70	16659.80	0.00	304.60
spec1-5_product56	122.74	VALID	75.49	2319.60	14961.60	0.00	309.50
spec1-5_product6	121.57	VALID	76.19	1536.40	14459.50	0.00	316.50
spec1-5_product7	122.28	VALID	74.59	2610.00	17367.20	0.00	297.60
spec1-5_product8	122.24	VALID	75.21	2571.10	17239.50	0.00	303.40
spec1-5_product9	121.44	VALID	79.18	1618.00	16944.00	0.00	321.70

B. Depth-First Results

Table B.1: Depth-First Heuristic on Deep Set, 10 run average with a depth of 1500

name	time	result	coverage	completed paths	paths explored	error_layer	deepest_layer
SatDeep01.oox	120.04	VALID	90.20	14869.60	29722.90	0.00	347.00
SatDriller.oox	120.02	VALID	94.70	3257926.00	11402608.90	0.00	448.00
UnsatDeep01.oox	120.03	VALID	90.00	15985.20	31954.40	0.00	336.00
UnsatDeep02.oox	0.05	INVALID	76.70	1.00	116.00	345.00	346.00
UnsatDriller.oox	103.32	INVALID	97.13	4174992.30	14612543.70	400.50	448.00

Table B.2: Depth-First Heuristic on Jayhorn Recursive Set, 10 run average with a depth of 1500

name	time	result	coverage	completed paths	paths explored	error_layer	deepest_layer
Addition.oox	120.05	VALID	88.20	284536.60	284535.60	0.00	1500.00
SatAckermann01.oox	120.02	VALID	92.90	4562661.30	6756293.60	0.00	1500.00
SatAckermann02.oox	120.02	VALID	92.90	4870375.50	7213899.90	0.00	1500.00
SatAckermann03.oox	120.02	VALID	92.90	5203290.30	7702983.50	0.00	1500.00
SatAddition01.oox	120.03	VALID	75.70	423257.90	423256.90	0.00	1500.00
SatEvenOdd01.oox	15.54	VALID	97.90	925.00	1350.00	0.00	1500.00
SatFibonacci01.oox	120.03	VALID	76.70	1194193.70	1194192.70	0.00	1500.00
SatFibonacci02.oox	0.01	VALID	96.20	1.00	198.00	0.00	727.00
SatFibonacci03.oox	120.02	VALID	76.70	1253013.60	1253012.60	0.00	1500.00
SatGcd.oox	120.04	VALID	95.70	16095.50	16094.50	0.00	1500.00
SatHanoi01.oox	120.02	VALID	54.20	1834508.50	2870709.40	0.00	1500.00
SatMccarthy91.oox	120.03	VALID	87.50	476823.50	476822.50	0.00	1500.00
SatPrimes01.oox	120.04	VALID	97.40	3237915.40	3237927.40	0.00	1500.00
Ackermann01.oox	120.04	VALID	81.00	3995388.70	5918750.70	0.00	1500.00
InfiniteLoop.oox	0.04	INVALID	88.90	0.00	1.00	6.00	6.00
UnsatAckermann01.oox	120.01	VALID	92.30	5416389.20	8045627.80	0.00	1500.00
UnsatAddition01.oox	120.03	VALID	75.00	505512.60	505511.60	0.00	1500.00
UnsatAddition02.oox	0.01	INVALID	85.70	0.00	0.00	5.00	5.00
UnsatEvenOdd01.oox	0.16	INVALID	91.50	76.00	504.00	1495.00	1500.00
UnsatMccarthy91.oox	120.03	VALID	87.50	473480.40	473479.40	0.00	1500.00

Table B.3: Depth-First Heuristic on MinePump Set, 10 run average with a depth of 1500

name	time	result	coverage	completed paths	paths explored	error_layer	deepest_layer
spec1-5_product59.oox	11.62	VALID	89.70	9767.00	474656.00	0.00	1500.00
spec1-5_product60.oox	14.80	VALID	89.80	9767.00	578720.00	0.00	1500.00
spec1-5_product61.oox	20.55	VALID	88.10	9921.00	821498.00	0.00	1500.00
spec1-5_product62.oox	21.45	VALID	87.60	9921.00	821498.00	0.00	1500.00
spec1-5_product63.oox	11.48	VALID	89.90	9779.00	478455.00	0.00	1500.00
spec1-5_product64.oox	15.46	VALID	89.90	9779.00	583751.00	0.00	1500.00
spec1-5_product1.oox	7.15	INVALID	83.00	513.00	37208.00	338.00	1311.00
spec1-5_product10.oox	7.04	INVALID	82.50	513.00	37208.00	338.00	1311.00
spec1-5_product11.oox	4.55	INVALID	84.60	513.00	26064.00	338.00	1275.00
spec1-5_product12.oox	4.98	INVALID	84.70	513.00	27792.00	338.00	1275.00
spec1-5_product13.oox	7.31	INVALID	82.90	513.00	39448.00	355.00	1396.00
spec1-5_product14.oox	9.08	INVALID	82.40	513.00	39448.00	355.00	1396.00
spec1-5_product15.oox	6.38	INVALID	84.90	513.00	27472.00	355.00	1360.00
spec1-5_product16.oox	8.89	INVALID	85.00	513.00	29328.00	355.00	1360.00
spec1-5_product2.oox	12.86	INVALID	82.50	513.00	37208.00	338.00	1311.00
spec1-5_product3.oox	8.40	INVALID	84.60	513.00	26064.00	338.00	1275.00
spec1-5_product4.oox	9.14	INVALID	84.70	513.00	27792.00	338.00	1275.00
spec1-5_product48.oox	4.12	INVALID	88.90	530.00	42871.00	343.00	1500.00
spec1-5_product49.oox	4.26	INVALID	88.90	530.00	42871.00	343.00	1500.00
spec1-5_product5.oox	12.71	INVALID	82.90	513.00	39448.00	355.00	1396.00
spec1-5_product50.oox	4.17	INVALID	88.40	530.00	42871.00	343.00	1500.00
spec1-5_product51.oox	3.06	INVALID	90.70	521.00	29915.00	343.00	1500.00
spec1-5_product52.oox	3.45	INVALID	90.80	521.00	31991.00	343.00	1500.00
spec1-5_product53.oox	4.01	INVALID	89.10	527.00	43314.00	340.00	1500.00
spec1-5_product54.oox	4.03	INVALID	88.60	527.00	43314.00	340.00	1500.00
spec1-5_product55.oox	2.91	INVALID	90.90	521.00	30176.00	340.00	1500.00
spec1-5_product56.oox	3.24	INVALID	90.90	521.00	32276.00	340.00	1500.00
spec1-5_product6.oox	12.54	INVALID	82.40	513.00	39448.00	355.00	1396.00
spec1-5_product7.oox	7.99	INVALID	84.90	513.00	27472.00	355.00	1360.00
spec1-5_product8.oox	8.74	INVALID	85.00	513.00	29328.00	355.00	1360.00
spec1-5_product9.oox	11.12	INVALID	83.00	513.00	37208.00	338.00	1311.00

C. Concolic- Genetic Results

Table C.1: Concolic Genetic Deep Set, 10 run average with a depth of 1500

name	time	result	coverage	completed paths	paths explored	error_layer	deepest_layer	c_invocs	iterations	fuzzer%
SatDeep01.oox	73.85	VALID	98.40	13890.40	38674.60	0.00	399.00	3.70	602.80	99.08
SatDriller.oox	120.08	VALID	94.70	20849.30	101893.30	0.00	448.00	14.80	631.80	97.30
UnsatDeep01.oox	0.08	INVALID	97.82	76.20	272.70	375.80	377.90	0.00	1031.60	0.00
UnsatDeep02.oox	20.54	INVALID	98.30	3901.90	10907.90	390.20	396.20	1.00	1177.00	98.44
UnsatDriller.oox	118.53	INVALID	97.13	29619.40	145319.80	400.50	448.00	21.00	1215.80	94.99

Table C.2: Concolic Genetic Jayhorn Set, 10 run average with a depth of 1500

name	time	result	coverage	completed paths	paths explored	error_layer	deepest_layer	c_invocs	iterations	fuzzer%
Addition.oox	85.03	VALID	91.20	1494.50	5321.80	0.00	267.90	21.70	390.30	60.57
SatAckermann01.oox	26.80	VALID	100.00	26.40	1887.80	0.00	1436.00	4.80	2056.80	78.12
SatAckermann02.oox	20.93	VALID	100.00	24.40	1777.70	0.00	1372.00	4.60	1964.40	87.50
SatAckermann03.oox	26.37	VALID	100.00	23.30	1696.90	0.00	1436.00	5.00	2055.70	78.12
SatAddition01.oox	71.36	VALID	89.20	1332.40	4846.20	0.00	199.30	18.90	312.80	44.32
SatEvenOdd01.oox	87.31	VALID	93.60	884.70	151532.10	0.00	1500.00	11.60	2310.20	69.70
SatFibonacci01.oox	13.70	VALID	78.70	208.40	1445.60	0.00	607.80	1.80	955.00	85.90
SatFibonacci02.oox	8.56	VALID	96.20	157.80	31244.40	0.00	727.00	1.00	1144.00	100.00
SatFibonacci03.oox	35.54	VALID	96.70	288.20	2552.50	0.00	1500.00	3.00	2353.60	58.21
SatGcd.oox	39.92	VALID	95.70	933.30	3011.90	0.00	417.20	12.70	628.10	28.12
SatHanoi01.oox	33.31	VALID	94.90	398.20	3492.40	0.00	1500.00	4.80	2497.40	35.56
SatMccarthy91.oox	56.99	VALID	86.84	423.20	53796.40	0.00	1500.00	3.10	2520.80	95.00
SatPrimes01.oox	90.34	VALID	82.50	1247.70	27831.20	0.00	1500.00	16.40	3927.30	6.75
Ackermann01.oox	0.43	INVALID	59.50	162.20	169.00	16.00	16.00	2.00	52.00	23.09
InfiniteLoop.oox	0.04	INVALID	84.46	2.00	376.00	6.00	603.60	0.00	1509.60	0.00
UnsatAckermann01.oox	29.68	INVALID	100.00	273.50	78794.80	32.00	1500.00	3.00	4787.90	86.67
UnsatAddition01.oox	57.90	INVALID	85.82	180.60	45154.00	15.00	1500.00	1.00	5513.20	70.83
UnsatAddition02.oox	0.01	INVALID	85.70	0.00	0.00	5.00	5.00	0.00	12.00	0.00
UnsatEvenOdd01.oox	54.05	INVALID	82.98	85.10	20658.60	22.30	1500.00	1.00	4524.00	73.22
UnsatMccarthy91.oox	25.15	INVALID	88.72	143.00	18533.70	10.00	1500.00	1.00	4514.80	93.33

Table C.3: Concolic Genetic MinePump Set, 10 run average with a depth of 1500

name	time	result	coverage	completed paths	paths explored	error_layer	deepest_layer	c_invocs	iterations	fuzzer%
spec1-5_product59.oox	30.90	VALID	89.70	3298.20	30628.20	0.00	1484.70	2.90	2594.90	100.00
spec1-5_product60.oox	37.14	VALID	89.80	3322.00	36441.40	0.00	1488.80	3.00	2573.70	100.00
spec1-5_product61.oox	52.73	VALID	88.10	3642.60	55728.30	0.00	1500.00	3.40	2695.90	100.00
spec1-5_product62.oox	41.03	VALID	87.60	3120.30	47673.50	0.00	1500.00	2.70	2688.80	100.00
spec1-5_product63.oox	35.67	VALID	89.90	3865.60	36628.00	0.00	1500.00	3.70	2574.10	100.00
spec1-5_product64.oox	39.62	VALID	89.90	3832.80	41818.40	0.00	1485.60	3.70	2539.10	99.94
spec1-5_product1.oox	0.15	INVALID	77.05	15.90	234.30	430.70	775.70	0.00	2274.20	0.00
spec1-5_product10.oox	0.16	INVALID	69.15	9.80	153.60	458.50	618.50	0.00	1744.80	0.00
spec1-5_product11.oox	0.15	INVALID	79.89	54.50	433.20	384.40	719.50	0.00	2377.00	0.00
spec1-5_product12.oox	0.15	INVALID	74.57	27.60	284.00	448.00	653.20	0.00	1989.40	0.00
spec1-5_product13.oox	0.15	INVALID	70.31	6.60	115.40	490.50	638.30	0.00	1795.40	0.00
spec1-5_product14.oox	0.19	INVALID	70.13	15.50	235.30	504.40	684.10	0.00	2008.20	0.00
spec1-5_product15.oox	0.17	INVALID	70.54	25.60	219.00	402.70	549.80	0.00	1809.80	0.00
spec1-5_product16.oox	0.17	INVALID	70.61	15.70	179.90	391.90	570.50	0.00	1714.40	0.00
spec1-5_product2.oox	0.15	INVALID	70.91	9.50	165.90	433.50	700.10	0.00	1998.40	0.00
spec1-5_product3.oox	0.17	INVALID	77.35	59.00	464.30	404.30	713.50	0.00	2342.00	0.00
spec1-5_product4.oox	0.17	INVALID	68.13	18.90	193.20	340.00	548.90	0.00	1705.00	0.00
spec1-5_product48.oox	0.18	INVALID	68.27	18.90	284.80	389.20	722.70	0.00	2144.80	0.00
spec1-5_product49.oox	0.18	INVALID	81.13	24.00	361.30	422.60	1049.00	0.00	3161.80	0.00
spec1-5_product5.oox	0.19	INVALID	70.69	16.90	240.40	374.70	734.00	0.00	2160.60	0.00
spec1-5_product50.oox	0.27	INVALID	79.41	26.00	378.80	281.80	1004.40	0.00	2992.80	0.00
spec1-5_product51.oox	0.21	INVALID	74.38	51.80	441.70	308.70	770.10	0.00	2496.00	0.00
spec1-5_product52.oox	0.21	INVALID	75.42	28.40	318.70	314.50	852.10	0.00	2670.40	0.00
spec1-5_product53.oox	0.22	INVALID	75.64	24.60	361.50	390.60	930.90	0.00	2815.00	0.00
spec1-5_product54.oox	0.22	INVALID	69.87	18.80	292.20	319.30	858.40	0.00	2563.20	0.00
spec1-5_product55.oox	0.21	INVALID	76.94	73.00	646.90	266.40	780.10	0.00	2520.80	0.00
spec1-5_product56.oox	0.26	INVALID	76.79	36.20	392.20	389.50	838.50	0.00	2682.80	0.00
spec1-5_product6.oox	0.24	INVALID	71.18	5.80	105.60	518.20	593.30	0.00	1640.60	0.00
spec1-5_product7.oox	0.21	INVALID	69.79	22.10	206.30	386.60	539.10	0.00	1682.80	0.00
spec1-5_product8.oox	0.20	INVALID	69.33	14.00	151.70	378.90	557.00	0.00	1690.60	0.00
spec1-5_product9.oox	0.20	INVALID	72.22	13.40	193.90	370.70	680.00	0.00	1990.20	0.00

D. Concolic-Random Results

Table D.1: Concolic Random Deep Set, 10 run average with a depth of 1500

name	time	result	coverage	completed paths	paths explored	error_layer	deepest_layer	c_invocs	iterations	fuzzer%
SatDeep01	63.76	VALID	98.40	1961.00	6061.20	0.00	396.60	3.10	587.80	98.46
SatDriller	120.09	VALID	94.70	2640.20	14471.70	0.00	448.00	15.10	632.10	97.30
UnsatDeep01	0.07	INVALID	97.66	57.30	215.10	375.40	376.60	0.00	639.40	0.00
UnsatDeep02	20.00	INVALID	98.30	655.50	2083.20	385.80	392.80	1.00	1144.00	100.00
UnsatDriller	114.62	INVALID	97.40	3682.60	20389.00	445.00	448.00	21.00	1276.00	97.30

Table D.2: Concolic Random Jayhorn Set, 10 run average with a depth of 1500

name	time	result	coverage	completed paths	paths explored	error_layer	deepest_layer	c_invocs	iterations	fuzzer%
Addition	105.19	VALID	91.20	159.00	1769.60	0.00	1500.00	13.40	2093.00	50.57
SatAckermann01	4.22	VALID	100.00	14.00	93.00	0.00	125.00	6.00	180.00	6.25
SatAckermann02	4.20	VALID	100.00	14.00	93.00	0.00	125.00	6.00	180.00	6.25
SatAckermann03	4.23	VALID	100.00	14.00	93.00	0.00	125.00	6.00	180.00	6.25
SatAddition01	84.31	VALID	78.13	208.30	10612.70	0.00	1500.00	11.80	2318.40	85.83
SatEvenOdd01	104.23	VALID	88.73	123.40	15163.30	0.00	1500.00	8.50	2319.60	61.10
SatFibonacci01	13.45	VALID	78.70	26.30	564.10	0.00	607.80	1.80	955.00	81.54
SatFibonacci02	7.14	VALID	96.20	14.00	2772.00	0.00	727.00	1.00	1144.00	100.00
SatFibonacci03	35.43	VALID	96.70	37.70	1180.50	0.00	1500.00	3.00	2353.60	53.85
SatGcd	111.41	VALID	95.70	115.90	2248.70	0.00	1359.20	3.00	2019.60	100.00
SatHanoi01	25.03	VALID	94.90	53.10	1659.30	0.00	1500.00	5.00	2496.10	9.52
SatMccarthy91	58.56	VALID	96.24	57.10	7631.20	0.00	1500.00	3.20	2763.50	96.67
SatPrimes01	54.72	VALID	70.44	133.90	12786.30	0.00	1500.00	7.50	3332.60	63.66
Ackermann01	0.60	INVALID	59.02	16.00	22.00	16.00	16.00	2.00	26.00	46.15
InfiniteLoop	0.03	INVALID	86.68	1.40	263.50	6.00	902.40	0.00	2251.20	0.00
UnsatAckermann01	18.90	VALID	100.00	37.00	8790.00	0.00	1500.00	3.00	2281.00	60.00
UnsatAddition01	60.48	INVALID	84.56	19.00	4754.00	15.00	1500.00	1.00	5498.00	100.00
UnsatAddition02	0.01	INVALID	85.70	0.00	0.00	5.00	5.00	0.00	0.00	0.00
UnsatEvenOdd01	52.60	INVALID	83.40	10.50	2705.80	22.30	1500.00	1.00	4502.00	100.00
UnsatMccarthy91	21.52	INVALID	94.36	14.50	1706.30	28.00	1369.60	0.90	4456.20	90.00

Table D.3: Concolic Random MinePump Set, 10 run average with a depth of 1500

name	time	result	coverage	completed paths	paths explored	error_layer	deepest_layer	c_invocs	iterations	fuzzer%
spec1-5_product59.oox	26.83	VALID	89.70	649.30	6389.50	0.00	1467.40	2.40	2529.30	99.83
spec1-5_product60.oox	35.39	VALID	89.78	820.70	9270.60	0.00	1421.10	3.30	2482.10	99.78
spec1-5_product61.oox	44.99	VALID	88.10	795.80	12885.90	0.00	1500.00	3.20	2652.20	100.00
spec1-5_product62.oox	41.97	VALID	87.60	740.30	12024.30	0.00	1500.00	2.90	2675.10	100.00
spec1-5_product63.oox	25.15	VALID	89.82	721.20	6746.70	0.00	1483.50	2.70	2558.00	99.78
spec1-5_product64.oox	28.36	VALID	89.90	775.00	8868.70	0.00	1469.60	2.70	2599.00	99.72
spec1-5_product1.oox	0.27	INVALID	78.16	26.00	352.00	464.30	902.10	0.00	2441.60	0.00
spec1-5_product10.oox	0.36	INVALID	67.99	17.20	253.50	384.70	611.10	0.00	1051.40	0.00
spec1-5_product11.oox	0.33	INVALID	71.45	35.30	288.70	392.70	525.30	0.00	1322.40	0.00
spec1-5_product12.oox	0.31	INVALID	76.42	19.70	205.80	399.30	608.30	0.00	1470.20	0.00
spec1-5_product13.oox	0.28	INVALID	67.83	11.30	166.00	340.80	681.60	0.00	1424.60	0.00
spec1-5_product14.oox	0.28	INVALID	69.65	11.60	178.40	412.60	682.70	0.00	1346.00	0.00
spec1-5_product15.oox	0.26	INVALID	72.91	24.00	201.90	393.90	513.00	0.00	1030.60	0.00
spec1-5_product16.oox	0.27	INVALID	70.69	17.70	167.90	451.40	519.70	0.00	1015.00	0.00
spec1-5_product2.oox	0.24	INVALID	65.06	8.00	122.00	376.40	537.80	0.00	815.80	0.00
spec1-5_product3.oox	0.25	INVALID	73.81	24.80	236.60	326.90	622.70	0.00	1718.00	0.00
spec1-5_product4.oox	0.23	INVALID	64.88	11.30	109.20	326.80	411.30	0.00	621.40	0.00
spec1-5_product48.oox	0.28	INVALID	70.78	12.10	205.30	385.60	770.70	0.00	1546.80	0.00
spec1-5_product49.oox	0.30	INVALID	72.45	16.40	261.00	355.80	843.00	0.00	1907.60	0.00
spec1-5_product5.oox	0.24	INVALID	73.60	10.70	163.90	340.60	779.60	0.00	2000.00	0.00
spec1-5_product50.oox	0.30	INVALID	74.40	13.00	209.50	419.60	777.80	0.00	1662.80	0.00
spec1-5_product51.oox	0.28	INVALID	71.02	21.80	182.10	287.70	508.60	0.00	1352.20	0.00
spec1-5_product52.oox	0.29	INVALID	69.84	19.60	211.40	334.80	625.70	0.00	1534.00	0.00
spec1-5_product53.oox	0.29	INVALID	72.14	17.60	275.90	426.90	813.00	0.00	2092.80	0.00
spec1-5_product54.oox	0.29	INVALID	72.27	14.40	227.20	384.00	808.80	0.00	1746.00	0.00
spec1-5_product55.oox	0.30	INVALID	74.37	40.60	387.00	358.20	735.30	0.00	1832.80	0.00
spec1-5_product56.oox	0.28	INVALID	82.31	32.00	341.90	393.70	859.20	0.00	2463.60	0.00
spec1-5_product6.oox	0.31	INVALID	75.35	18.50	281.50	480.30	846.50	0.00	1997.80	0.00
spec1-5_product7.oox	0.27	INVALID	79.80	41.50	367.20	340.70	812.70	0.00	2498.20	0.00
spec1-5_product8.oox	0.25	INVALID	82.07	31.30	305.10	410.90	725.30	0.00	1981.60	0.00
spec1-5_product9.oox	0.26	INVALID	71.54	8.10	136.80	433.50	595.10	0.00	742.20	0.00

E. Fuzzer Comparisons

Table E.1: Concolic Random vs Concolic Genetic MinePump Set, 10 run average with a depth of 1500

name	fuzzer	time	result	coverage	completed paths	paths explored	error_layer	deepest_layer	c_invocs	iterations	fuzzer%
spec1-5_product1	Genetic	0.15	INVALID	77.05	15.90	234.30	430.70	775.70	0.00	2274.20	0.00
spec1-5_product1	Random	0.27	INVALID	78.16	26.00	352.00	464.30	902.10	0.00	2441.60	0.00
spec1-5_product10	Genetic	0.16	INVALID	69.15	9.80	153.60	458.50	618.50	0.00	1744.80	0.00
spec1-5_product10	Random	0.36	INVALID	67.99	17.20	253.50	384.70	611.10	0.00	1051.40	0.00
spec1-5_product11	Genetic	0.15	INVALID	79.89	54.50	433.20	384.40	719.50	0.00	2377.00	0.00
spec1-5_product11	Random	0.33	INVALID	71.45	35.30	288.70	392.70	525.30	0.00	1322.40	0.00
spec1-5_product12	Genetic	0.15	INVALID	74.57	27.60	284.00	448.00	653.20	0.00	1989.40	0.00
spec1-5_product12	Random	0.31	INVALID	76.42	19.70	205.80	399.30	608.30	0.00	1470.20	0.00
spec1-5_product13	Genetic	0.15	INVALID	70.31	6.60	115.40	490.50	638.30	0.00	1795.40	0.00
spec1-5_product13	Random	0.28	INVALID	67.83	11.30	166.00	340.80	681.60	0.00	1424.60	0.00
spec1-5_product14	Genetic	0.19	INVALID	70.13	15.50	235.30	504.40	684.10	0.00	2008.20	0.00
spec1-5_product14	Random	0.28	INVALID	69.65	11.60	178.40	412.60	682.70	0.00	1346.00	0.00
spec1-5_product15	Genetic	0.17	INVALID	70.54	25.60	219.00	402.70	549.80	0.00	1809.80	0.00
spec1-5_product15	Random	0.26	INVALID	72.91	24.00	201.90	393.90	513.00	0.00	1030.60	0.00
spec1-5_product16	Genetic	0.17	INVALID	70.61	15.70	179.90	391.90	570.50	0.00	1714.40	0.00
spec1-5_product16	Random	0.27	INVALID	70.69	17.70	167.90	451.40	519.70	0.00	1015.00	0.00
spec1-5_product2	Genetic	0.15	INVALID	70.91	9.50	165.90	433.50	700.10	0.00	1998.40	0.00
spec1-5_product2	Random	0.24	INVALID	65.06	8.00	122.00	376.40	537.80	0.00	815.80	0.00
spec1-5_product3	Genetic	0.17	INVALID	77.35	59.00	464.30	404.30	713.50	0.00	2342.00	0.00
spec1-5_product3	Random	0.25	INVALID	73.81	24.80	236.60	326.90	622.70	0.00	1718.00	0.00
spec1-5_product4	Genetic	0.17	INVALID	68.13	18.90	193.20	340.00	548.90	0.00	1705.00	0.00
spec1-5_product4	Random	0.23	INVALID	64.88	11.30	109.20	326.80	411.30	0.00	621.40	0.00
spec1-5_product48	Genetic	0.18	INVALID	68.27	18.90	284.80	389.20	722.70	0.00	2144.80	0.00
spec1-5_product48	Random	0.28	INVALID	70.78	12.10	205.30	385.60	770.70	0.00	1546.80	0.00
spec1-5_product49	Genetic	0.18	INVALID	81.13	24.00	361.30	422.60	1049.00	0.00	3161.80	0.00
spec1-5_product49	Random	0.30	INVALID	72.45	16.40	261.00	355.80	843.00	0.00	1907.60	0.00
spec1-5_product5	Genetic	0.19	INVALID	70.69	16.90	240.40	374.70	734.00	0.00	2160.60	0.00
spec1-5_product5	Random	0.24	INVALID	73.60	10.70	163.90	340.60	779.60	0.00	2000.00	0.00
spec1-5_product50	Genetic	0.27	INVALID	79.41	26.00	378.80	281.80	1004.40	0.00	2992.80	0.00
spec1-5_product50	Random	0.30	INVALID	74.40	13.00	209.50	419.60	777.80	0.00	1662.80	0.00
spec1-5_product51	Genetic	0.21	INVALID	74.38	51.80	441.70	308.70	770.10	0.00	2496.00	0.00
spec1-5_product51	Random	0.28	INVALID	71.02	21.80	182.10	287.70	508.60	0.00	1352.20	0.00
spec1-5_product52	Genetic	0.21	INVALID	75.42	28.40	318.70	314.50	852.10	0.00	2670.40	0.00
spec1-5_product52	Random	0.29	INVALID	69.84	19.60	211.40	334.80	625.70	0.00	1534.00	0.00
spec1-5_product53	Genetic	0.22	INVALID	75.64	24.60	361.50	390.60	930.90	0.00	2815.00	0.00
spec1-5_product53	Random	0.29	INVALID	72.14	17.60	275.90	426.90	813.00	0.00	2092.80	0.00
spec1-5_product54	Genetic	0.22	INVALID	69.87	18.80	292.20	319.30	858.40	0.00	2563.20	0.00
spec1-5_product54	Random	0.29	INVALID	72.27	14.40	227.20	384.00	808.80	0.00	1746.00	0.00
spec1-5_product55	Genetic	0.21	INVALID	76.94	73.00	646.90	266.40	780.10	0.00	2520.80	0.00
spec1-5_product55	Random	0.30	INVALID	74.37	40.60	387.00	358.20	735.30	0.00	1832.80	0.00
spec1-5_product56	Genetic	0.26	INVALID	76.79	36.20	392.20	389.50	838.50	0.00	2682.80	0.00
spec1-5_product56	Random	0.28	INVALID	82.31	32.00	341.90	393.70	859.20	0.00	2463.60	0.00
spec1-5_product59	Genetic	30.90	VALID	89.70	3298.20	30628.20	0.00	1484.70	2.90	2594.90	100.00
spec1-5_product59	Random	26.83	VALID	89.70	649.30	6389.50	0.00	1467.40	2.40	2529.30	99.83
spec1-5_product6	Genetic	0.24	INVALID	71.18	5.80	105.60	518.20	593.30	0.00	1640.60	0.00
spec1-5_product6	Random	0.31	INVALID	75.35	18.50	281.50	480.30	846.50	0.00	1997.80	0.00
spec1-5_product60	Genetic	37.14	VALID	89.80	3322.00	36441.40	0.00	1488.80	3.00	2573.70	100.00
spec1-5_product60	Random	35.39	VALID	89.78	820.70	9270.60	0.00	1421.10	3.30	2482.10	99.78
spec1-5_product61	Genetic	52.73	VALID	88.10	3642.60	55728.30	0.00	1500.00	3.40	2695.90	100.00
spec1-5_product61	Random	44.99	VALID	88.10	795.80	12885.90	0.00	1500.00	3.20	2652.20	100.00
spec1-5_product62	Genetic	41.03	VALID	87.60	3120.30	47673.50	0.00	1500.00	2.70	2688.80	100.00
spec1-5_product62	Random	41.97	VALID	87.60	740.30	12024.30	0.00	1500.00	2.90	2675.10	100.00
spec1-5_product63	Genetic	35.67	VALID	89.90	3865.60	36628.00	0.00	1500.00	3.70	2574.10	100.00
spec1-5_product63	Random	25.15	VALID	89.82	721.20	6746.70	0.00	1483.50	2.70	2558.00	99.78
spec1-5_product64	Genetic	39.62	VALID	89.90	3832.80	41818.40	0.00	1485.60	3.70	2539.10	99.94
spec1-5_product64	Random	28.36	VALID	89.90	775.00	8868.70	0.00	1469.60	2.70	2599.00	99.72
spec1-5_product7	Genetic	0.21	INVALID	69.79	22.10	206.30	386.60	539.10	0.00	1682.80	0.00
spec1-5_product7	Random	0.27	INVALID	79.80	41.50	367.20	340.70	812.70	0.00	2498.20	0.00
spec1-5_product8	Genetic	0.20	INVALID	69.33	14.00	151.70	378.90	557.00	0.00	1690.60	0.00
spec1-5_product8	Random	0.25	INVALID	82.07	31.30	305.10	410.90	725.30	0.00	1981.60	0.00
spec1-5_product9	Genetic	0.20	INVALID	72.22	13.40	193.90	370.70	680.00	0.00	1990.20	0.00
spec1-5_product9	Random	0.26	INVALID	71.54	8.10	136.80	433.50	595.10	0.00	742.20	0.00

Figure E.1: Comparison of what percentage of the transition coverage is caused by the fuzzing algorithm after each concolic invocation

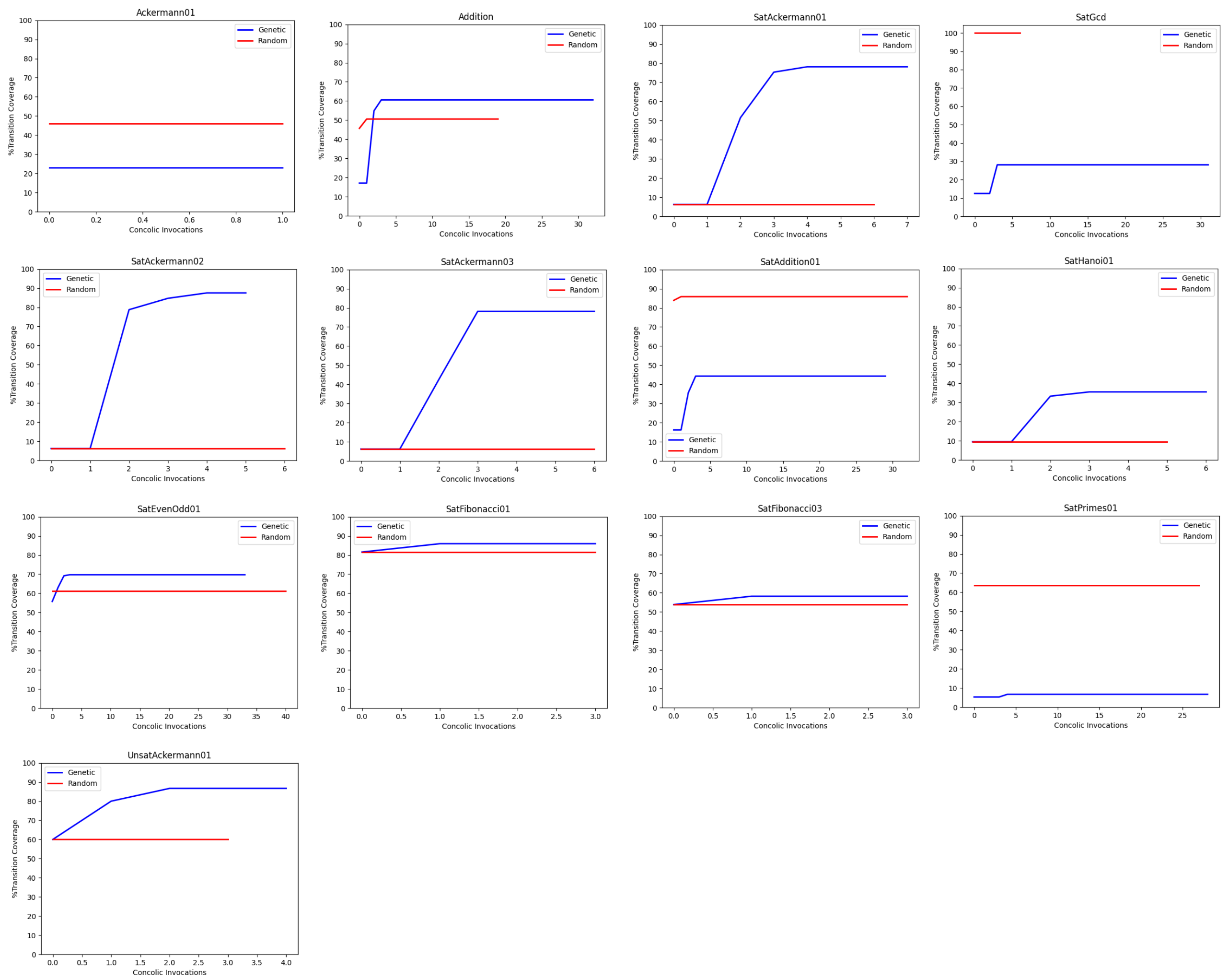


Table E.2: Concolic Random vs Concolic Genetic Deep Set, 10 run average with a depth of 1500

name	fuzzer	time	result	coverage	completed paths	paths explored	error_layer	deepest_layer	c_invocs	iterations	fuzzer%
SatDeep01	Genetic	73.85	VALID	98.40	13890.40	38674.60	0.00	399.00	3.70	602.80	99.08
SatDeep01	Random	63.76	VALID	98.40	1961.00	6061.20	0.00	396.60	3.10	587.80	98.46
SatDriller	Genetic	120.08	VALID	94.70	20849.30	101893.30	0.00	448.00	14.80	631.80	97.30
SatDriller	Random	120.09	VALID	94.70	2640.20	14471.70	0.00	448.00	15.10	632.10	97.30
UnsatDeep01	Genetic	0.08	INVALID	97.82	76.20	272.70	375.80	377.90	0.00	1031.60	0.00
UnsatDeep01	Random	0.07	INVALID	97.66	57.30	215.10	375.40	376.60	0.00	639.40	0.00
UnsatDeep02	Genetic	20.54	INVALID	98.30	3901.90	10907.90	390.20	396.20	1.00	1177.00	98.44
UnsatDeep02	Random	20.00	INVALID	98.30	655.50	2083.20	385.80	392.80	1.00	1144.00	100.00
UnsatDriller	Genetic	118.53	INVALID	97.13	29619.40	145319.80	400.50	448.00	21.00	1215.80	94.99
UnsatDriller	Random	114.62	INVALID	97.40	3682.60	20389.00	445.00	448.00	21.00	1276.00	97.30

Table E.3: Concolic Random vs Concolic Genetic Jayhorn Set, 10 run average with a depth of 1500

name	fuzzer	time	result	coverage	completed paths	paths explored	error_layer	deepest_layer	c_invocs	iterations	fuzzer%
Addition	Genetic	85.03	VALID	91.20	1494.50	5321.80	0.00	267.90	21.70	390.30	60.57
Addition	Random	105.19	VALID	91.20	159.00	1769.60	0.00	1500.00	13.40	2093.00	50.57
SatAckermann01	Genetic	26.80	VALID	100.00	26.40	1887.80	0.00	1436.00	4.80	2056.80	78.12
SatAckermann01	Random	4.22	VALID	100.00	14.00	93.00	0.00	125.00	6.00	180.00	6.25
SatAckermann02	Genetic	20.93	VALID	100.00	24.40	1777.70	0.00	1372.00	4.60	1964.40	87.50
SatAckermann02	Random	4.20	VALID	100.00	14.00	93.00	0.00	125.00	6.00	180.00	6.25
SatAckermann03	Genetic	26.37	VALID	100.00	23.30	1696.90	0.00	1436.00	5.00	2055.70	78.12
SatAckermann03	Random	4.23	VALID	100.00	14.00	93.00	0.00	125.00	6.00	180.00	6.25
SatAddition01	Genetic	71.36	VALID	89.20	1332.40	4846.20	0.00	199.30	18.90	312.80	44.32
SatAddition01	Random	84.31	VALID	78.13	208.30	10612.70	0.00	1500.00	11.80	2318.40	85.83
SatEvenOdd01	Genetic	87.31	VALID	93.60	884.70	151532.10	0.00	1500.00	11.60	2310.20	69.70
SatEvenOdd01	Random	104.23	VALID	88.73	123.40	15163.30	0.00	1500.00	8.50	2319.60	61.10
SatFibonacci01	Genetic	13.70	VALID	78.70	208.40	1445.60	0.00	607.80	1.80	955.00	85.90
SatFibonacci01	Random	13.45	VALID	78.70	26.30	564.10	0.00	607.80	1.80	955.00	81.54
SatFibonacci02	Genetic	8.56	VALID	96.20	157.80	31244.40	0.00	727.00	1.00	1144.00	100.00
SatFibonacci02	Random	7.14	VALID	96.20	14.00	2772.00	0.00	727.00	1.00	1144.00	100.00
SatFibonacci03	Genetic	35.54	VALID	96.70	288.20	2552.50	0.00	1500.00	3.00	2353.60	58.21
SatFibonacci03	Random	35.43	VALID	96.70	37.70	1180.50	0.00	1500.00	3.00	2353.60	53.85
SatGcd	Genetic	39.92	VALID	95.70	933.30	3011.90	0.00	417.20	12.70	628.10	28.12
SatGcd	Random	111.41	VALID	95.70	115.90	2248.70	0.00	1359.20	3.00	2019.60	100.00
SatHanoi01	Genetic	33.31	VALID	94.90	398.20	3492.40	0.00	1500.00	4.80	2497.40	35.56
SatHanoi01	Random	25.03	VALID	94.90	53.10	1659.30	0.00	1500.00	5.00	2496.10	9.52
SatMccarthy91	Genetic	56.99	VALID	86.84	423.20	53796.40	0.00	1500.00	3.10	2520.80	95.00
SatMccarthy91	Random	58.56	VALID	96.24	57.10	7631.20	0.00	1500.00	3.20	2763.50	96.67
SatPrimes01	Genetic	90.34	VALID	82.50	1247.70	27831.20	0.00	1500.00	16.40	3927.30	6.75
SatPrimes01	Random	54.72	VALID	70.44	133.90	12786.30	0.00	1500.00	7.50	3332.60	63.66
Ackermann01	Genetic	0.43	INVALID	59.50	162.20	169.00	16.00	16.00	2.00	52.00	23.09
Ackermann01	Random	0.60	INVALID	59.02	16.00	22.00	16.00	16.00	2.00	26.00	46.15
InfiniteLoop	Genetic	0.04	INVALID	84.46	2.00	376.00	6.00	603.60	0.00	1509.60	0.00
InfiniteLoop	Random	0.03	INVALID	86.68	1.40	263.50	6.00	902.40	0.00	2251.20	0.00
UnsatAckermann01	Genetic	29.68	INVALID	100.00	273.50	78794.80	32.00	1500.00	3.00	4787.90	86.67
UnsatAckermann01	Random	18.90	VALID	100.00	37.00	8790.00	0.00	1500.00	3.00	2281.00	60.00
UnsatAddition01	Genetic	57.90	INVALID	85.82	180.60	45154.00	15.00	1500.00	1.00	5513.20	70.83
UnsatAddition01	Random	60.48	INVALID	84.56	19.00	4754.00	15.00	1500.00	1.00	5498.00	100.00
UnsatAddition02	Genetic	0.01	INVALID	85.70	0.00	0.00	5.00	5.00	0.00	12.00	0.00
UnsatAddition02	Random	0.01	INVALID	85.70	0.00	0.00	5.00	5.00	0.00	0.00	0.00
UnsatEvenOdd01	Genetic	54.05	INVALID	82.98	85.10	20658.60	22.30	1500.00	1.00	4524.00	73.22
UnsatEvenOdd01	Random	52.60	INVALID	83.40	10.50	2705.80	22.30	1500.00	1.00	4502.00	100.00
UnsatMccarthy91	Genetic	25.15	INVALID	88.72	143.00	18533.70	10.00	1500.00	1.00	4514.80	93.33
UnsatMccarthy91	Random	21.52	INVALID	94.36	14.50	1706.30	28.00	1369.60	0.90	4456.20	90.00

F. State-Of-The-Art Results

Table F.1: Average Comparison Concolic, JDart, JBMC, Java-Ranger on Invalid Minepump Programs

heuristic	time
DSE	0.15
JBMC	2.41
JDart	4.13
JavaRanger	10.06

Table F.2: Average Comparison Concolic, JDart, JBMC, Java-Ranger on Valid Minepump Programs

heuristic	time
DSE	33.05
JBMC	3.17
JDart	24.33
JavaRanger	325.00

Table F.3: Average Comparison Concolic, JDart, JBMC, Java-Ranger on the Deep set

heuristic	time
DSE	48.83
JBMC	2.94
JDart	542.90
JavaRanger	336.57

Table F.4: JDart, Java-Ranger and JBMC results on Deep Set, 5 run average

program	tool	time	verdict
SatDeep01	DSE	56.51	VALID
SatDeep01	JDart	901.70	TIMEOUT
SatDeep01	JBMC	2.76	VALID
SatDeep01	JavaRanger	8.60	UNKOWN
SatDriller	DSE	91.91	VALID
SatDriller	JDart	902.19	TIMEOUT
SatDriller	JBMC	3.60	VALID
SatDriller	JavaRanger	829.43	VALID
UnsatDeep01	DSE	0.05	INVALID
UnsatDeep01	JDart	4.01	INVALID
UnsatDeep01	JBMC	2.50	INVALID
UnsatDeep01	JavaRanger	7.90	UNKOWN
UnsatDeep02	DSE	9.56	INVALID
UnsatDeep02	JDart	4.48	INVALID
UnsatDeep02	JBMC	2.50	INVALID
UnsatDeep02	JavaRanger	7.84	UNKOWN
UnsatDriller	DSE	86.10	INVALID
UnsatDriller	JDart	902.10	TIMEOUT
UnsatDriller	JBMC	3.34	INVALID
UnsatDriller	JavaRanger	829.06	VALID

Table F.5: JDart, Java-Ranger and JBMC results on Jayhorn Recursive Set, SVComp Results

name	heuristic	time	result
Ackermann01	DSE	0.35	INVALID
Ackermann01	JBMC	1.40	INVALID
Ackermann01	JDart	3.60	INVALID
Ackermann01	JavaRanger	8.80	INVALID
Addition	DSE	109.10	VALID
Addition	JBMC	880.00	UNKOWN
Addition	JDart	130.00	VALID
Addition	JavaRanger	630.00	UNKOWN
InfiniteLoop	DSE	0.02	INVALID
InfiniteLoop	JBMC	1.00	INVALID
InfiniteLoop	JDart	3.70	INVALID
InfiniteLoop	JavaRanger	8.40	INVALID
SatAckermann01	DSE	12.65	VALID
SatAckermann01	JBMC	500.00	OOM
SatAckermann01	JDart	900.00	TIMEOUT
SatAckermann01	JavaRanger	800.00	UNKOWN
SatAckermann02	DSE	12.32	VALID
SatAckermann02	JBMC	450.00	OOM
SatAckermann02	JDart	680.00	TIMEOUT
SatAckermann02	JavaRanger	560.00	UNKOWN
SatAckermann03	DSE	9.94	VALID
SatAckermann03	JBMC	470.00	OOM
SatAckermann03	JDart	900.00	TIMEOUT
SatAckermann03	JavaRanger	800.00	UNKOWN
SatAddition01	DSE	110.08	VALID
SatAddition01	JBMC	880.00	ERROR
SatAddition01	JDart	140.00	VALID
SatAddition01	JavaRanger	800.00	UNKOWN
SatEvenOdd01	DSE	65.20	VALID
SatEvenOdd01	JBMC	880.00	ERROR
SatEvenOdd01	JDart	35.00	VALID
SatEvenOdd01	JavaRanger	73.00	VALID
SatFibonacci01	DSE	16.35	VALID
SatFibonacci01	JBMC	700.00	OOM
SatFibonacci01	JDart	420.00	TIMEOUT
SatFibonacci01	JavaRanger	870.00	VALID
SatFibonacci03	DSE	27.76	VALID
SatFibonacci03	JBMC	670.00	OOM
SatFibonacci03	JDart	900.00	TIMEOUT
SatFibonacci03	JavaRanger	870.00	VALID

name	heuristic	time	result
SatGcd	DSE	22.50	VALID
SatGcd	JBMC	880.00	ERROR
SatGcd	JDart	900.00	TIMEOUT
SatGcd	JavaRanger	610.00	UNKOWN
SatHanoi01	DSE	29.97	VALID
SatHanoi01	JBMC	580.00	OOM
SatHanoi01	JDart	900.00	TIMEOUT
SatHanoi01	JavaRanger	810.00	VALID
SatMccarthy91	DSE	48.97	VALID
SatMccarthy91	JBMC	710.00	OOM
SatMccarthy91	JDart	900.00	TIMEOUT
SatMccarthy91	JavaRanger	820.00	VALID
SatPrimes01	DSE	65.46	VALID
SatPrimes01	JBMC	450.00	OOM
SatPrimes01	JDart	900.00	TIMEOUT
SatPrimes01	JavaRanger	800.00	UNKOWN
UnsatAckermann01	DSE	23.90	INVALID
UnsatAckermann01	JBMC	4.80	INVALID
UnsatAckermann01	JDart	900.00	TIMEOUT
UnsatAckermann01	JavaRanger	800.00	UNKOWN
UnsatAddition01	DSE	54.63	INVALID
UnsatAddition01	JBMC	0.91	INVALID
UnsatAddition01	JDart	3.70	INVALID
UnsatAddition01	JavaRanger	800.00	UNKOWN
UnsatAddition02	DSE	0.01	INVALID
UnsatAddition02	JBMC	880.00	ERROR
UnsatAddition02	JDart	460.00	INVALID
UnsatAddition02	JavaRanger	800.00	UNKOWN
UnsatEvenOdd01	DSE	35.78	INVALID
UnsatEvenOdd01	JBMC	1.10	INVALID
UnsatEvenOdd01	JDart	3.70	INVALID
UnsatEvenOdd01	JavaRanger	8.20	INVALID
UnsatMccarthy91	DSE	10.76	INVALID
UnsatMccarthy91	JBMC	1.10	INVALID
UnsatMccarthy91	JDart	3.70	INVALID
UnsatMccarthy91	JavaRanger	8.00	INVALD

Table F.6: JDart, Java-Ranger and JBMC results on MinePump set, SVComp Results

name	heuristic	time	result
spec1-5_product1	DSE	0.13	INVALID
spec1-5_product1	JBMC	2.30	INVALID
spec1-5_product1	JDart	4.50	INVALID
spec1-5_product1	JavaRanger	8.80	INVALID
spec1-5_product10	DSE	0.13	INVALID
spec1-5_product10	JBMC	2.30	INVALID
spec1-5_product10	JDart	3.90	INVALID
spec1-5_product10	JavaRanger	9.00	INVALID
spec1-5_product11	DSE	0.16	INVALID
spec1-5_product11	JBMC	2.50	INVALID
spec1-5_product11	JDart	3.80	INVALID
spec1-5_product11	JavaRanger	9.40	INVALID
spec1-5_product12	DSE	0.14	INVALID
spec1-5_product12	JBMC	2.50	INVALID
spec1-5_product12	JDart	3.80	INVALID
spec1-5_product12	JavaRanger	9.10	INVALID
spec1-5_product13	DSE	0.16	INVALID
spec1-5_product13	JBMC	2.40	INVALID
spec1-5_product13	JDart	3.80	INVALID
spec1-5_product13	JavaRanger	8.40	INVALID
spec1-5_product14	DSE	0.15	INVALID
spec1-5_product14	JBMC	2.20	INVALID
spec1-5_product14	JDart	3.90	INVALID
spec1-5_product14	JavaRanger	9.20	INVALID
spec1-5_product15	DSE	0.15	INVALID
spec1-5_product15	JBMC	2.40	INVALID
spec1-5_product15	JDart	3.90	INVALID
spec1-5_product15	JavaRanger	8.90	INVALID
spec1-5_product16	DSE	0.14	INVALID
spec1-5_product16	JBMC	2.30	INVALID
spec1-5_product16	JDart	3.90	INVALID
spec1-5_product16	JavaRanger	8.50	INVALID
spec1-5_product2	DSE	0.14	INVALID
spec1-5_product2	JBMC	2.40	INVALID
spec1-5_product2	JDart	3.80	INVALID
spec1-5_product2	JavaRanger	8.50	INVALID
spec1-5_product3	DSE	0.13	INVALID
spec1-5_product3	JBMC	2.40	INVALID
spec1-5_product3	JDart	4.10	INVALID
spec1-5_product3	JavaRanger	9.30	INVALID
spec1-5_product4	DSE	0.14	INVALID
spec1-5_product4	JBMC	2.20	INVALID
spec1-5_product4	JDart	3.90	INVALID
spec1-5_product4	JavaRanger	9.50	INVALID
spec1-5_product48	DSE	0.15	INVALID
spec1-5_product48	JBMC	2.80	INVALID
spec1-5_product48	JDart	3.80	INVALID
spec1-5_product48	JavaRanger	39.00	INVALID
spec1-5_product49	DSE	0.18	INVALID
spec1-5_product49	JBMC	2.70	INVALID
spec1-5_product49	JDart	4.40	INVALID
spec1-5_product49	JavaRanger	8.20	INVALID
spec1-5_product5	DSE	0.13	INVALID
spec1-5_product5	JBMC	2.20	INVALID
spec1-5_product5	JDart	4.60	INVALID
spec1-5_product5	JavaRanger	8.90	INVALID
spec1-5_product50	DSE	0.16	INVALID
spec1-5_product50	JBMC	2.50	INVALID
spec1-5_product50	JDart	4.40	INVALID
spec1-5_product50	JavaRanger	8.40	INVALID
spec1-5_product51	DSE	0.15	INVALID
spec1-5_product51	JBMC	2.40	INVALID
spec1-5_product51	JDart	4.50	INVALID
spec1-5_product51	JavaRanger	9.40	INVALID
spec1-5_product52	DSE	0.17	INVALID
spec1-5_product52	JBMC	2.40	INVALID
spec1-5_product52	JDart	4.50	INVALID
spec1-5_product52	JavaRanger	8.90	INVALID
spec1-5_product53	DSE	0.16	INVALID
spec1-5_product53	JBMC	2.60	INVALID
spec1-5_product53	JDart	4.60	INVALID
spec1-5_product53	JavaRanger	9.10	INVALID
spec1-5_product54	DSE	0.16	INVALID
spec1-5_product54	JBMC	2.50	INVALID
spec1-5_product54	JDart	4.50	INVALID
spec1-5_product54	JavaRanger	8.80	INVALID
spec1-5_product55	DSE	0.16	INVALID
spec1-5_product55	JBMC	2.70	INVALID
spec1-5_product55	JDart	4.50	INVALID
spec1-5_product55	JavaRanger	8.50	INVALID
spec1-5_product56	DSE	0.17	INVALID
spec1-5_product56	JBMC	2.40	INVALID
spec1-5_product56	JDart	4.50	INVALID
spec1-5_product56	JavaRanger	9.40	INVALID
spec1-5_product59	DSE	25.69	VALID
spec1-5_product59	JBMC	3.30	VALID
spec1-5_product59	JDart	24.00	VALID
spec1-5_product59	JavaRanger	330.00	VALID
spec1-5_product6	DSE	0.14	INVALID
spec1-5_product6	JBMC	2.20	INVALID
spec1-5_product6	JDart	3.80	INVALID
spec1-5_product6	JavaRanger	8.40	INVALID
spec1-5_product60	DSE	34.97	VALID
spec1-5_product60	JBMC	3.20	VALID
spec1-5_product60	JDart	24.00	VALID
spec1-5_product60	JavaRanger	460.00	VALID
spec1-5_product61	DSE	37.54	VALID
spec1-5_product61	JBMC	3.00	VALID
spec1-5_product61	JDart	25.00	VALID
spec1-5_product61	JavaRanger	160.00	VALID
spec1-5_product62	DSE	34.76	VALID
spec1-5_product62	JBMC	3.00	VALID
spec1-5_product62	JDart	25.00	VALID
spec1-5_product62	JavaRanger	160.00	VALID
spec1-5_product63	DSE	33.08	VALID
spec1-5_product63	JBMC	3.20	VALID
spec1-5_product63	JDart	24.00	VALID
spec1-5_product63	JavaRanger	360.00	VALID
spec1-5_product64	DSE	32.26	VALID
spec1-5_product64	JBMC	3.30	VALID
spec1-5_product64	JDart	24.00	VALID
spec1-5_product64	JavaRanger	480.00	VALID
spec1-5_product7	DSE	0.13	INVALID
spec1-5_product7	JBMC	2.50	INVALID
spec1-5_product7	JDart	3.90	INVALID
spec1-5_product7	JavaRanger	8.50	INVALID
spec1-5_product8	DSE	0.13	INVALID
spec1-5_product8	JBMC	2.30	INVALID
spec1-5_product8	JDart	3.90	INVALID
spec1-5_product8	JavaRanger	9.00	INVALID
spec1-5_product9	DSE	0.14	INVALID
spec1-5_product9	JBMC	2.20	INVALID
spec1-5_product9	JDart	4.00	INVALID
spec1-5_product9	JavaRanger	8.30	INVALID

Bibliography

[1] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[2] K. Y. Rozier, "Linear temporal logic symbolic model checking," *Computer Science Review*, vol. 5, no. 2, pp. 163–203, 2011.

[3] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976, ISSN: 0001-0782. DOI: `10.1145/360248.360252`. [Online]. Available: `https://doi.org/10.1145/360248.360252`.

[4] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[5] S. Koppier, "The path explosion problem in symbolic execution: An approach to the effects of concurrency and aliasing," M.S. thesis, 2020.

[6] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, 2008, pp. 209–224.

[7] E. F. Rizzi, S. Elbaum, and M. B. Dwyer, "On the techniques we create, the tools we build, and their misalignments: A study of klee," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 132–143.

[8] V. Sharma, S. Hussein, M. W. Whalen, S. McCamant, and W. Visser, "Java ranger: Statically summarizing regions for efficient symbolic execution of java," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 123–134.

[9] V. Sharma, S. Hussein, M. W. Whalen, S. McCamant, and W. Visser, "Java ranger at sv-comp 2020 (competition contribution)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2020, pp. 393–397.

[10] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[11] R. Brenguier, L. Cordeiro, D. Kroening, and P. Schrammel, "Jbmc: A bounded model checking tool for java bytecode," *arXiv preprint arXiv:2302.02381*, 2023.

[12] K. Luckow, M. Dimjašević, D. Giannakopoulou, *et al.*, "Jd art: A dynamic symbolic analysis framework," in *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings 22*, Springer, 2016, pp. 442–459.

[13] M. Mues and F. Howar, "JDart: Portfolio solving, breadth-first search and SMT-Lib strings (competition contribution)," in *Proc. TACAS (2)*, ser. LNCS 12652, Springer, 2021, pp. 448–452. DOI: `10.1007/978-3-030-72013-1_30`.

[14] A. Avancini and M. Ceccato, "Comparison and integration of genetic algorithms and dynamic symbolic execution for security testing of cross-site scripting vulnerabilities," *Information and Software Technology*, vol. 55, no. 12, pp. 2209–2222, 2013.

[15] M. Christakis, P. Müller, and V. Wüstholz, "Guiding dynamic symbolic execution toward unverified program executions," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 144–155.

[16] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, "Automated whitebox fuzz testing.," in *NDSS*, vol. 8, 2008, pp. 151–166.

[17] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 122–131.

[18] C. S. Păsăreanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," ser. ISSTA '11, Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 34–44, ISBN: 9781450305624. DOI: `10.1145/2001420.2001425`. [Online]. Available: `https://doi.org/10.1145/2001420.2001425`.

[19] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.

[20] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[21] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*, IEEE, 2012, pp. 380–394.

[22] N. Stephens, J. Grosen, C. Salls, *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution.," in *NDSS*, vol. 16, 2016, pp. 1–16.

[23] D. van Vliet, "A heuristic approach to the path explosion problem for complex heap programs," Jul. 2023.

[24] A. Fioraldi, A. Mantovani, D. Maier, and D. Balzarotti, "Dissecting american fuzzy lop: A fuzzbench evaluation," *ACM transactions on software engineering and methodology*, vol. 32, no. 2, pp. 1–26, 2023.

[25] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.

[26] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *2011 11th International Conference on Quality Software*, IEEE, 2011, pp. 31–40.

[27] G. Fraser, A. Arcuri, and P. McMinn, "A memetic algorithm for whole test suite generation," *Journal of Systems and Software*, vol. 103, pp. 311–327, 2015.

[28] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik, "Jbmc: A bounded model checking tool for verifying java bytecode," in *International Conference on Computer Aided Verification*, Springer, 2018, pp. 183–190.