

Automatic Generation of Fuzzing Benchmark Suites

- Generated Based on Genetic Algorithms

Automatisk generering av testsviter för fuzzers

Tommy Johansson

Supervisor : Nahid Shahmehri
Examiner : Ulf Kargén

Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: <http://www.ep.liu.se/>.

Abstract

Security testing is performed to find security bugs, which are harder to find since security bugs cannot be connected to functional requirements. That makes it much harder to find security bugs. To make security testing easier, a technique known as fuzzing is used to generate random or semi-valid data to test the program's ability to handle malformed or malicious input. By catching security bugs before the program is released, the number of potential exploits that could be used by an attacker is decreased. To increase the efficiency of these fuzzers there have been works to create benchmark suites to evaluate fuzzer performance on different types of security bugs. These suites evaluate fuzzers in different ways, and the results are usually not comparable. Some benchmark suites generate synthetic bugs while others manually reintroduce bugs or write intentionally buggy programs. The most difficult metric to evaluate fuzzers on is how many unique bugs were found by a fuzzer. A crash in the program from a fuzzer might be caused by a single bug or a set of bugs in the program, and it is very hard to know which bug caused the crash without manual inspection.

The aim of this thesis was to develop a benchmark test suite that could automatically generate test suites for fuzzers that could automatically determine which bug caused the program to crash. This was done by reintroducing fixed known vulnerabilities into the latest stable version of a program. A diff of the code for each vulnerability from before it was fixed to the latest version was used and reduced with the use of a genetic algorithm. The genetic algorithm managed to reduce 99,6% of 250 000 lines of code changes to 974 lines of code changes from 25 diffs from five different projects in an average time of 4 hours per project. From the 25 reduced diffs, only 19 could be used in the generated test suites, with 4 of the selected bugs included on average for each project, with a yield of 76% on average. However, more research is needed to investigate how the yield changes when more vulnerabilities are added and how well these reintroduced bugs are found by different fuzzers.

Acknowledgments

I want to thank my examiner, Ulf Kargén, and my supervisor, Nahid Shahmehri, for their great support and feedback during this thesis, as well as for allowing me to complete this thesis under the circumstances that arose during this thesis. I also want to thank my family for believing in me and pushing me to complete this thesis.

Contents

| | |
|--|-------------|
| Abstract | iii |
| Acknowledgments | iv |
| Contents | v |
| List of Algorithms | vii |
| List of Figures | viii |
| List of Tables | x |
| 1 Introduction | 1 |
| 1.1 Background | 1 |
| 1.2 Motivation | 1 |
| 1.3 Aim | 2 |
| 1.4 Research questions | 2 |
| 1.5 Delimitations | 3 |
| 2 Theory | 4 |
| 2.1 Fuzzing | 4 |
| 2.2 Fuzzing Evaluation | 5 |
| 2.3 OSS-Fuzz | 5 |
| 2.4 Genetic Algorithms | 6 |
| 2.5 Automatic Program Repair | 15 |
| 2.6 Version Control | 15 |
| 3 Previous Work | 17 |
| 3.1 Fuzzing Evaluations with reintroduced Bugs | 17 |
| 3.2 Automatic Program Repair | 17 |
| 3.3 Genetic Algorithms | 18 |
| 4 Method | 19 |
| 4.1 Overview | 19 |
| 4.2 Pre-study | 19 |
| 4.3 Design and Implementation | 20 |
| 5 Results | 39 |
| 5.1 Overview | 39 |
| 5.2 Pre-study | 39 |
| 5.3 Parameter Tests | 41 |
| 5.4 Evaluation | 45 |
| 6 Discussion | 48 |

| | | |
|----------|---|-----------|
| 6.1 | Results | 48 |
| 6.2 | Method | 49 |
| 6.3 | The Work in a Wider Context | 50 |
| 7 | Conclusion | 52 |
| 7.1 | Conclusions | 52 |
| 7.2 | How Can GAs Be Used To Minimize Diffs? | 52 |
| 7.3 | How Can Several Vulnerable Versions of a Program With Only One Known Vulnerability Be Merged Into One Program With the Same Vulnerabilities? . . . | 53 |
| 7.4 | Future Work | 53 |
| | Bibliography | 54 |
| A | Appendix | 57 |
| A.1 | Detailed Stats | 57 |

List of Algorithms

| | | |
|---|--------------------------------|----|
| 1 | Reduce Required Bits | 32 |
| 2 | Evaluate Test Suite | 38 |

List of Figures

| | | |
|------|---|----|
| 2.1 | Different types of representations. | 6 |
| 2.2 | Proportionate Roulette Wheel Selection with population size n | 7 |
| 2.3 | Stochastic Universal Selection with population size n | 8 |
| 2.4 | Tournament Selection with size $m = 2$ and population size n | 9 |
| 2.5 | K-point Crossover with $K = 1$ | 10 |
| 2.6 | K-point Crossover with $K = 2$ | 10 |
| 2.7 | Uniform Crossover with 4 successful coin flips. | 10 |
| 2.8 | Partially Mapped Crossover with 2 mappings. | 11 |
| 2.9 | Bit-flip Mutation with 3 successful weighted coin flips. | 12 |
| 2.10 | Simple Inversion Mutation with a selected segment of length 4 inserted at position 2. | 12 |
| 2.11 | Displacement Mutation with a selected segment of length 4 inserted at position 8. | 13 |
| 2.12 | Displacement Mutation with an exchange of segments of length 3. | 13 |
| 2.13 | Scramble Mutation with a selected segment of length 4. | 13 |
| 2.14 | Example output from <i>git -diff</i> showing the file <i>main.c</i> starting at line 6 in the function <i>factorial</i> with 7 lines from the old version and 9 lines from the new version. | 16 |
| | | |
| 4.1 | A parsed diff with 3 files and 4 hunks and 17 changed lines, resulting in the file bit-vector 111, hunk bit-vector 1111, and line bit-vector 1111111111111111. | 22 |
| 4.2 | A generated diff with 2 files generated from the file bit-vector 011 applied on the diff in Figure 4.1. | 23 |
| 4.3 | A generated diff with 2 files and 2 hunks generated from the hunk bit-vector 0101 applied on the diff in Figure 4.1. | 23 |
| 4.4 | A generated diff with 2 files, 3 hunks, and 8 changed lines generated from the line bit-vector 00110101001001110 applied on the diff in Figure 4.1. | 24 |
| 4.5 | A diff applied with a bit-vector to transform a removed line into a context line, causing a line increase in the source code. | 25 |
| 4.6 | A diff applied with a bit-vector to remove an added line, causing a line decrease in the source code. | 25 |
| 4.7 | The directory structure. | 26 |
| 4.8 | Example of a script to parse a compilation log. | 27 |
| 4.9 | Example of a script to parse a test suite. | 27 |
| 4.10 | Initialization of individuals with standard initialization size and double initialization size. | 29 |
| 4.11 | A contradicting example of the Partially Mapped Crossover function with a population in the binary representation. | 30 |
| 4.12 | Example of when several bit-vectors, which reintroduce the bug, help reduce the required bits more than the current best bit-vector. | 32 |
| 4.13 | Inverse initialization. | 33 |
| 4.14 | Reduction of search space with inverse initialization. | 33 |
| 4.15 | Example of the focused adaptive mutation. | 34 |
| 4.16 | The design of the GA. | 35 |
| 4.17 | Minimization flowchart of a diff. | 36 |

| | | |
|-----|--|----|
| 5.1 | Selection-and Crossover function parameter test with the fitness value of the best individual per generation averaged out over the total number of runs. The parameter test was performed on the vulnerability from the OSS-Fuzz issue 56272 for the nDPI project. | 41 |
| 5.2 | Mutation function-and probability parameter test with the fitness value of the best individual per generation averaged out over the total number of runs. The parameter test was performed on the vulnerability from the OSS-Fuzz issue 56272 for the nDPI project. | 42 |
| 5.3 | Crossover probability parameter test with the fitness value increase per generation of the best individual per crossover probability averaged out over the total number of runs. The parameter test was performed on the vulnerability from the OSS-Fuzz issue 56272 for the nDPI project. | 43 |
| 5.4 | Population size and maximum generations parameter test with the fitness value of the best individual per generation averaged out over the total number of runs. The parameter test was performed on the vulnerability from the OSS-Fuzz issue 55218 for the nDPI project. | 44 |

List of Tables

| | | |
|------|--|----|
| 5.1 | Compile time, test time, time to build fuzzers, and implementation language for the selected projects. | 39 |
| 5.2 | Selected Vulnerabilities for nDPI. | 40 |
| 5.3 | Selected Vulnerabilities for mruby. | 40 |
| 5.4 | Selected Vulnerabilities for libxml2. | 40 |
| 5.5 | Selected Vulnerabilities for libdwarf. | 40 |
| 5.6 | Selected Vulnerabilities for hunspell. | 40 |
| 5.7 | Result of the GA from the initial diffs to the final diffs for all projects. | 45 |
| 5.8 | Result of the GA from the initial diffs to the file output diffs for all projects. | 45 |
| 5.9 | Result of the GA from the file output diffs to the coarse output diffs for all projects. | 46 |
| 5.10 | Result of the GA from the coarse output diffs to the final diffs for all projects. | 46 |
| 5.11 | Result of the automatically constructed test suites. | 46 |
| 5.12 | Reduction of code from the initial diffs to the final diffs for the merged diffs for all projects. | 47 |
| A.1 | Result of the GA from the initial diffs to the final diffs. | 57 |
| A.2 | Result of the GA from the initial diffs to the file output diffs. | 58 |
| A.3 | Result of the GA from the file output diffs to the coarse output diffs. | 59 |
| A.4 | Result of the GA from the coarse output diffs to the final diffs. | 60 |
| A.5 | Diffs which failed to be merged into their test suite. | 60 |



1 Introduction

1.1 Background

In the secure software development process, security testing is done to check the code for security bugs. This is often more difficult than regular software testing because most security bugs are not caught by the requirements of the software. They are most often the side effects of the coded application [1]. The application might perform the expected actions under testing, but it might also perform additional actions that were not caught by the tests or the tester.

A complement to make security testing easier is to automatically generate random or semi-valid data as input for the program and monitor the outcome. This concept is known as Fuzzing [2]. Fuzzing is used to check if an invalid or badly formatted input crashes a program. The most common security bugs fuzzing finds are memory corruption bugs, like buffer overflows [3]. An effect of fuzzing is that potential exploits are already caught in the testing phase, which can thus decrease the number of exploits available to an attacker.

Furthermore, due to its automation and scalability, fuzzing has become a popular approach in the industry [4]. This has led to the research and development of better and more efficient fuzzing techniques to generate better input to catch more bugs, which is currently a hot topic in the security testing community.

1.2 Motivation

A common approach to evaluate fuzzers is to use old programs with known bugs [5] to see if the fuzzers can find them. However, the evaluation of different fuzzers is not always done in the same way. Most tests are done on real-world programs, but their size and quantity vary greatly between fuzzing evaluations [5]. This makes it hard to draw any conclusions about the performance of one fuzzer compared to another.

Therefore, it is of great interest to develop a benchmark suite to evaluate the fuzzers on programs with several known real-world vulnerabilities to compare performance between fuzzers. However, when a fuzzed input crashes a program, it is not easy to know which bug caused the program to crash. Thus, a fuzzer might crash the program for 100 unique inputs, and there might be several different bugs causing the crashes. This makes it almost impossible to know how many bugs a fuzzer has found based on the number of crashes.

This problem with fuzzers makes it very tedious to find out which bugs the fuzzer triggered by executing the inputs again manually. This manual work is not needed if the benchmark suite has access to *ground truth*, an automatic way of determining which bugs the fuzzer triggered. Therefore, it is of great interest to develop a benchmark suite with ground truth to automatically determine which bugs the fuzzer actually triggered.

Although there has been work in reintroducing old bugs into newer versions of a program to evaluate a fuzzer's performance, the bugs were either reintroduced manually [6] or made use of synthetically generated bugs, which do not correctly represent real-world performance [7]. If the bugs were injected manually, it would be both tedious and drastically limit the number of bugs to support, and would be hard to extend and maintain. Also, if the fuzzers were to be evaluated on synthetic tests, advancements in fuzzers might make them good at detecting synthetic bugs instead of real-world bugs. In order to get a better real-world evaluation of fuzzers, real-world vulnerable code needs to be used instead. Therefore, the focus of this thesis is to develop a method to automatically generate fuzzing benchmark suites with ground truth.

1.3 Aim

The goal of this thesis is to, in a controlled way, automatically reintroduce old known vulnerabilities into the latest version of a target program. This is done by generating several versions of the target program with only one unique vulnerability reintroduced in each version. All the versions are then merged into one program with the known vulnerabilities. Whenever an input causes a crash in the merged program, the versions with only one vulnerability can be run with the same input to determine which vulnerability caused the crash by looking at which version crashes. For this purpose, Google's OSS-Fuzz will be used to select a set of bugs for a few programs from Google's OSS-Fuzz database [8]. OSS-Fuzz includes an input that triggers the bug and a date for every bug, which helps to find the latest version before the bug was fixed. OSS-Fuzz was originally intended for security testing of Chrome Components but is now offered as a Continuous Integration (CI) fuzzing service for open-source projects that are critical global IT-infrastructures or have a large user base [9].

Furthermore, the thesis will make use of techniques inspired by automatic program repair, more specifically, a genetic algorithm (GA). A GA is a type of algorithm that finds solutions through an evolutionary process [10]. In contrast to these techniques for automatic program repair, which need to synthetically generate code [11, 12], this thesis will reintroduce old code into the latest version of a program. To achieve this, a diff is used between the version before the fix and the latest stable version of the program. By using diffs, the search space for reintroducing the bug becomes much smaller since no code needs to be generated. Concretely, the purpose is to establish a method that can minimize this diff in such a way that the bug is reintroduced, but the rest of the program runs correctly. To solve this problem, the following research questions need to be answered.

1.4 Research questions

1. How can GAs be used to minimize a diff?
 - a. How can a diff be represented for use in a GA?
 - b. How can diffs be evaluated to drive a GA towards better diffs?
2. How can several vulnerable versions of a program with only one known vulnerability be merged into one program with the same vulnerabilities?

1.5 Delimitations

The scope of this thesis is limited to developing the GA to minimize a diff. The collection of projects, versions, and diffs to be used will be done manually. Automation of this is not central to the project.



2 Theory

2.1 Fuzzing

Fuzzing is testing a program through the use of fuzzed inputs to test for unintended behavior. A fuzzed input is a semi-randomly generated input intended to supply the program with inputs outside the program's expected input space [13]. Fuzz testing uses fuzzing to find security bugs in a program. A bug oracle is used with the fuzzer to determine if a bug was triggered. Bug oracles usually come in the form of *sanitizers*, which work by applying program transformations to catch unwanted behaviors that do not normally terminate the program with a fatal signal. These sanitizers range from handling manipulation of memory addresses to undefined behaviors to control flow integrity to input validations [13].

When a fuzzer tests a program, it can make use of different amounts of semantic information from the program. Depending on the amount of information that is used, a fuzzer is divided into one of three categories of fuzzers; White-box-, Grey-box-, and Black-box fuzzer [13]. White-box fuzzers make use of very detailed information from the program, such as branches and execution information, to systemically test the program. Black-box fuzzers, on the other hand, barely use any information from the program except the output, treating the program as a black box. Gray-box fuzzers are in the middle and also make use of semantic information, but not to the same extent as White-box fuzzers, with more emphasis on approximated information to increase speed [13].

Furthermore, for complex programs, there might be long waits when initializing the program before the program can accept any input. If this initialization phase needs to be executed every time the fuzzer tests the program, there would be a lot of wasted time spent on initializing the program rather than actually testing. To circumvent this problem, some fuzzers take a snapshot of the memory when the program reaches a state where it accepts inputs and then uses this snapshot to reset the program after each execution on a fuzzed input to a state where it can accept inputs again, skipping the initialization phase. This approach is called *In-Memory Fuzzing* and can also be used to test a single part of a program (for example, an interface or an API) [13].

2.2 Fuzzing Evaluation

Fuzz testing has become a very popular choice for testing programs and has sparked much interest in researchers in how to improve fuzzers [5]. Due to the randomness of fuzzers, they are evaluated experimentally to account for different results between runs. Several benchmark test suites have been developed with different approaches in order to evaluate fuzzers on sample programs. These programs can be divided into four groups: programs with synthetically generated bugs, programs written with intentional bugs, real-world programs with reintroduced bugs, and old real-world programs with known bugs [14]. Currently, programs with synthetically generated bugs are the popular and preferred approach due to it being easier to automate the bug injection process. The most well-known benchmark test suites currently are LAVA-M [7], DARPA CGC [15], and MAGMA [6].

However, since these test suites use either synthetically generated bugs, intentional bugs, or manually reintroduced bugs, the fuzzers are evaluated on different performance aspects, which may change the way the fuzzers are optimized. For example, with a test suite with synthetically generated bugs, it could lead to fuzzers becoming optimized for synthetic bugs instead of real-world bugs. Synthetically generated bugs are bugs that, given a pattern of code, introduce bugs in a deterministic manner, resulting in very similar bugs for the same type of vulnerability, which is not always the case in real-world code [7]. Intentional bugs are small programs manually created with intentional vulnerabilities, which may also result in similar-looking bugs for a given class of vulnerabilities, because of human bias [15]. Manually reintroduced bugs are based on real-world bugs, which are reintroduced into the code manually, requiring much work to support many types of bugs, and are not that scalable [6].

When it comes to evaluating fuzzers on real-world programs, the programs are few and handpicked, which are then used to manually evaluate the fuzzers' performance to find bugs [5]. This approach is more tedious and requires more manual work than synthetic programs, which also limits the number of fuzzer runs and unique programs that can be used in the evaluation due to the amount of work required per fuzzer evaluation. These are also not complete test suites, such as the likes of LAVA-M and MAGMA, but merely known buggy real-world programs. A more automated approach for fuzzer evaluation on real-world programs would be a test suite similar to MAGMA, but without the need to manually introduce and verify bugs. A test suite to automatically construct a deliberately vulnerable program with old known, fixed bugs, which could automatically give ground-truth about which bugs were triggered during the evaluation. Ground-truth means that a bug can be uniquely identified and is the strongest metric compared to other metrics like Coverage Profiles and Stack hashes when evaluating a fuzzer [5]. The problem with other metrics is the uncertainty of what a unique bug is and how many unique bugs were triggered. Since this can not be determined precisely without ground-truth through manual or automatic inspection of code, these metrics use approximation based on the inputs and the program's response.

2.3 OSS-Fuzz

OSS-Fuzz is a Continuous Integration (CI) fuzzing service for select open-source programs that have a large user base or are important for the global IT infrastructure. To use OSS-Fuzz for the project, OSS-Fuzz must know where to test and how to build the project. To specify where OSS-Fuzz should test the project, a *fuzz target* must be implemented in the source code for every place OSS-Fuzz should fuzz test. A fuzz target is a function in the source code used to allow fuzzers to inject input bytes into the program at a specific point in the code. Furthermore, OSS-Fuzz needs a *.yaml*-file with metadata for the project, a Dockerfile with project dependencies, and a *build.sh*-file to build the project and link it to the fuzzer [9].

However, for the project to be added to the list of projects associated with OSS-Fuzz, the owner of the project needs to create a pull request with the *.yaml* file to the OSS-Fuzz repository. Now, whenever a pull request is done, OSS-Fuzz will build and run the fuzz

targets, and report found bugs to the OSS-Fuzz issue tracker and to the owner of the project. After the bug is fixed, or if it is not fixed within 90 days, the report becomes public. In every report, the project name, fuzz target, and fuzzer engine used are included. Additionally, each report also includes information on which state the program crashed and a test case with the input bytes that caused the crash if the crash was reproducible. The fuzzer engines used by OSS-Fuzz are three gray-box fuzzer engines; libFuzzer, AFL++, and Honggfuzz [9].

2.4 Genetic Algorithms

Genetic algorithms (GAs) are used to find a solution to optimization problems by searching through the search space by simulating an evolutionary process. This means that current solutions influence the new solutions in the same way natural selection does in evolution. This evolution of solutions is repeated until a solution to a problem is found. To achieve this, a balance between the exploitation of current solutions and the exploration of the search space is essential [10]. At the start of a GA implementation, an initial population is usually generated in the form of strings with a fixed length, but other representations exist. Each iteration in a GA is called a generation and consists of a set of individuals, strings, which are potential solutions to the problem. The best individuals are modified each generation to produce better individuals for the next generation. Each generation cycle consists of three kinds of functions; the selection, crossover, and mutation functions. The selection function determines which individuals to use for creating the next generation. The crossover function uses the individuals selected by the selection function to create new individuals for the next generation. The mutation is responsible for adding a little diversity to the new generation by applying small changes to the generation generated by the crossover function [10].

Representation of Individuals

Most problems have a representation that is hard to model in GAs. In order to alleviate this translation problem, some common encoding schemes are used to represent the problem domain in the GA. The most common encoding schemes are binary-, permutation-, value-, and tree-representations [16]. In the binary, permutation and value representations individuals are represented as strings of symbols where binary is by far the most popular encoding scheme. Obviously, the binary representation consists of 1's and 0's, while the value represen-

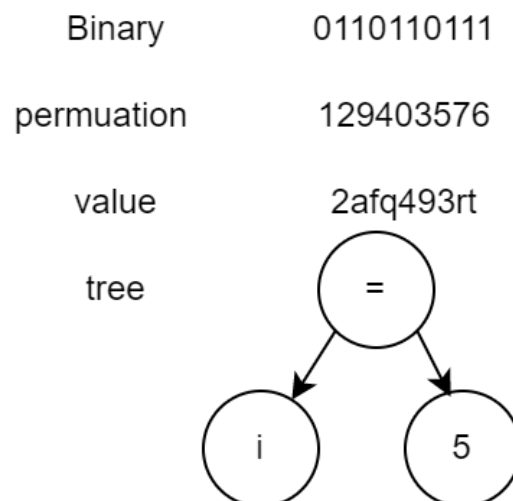


Figure 2.1: Different types of representations.

tation consists of numbers and characters. Value representations are usually used in problem domains that require a more complex representation than a binary representation [16]. The permutation representation is similar to the value representation in that it uses numbers to indicate positions in a sequence. This representation is mostly used in ordering problems. Tree representations are constructed with commands and functions in the form of an Abstract Syntax Tree (AST), see Figure 2.1, a semantic representation of a program, and are usually used when representing program code or expressions for altering programs [16].

Objective Function

The objective function is used to evaluate how good an individual is, i.e., how close the individual is to the optimal individual. This function is used in the selection function to determine the best individuals in a generation [10].

Fitness Function

The fitness function is used to determine how good an individual is compared to other individuals in the population. This function is usually calculated with the help of the objective function and is sometimes used instead of the objective function in the selection function [10].

Selection Function

The GA needs to preserve the good qualities among the population and discard the bad in a fixed population size in order to converge to an optimal or near-optimal solution. This process is done through the selection method of a GA and determines the parents for the next generation [17]. There are several selection methods; the common methods are Proportionate Roulette Wheel Selection, Ranking Selection, Stochastic Remainder Selection, Stochastic Universal Selection, Tournament Selection [17, 18, 19].

Proportionate Roulette Wheel Selection

Proportionate Roulette Wheel Selection (PRWS) is based on simulating a roulette wheel with each section's size proportional to an individual's fitness value, which is determined by dividing an individual's objective-function value by the sum of all individuals' objective-function values. The "wheel" is "rotated" n times, see Figure 2.2, where n is the population size, and thus selects the parents for the new generation [17]. Depending on implementation strategies, this selection method varies in time complexity. When linear search for the chosen individual is used, the time complexity is $O(n^2)$ since, on average, half of the list will be searched n times. The time complexity can be reduced to $O(n \log n)$ if binary search is used instead of linear search [18].

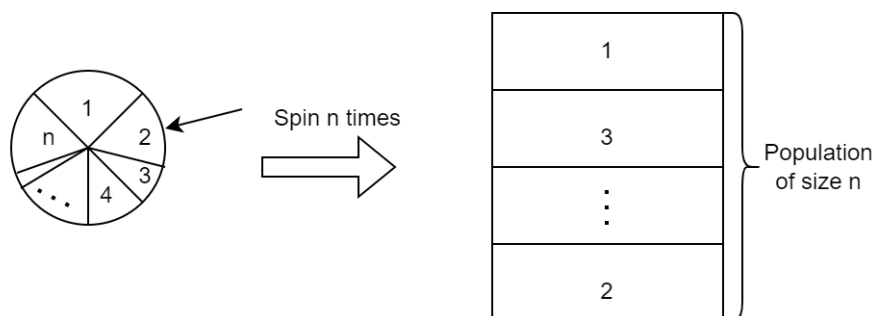


Figure 2.2: Proportionate Roulette Wheel Selection with population size n .

Stochastic Remainder Selection

Stochastic Remainder Selection (SRS) is done through the same fitness calculation as done in PRWS, where the integer value of the fitness value corresponds to the number of copies that are selected of an individual. The decimal remainder is used to indicate the chance of an additional copy of an individual being selected to fill out the parent population [10, 18]. Filling out the remaining parent population can be done by picking an individual with or without replacement. In the event replacement is used, the individuals are chosen through PRWS, and thus, the time complexity is determined by the time complexity of the PRWS implementation. Without replacements, a single search through the list is enough, with a biased coin flip determined by the individual's probability. The complexity of this variation is in $O(n)$ [18].

Stochastic Universal Selection

Stochastic Universal Selection (SUS) is related to PRWS, where the "wheel" is created in the same way. But instead of one pointer, there are as many pointers as the size of the population spaced evenly around the wheel. Instead of several spins as done in PRWS, one spin is enough to determine the parent population by counting the number of pointers landing in a specific section of the wheel, see Figure 2.3. Since this corresponds to a single search through the list, the time complexity is in $O(n)$ [10, 18, 20].

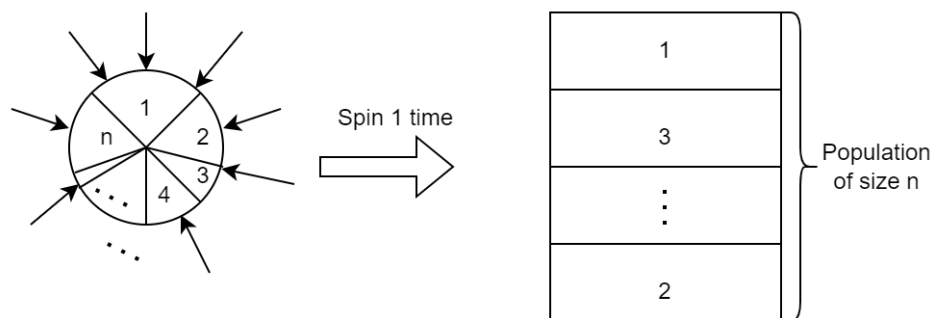


Figure 2.3: Stochastic Universal Selection with population size n .

Ranking Selection

Ranking Selection (RS) assigns the ranks N to 1 from the best to the worst individual in the population, where N is the population size. RS consists of a sorting phase followed by a ranking phase. The sorting phase sorts the individuals based on their fitness value. The ranking phase assigns the ranks to the sorted list. The individuals are then selected in the same way as in PRWS, with the ranks determining the sizes on the wheel, where a higher rank gives a larger size on the wheel. Due to the best known sorting algorithm, the time complexity of RS has a lower bound in $O(n \log n)$ [18, 17, 20].

Tournament Selection

Tournament Selection (TS) places m randomly selected individuals in a tournament where the individual with the best fitness value gets placed in the parent population, see Figure 2.4. The number m determines the size of the tournament, where the most popular size is $m = 2$. Since each tournament can be completed in constant time and n tournaments are needed to fill the parent population, the time complexity of TS is in $O(n)$ [17, 18, 21].

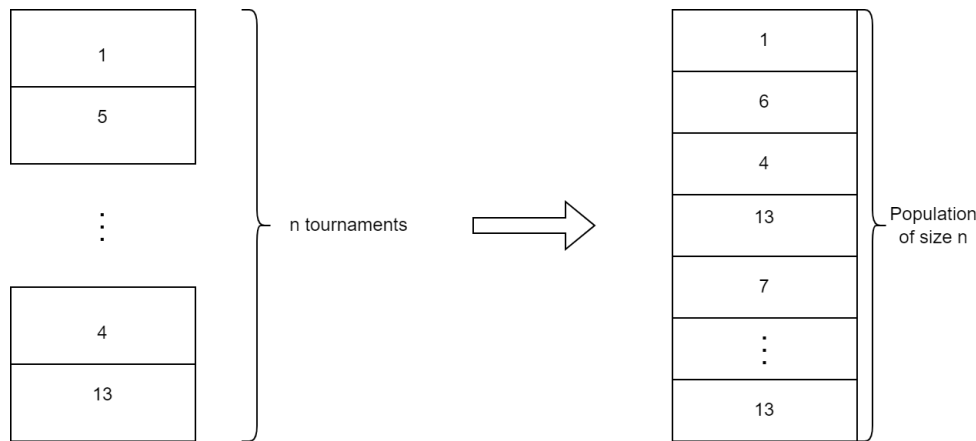


Figure 2.4: Tournament Selection with size $m = 2$ and population size n .

Comparison of Selection Functions

Depending on the performance constraints and desirable features for the genetic algorithm, different selection functions work better than others. One important feature that is almost essential for a good GA is a steady pressure towards convergence. This feature is present in almost all selection functions except the ones where variety and randomness play a bigger part. The selection functions TS and RS have good steady convergence pressure [17] while the other selection functions have a little weaker or more unstable convergence pressure.

Another important aspect of a GA is how good it is at selecting the individuals for the next generation in order to reach the maximum fitness in the minimum number of generations. The selection functions that are best at this are TS and PRWS (and SRS and SUS) [17]. When it comes to time complexity, most selection functions are either in $O(n)$ or $O(n \log n)$ [17] since the entire next population needs to be selected, and some selection functions do some sorting or some other process on the list of individuals.

Additionally, diversity among the individuals is important in order for the GA to achieve good fitness values. Without diversity, where only the best individuals are kept for the next generation, they might not have some feature that some individuals with a worse fitness value have. With these features now missing from the population, the only way to introduce them again is through the mutation function. The selection functions that are good at preserving diversity are TS and RS, where RS, along with PRWS, are free from bias for the better-fit individuals. However, TS also suffers from loss of diversity in larger tournament sizes, and PRWS has a greater risk of premature convergence [16].

Crossover Function

The crossover function creates the new generation from the parent population picked by the selection function. The main idea of the crossover function is the recombination of two individuals to create two new individuals. The most common crossover functions are K-point crossover, Reduced Surrogate Crossover, Uniform Crossover, Shuffle Crossover, and Partially Mapped Crossover [16]. The crossover function may be utilized with a crossover rate to determine if the crossover function will be used on a particular pair of offspring. The optimal value of the crossover rate lies between 60% and 100%. [22].

K-point Crossover

The K-point Crossover function randomly selects k points in the parent individuals to be used when mixing the parents. The most common value is $k = 1$, where one random point is

selected, and the parents switch the substring after the selected point, see Figure 2.5. When $k > 1$ points are chosen randomly the switching of substrings are alternated between the crossover points, see Figure 2.6. Due to its simple implementation and concept, it is used with most representations and smaller problems, which it can be applied to [16].

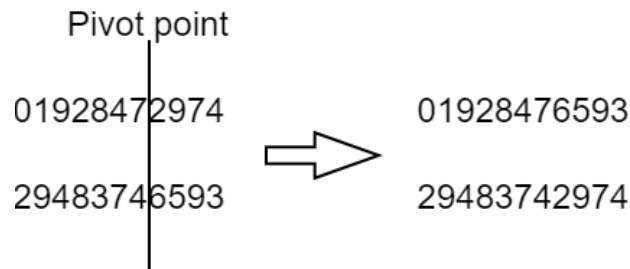


Figure 2.5: K-point Crossover with $K = 1$.

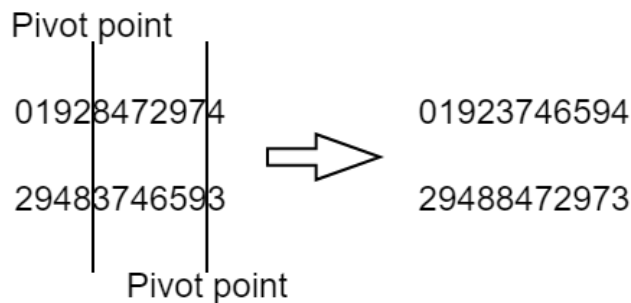


Figure 2.6: K-point Crossover with $K = 2$.

Uniform Crossover

Uniform Crossover is related to the K-point Crossover function, but instead treats each symbol in the parent individual as a separate segment. When creating the offspring, a coin flip is used to determine if a symbol will be swapped, see Figure 2.7, with the other parent individual or not [23]. The Uniform Crossover function is applicable to most representations and is mostly used for large sets of data [16].

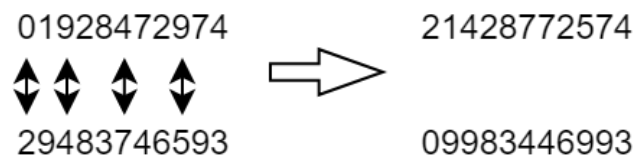


Figure 2.7: Uniform Crossover with 4 successful coin flips.

Reduced Surrogate Crossover

Reduced Surrogate Crossover only applies recombination between two parent individuals if the number of different symbols is more than 1, otherwise no operation is performed, and the

parent individuals live on till the next generation. When there are two parents with differing symbols, it applies 1-point Crossover randomly at one of the points on the individuals where the symbols differ [24]. This function is best when the two parent individuals are different, since it would keep diversity in the population and increase possible pivot points. However, if they are almost the same, the offsprings will be very similar to the parent individuals and could therefore very easily prematurely converge to a sub-optimal solution. This function is usually used with any representation except tree and is recommended for use on small optimization problems [16].

Shuffle Crossover

The Shuffle Crossover function shuffles the symbols in the two parent individuals before an arbitrary Crossover function, often the 1-point Crossover function, is applied to create the offsprings. After the crossover function, the two offsprings are shuffled in the opposite direction the parents were shuffled [24]. For example, if parent 1 had symbol 2 shuffled with symbol 3, then offspring 1 will have symbol 3 shuffled with symbol 2 after the crossover function. The use of this function is similar to the K-point Crossover function, but it has been quite limited in recent years [16].

Partially Mapped Crossover

Partially Mapped Crossover randomly selects a range in the individual where the two parents exchange a substring. These substrings are then used to map symbols between the first parent and the second parent. These mappings are then applied to all symbols that lie outside this substring range when copied to the offspring. If a symbol is not present in the mapping substring, it is simply copied without any modification [24]. For example, given the two parent individuals (3 9 1 5 6) and (2 7 6 4 5) with the range [3,4] would form the map $1 \iff 6$ and $5 \iff 4$ from the substrings (1 5) and (6 4). This would create the offsprings (3 9 6 4 1) and (2 7 1 5 4), see Figure 2.8. The Partially Mapped Crossover function is mostly used with the permutation representation for task ordering problems [16].

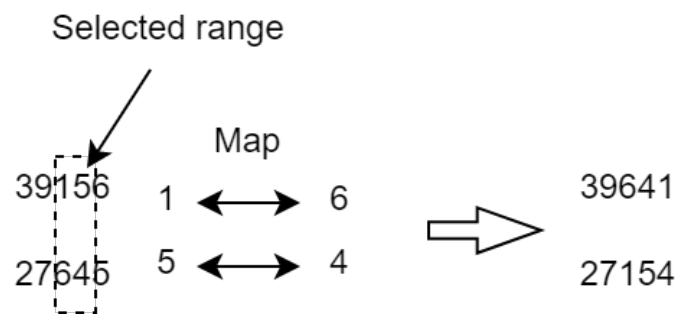


Figure 2.8: Partially Mapped Crossover with 2 mappings.

Comparison of Crossover Functions

The crossover function is an integral part of a GA in order to reach good solutions. However, some crossover functions are representation and problem-dependent, and other crossover functions could cause worse performance if not configured properly. The most common crossover function, the K-point crossover function, is easy to implement but could lead to less diverse offspring due to its simple recombination concept [16]. Additionally, a study by Kora and Yadlapalli [25] has shown that more pivot points than 2 actually have diminishing returns regarding offspring diversity because the extra pivot points do not add much more

mixing when recombining the parents. It instead may provide a more consistent start and end among the offsprings [25].

Another crossover function that also suffers from less diverse offsprings is the Uniform crossover function. But it instead has an unbiased exploration since each symbol has an equal chance to exchange with the other parent, which offers a better recombination potential than the k-point crossover function [16]. On the other hand, the Reduced Surrogate crossover function suffers from premature convergence due to it not being able to alter very similar individuals and thus risking converging to a sub-optimal solution. It is, however, good for small optimization problems where the risk of premature convergence is minimal [16].

One crossover function that stands out from the rest is the Partially Mapped crossover function, which, unlike the rest, does not have any drawbacks. It generally performs better than any other crossover function and has a better convergence rate. One inconvenience with the Partially Mapped crossover function is that it is mostly limited to the permutation representation of individuals [16]. Lastly, the Shuffle crossover function is just a shuffle applied before an arbitrary crossover function, and thus its goal is to reduce the introduced bias from the other crossover functions [16].

Mutation Function

As a final touch, a mutation function can be applied to the offsprings after the crossover operation to introduce some diversity in the population [10]. This would ensure that there would be a large enough diversity to prevent the GA from converging to a sub-optimal solution. The mutation function is always accompanied by a mutation rate to determine if the mutation function will be used on a particular offspring in order to reduce the risk that the GA becomes a random search algorithm. The optimal values for the mutation rate are typically low, around 0.1% to 5% [22].

Simple Inversion Mutation

The most common and widely used mutation function is the Bit-flip Mutation function, where each symbol in an offspring is inverted with a low probability p_m [10], see Figure 2.9. This function is, however, limited only to the binary representation. A more general inver-



Figure 2.9: Bit-flip Mutation with 3 successful weighted coin flips.

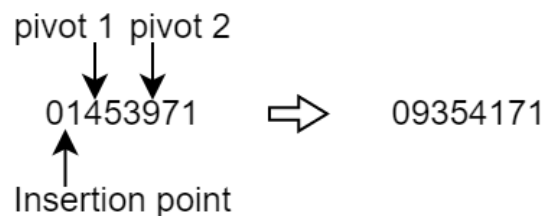


Figure 2.10: Simple Inversion Mutation with a selected segment of length 4 inserted at position 2.

sion function is the Simple Inversion Mutation function, where a randomly selected segment of the offspring is reversed in order and inserted at a random place in the offspring [16], see Figure 2.10.

Displacement Mutation/Scramble Mutation

The Displacement Mutation function selects a random segment of the offspring and either inserts the segment at a new place in the offspring, see Figure 2.11, or exchanges the segment with an equally long segment at the place in the offspring, see Figure 2.12. The Scramble Mutation function instead scrambles the selected segment without displacing it [16], see Figure 2.13.

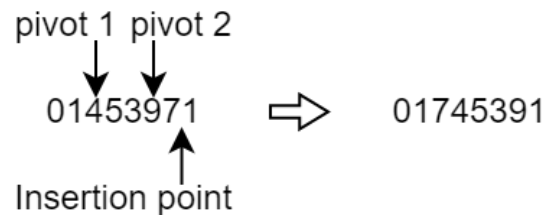


Figure 2.11: Displacement Mutation with a selected segment of length 4 inserted at position 8.

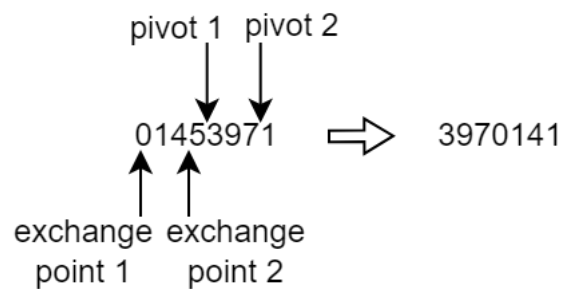


Figure 2.12: Displacement Mutation with an exchange of segments of length 3.



Figure 2.13: Scramble Mutation with a selected segment of length 4.

Comparison of Mutation Functions

The mutation functions do not mostly depend on the representations of the individuals, except for the Bit-flip Mutation function, which requires the binary representation. There are, however, certain representations where the different mutation functions perform the best. For the Simple Inversion Mutation function, binary and permutation representations work

the best, while the Displacement Mutation function works best with the value representation. When it comes to the Scramble Mutation function, the tree representation is preferred [16].

Furthermore, these mutation functions also have a small impact on the performance of the GA. The Simple Inversion Mutation function might cause premature convergence due to the simple nature of the mutation, but it is very easy to implement. The Displacement Mutation function also has a risk of premature convergence, but it is much better suited to smaller problems where the problem of premature convergence is less of a problem [16]. Lastly, the Scramble Mutation function is better on larger problems but might cause disturbance in the population. This is because the GA could have been close to a global optimum but the mutation function scrambled the better offsprings and now it will take more generations to reach the optimum [16].

Variants

The GA has some general variations where the fundamental algorithm cycle is altered to change the convergence speed and diversity in the population [26].

Elitist Recombination GA

The elitist recombination GA differs from the standard GA since it combines the selection function and the crossover function into a single operation with no mutation function after the combined function [26]. Similar to the other selection functions, the elitist recombination GA uses two parent individuals to create two offsprings. However, instead of discarding the parents for the offsprings, this is only done if the offspring have a better fitness score than the parent. Otherwise, the parent lives on till the next generation [26]. This ensures that the best possible individual is never discarded and that the fitness value in the next generation is better than or equal to that of the previous generation. This speeds up the convergence of the GA and increases the risk for premature convergence. To lower this risk, elitist recombination is done with a Tournament Selection of size 2 [26]. This approach preserves some diversity in the population since the best of the worst individuals are kept instead of being discarded.

Adaptive GA

The adaptive GA is very similar to the standard GA except for its ability to change its parameters based on the state of the population. A common problem with the standard GA is the slow convergence when the population is close to an optimum due to the concentrated distribution of fitness values [27]. The adaptive GA mitigates this by keeping the population's diversity relatively sparse. This is done by changing the probabilities as well as the order of the crossover and mutation functions based on whether the population is close to an optimum or not [27]. By increasing the probability that an individual's symbols are mutated and exchanged, the convergence rate can be increased, since more exploration is done near an optimum. This exploration would then be exploited in the crossover function to pull the population closer to the optimum.

Hybrid GA

The hybrid GA is almost identical to the adaptive GA since it also focuses on speeding up convergence around an optimum. The difference for the hybrid is the help of application-specific local search algorithms to help guide the population towards the optimum [28].

Convergence of Genetic Algorithms

Several parameters determine how a GA will behave when executing, and if it will converge at all or keep going forever. The GA needs to be finely tuned to be able to achieve the desired outcome.

Selection Intensity

The selection intensity of a GA determines how much emphasis is put on selecting the best offsprings from a population. The selection intensity naturally decreases when the population is closing in on an optimum since the variance of the fitness value distribution is decreasing [19].

Premature Convergence

Premature convergence in GAs is more likely when the selection intensity is too high, thus limiting the diversity in the population, causing the convergence to a sub-optimal solution. To reduce the risk of premature convergence, the selection intensity of the selection function needs to be balanced to preserve some diversity in the population to give the GA an opportunity to "jump" out of a sub-optimal solution. The mutation function also helps to ensure some diversity in the population [19]. Another parameter which could cause the GA to converge prematurely is a too small population used during execution, since the population would be too small to have a big enough diversity [26].

Stopping Criteria

Due to the stochastic nature of GAs, convergence to a global optimum is not guaranteed. To make sure the GA does not run forever, a common method is to limit the execution time to a specific number of generations [29]. This limit varies depending on the hardness of the problem and the chosen selection, crossover, and mutation functions for the GA. Another approach to establish a stopping criteria is to monitor the change in fitness value between generations [29]. When this change is below a certain threshold, the GA is halted.

2.5 Automatic Program Repair

Automatic Program Repair (APR) is the generation of code changes to fix bugs while maintaining the same functionality of the original program. This has become a very interesting research area in recent years due to the immense amount of resources being spent on fixing bugs in shipped software [11]. The use of APR patches for fixing these bugs could be generated automatically to reduce the time developers spend on debugging.

Fault Localization

Fault Localization is a technique used to find the code that is causing the bug, to limit the search space for the potential changes the algorithm might try. The main idea is to mark the lines which has potential of being the root cause of the bug and focus the algorithm on these lines [11].

2.6 Version Control

Version control is a way to keep track of changes in a file. Git [30] is a well-known distributed version control system used to track changes to source code for software development.

Commits

A commit is a record of changes made to one or several files [31] used to track versions of files. A commit contains changes made to the files since their last version, as well as who made the changes, along with a date.

Diffs

A diff shows the changes between two versions of a file. Added lines starts with a "+" and removed lines starts with a "-". The changes in the file are divided into *hunks* instead of showing the entire file, since changes can occur at many places in the file. The start of a hunk is indicated by the line @@ -{old-start}, {old-lines} +{new-start}, {new-lines} @@ [label] where "label" is optional, indicating the context of the lines [32], see Figure 2.14. Many Linux operating system distribution come installed with the GNU diffutils module

```
diff --git a/main.c b/main.c
index d309225..31ada71 100644
--- a/main.c
+++ b/main.c
@@ -6,7 +6,9 @@ int factorial(int i) {
 }

 int main(int argc, char const *argv[]) {
- int i = 5;
+ if (argc > 2)
+   exit(-1);
+ int i = stoi(argv[1]);
  printf("%d\n", factorial(i));
  return 0;
 }
```

Figure 2.14: Example output from `git -diff` showing the file `main.c` starting at line 6 in the function `factorial` with 7 lines from the old version and 9 lines from the new version.

which contains a local command line tool to see differences between two files on the system. This tool can be invoked with the command `diff` followed by some optional arguments and the two files for comparison [33]. This tool outputs in the same format as Git's own `diff`. What makes Git more useful and convenient is the support for differences between more than two versions of a file, as well as between two commits in the repository invoked directly in the project's Git directory. Git's `diff` also supports tracking changes between two versions of a file even if the file name has changed [32]. In GNU `diffutils`, this would not be possible since it does not track the files but only looks at the difference when invoked. Therefore, the file with the changed name would be shown in the output as an entirely new file with only added lines, while it only changed name [33].

Patches

A git patch is a diff output between two commits, which can be used to automatically apply the changes between the commits to the source tree. A patch file can be applied to the source code with the command `git apply {patch-file}` and the changes can be reverted with the command `git apply -R {patch-file}` [34]. A patch-file can be created with the command `git diff {commit1} {commit2} > {patch-file}.patch` which will contain the changes from `commit1` to `commit2`. The patch file can be modified to only contain changes in certain files with the flag `:{file}` and exclude changes in certain files with the flag `!{file}`. These flags can also take a regular expression to include/exclude changes in files matching the regular expression [34].



3 Previous Work

3.1 Fuzzing Evaluations with reintroduced Bugs

Work previously done in this area relied on either synthetically generated bugs or manual insertion and verification of bugs to determine if the test suite had ground truth or not. The work done by Hazimeh, Herrera, and Payer, MAGMA, used real bugs from popular open source projects, which were manually inspected and introduced into the latest version of a program [6]. Also, after each bug, a so-called canary was inserted to collect information to evaluate if the bug was *reached*, *triggered*, or *detected* by the fuzzer. Out of the 118 unique bugs in MAGMA, the authors managed to verify 74 as reachable and 43 as triggerable [6]. Since the search and injection of bugs is done manually, it would require much work to extend MAGMA to support more bugs.

Another but related approach was done by Dolan-Gavitt et al. [7] with LAVA, which was more automated than MAGMA but instead used synthetic bugs instead of real bugs. These bugs were injected after places in the code where the input does not affect the execution flow and where the input has not been altered much. LAVA supported around 2 million bugs in total for four open-source programs, but due to the time taken to find a target to inject a bug and potentially re-compile, only a subset were evaluated. LAVA's ability to inject bugs was evaluated by injecting around 2 000 bugs in total on the four supported programs, which were later run by a fuzzer and symbolic execution with SAT solving. The highest success rate of finding bugs was around 25% for the fuzzer and around 20% for the symbolic execution with SAT solving. The lowest success rate was 0% for both techniques on the four programs [7]. This result for the fuzzer was probably related to the size of the program and how the mutation of the input bytes did not trigger many bugs. The result of the symbolic execution with SAT solving depended on at what depth in the program the bugs resided and the control flow of the program.

3.2 Automatic Program Repair

Clair Le Goues et al. used genetic programming on the AST in the work GenProg [11] to automatically repair programs with the help of both passing and failing test cases. Due to the very large search space of potential code statements, GenProg limited the code generation to only use code statements already present in the program to limit the search space. No

new code was ever generated, only copied from other places in the code, moved around, or deleted [11]. GenProg used a fitness function constructed of a weighted average of passing and failing test cases of both functional requirements and vulnerability-specific tests. This was used to help localize the faults and guide the genetic operators to modify the statements that triggered bad behaviour. The parameters used by GenProg were a population size of 40, run for a maximum of 10 generations with a maximum mutation probability of 6% for a given statement. Convergence time for GenProg took on average 356 s with a worst case over 2 000 s to automatically repair 8 unique types of bugs successfully 77% of the time over 1.2 million lines of code [11]. GenProg's convergence time and success rate were quite resistant to modification of the parameters, except for the mutation probability, which, when above 12%, worsened the success rate.

Another work done by Eric Schulte et al. [35] automatically modified assembly code to fix bugs with a genetic algorithm to be able to fix programs regardless of the original programming language of the source code. The individuals were represented as strings of x86 assembly codes or Java bytecode, where the codes were then copied, deleted, swapped, and recombined to regenerate new individuals [35]. The individuals used in the crossover-and mutation function were selected by tournament selection of size 3. The fitness function was a weighted sum of positive and negative tests, where the negative tests were weighted more. An individual that failed to compile or link was assigned a fitness value of 0 [35]. A population size of 40 with a maximum of 10 generations was used, but the authors recommended a much larger population size and number of generations to maximize the effectiveness of repairs on the assembly level. With the population size of 40, the success rate was on average 34% but with a larger population, the success rate became much better. One test rose from 1% success to 21% with the larger population size and more generations [35].

3.3 Genetic Algorithms

Congrui Yang et al. [27] proposed an improved adaptive genetic algorithm, which did more than only switch the order of operators and alter the crossover and mutation probabilities based on the state of the population. The proposed algorithm also employed an elitist strategy to keep the best individual between generations. Without the elitist strategy, the increased crossover and mutation probabilities would destroy the good individuals and decrease the probability of convergence [27]. The proposed algorithm was tested against 3 other genetic algorithms in terms of convergence times, number of generations, and stability. All the GAs used the PRWS selection method, the 1-point crossover method, and the bit-flip mutation function. The 3 GAs used as comparison were the standard GA with no adaptive features and two adaptive GAs. The population size used was 100, the maximum number of generations was 450, with a crossover probability of 60% and a mutation probability of 1% [27].

The test was run 50 times on different continuous variable functions in one dimension, where the proposed algorithm outperformed the rest of the GAs with a convergence in 36, respectively 30 generations in the two tests, with a 100% convergence rate on both tests. It was also the only GA to find the theoretical optimal value of both tested functions. Furthermore, the standard GA with no adaptive features converged in 197, respectively 135 generations, with only a 62%, respectively 52% convergence rate. Additionally, the first adaptive GA converged in 95, respectively 83 generations with a 68%, respectively 82% convergence rate. The second adaptive GA converged in 52, respectively 48 generations with a 92%, respectively 100% convergence rate [27]. From these tests, it is clear that even tweaking the crossover and mutation probabilities or changing the order of operations adaptively can greatly reduce the number of generations needed to converge and increase the convergence rate. By preserving the best individual found so far between generations can also further greatly decrease the number of needed generations and increase the convergence rate.



4 Method

4.1 Overview

The method was divided into a pre-study, a design and implementation phase, and an evaluation phase. In the Pre-study, relevant previous research was gathered, and relevant vulnerabilities and projects were chosen to be used during the implementation. In the design and implementation phase, code to parse the projects' test suites' results was developed to be used in the evaluation of the diffs in the GA implementation. The design of the GA was also established. To get the best performance possible from the GA, a few key parameters were adjusted in a parameter test. In the evaluation phase, a few other projects were used to test that the implementation was not biased towards the vulnerabilities used to develop the GA.

4.2 Pre-study

The pre-study consisted of searching on OSS-Fuzz's Database of security vulnerabilities in open-source projects [8] for projects that were suitable to use for reintroducing bugs. Five projects were picked for this study to be used in the design and implementation phase as well as in the evaluation phase. The selected projects for this study were nDPI, a deep-packet inspector [36], mruby, a lightweight Ruby implementation [37], LibXML2, an XML toolkit [38], Libdwarf, a library for handling DWARF debugging data formats [39], and hunspell, a fast, high-quality spell checker [40]. The project nDPI was used to guide the design and implementation of the GA, while the remaining 4 were used to determine that the GA was not biased towards the first project. The project nDPI was picked to guide the development since the hunks included some dependencies between each other, but were also, in some cases, mutually exclusive. This made it great to test with for different approaches. The sizes of the diffs were also of moderate size, which reduced the time between test runs. For each project, 5 vulnerabilities with reproducible test cases were chosen from OSS-Fuzz's database. These vulnerabilities were then used to create diffs between the latest stable version of the project and the commit just before the specific vulnerability was fixed for that project.

Project & Vulnerability Selection Criteria

The projects were picked based on the criteria that the bugs reported were fixed, that the project had a reproducible test case generated by OSS-Fuzz, that the combined time taken to compile and test the project was less than 1 minute, and that the time taken to build the project's fuzzers on one core was less than 1 minute. Projects with a total time over 1 minute taken to compile and test or build fuzzers were not considered, to reduce the time to compile and test and build fuzzers for every individual during the execution of the GA. Furthermore, if the bugs were not reproducible, they would be impossible to re-implement since it would be impossible to reliably check them. Finally, the criteria that the bugs must have been fixed was important. If a bug was not fixed and present in the latest stable release of the source code, it would not be possible to use it with the GA, because to find the minimal amount of code required to re-implement the bug, a fix for the bug had to be found first, which the GA was not designed to do. Additionally, due to OSS-Fuzz's bug retention policy, it would not have been possible to access the reproducible test case of the bug unless it was reported more than 90 days ago. If the bug was fixed within 90 days after it was reported, the reproducible test case would be accessible to the public 30 days after it was fixed, but at the latest after 90 days since it was reported.

The bugs listed on OSS-Fuzz's issue tracker were sorted based on the issue number in descending order and traversed in descending order from 30 days earlier (1 May 2023) to find projects with recent bugs. Only bugs of medium to high security severity were considered to focus on bugs that may cause more harm when exploited. To reduce the size of the initial diffs, the projects with vulnerabilities close to the 30-day reproducible test case disclosure and projects' disk space with less than 300 MB were prioritized. However, it was not always the case that all the different types of fixed vulnerabilities were close to 30 days old. Due to the many potential changes that can happen in the repository over time, all selected vulnerabilities were less than 12 months old. This was done to keep the diffs to manageable sizes and reduce the time taken to run the GA. All projects that did not have at least 5 bugs reported and fixed within 12 months and had a source code directory larger than 300 MB were discarded.

4.3 Design and Implementation

After the projects were collected, scripts for parsing the test suites for these projects were set up locally in order to collect the number of passed tests. A script for parsing the result of OSS-Fuzz was also set up to check that the vulnerabilities were successfully reintroduced. The number of passed tests, compilation results, and the length of the diff were used in the GA to evaluate the diff to produce as close to an optimal diff as possible. Additionally, some scripts to apply the diff to the source code, compile it, run the test suite, run OSS-Fuzz, and revert the diff were set up. These scripts were used in the GA, which was designed and performance-tuned, then run on the collected vulnerabilities one project at a time. When the reduced diffs for the vulnerabilities were found, they were merged into the latest version of the vulnerabilities' project. Additionally, in order to verify ground truth, every reduced diff was tested to make sure it did not introduce more than one known vulnerability before it was merged.

Genetic Algorithm

The most important part was the use of a good representation of the problem, which could be used with the GA. For this GA, diffs were used instead of operating directly on the code, as methods like GenProg [11] needed approximations and some tricks to have an acceptable search space. Operating directly on diffs limited the search space drastically, and no approximations on the diffs were needed. Using Git diffs was convenient, since all projects

on OSS-Fuzz have a Git repository and Git diffs were more expressive than GNU diffs. The diffs were created manually for each project and were used as templates for the GA during the initialization of the population. The diffs were used to apply changes to the project so each modification of the source code could be evaluated, run, and tested.

However, since the GA is a probabilistic selective optimization algorithm, it was not guaranteed that the GA had found a solution when it had stopped. In order to increase the probability of reaching an optimum, the GA had to have a steady pressure towards convergence, but also had to keep the population diverse to prevent premature convergence. The use of a good combination of the selection function and crossover function ensured a steady pressure towards convergence.

Collection of Initial Diffs

The GA developed, made use of *git diff* to get a diff between two commits and created a patch file with `git diff {latestStableCommit} {commitBeforeBugFix} ':!{binaryFiles}' ':!{tests}' > {patch-file}.patch` from the diff. All changes in binary files, e.g. *.pdf* or *.pcap*, were removed from the patch file since git was unable to apply changes to these files when the patch was applied. The changes in the tests were also excluded in order to find reduced diffs where the diff passed all the latest tests. Changes in the projects' build systems and fuzzer configurations were kept in order to increase the number of compiled individuals and reduce the chance of premature convergence. If the changes in the projects' build systems were excluded, very few individuals compiled, and made it impossible to find any good individuals to use in the GA, and resulted in a random search.

The OSS-Fuzz issue tracker [41] was used to search for fixed vulnerabilities found by OSS-Fuzz. It was possible to deduce the time the vulnerability was reported as fixed through the comments on the issue tracker. Based on the time mentioned in the commits and the time the vulnerability was reported as fixed in the issue tracker, it was possible to reduce the number of relevant commits. The commits were checked out, compiled, and reproduced with OSS-Fuzz, and their associated reproducible test case to find the latest commit containing the bug. The GA would, of course, work with any diff that was guaranteed to have the bug present. This was done to minimize the potential overhead of source code that was not needed to re-implement the bug and to reduce execution time. Some commits made this process very easy by mentioning the OSS-Fuzz Issue number in the commit message.

However, the majority of commits did not mention the OSS-Fuzz Issue number in the commit message. The approach to find the latest commit containing the bug was then to checkout a commit just after it was reported and a commit around the time it was reported to be fixed, and apply a binary search to get the commit responsible for fixing the bug. For some projects, the reported dates on the OSS-Fuzz Issue Tracker had an offset of a few hours to a few days later compared to when the commit was published. This was remedied by checking out commits some time before the reported dates on OSS-Fuzz Issue Tracker when no commits during the reported date range resulted in a found bug. If no commits could be found to re-implement the bug, it was replaced with another bug from the OSS-Fuzz Issue Tracker. This meant that the bug report most likely was a result of a bug in a third-party library used by the project, or that the bug could not be reproduced reliably locally.

Limitations on Diffs

The Diffs used in the GA had to be able to compile and reintroduce the bug when the whole diff was applied in order for the GA to function properly.

Representation of Individuals

Each diff was used to create a patch file. The patch file was then represented as a boolean array representation (or bit-vector) for use in the GA, which was a recommended representation for GAs [10], so it could be modified on a file-by-file, hunk-by-hunk, or line-by-line basis. This meant that if a symbol in the bit-vector was a "1", then that specific line, hunk, or file was included in the generated diff, otherwise it was not. The representation was divided into three categories; file, coarse, and fine-grained. The file version was a boolean array representation of the files in the diff in order to find the necessary files to re-implement the bug. The coarse version was also a boolean array representation of the hunks in the diff found by the file version in order to find the hunks that were needed to re-implement the bug. The

| | Bit-vector | | |
|---|------------|---|---|
| | F | H | L |
| diff --git a/mrbgems/mruby-proc-ext/src/proc.c b/mrbgems/mruby-proc-ext/src/proc.c index cf0989744..0ef754410 100644 --- a/mrbgems/mruby-proc-ext/src/proc.c +++ b/mrbgems/mruby-proc-ext/src/proc.c | 1 | | |
| @@ -51,13 +51,16 @@ proc_inspect(mrb_state *mrb, mrb_value self) int32_t line; mrb_str_cat_lit(mrb, str, " "); | | 1 | |
| - if (mrb_debug_get_position(mrb, irep, 0, &line, &filename)) { | | | 1 |
| - mrb_str_cat_cstr(mrb, str, filename); | | | 1 |
| - mrb_str_cat_lit(mrb, str, ":"); | | | 1 |
| + filename = mrb_debug_get_filename(mrb, irep, 0); | | | 1 |
| + mrb_str_cat_cstr(mrb, str, filename ? filename : "-"); | | | 1 |
| + mrb_str_cat_lit(mrb, str, ":"); | | | 1 |
| + | | | 1 |
| + line = mrb_debug_get_line(mrb, irep, 0); | | | 1 |
| + if (line != -1) { | | | 1 |
| mrb_str_concat(mrb, str, mrb_fixnum_value(line)); | | | 1 |
| } | | | |
| else { | | | |
| mrb_str_cat_lit(mrb, str, "-:-"); | | | 1 |
| + mrb_str_cat_lit(mrb, str, "-:-"); | | | 1 |
| } | | | |
| } | | | |
| diff --git a/mrbgems/mruby-rational/src/rational.c b/mrbgems/mruby-rational/src/rational.c index a54324f47..3b13fb57c 100644 --- a/mrbgems/mruby-rational/src/rational.c +++ b/mrbgems/mruby-rational/src/rational.c | 1 | | |
| @@ -769,7 +769,7 @@ void mrb_mruby_rational_gem_init(mrb_state *mrb) struct RClass *rat; | | 1 | |
| rat = mrb_define_class_id(mrb, MRB_SYM(Rational), mrb_class_get_id(mrb, MRB_SYM(Numeric))); | | | |
| - MRB_SET_INSTANCE_TT(rat, MRB_TT_UNDEF); | | | 1 |
| + MRB_SET_INSTANCE_TT(rat, MRB_TT_RATIONAL); | | | 1 |
| mrb_undef_class_method(mrb, rat, "new"); | | | |
| mrb_define_class_method(mrb, rat, "new", rational_s_new, MRB_ARGS_REQ(2)); | | | |
| mrb_define_method(mrb, rat, "numerator", rational_numerator, MRB_ARGS_NONE()); | | | |
| diff --git a/mrbgems/mruby-sleep/src/sleep.c b/mrbgems/mruby-sleep/src/sleep.c index 69ce5d756..d05bc1944 100644 --- a/mrbgems/mruby-sleep/src/sleep.c +++ b/mrbgems/mruby-sleep/src/sleep.c | 1 | | |
| @@ -40,7 +40,7 @@ | | 1 | |
| /* not implemented forever sleep (called without an argument)*/ static mrb_value | | | |
| -f_sleep(mrb_state *mrb, mrb_value self) | | | 1 |
| +mrb_f_sleep(mrb_state *mrb, mrb_value self) | | | 1 |
| { time_t beg = time(0); time_t end; | | | |
| @@ -71,7 +71,7 @@ f_sleep(mrb_state *mrb, mrb_value self) | | 1 | |
| /* mruby special; needed for mruby without float numbers */ static mrb_value | | | |
| -f_usleep(mrb_state *mrb, mrb_value self) | | | 1 |
| +mrb_f_usleep(mrb_state *mrb, mrb_value self) | | | 1 |
| { mrb_int usec; #ifdef _WIN32 | | | |

Figure 4.1: A parsed diff with 3 files and 4 hunks and 17 changed lines, resulting in the file bit-vector 111, hunk bit-vector 1111, and line bit-vector 11111111111111111.

```

diff --git a/mrbgems/mruby-rational/src/rational.c b/mrbgems/mruby-rational/src/rational.c
index a54324f47..3b13fb57c 100644
--- a/mrbgems/mruby-rational/src/rational.c
+++ b/mrbgems/mruby-rational/src/rational.c
@@ -769,7 +769,7 @@ void mrb_mruby_rational_gem_init(mrb_state *mrb)
     struct RClass *rat;

     rat = mrb_define_class_id(mrb, MRB_SYM(Rational), mrb_class_get_id(mrb, MRB_SYM(Numeric)));
-   MRB_SET_INSTANCE_TT(rat, MRB_TT_UNDEF);
+   MRB_SET_INSTANCE_TT(rat, MRB_TT_RATIONAL);
     mrb_undef_class_method(mrb, rat, "new");
     mrb_define_class_method(mrb, rat, "new", rational_s_new, MRB_ARGS_REQ(2));
     mrb_define_method(mrb, rat, "numerator", rational_numerator, MRB_ARGS_NONE());
diff --git a/mrbgems/mruby-sleep/src/sleep.c b/mrbgems/mruby-sleep/src/sleep.c
index 69ce5d756..d05bc1944 100644
--- a/mrbgems/mruby-sleep/src/sleep.c
+++ b/mrbgems/mruby-sleep/src/sleep.c
@@ -40,7 +40,7 @@
/* not implemented forever sleep (called without an argument)*/
static mrb_value
-f_sleep(mrb_state *mrb, mrb_value self)
+mrb_f_sleep(mrb_state *mrb, mrb_value self)
{
    time_t beg = time(0);
    time_t end;
@@ -71,7 +71,7 @@ f_sleep(mrb_state *mrb, mrb_value self)
/* mruby special; needed for mruby without float numbers */
static mrb_value
-f_usleep(mrb_state *mrb, mrb_value self)
+mrb_f_usleep(mrb_state *mrb, mrb_value self)
{
    mrb_int usec;
#ifdef _WIN32

```

Figure 4.2: A generated diff with 2 files generated from the file bit-vector 011 applied on the diff in Figure 4.1.

fine-grained version was also a boolean array representation of each line in the diff found with the coarse version to further optimize the final diff. The search space for this approach was 2^n where n is the number of files, hunks, or lines. Each version helped reduce the search space further and refine the necessary source code changes to re-implement the bug. To simply try to search for the necessary lines with the initial diff would result in a very low compile percentage, and consequently, a very low convergence rate for the GA due to the large search space.

```

diff --git a/mrbgems/mruby-rational/src/rational.c b/mrbgems/mruby-rational/src/rational.c
index a54324f47..3b13fb57c 100644
--- a/mrbgems/mruby-rational/src/rational.c
+++ b/mrbgems/mruby-rational/src/rational.c
@@ -769,7 +769,7 @@ void mrb_mruby_rational_gem_init(mrb_state *mrb)
     struct RClass *rat;

     rat = mrb_define_class_id(mrb, MRB_SYM(Rational), mrb_class_get_id(mrb, MRB_SYM(Numeric)));
-   MRB_SET_INSTANCE_TT(rat, MRB_TT_UNDEF);
+   MRB_SET_INSTANCE_TT(rat, MRB_TT_RATIONAL);
     mrb_undef_class_method(mrb, rat, "new");
     mrb_define_class_method(mrb, rat, "new", rational_s_new, MRB_ARGS_REQ(2));
     mrb_define_method(mrb, rat, "numerator", rational_numerator, MRB_ARGS_NONE());
diff --git a/mrbgems/mruby-sleep/src/sleep.c b/mrbgems/mruby-sleep/src/sleep.c
index 69ce5d756..d05bc1944 100644
--- a/mrbgems/mruby-sleep/src/sleep.c
+++ b/mrbgems/mruby-sleep/src/sleep.c
@@ -71,7 +71,7 @@ f_sleep(mrb_state *mrb, mrb_value self)
/* mruby special; needed for mruby without float numbers */
static mrb_value
-f_usleep(mrb_state *mrb, mrb_value self)
+mrb_f_usleep(mrb_state *mrb, mrb_value self)
{
    mrb_int usec;
#ifdef _WIN32

```

Figure 4.3: A generated diff with 2 files and 2 hunks generated from the hunk bit-vector 0101 applied on the diff in Figure 4.1.

Parsing and Generation of Diffs

The diffs used in the GA had to be parsed to both generate the bit-vectors for the GA and to generate new diffs from the generated bit-vectors. The diffs were first divided into the different files present in the diff. Whenever a file was found, the bit-vector was extended by one element, see Figure 4.1. If the GA was run in coarse-grained mode, the diff was also divided into the different hunks for each file. Whenever a hunk was found, the bit-vector was extended by one element, see Figure 4.1. Each file also kept track of how many bits of the bit-vector corresponded to which file. This was necessary when going from a bit-vector of hunks to a diff in order to know if a file should be included in the generated diff or not. When the GA was run in fine-grained mode, the diff was also divided on the lines in each hunk, see Figure 4.1. This meant that each hunk also returned to its file which lines were used in each hunk. If the hunk did not introduce any changes to the source code, the returned hunk was empty. If all hunks in a file were empty, the file was not included in the generated diff. The generated diff from the bit-vector, therefore, only included files and hunks which introduced a change in the source code, see Figure 4.2, Figure 4.3, and Figure 4.4.

When a diff was modified by removing hunks, an offset of $\{old-lines\} - \{new-lines\}$ was applied to the rest of the hunk-headers in the file to keep it a valid patch. This offset was modified with which lines were kept or not in the fine-grained version. If a line in the diff that removed a line in the source code (i.e., starting with '-') was removed from the patch, the

```
diff --git a/mrbgems/mruby-proc-ext/src/proc.c b/mrbgems/mruby-proc-ext/src/proc.c
index cf0989744..0ef754410 100644
--- a/mrbgems/mruby-proc-ext/src/proc.c
+++ b/mrbgems/mruby-proc-ext/src/proc.c
@@ -51,13 +51,16 @@ proc_inspect(mrb_state *mrb, mrb_value self)
     int32_t line;
     mrb_str_cat_lit(mrb, str, " ");

     if (mrb_debug_get_position(mrb, irep, 0, &line, &filename)) {
         mrb_str_cat_cstr(mrb, str, filename);
-        mrb_str_cat_lit(mrb, str, ":");
+        filename = mrb_debug_get_filename(mrb, irep, 0);
+        mrb_str_cat_lit(mrb, str, ":");
+        line = mrb_debug_get_line(mrb, irep, 0);
+        mrb_str_concat(mrb, str, mrb_fixnum_value(line));
     }
     else {
         mrb_str_cat_lit(mrb, str, "-:-");
+        mrb_str_cat_lit(mrb, str, "-");
     }
 }

diff --git a/mrbgems/mruby-sleep/src/sleep.c b/mrbgems/mruby-sleep/src/sleep.c
index 69ce5d756..d05bc1944 100644
--- a/mrbgems/mruby-sleep/src/sleep.c
+++ b/mrbgems/mruby-sleep/src/sleep.c
@@ -40,7 +40,7 @@
 /* not implemented forever sleep (called without an argument)*/
 static mrb_value
-f_sleep(mrb_state *mrb, mrb_value self)
+mrb_f_sleep(mrb_state *mrb, mrb_value self)
 {
     time_t beg = time(0);
     time_t end;
@@ -71,7 +71,6 @@ f_sleep(mrb_state *mrb, mrb_value self)

 /* mruby special; needed for mruby without float numbers */
 static mrb_value
-f_usleep(mrb_state *mrb, mrb_value self)
 {
     mrb_int usec;
 #ifdef _WIN32
```

Figure 4.4: A generated diff with 2 files, 3 hunks, and 8 changed lines generated from the line bit-vector 00110101001001110 applied on the diff in Figure 4.1.

offset was adjusted by adding 1 to the *new-lines*. To realize this change in the generated patch, the '-' was replaced with a space, making the line effectively a context line in the patch, see Figure 4.5. If a line in the diff that added a line in the source code (i.e., starting with '+') was removed from the patch, the offset was adjusted by subtracting 1 from *new-lines*. To realize this change in the generated patch, the entire line was removed from the patch, see Figure 4.6.

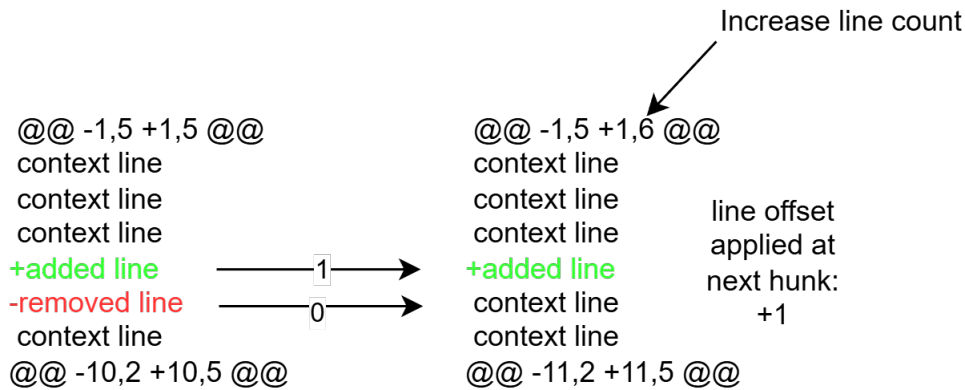


Figure 4.5: A diff applied with a bit-vector to transform a removed line into a context line, causing a line increase in the source code.

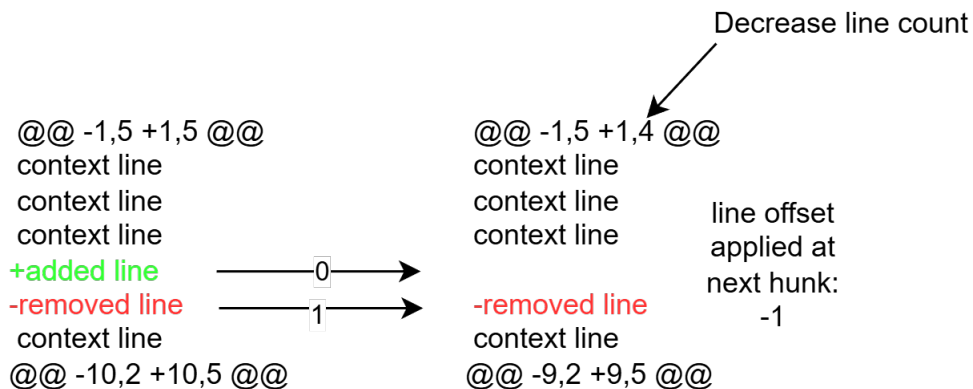


Figure 4.6: A diff applied with a bit-vector to remove an added line, causing a line decrease in the source code.

Directory Structure Used by The GA

The GA assumed a specific structure of folders and files associated with a project to function properly, see Figure 4.7.

- **diffs/temp**: The directory used to store the individuals during the execution of the GA.
- **diffs/out/file**: The directory used to store the individuals returned from the GA run in file mode.
- **diffs/out/coarse**: The directory used to store the individuals returned from the GA run in coarse-grained mode.



Figure 4.7: The directory structure.

- **diffs/out/fine**: The directory used to store the individuals returned from the GA run in fine-grained mode.
- **diffs/out/done**: The directory used to store the merged diff containing several of the returned individuals from the execution of the GA.
- **<git-repository-project>**: A git repository of the project used for creating copies of the repository, used for testing and fuzzing for the GA.
- **build.sh**: A shell script used to build the project, parse the compilation log, and echo the compilation percentage to standard output.
- **test.sh**: A shell script used to run the project's test suite, parse the result, and echo the result to standard output.

Parsing of The Compilation Log

In order to prioritize individuals which were closer to compiling, the projects' build systems had to be parsed to get an estimated compilation percentage. These scripts were based on the build instructions presented on the project's GitHub page. Some projects printed the exact compilation percentage, but most did not. To get a compilation percentage estimate, in this case, the compilation log had to be parsed based on the different directories compiled. If a project, for example, compiled the directories */src*, */tests*, and */docs*, the compilation percentage would be increased by 33% whenever these directories appeared in the compilation log, see Figure 4.8. This compilation estimate was only used if the project's build system returned a non-zero exit code.

Parsing of Test Suites

Due to the different types of output formats from the test suites used by the projects, separate simple shell scripts had to be written to parse each test suite to echo back the result to the GA. The source code was first compiled with the script **build.sh**. If the compilation was successful, the script **test.sh** was called to start the test suite and parse the result. This parsing was done with pattern matching using *grep* on the output. This script had to be written manually for all of the different test suites. These scripts were small and hard-coded to only support a specific test suite output. Not much knowledge of shell scripting was required to make these scripts. Basic knowledge of running commands in the terminal in a Unix-based operating system should be sufficient. The time taken to create a shell script to parse a test suite was around 5 to 10 minutes. Most test suites returned the number of run, passed, and

```
#!/bin/bash
worker=$1
cd "libxml2_test_${worker}"
result=$(./autogen.sh)
if [ ! $? ]; then
    exit $?
fi
result=$(make)
status=$?
while IFS= read -r line; do
    if [ "$line" = "Making all in src" ]; then
        echo "33"
    fi
    if [ "$line" = "Making all in tests" ]; then
        echo "67"
    fi
    if [ "$line" = "Making all in docs" ]; then
        echo "100"
    fi
done <<EOF
$result
EOF
exit $status
```

Figure 4.8: Example of a script to parse a compilation log.

failed tests on the last line in the log. A script to parse this could be to read the last line and get the number of run, passed and failed tests, and return the number of passed tests to the GA with the number of failed tests as the exit code, see Figure 4.9. This made the GA aware if all tests passed or not. The GA did not build the fuzzers if a non-zero exit code was returned, since it could lead to the fuzzers finding a crash that also caused the failed tests. This would lead to GA following a false positive vulnerability. The GA assumed this script was present due to the required file structure.

```
#!/bin/bash
worker=$1
cd "libxml2_test_${worker}"
result=$(make runsuite)
if [ ! $? ]; then
    echo 0
    exit $?
fi
result=$(./runsuite | grep "Total")
num_tests=$(echo $result | grep -o "[0-9]* tests," | grep -o "[0-9]*")
num_errors=$(echo $result | grep -c "no errors")
if [ $num_errors = 0 ]; then
    num_errors=$(echo $result | grep -o ", [0-9]* errors," | grep -o "[0-9]*")
else
    num_errors=0
fi
echo $((num_tests - num_errors))
exit $num_errors
```

Figure 4.9: Example of a script to parse a test suite.

Building Fuzzers with OSS-Fuzz

Each project's source code was cloned to the desktop computer, and then OSS-Fuzz's fuzzers were pointed to the local source code directory. This was done with the command `python infra/helper.py build_fuzzers -sanitizer {address/memory/undefined} project_name {path_to_source_code}`, which built the fuzzers for the fuzz targets

associated with the project and pointed the fuzzers' execution to the source code location. OSS-Fuzz started a Docker image and copied a build script for the project into the Docker image. This build script described how to build the project, build the fuzzers, and link the fuzzers to the source code. OSS-Fuzz defaulted to compile the fuzzers to its own out directory associated with the project. This was changed to place the fuzzer binaries into the project's local source code directory instead. This Docker image had to be run every time there were any changes to the diffs to get the updated fuzzer binaries with the changed source code.

Furthermore, due to OSS-Fuzz having to be run in its own Docker image, the location of the binaries of the source code after compilation for the projects was different than the binaries compiled outside of the Docker image. This caused the build systems of the projects to be unable to compile through OSS-Fuzz if they had already been compiled outside the Docker image. The same happened if the projects were first compiled through OSS-Fuzz and then tried to compile through their build systems. Running the projects' test suites inside OSS-Fuzz's Docker image did not work since the library dependencies for the projects did not support the compiler flags used to build and link the fuzzers. This resulted in either a crash or failed test cases, which otherwise passed outside the Docker image. This was remedied with the use of two copies of the same project, one for OSS-Fuzz and one for outside OSS-Fuzz. This, however, obviously doubled the compilation time since the same project had to be compiled twice to test the same source code.

Parsing of OSS-Fuzz Results

To actually check if a bug was reintroduced, the reproducible test case associated with the project and the bug was used. The command `python infra/helper.py reproduce {path_to_source_code_fuzz_binaries} {fuzz_target} {path_to_testcase}` ran the fuzzer on the project at the fuzz target with the input bytes that triggered the bug. If the fuzzer aborted execution after running the input, this meant that the bug was reintroduced. However, the address sanitizer was also bundled with a memory leak sanitizer, which could also abort the execution. Since memory leaks were not an indicator of a successful re-implementation of the bug, these crashes were ignored by the GA. If the fuzzer, however, reported that only a small number of test cases were run before exiting, this indicated that the bug had not been reintroduced.

Specifying Vulnerabilities to the GA

The GA got the desired vulnerabilities from the command line argument `-bugs` parsed by the program with a set of vulnerabilities with their corresponding test case, diff, fuzz target, and sanitizer.

Initialization of Individuals

The patch file to be reduced by the GA was parsed and divided into the files, hunks, and lines present in the patch file. The GA first evaluated the whole patch file without any changes applied, to at least have an individual that re-implemented the bug. The individuals were then initialized in the GA through a divide-and-conquer approach, where the set of included code elements was spread out evenly over the whole population. Since the initialization size may vary between projects, the initialization size could be increased when the GA failed to find any individuals that compiled in the first few generations. However, with a larger initialization size, the individuals at the end of the population would be smaller than the desired initialization size due to the end of the patch file. This was remedied by allowing individuals at the end of the population to wrap around to the beginning of the patch file for the parts of the individuals which would otherwise be cut off by the end of the patch file, see Figure 4.10. The initialized individuals were immediately evaluated in order for a chance to quickly determine which parts of the patch were relevant to the bug. From generation

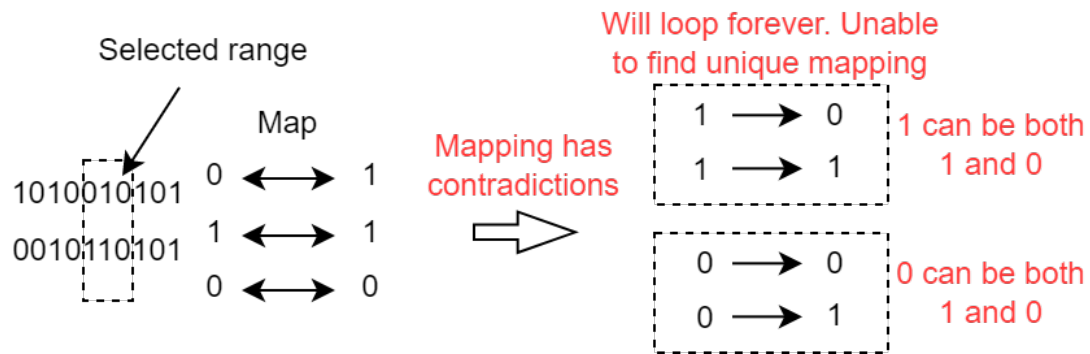


Figure 4.11: A contradicting example of the Partially Mapped Crossover function with a population in the binary representation.

not applicable to this GA, see Figure 4.11. Additionally, since the selection of code changes in a diff was not an ordering problem, the permutation problem representation was not a great fit. The crossover function, which was next considered, was the Uniform Crossover function. It did have the risk of producing less diversity, but to a lesser extent compared to the K-point Crossover function due to its higher reproduction potential. Since all symbols were treated individually to determine which would switch individuals, the Uniform Crossover function could mix the parents more than the K-point Crossover function. To determine how much mixing was required to create good solutions, a parameter test had to be conducted by comparing the Uniform Crossover, 1-point Crossover, 2-point Crossover, and 4-point Crossover.

Mutation Function

The mutation function adds a little diversity to the population after the crossover function has produced the new generation. It was also in some situations the only way to find an important feature for the optimal solution. However, the mutation function could also destroy a good individual by mutating an important symbol or cause the GA to prematurely converge. The performance of the mutation function depended on the representation of the individuals and the problem size. The mutation functions considered were the most common mutation function for the binary problem representation, the Bit-flip Mutation function [10], and its more general case, the Simple Inversion Mutation function [16]. The two mutation functions were tested in a parameter test to determine which would fit the GA best.

Evaluation of Individuals

After the population had been mutated, the individuals were evaluated, and the status of an individual after the evaluation was one of six possible values; `BUG_REINTRODUCED`, `BUG_NOT_REINTRODUCED`, `FAILED_TO_PASS_ALL_TESTS`, `FAILED_TO_BUILD_FUZZERS`, `FAILED_TO_COMPILE` and `FAILED_TO_APPLY_PATCH`. The individuals' bit-vectors were used to generate the diffs associated with each individual in the current population. These diffs were then one by one applied to the source code, followed by compilation, testing, building the fuzzers, and reproducing the bug's test case. These results were then used in the fitness function to calculate a score for each individual in the population. Building the fuzzers and reproducing the bug's test case was only performed if all tests in the project's test suite passed in order to avoid false positives caused by failing test cases unrelated to the actual bug.

Fitness Function

The results from the tests and compilation of the current diff were used to evaluate the diffs in the GA in order to get a vulnerable compiling diff. The length of the diff was also used in the evaluation to get as small a diff as possible. The fitness function for a diff d for the GA was

$$\max(f(d)), \text{ where}$$

$$f(d) = \begin{cases} d_{\text{compilepercent}}, & \text{if } d \text{ did not compile} \\ |\{i \in p \mid d \text{ passes } i\}| \cdot \frac{|d_{\text{init}}|}{|d|}, & \text{if bug not reintroduced} \\ |\{i \in p \mid d \text{ passes } i\}| \cdot \frac{|d_{\text{init}}|}{|d|} \cdot (|d_{\text{init}}| + 1), & \text{otherwise} \end{cases}$$

The factors $d_{\text{compilepercent}}$ and $|d_{\text{init}}|$ were the compile percentage for a diff and the length of the initial diff, respectively, while $|d|$ was the length of the current diff. The function was designed to promote diffs that passed the project's test suite with the most passed tests, as well as reintroduced the bug. Then the fitness-function value was divided by the length of the diff to give diffs with the same number of passed tests and verification result, but shorter in length, an advantage. Diffs that reintroduced bugs had to always have a higher fitness value than any other diff of any length for the GA to work properly. To achieve this, the fitness value was multiplied by the length of the initial length +1 ($|d_{\text{init}}| + 1$). This made sure that a diff of length 1, which passed all tests, did not outperform a diff of length $|d_{\text{init}}|$, which also passed all tests, but also re-implemented the bug. To make the GA focus more on compiling the diffs, all the diffs that did not compile used the compile percentage $d_{\text{compilepercent}}$ as the fitness-function value. Diffs of length 0 were not evaluated since they did not apply any changes to the source code and were therefore uninteresting for the GA. This also avoided a divide-by-zero problem with the fitness function.

Parallelization of Evaluations

The evaluation of individuals proved to be a performance bottleneck for the GA and amounted to around 99% of the execution time. To speed up the GA, the evaluation of the individuals was done in parallel using a thread pool. However, one copy of the source code was not enough to evaluate several individuals in parallel because when a diff was applied to the source code, another diff could not be applied. Therefore, to evaluate individuals in parallel, extra copies of the source code equal to the number of threads used to evaluate the individuals had to be used. Each thread's ID was used to determine which copy of the source code to evaluate the individual with.

Fault Localization

The initial design of the GA allowed individuals to use the whole diff to find the minimal diff throughout the entire execution of the GA. However, this made it possible for the GA to evaluate individuals that reintroduced the bug but were larger than the currently best found individual. This caused the GA to be unable to converge before the max number of generations had been reached. Therefore, it was of interest to force the GA to only evaluate individuals of the same size or smaller. A Fault localization was implemented to achieve this by tracking which bits in the bit-vector were required to re-implement the bug, see Algorithm 1. These required bits were determined by looping over all bit-vectors in the population, which re-implemented and not re-implemented the bug in the current generation, and applied a bit-wise-and operation between the individuals and the current required bits. This enabled several different individuals to help reduce the required bits further than the current best individual, see Figure 4.12. The crossover-and mutation function used these required bits to focus the GA on these bits to reduce the required bits even further. All bits that were not part of the required bits were set to 0 since those bits had been confirmed not to be required

Algorithm 1 Reduce Required Bits

```

Require: Individuals
Require: Required bits
Ensure:  $Required\ bits \leq Required\ bits$ 
for all Individual in Individuals do
  if Individual re-implements bug then
     $Required\ bits \leftarrow Required\ bits \ \& \ Individual$ 
  end if
  if Individual !re-implements bug then
     $Required\ bits \leftarrow Required\ bits \ \& \ !Individual$ 
  end if
end for
if  $Required\ bits = 0$  then
  for all Individual in Individuals do
    if Individual re-implements bug then
       $Required\ bits \leftarrow Required\ bits \ | \ Individual$ 
    end if
  end for
end if

```

| | |
|---------------------|-----------------------|
| 0111011011100000010 | Bug re-introduced |
| 1100010001110000001 | Bug not re-introduced |
| 0110111011000000000 | |
| 1001111000000001010 | |
| 0001111101011101000 | |
| 0000011000000000000 | & |

Figure 4.12: Example of when several bit-vectors, which reintroduce the bug, help reduce the required bits more than the current best bit-vector.

to re-implement the bug. However, if the bug could be re-implemented in several mutually exclusive parts of the code, the required bits could be set to 0, indicating that no change in the code is required to re-implement the bug, which is false. To remedy this, the required bits were determined by the bitwise-or of the individuals, which re-implemented the bug in this case, see Algorithm 1. If the required bits were reduced too aggressively and did not include the bug, the required bits were reset to the currently best individual's bit-vector. If the required bits were less than the currently best evaluated individual after the final generation, the required bits were evaluated and replaced the best individual if the required bits managed to re-implement the bug. This was done to ensure that only the individuals that had been evaluated could be a final solution returned by the GA.

Crossover Function with Fault Localization

The original implementation of the crossover function did not consider which bits were currently required to reintroduce the bug. To make sure the result of the crossover included a

viduals that had already been evaluated. Therefore, these evaluations would be unnecessary to perform. This was remedied by marking all duplicate bit-vectors, logging all evaluated bit-vectors, and retroactively assigning fitness values to duplicate and already evaluated individuals.

Focused Adaptive Mutation

The fault localization helped the GA to only evaluate individuals that were the same size or smaller than the current best individual. However, when the number of individuals that re-implemented the bug was small, and the best individual in the current population was only marginally better than the previous best known individual, the GA stagnated. The speed of descent became slow, and the better individuals were only a few bits better than the best individual. To remedy this, the mutation probability was adjusted based on the performance of the previous generation. If there were more duplicates or already evaluated individuals in the population compared to the previous generation, the mutation rate was increased. If only a few individuals compiled and there was a low amount of duplicates in the population, the mutation was decreased, since this indicated that the mutation probability was too high.

For large individuals, there was a chance that some important parts of an individual were never explored. With a more deterministic and spread out mutation for large individuals, the GA could focus on the parts of the individual that increased the fitness the most. The bits to mutate in the individual were determined by the index of where the individual resided in a randomly ordered list of individuals. For example, the individual first in the randomly ordered list was mutated in the first few bits, and the last individual in the randomly ordered list was mutated in the last few bits, see Figure 4.15.

size: 18 population: 6 mutation probability: 1/3

```

100|000|000|000|000|000
000|101|000|000|000|000
000|000|010|000|000|000
000|000|000|110|000|000
000|000|000|000|011|000
000|000|000|000|000|111

```

Figure 4.15: Example of the focused adaptive mutation.

Adaptive Order of Operations

When an individual was found that re-implemented the bug and was smaller than the initial diff, the mutation was done before the crossover, see Figure 4.16. This was done in order to get more diversity in the population before the crossover function to get more useful crossover operations. The more diverse the population is, the more effective the crossover function is [16]. The crossover was done before the mutation in cases where the initialization failed to find an individual that re-implemented the bug and was smaller than the initial diff, and would therefore need a slower search with a less diverse population. With a too diverse population, with the mutation done before the crossover, there would be a risk that an individual that re-implemented the bug and was smaller than the initial diff would never be found.

Adaptive Initialization

If the initialized population resulted in zero compiling individuals in the first 5 generations of the GA, see Figure 4.16, the GA was restarted with a larger initialization size, see Figure 4.17.

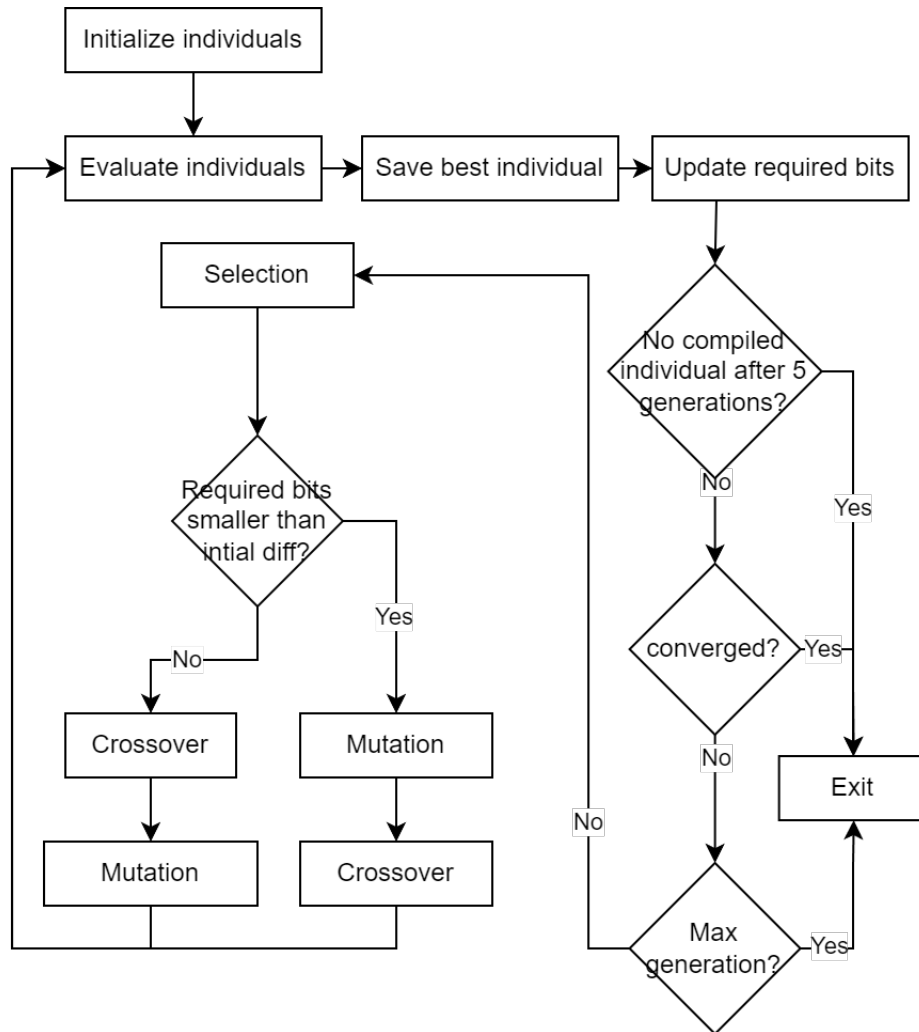


Figure 4.16: The design of the GA.

Stopping Criteria

The GA stopped when the entire population consisted of a single bit-vector or a small set of bit-vectors. These bit-vectors had to include the currently best bit-vector, and no new bit-vectors had to have been generated since the previous generation. This could only occur whenever the diversity in the population was extremely low and the required bits forced the GA to only output already evaluated individuals. This could also occur when the selection function was too biased towards the best individuals, coupled with a too low mutation rate, causing the GA to prematurely converge.

Initial Parameter Values

Before the parameter test was run, the GA used a population size of 50 with a maximum of 15 generations. The selection function was Tournament size 4 with the 2-point-crossover function with a 95% crossover probability, and the mutation function bit flip with a probability of 1% was used. These parameter values were based on the values used in GenProg [11], automatic program repair of Assembler programs [35], an improved adaptive GA [27], and some preliminary test runs. These values functioned as baseline values in the parameter test when a parameter had not yet been tested. The initialization size of the individuals was the

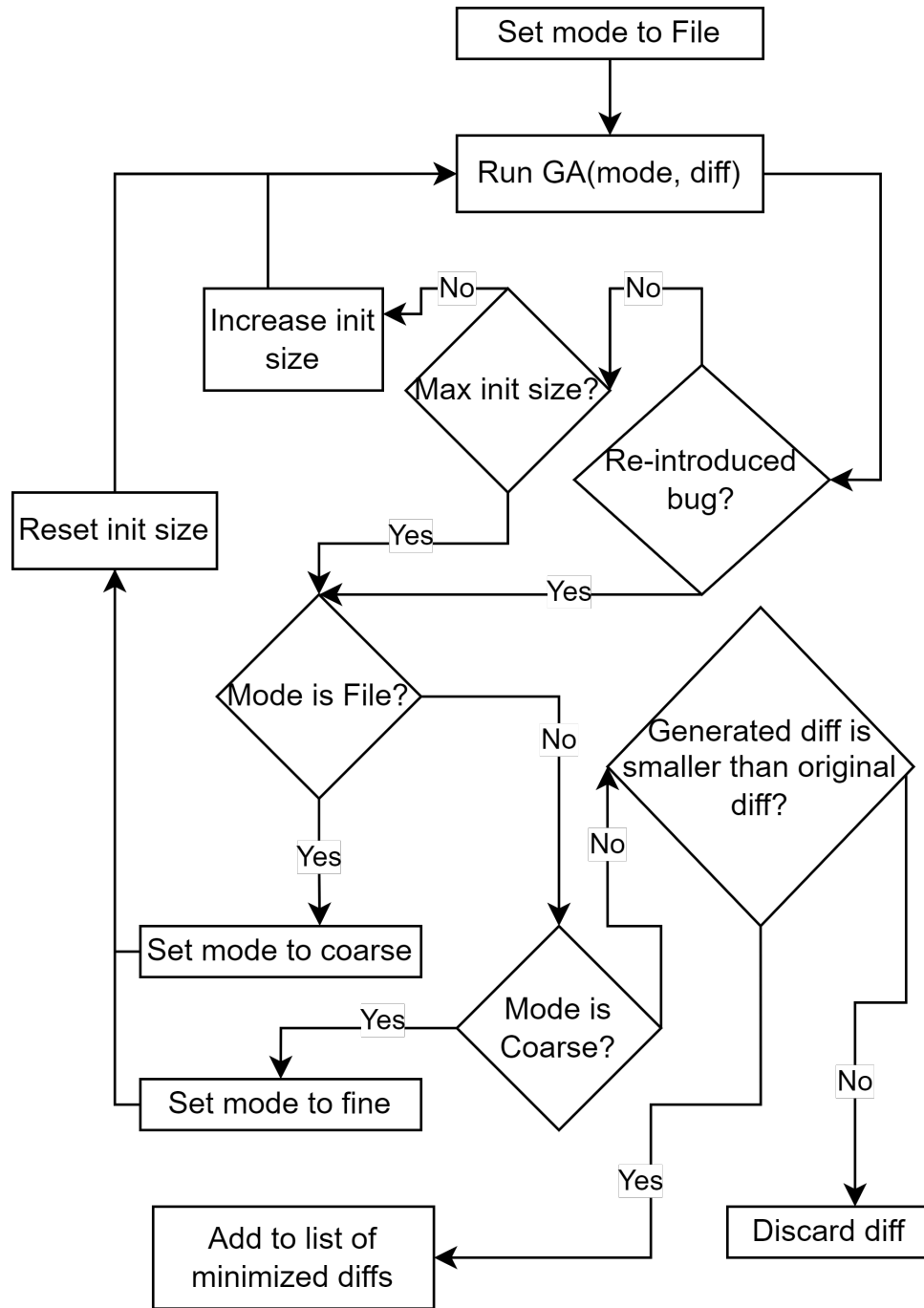


Figure 4.17: Minimization flowchart of a diff.

initial diff length divided by the population size to get isolated parts of the initial diff to have the potential to reintroduce the bug already during the initialization.

Parameter Test

The test ran the GA with a set of parameters on the same diff to compare the convergence percentage, the average diff lengths, the number of generations needed, and the total time taken. The dynamic initialization size was turned off during the parameter tests to get as accurate comparisons as possible. The 3 best combinations of parameters from each test were

used as the new baseline in the next parameter test until all tests had been run, until as close to a best set of parameters as possible had been selected.

Each version of the GA was run 10 times per parameter value to get reliable results. The parameter test was run with the GA, with only the coarse mode enabled to reduce the time taken for each parameter test and to have a moderate search space for the GA. If only the file mode was used, the solution was found during initialization, and if only the fine mode was used, no individual was able to compile. The parameter test ran the GA with the coarse version to evaluate the parameters in cases where an amount of searching was required to find an as-close-as-possible to optimal solution. Every combination of the considered selection functions (PRWS, Tournament size 2, Tournament size 4, and Tournament size 8) and crossover functions (Uniform Crossover, 1-point Crossover, 2-point Crossover, and 4-point Crossover) was evaluated to get the best possible steady pressure towards convergence while still having some diversity in the population to not prematurely converge. When a good combination of the selection function and the crossover function was found, the mutation probability was tested along with the different mutation functions. From preliminary testing, the mutation probability played a big part in the performance of the GA, and it was therefore more thoroughly tested with more possible values. A focused mutation variant with adaptive probability was also tested and compared with mutations with static mutation probability. When a combination of mutation probability and mutation function was found that increased the performance of the GA, the crossover probability was adjusted. Lastly, the population size and maximum number of generations were adjusted to find the population size that resulted in the greatest descent compared to the total execution time. The GA with different population sizes was allowed to run for a maximum of 50 generations or 2 hours of execution time to compare larger population sizes with more possible generations. For the last parameter test, a much larger diff from the same project was used in order to properly test the performance of the GA, since the GA managed to converge before the preliminary max number of generations with the diff used for the previous parameter tests.

Creation of Test Suite With Ground Truth

The patches generated by the GA were used to create the test suite with automatic ground truth. However, in order to guarantee ground truth, all patches had to contain one and only one known bug from the set of bugs returned from the GA. This was verified by reproducing with OSS-Fuzz with the sanitizer, fuzz target, and test case for each bug with only one patch applied to the source code. A patch had ground truth if OSS-Fuzz reproduced the bug exactly once during this process. Otherwise, the patch did not have ground truth and was discarded.

Merge of Verified Patches

The patches that had verified ground truth were applied to the source code. However, it was not guaranteed that all patches could be applied to the source code since several patches could cause compilation errors when combined. Some bugs could also be incompatible with each other. Therefore, the merging process had to use a heuristic to narrow down the search space since, in theory, the number of ways the patches could be applied was $n!$ for n different bugs. The heuristic used was an ordering of the file size of each patch. The patches were first applied to the source code in ascending file size order. Whenever a patch was applied, the set of currently applied patches was evaluated to verify that all applied patches could be triggered and verified with ground truth.

The test suite was evaluated to determine if all bugs in the merged program could be triggered and verified with ground truth. For each bug included in the test suite, all test cases were looped over to check that only one bug was triggered for each test case. Whenever a bug was triggered in the test suite, all bugs in the test suite were reproduced individually with the test case that triggered the bug in the test suite, see Algorithm 2. If more than one or no

Algorithm 2 Evaluate Test Suite

Require: *Test Suite***Require:** *Bugs***Require:** *Testcases***Ensure:** *All bugs can be triggered in the Test Suite with ground truth**apply*(*Test Suite*)**for all** *Bug* **in** *Bugs* **do***build_fuzzers*(*Bug*[*sanitizer*])**for all** *testcase* **in** *Testcases* **do****if** *reproduce*(*Bug*, *testcase*, *Bug*[*fuzz_target*]) **triggered a bug** **then***reproduced_bugs* \leftarrow 0**for all** *Bug* **in** *Bugs* **do***apply*(*Bug*)*build_fuzzers*(*Bug*[*sanitizer*])**if** *reproduce*(*Bug*, *testcase*, *Bug*[*fuzz_target*]) **then***reproduced_bugs* \leftarrow *reproduced_bugs* + 1**end if****end for****if** *reproduced_bugs* \neq 1 **then***Throw Exception***end if****end if****end for****end for**

 \triangleright Run in parallel \triangleright Run in parallel \triangleright Run in parallel \triangleright Run in parallel \triangleright Run in parallel

bug was triggered by a test case, then it would be impossible to determine ground truth. The test suite would then be unusable as an evaluation tool for Fuzzers with automatic ground truth. If a patch could not be applied to the source code or caused the current test suite to be unable to verify bugs with ground truth, the patch was not included in the test suite. If not all patches could be applied to the source code and verified in ascending file size order, they were instead applied to the source code and verified in descending file size order. If none of these application approaches successfully created a test suite with ground truth with all the verified patches, the approach that produced a set with the most patches was used. If both approaches produced the same test suite, the first approach was used to create the patch.



5 Results

5.1 Overview

To test the performance of creating automatic test suites with ground truth and to make sure the GA had no bias towards the bug used in the parameter test, the program was used to create test suites with ground truth for five projects, including the project used during implementation and testing. The performance test created test suites with ground truth 10 times per project to get a good statistical accuracy while taking an acceptable time to complete. The test measured the number of bugs included in the test suite, the size of the test suite, the success rate, and the time taken to create each test suite.

5.2 Pre-study

The vulnerabilities for the projects were selected based on the 4.2 Project & Vulnerability Selection Criteria.

Selected Projects

The 5 projects selected had compile-and-test time as well as Fuzzer build time within the 1-minute limit and were implemented in C or C++ since most vulnerabilities are found in these languages, see Table 5.1.

| Project | Compile Time (s) | Test Time (s) | Build Fuzzers Time (s) | Language |
|----------|------------------|---------------|------------------------|----------|
| nDPI | 17.3 | 16.8 | 36 | C |
| mruby | 22.4 | 2.2 | 32.4 | C |
| libxml2 | 22.8 | 0.2 | 28.2 | C |
| libdwarf | 14.9 | 2.8 | 19.3 | C/C++ |
| hunspell | 24.8 | 4.9 | 24.1 | C++ |

Table 5.1: Compile time, test time, time to build fuzzers, and implementation language for the selected projects.

Selected Vulnerabilities

Most of the selected vulnerabilities were memory corruption bugs with different variants of buffer overflows and were detected with the address sanitizer. The age of the vulnerabilities was a few days after they were released to the public, with a few exceptions. For specifics for each project, see Tables 5.2-5.6.

| OSS-Fuzz Issue Nr. | Type Of Bug | Sanitizer | Diff Size | Days |
|--------------------|----------------------|-----------|-----------|------|
| 57448 | Segmentation fault | Address | 4 581 | 37 |
| 57369 | Heap buffer overflow | Address | 4 639 | 40 |
| 57317 | Segmentation fault | Address | 4 649 | 40 |
| 56272 | Segmentation fault | Address | 7 270 | 63 |
| 55218 | Heap buffer overflow | Address | 73 295 | 101 |

Table 5.2: Selected Vulnerabilities for nDPI.

| OSS-Fuzz Issue Nr. | Type Of Bug | Sanitizer | Diff Size | Days |
|--------------------|----------------------------|-----------|-----------|------|
| 57037 | Segmentation fault | Address | 4 561 | 35 |
| 56991 | Heap use after free | Address | 4 648 | 35 |
| 56889 | Use of uninitialized value | Memory | 4 774 | 35 |
| 56406 | Heap use after free | Address | 7 698 | 35 |
| 53183 | Integer overflow | Address | 25 917 | 35 |

Table 5.3: Selected Vulnerabilities for mruby.

| OSS-Fuzz Issue Nr. | Type Of Bug | Sanitizer | Diff Size | Days |
|--------------------|----------------------------|-----------|-----------|------|
| 57521 | Use of Uninitialized Value | Memory | 10 490 | 39 |
| 57469 | Global buffer overflow | Address | 10 512 | 36 |
| 57410 | Use of uninitialized value | Memory | 10 526 | 41 |
| 57304 | Global buffer overflow | Address | 11 347 | 42 |
| 57294 | Global buffer overflow | Address | 11 347 | 42 |

Table 5.4: Selected Vulnerabilities for libxml2.

| OSS-Fuzz Issue Nr. | Type Of Bug | Sanitizer | Diff Size | Days |
|--------------------|----------------------|-----------|-----------|------|
| 57527 | Heap buffer overflow | Address | 7 157 | 32 |
| 57442 | Heap buffer overflow | Address | 9 576 | 34 |
| 57437 | Heap double free | Address | 9 837 | 35 |
| 57429 | Invalid free | Address | 9 837 | 34 |
| 56906 | Heap buffer overflow | Address | 14 522 | 40 |

Table 5.5: Selected Vulnerabilities for libdwarf.

| OSS-Fuzz Issue Nr. | Type Of Bug | Sanitizer | Diff Size | Days |
|--------------------|----------------------------|-----------|-----------|------|
| 56737 | Heap use after free | Address | 157 | 59 |
| 55818 | Stack buffer underflow | Address | 619 | 82 |
| 55191 | Stack buffer overflow | Address | 705 | 115 |
| 54672 | Use of uninitialized value | Memory | 1 090 | 122 |
| 54244 | Heap buffer overflow | Address | 2 307 | 138 |

Table 5.6: Selected Vulnerabilities for hunspell.

5.3 Parameter Tests

The different choices of functions and their parameters were tested through a series of parameter tests to determine which combination of them gave the best possible performance for the GA.

The first 3 parameter tests for the GA, which tested the selection and crossover functions, mutation function, and mutation probability, and crossover probability, were run with the diff from the OSS-Fuzz issue number 56272 with a size of 7 271 lines. The last parameter test was run with the diff from the OSS-Fuzz issue number 55218, with a size of 73 295 lines, in order to fully test the population size and maximum generations. Both diffs used in the parameter tests were from the nDPI project. If the diff for the first 3 parameter tests were used for this test, it would finish during initialization for large population sizes and would not provide any useful results of the GA run under a large number of generations.

Selection and Crossover Functions

Each combination of the considered selection functions (PRWS, Tournament size 2, Tournament size 4, Tournament size 8) and crossover functions (Uniform Crossover, 1-point Crossover, 2-point Crossover, 4-point Crossover) was tested, see Figure 5.1. The three best

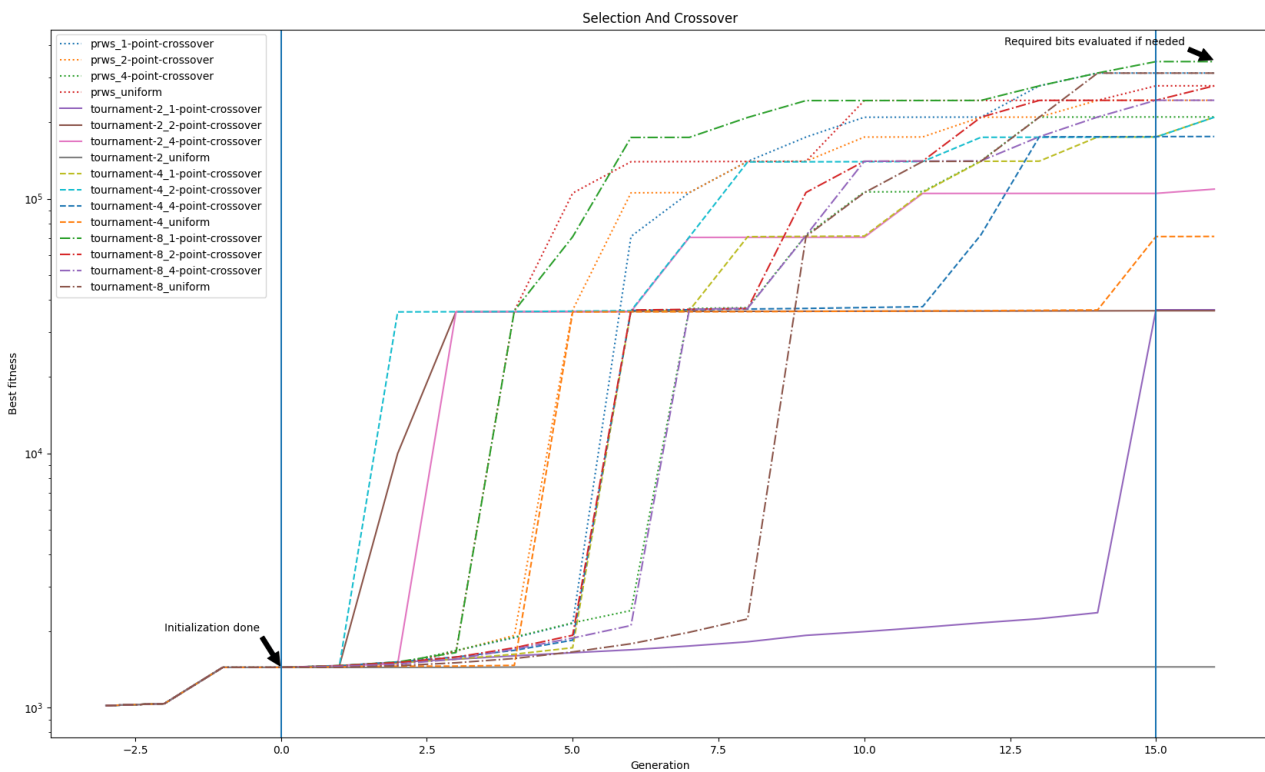


Figure 5.1: Selection-and Crossover function parameter test with the fitness value of the best individual per generation averaged out over the total number of runs. The parameter test was performed on the vulnerability from the OSS-Fuzz issue 56272 for the nDPI project.

combinations of selection and crossover functions from this parameter test were Tournament size 8 and 1-point crossover, PRWS and 1-point Crossover, and Tournament size 8 and Uniform Crossover. The only combination of selection and crossover functions that managed to achieve a 100% convergence rate was Tournament size 8 and 1-point Crossover. The average time taken to converge was 44.5 minutes in an average of 8.8 generations. The other two best combinations managed to achieve a 90% convergence rate in an average of 33.2 minutes with 10.2 generations on average and 40.2 minutes on average with an average of 12.3 generations respectively. The diff sizes produced by the three best combinations were on average 29, 206, and 278 lines, respectively. The three best combinations of selection and crossover functions (Tournament size 8 and 1-point Crossover, PRWS and 1-point Crossover, and Tournament size 8 and Uniform Crossover) were tested further in the next parameter test, Mutation Function and Mutation Probability.

Mutation Function and Mutation Probability

The three best combinations of selection and crossover functions from the previous parameter test were tested with the two considered mutation functions, bit-flip and simple inversion, with probability values 0.005, 0.01, 0.02, 0.03, and 0.01, 0.25, 0.5, respectively. The bit-flip mutation function was also tested with an adaptive probability variant. The three best muta-

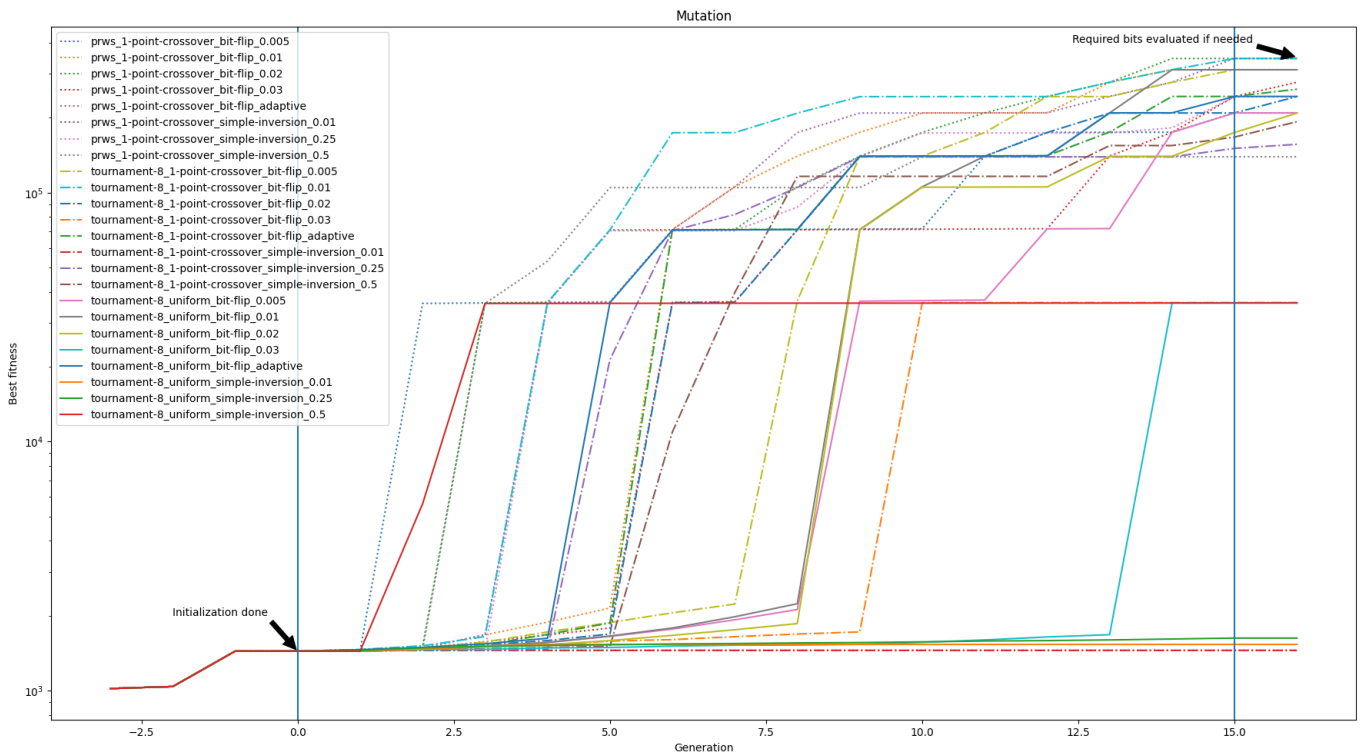


Figure 5.2: Mutation function-and probability parameter test with the fitness value of the best individual per generation averaged out over the total number of runs. The parameter test was performed on the vulnerability from the OSS-Fuzz issue 56272 for the nDPI project.

tion functions and probabilities based on currently chosen parameter values for the GA were Tournament size 8 and 1-point Crossover and bit-flip with 0.01 probability, PRWS and 1-point Crossover and adaptive bit-flip, and PRWS and 1-point Crossover and bit-flip with 0.02 probability, see Figure 5.2. All three best current parameter values for the GA managed to achieve a 100% convergence rate. The different GA configurations managed this in an average of 44.5 minutes with 8.8 generations on average, 45.1 minutes on average with an average of 9.4 generations, and in an average of 37.5 minutes with 10.3 generations on average, respectively. The diffs produced by the three best GA configurations had the same size of 29 lines. The three best GA configurations from this parameter test were used in the next parameter test, Crossover Probability.

Crossover Probability

The three best GA configurations from the previous parameter test were tested with different probability values 0.6, 0.7, 0.8, 0.9, and 1.0 for the crossover function, see Figure 5.3. Almost all GA configurations tested in this parameter test had a convergence rate of 100%, only 5 did not have a 100% convergence rate. The best three converged in an average of 16.9 minutes with

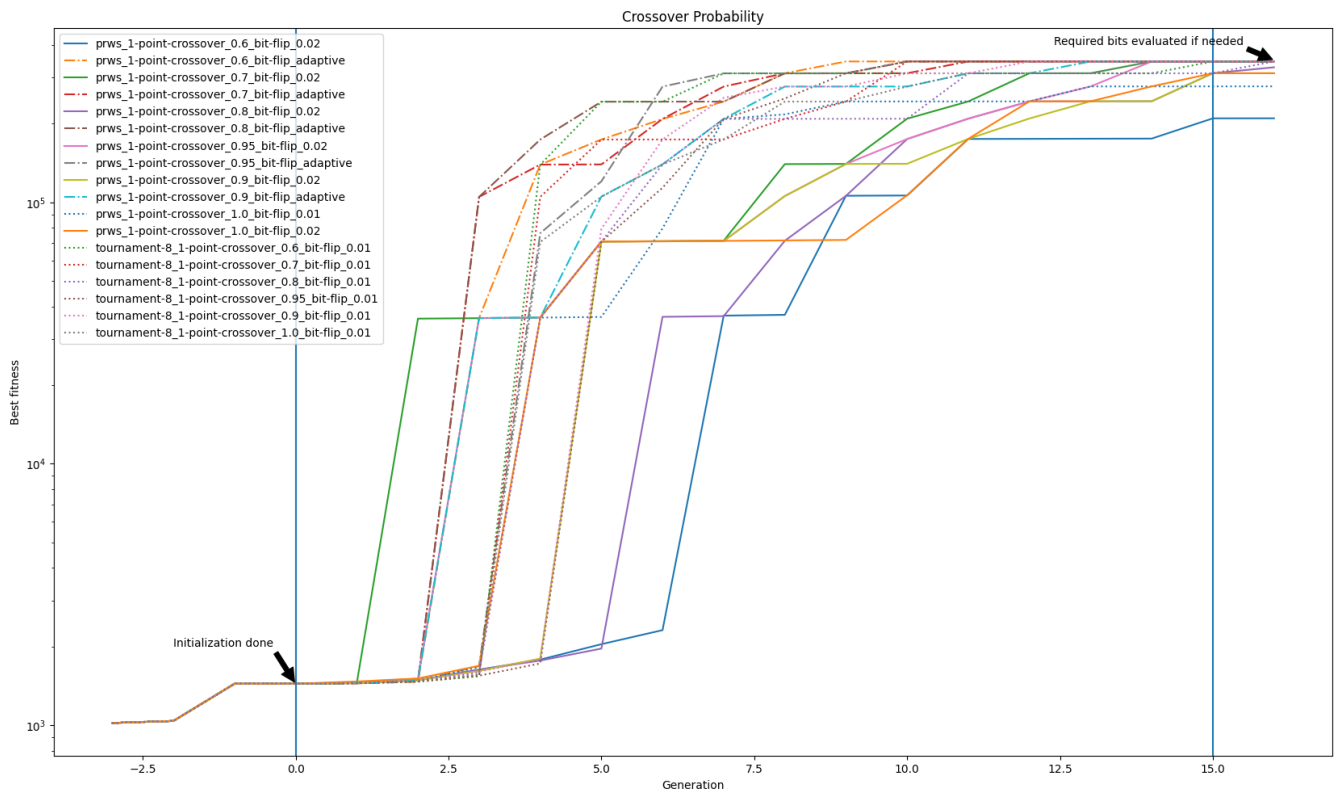


Figure 5.3: Crossover probability parameter test with the fitness value increase per generation of the best individual per crossover probability averaged out over the total number of runs. The parameter test was performed on the vulnerability from the OSS-Fuzz issue 56272 for the nDPI project.

6.2 generations on average, 16.5 minutes on average with an average of 6.4 generations, and in an average of 18.1 minutes with 6.8 generations on average. These three GA configurations (PRWS and 1-point Crossover with 0.8 crossover probability and adaptive bit-flip mutation, PRWS and 1-point Crossover with 0.6 crossover probability and adaptive bit-flip mutation, and with 0.95 crossover probability and adaptive bit-flip mutation) were used in the last parameter test, Population Size and Maximum Generations.

Population Size and Maximum Generations

The last parameter test ran the best GA configurations from the previous parameter test, with the diff from the OSS-Fuzz issue number 55218, with a diff size of 73 295 lines. The test ran for a maximum of 50 generations or 2 hours of execution time with the population sizes 50, 70, 90, and 110, see Figure 5.4. The best GA configuration from this parameter test was PRWS and 1-point-crossover with 0.95 crossover probability, adaptive mutation, and a population size of 110. The GA had a 100% convergence rate, resulting in an output diff of 30 lines. It took, on average, 92.4 minutes in an average of 9.7 generations to reduce the diff in this parameter test. The maximum number of generations used for the best GA configuration

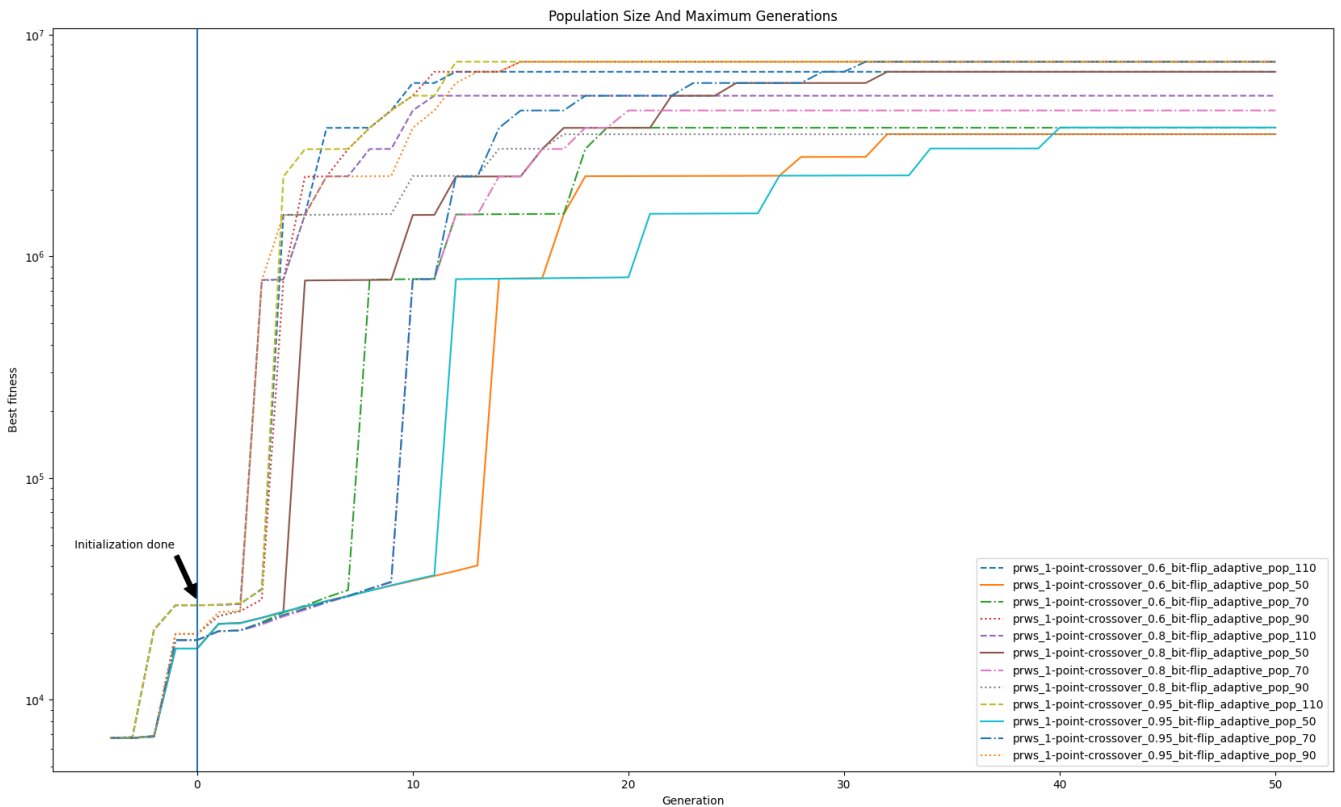


Figure 5.4: Population size and maximum generations parameter test with the fitness value of the best individual per generation averaged out over the total number of runs. The parameter test was performed on the vulnerability from the OSS-Fuzz issue 55218 for the nDPI project.

was 14 generations. The most used generations for any GA configuration in this parameter test was 45 generations, and the least used generations was 4 generations.

5.4 Evaluation

The performance of the GA and the algorithm for merging the reduced diffs was evaluated by reducing 5 diffs and trying to merge the resulting diffs into a single diff. This was done for 5 projects, 10 times. The diffs for each project were grouped together, and the output was summed together for each run to calculate the average and standard deviation, noted as average(\pm standard deviation), for each project. Each line in Tables 5.7-5.10 lists the total number of lines for that project and the output lines for that project, as well as how much the diff was reduced with the time taken for each run and phase.

Reduction of Diffs

The three phases of the GA reduced the diffs for the 25 vulnerabilities for the 5 projects, see Tables A.1-A.4 in appendix A for reduction stats per vulnerability. The complete reduction is shown first, followed by reduction on file-, hunk-, and line-level. In total, the GA reduced the diffs from 252 061 lines to 974 lines on average in around 80 000 s (22.2 h). The fastest time was 4 714 s (1.3 h) on average for the *hunspell* project, and the longest time was 30 839 s (8.6 h) on average for the *libxml2* project, see Table 5.7.

| Project | Input Diff Size | Output Diff Size | Size Reduction | Time Taken (s) |
|--------------|-----------------|---------------------------------|-----------------------------------|--|
| nDPI | 94 434 | 75 | 99.9% | 13 346(\pm 1 189) s |
| mruby | 47 598 | 395(\pm 440) | 99.2(\pm 0.9)% | 23 938(\pm 2 201) s |
| libxml2 | 54 222 | 222(\pm 7) | 99.6% | 30 839(\pm 562) s |
| libdwarf | 50 929 | 206(\pm 39) | 99.6(\pm 0.1)% | 7 055(\pm 373) s |
| hunspell | 4 878 | 76 | 98.4% | 4 714(\pm 124) s |
| Total | 252 061 | 974(\pm456) | 99.6(\pm0.2)% | 79 892(\pm2 587) s |

Table 5.7: Result of the GA from the initial diffs to the final diffs for all projects.

File Phase

The diffs were first reduced at the file level and managed to remove 75.8% on average of the lines from the initial diffs, with the vulnerabilities still present and triggerable. The execution time of this phase took around 50% of the total execution time of the GA, see Table 5.8. This phase removed files that were irrelevant to the vulnerabilities, for example, config and text files, as well as build system files and source files that were not needed to successfully compile and pass all tests.

| Project | Input Diff Size | Output Diff Size | Size Reduction | Time Taken (s) |
|--------------|-----------------|------------------------------------|-----------------------------------|--------------------------------------|
| nDPI | 94 434 | 44 956(\pm 23) | 52.4% | 7 111(\pm 174) s |
| mruby | 47 598 | 1 724(\pm 706) | 96.4(\pm 1.5)% | 10 101(\pm 371) s |
| libxml2 | 54 222 | 9 123 | 83.2% | 15 492(\pm 13) s |
| libdwarf | 50 929 | 4 545 | 91.1% | 2 548(\pm 82) s |
| hunspell | 4 878 | 590 | 87.9% | 2 371(\pm 98) s |
| Total | 252 061 | 60 938(\pm692) | 75.8(\pm0.3)% | 37 624(\pm411) s |

Table 5.8: Result of the GA from the initial diffs to the file output diffs for all projects.

Coarse-Grained Phase

The diffs from the File Phase were reduced on the hunk level through the Course-Grained Phase, and managed to remove 97.8% on average of the lines with vulnerabilities still present and triggerable. The execution time of this phase took around 25% of the total execution time of the GA, see Table 5.9. This phase removed sections in the source code that were irrelevant to the vulnerabilities.

| Project | Input Diff Size | Output Diff Size | Size Reduction | Time Taken (s) |
|--------------|---------------------|--------------------|--------------------|-------------------------|
| nDPI | 44 956(±23) | 108 | 99.8% | 5 223(±1 119) s |
| mruby | 1 724(±706) | 489(±453) | 75.3(±10.8)% | 3 573(±1 752) s |
| libxml2 | 9 123 | 275(±14) | 97.0(±0.2)% | 9 782(±155) s |
| libdwarf | 4 545 | 324 | 92.9% | 1 506(±17) s |
| hunspell | 590 | 129 | 78.1% | 827(±7) s |
| Total | 60 938(±692) | 1 324(±451) | 97.8(±0.7)% | 20 912(±1 877) s |

Table 5.9: Result of the GA from the file output diffs to the coarse output diffs for all projects.

Fine-Grained Phase

The diffs from the Course-Grained Phase were reduced on the line level through the Fine-Grained Phase, and managed to remove 28.7% on average of the lines with vulnerabilities still present and triggerable. The execution time of this phase took around 25% of the total execution time of the GA, see Table 5.10. This phase removed lines that were irrelevant to the vulnerabilities.

| Project | Input Diff Size | Output Diff Size | Size Reduction | Time Taken (s) |
|--------------|--------------------|------------------|--------------------|-------------------------|
| nDPI | 108 | 75 | 30.6% | 1 012(±9) s |
| mruby | 489(±453) | 395(±440) | 28.3(±10)% | 10 263(±961) s |
| libxml2 | 275(±14) | 222(±7) | 19.2(±4.1)% | 5 564(±504) s |
| libdwarf | 324 | 206(±39) | 36.3(±12)% | 3 001(±293) s |
| hunspell | 129 | 76 | 41.1% | 1 516(±46) s |
| Total | 1 324(±451) | 974(±456) | 28.7(±7.2)% | 21 356(±1 010) s |

Table 5.10: Result of the GA from the coarse output diffs to the final diffs for all projects.

Construction of Test Suites

When the reduced diffs were merged into one diff per project, 19 of 25 vulnerabilities were able to be merged. None of the test suites managed to merge all vulnerabilities into one diff, the maximum was 4 vulnerabilities, and the minimum was 3. This resulted in an overall success rate of 76% for the generation of test suites, where the maximum was 80%, and the minimum was 60%, see Table 5.11.

| Project | Total Vulnerabilities | Merged Vulnerabilities | Time Taken (s) | Total Time (s) |
|--------------|-----------------------|------------------------|----------------------|-------------------------|
| nDPI | 5 | 4 | 1 040(±23) s | 14 386(±1 186) s |
| mruby | 5 | 4 | 1 939(±215) s | 25 876(±2 168) s |
| libxml2 | 5 | 3 | 1 676(±108) s | 32 515(±602) s |
| libdwarf | 5 | 4 | 787(±70) s | 7 842(±422) s |
| hunspell | 5 | 4 | 899(±23) s | 5 613(±122) s |
| Total | 25 | 19 | 6 340(±177) s | 86 232(±2 647) s |

Table 5.11: Result of the automatically constructed test suites.

The time taken to merge the vulnerabilities and verify that ground truth is achieved for all vulnerabilities in the test suite took, on average, 1 268 s (21.1 min), and the time taken from the original diffs to the generated test suite was 17 246 s (4.8 h) on average. Of the 974 reduced lines, only 352 lines could be used to construct test suites, resulting in 36.1% of the code from the GA being compatible within the same diffs. The sum of all reduced usable diffs resulted in 352 lines, which, when merged per project, resulted in 5 diffs of 328 lines. The merging of the diffs resulted in a further reduction in diff size of 6.8% on average, see Table 5.12.

A diff failed to merge due to two reasons; conflicting changes between diffs, and the test suite not triggering all vulnerabilities in the merged diff. Of the 6 diffs that failed to merge, 4 of them failed due to conflicting changes, and 2 due to the test suite being unable to trigger all vulnerabilities, see Table A.5 in appendix A for more details.

| Project | Input Diff Size | Unmerged Diff Size | Merged Diff Size | Size Change |
|----------------|------------------------|---------------------------|-------------------------|--------------------|
| nDPI | 75 | 52 | 52 | 0% |
| mruby | 395(±440) | 104(±4) | 104(±4) | 0% |
| libxml2 | 222(±7) | 61(±3) | 53(±3) | -13.2(±0.7)% |
| libdwarf | 206(±39) | 72 | 64 | -11.1% |
| hunspell | 76 | 63 | 55 | -12.7% |
| Total | 974(±456) | 352(±7) | 328(±7) | -6.8(±0.1)% |

Table 5.12: Reduction of code from the initial diffs to the final diffs for the merged diffs for all projects.



6 Discussion

6.1 Results

In this section, the results are discussed and compared with the related works.

Behavior of The GA

During the parameter tests, there were some configurations that failed to find any (or almost any) individuals better than the initial population and showed on the graphs as horizontal (or nearly horizontal) lines. This most likely was due to too high diversity in the population, causing every individual to fail to compile. These results were expected when the mutation probability was too high, resulting in the GA to behave as a random search algorithm instead. The rest of the configurations followed a pattern of a steep climb for the first generations, followed by a gradual slowdown towards the final generations. This was an expected behavior in GAs called selection intensity [19], which naturally decreases when the GA is closing in on the optimal solution since the distribution of the fitness values in the population will be more concentrated. This can be proven by the fact that the number of active bits in the individuals is limited by the required bits fault localization 4.3 and that the number of active bits must be one or more bits. As better individuals are found, the population's individuals are limited by the best found individual's bit-vector, which will get more restrictive as better individuals are found, resulting in a smaller and smaller search space and a smaller variance of fitness values. This can more clearly be seen in Figure 5.3, where most of the configurations reached a near optimum within half of the generations and then spent the rest of the generations getting closer and closer to the optimum.

Test Suite Extendability Compared to Related Works

The time taken for the GA to reduce the diffs while preserving the vulnerabilities in roughly 80 000 s (22.2 h) in total across all projects was quite a long time, but not unexpected due to the compilation, test, and fuzzer build times for the selected projects and the population size. However, the time taken to reduce the diff per vulnerability was a little less than an hour on average, which was not great but also not terrible. This means it would take an hour on average to extend the test suite with an additional vulnerability with ground truth, with close

to no manual steps. This provides a greater test suite extendability than MAGMA [6], where all reintroductions of vulnerabilities were done manually, and DARPA CGC [15], where small programs with intended vulnerabilities were created manually. However, the extendability would not be close to the extendability of LAVA-M [7] but would support more variety of real-world vulnerabilities compared to LAVA-M's synthetic vulnerabilities.

Although it would be easy to extend the number of vulnerabilities in the test suites, it is not guaranteed that they would be compatible with the vulnerabilities already included in the test suites. None of the generated test suites included all possible vulnerabilities, with an overall success rate of 76%. This was to be expected since some generated code could be incompatible with each other. A remedy for this could be with manual editing of the code to improve compatibility, as done in MAGMA [6], but to ensure better scalability and automation, splitting the set of vulnerabilities into several test suites with different sets of vulnerabilities instead would be better to increase the success rate.

6.2 Method

In this section, the method is discussed and criticized along with a discussion of selected sources for this study.

Imprecise Compilation Percentage Evaluation

The parsing of the compilation logs when a compilation percentage was not reported by the build system was a little coarse. A more precise approximation of compilation percentage could be to count the number of successfully compiled files compared to the expected number of compiled files. This low precision of compilation percentage could have affected the results in some GA configurations where many individuals failed to compile. Due to the high uncertainty of which individuals actually came furthest in the compilation, the GA could potentially follow the individuals with the worst compilation percentage and risk of never finding an individual that compiles. This makes the parsing of the compilation logs, where a precise compilation percentage is missing, quite unreliable, but would only affect the results if there were no compiling individuals.

Compiling The Projects and Building Fuzzers

The high execution time of the GA was largely due to the fact that each individual had to be compiled twice. First with the project's own build suite and then with OSS-Fuzz to enable fuzzing to avoid false negatives in the project's test suite. This was the main bottleneck for the performance of the GA, and several different approaches were made to try to avoid compiling twice. It is still unclear if this problem was only applicable to the project selected to guide the implementation, or if the rest of the selected projects would have worked correctly compiled only through OSS-Fuzz. An initial check could have been implemented to compare test suite results from the project's own build suite and from OSS-Fuzz, and do double compilation if there were false negatives when compiled through OSS-Fuzz.

Possible False-Positives with The Fitness Function

The fitness function designed for this GA was based on the fitness function used in GenProg [11] with some modifications to also depend on the length of the evaluated diff. One significant edge case with the fitness function was when the number of passed tests was larger than the initial length of the diff. This could have led to an individual who passed many tests but did not re-implement the bug outperforming an individual that passed fewer tests but re-implemented the bug. This would make the GA focus on minimizing the diff while passing as many tests as possible without any regard for whether the bug was re-implemented

or not. To ensure that an individual that re-implemented the bug always had a larger fitness value than any individual that did not re-implement the bug, all tests had to pass to check if the bug had been re-implemented. This requirement also prevented cases where a failing test could have led to a false positive when checking if the bug had been re-implemented. This ensured that the result of whether the bug was re-implemented or not was accurate.

Possibility of a Better GA Configuration

The method used for the parameter tests could affect which parameters were used and their values, since only certain parameters were tested in each parameter test, which could affect the outcome in the following parameter tests. This could have resulted in the parameters used for the GA not being the optimal choice for this study. The initial parameter values for the parameter test could have affected the entire outcome, and some parameters that performed badly during the first test could have performed better if used with parameter values tested in the later tests. To ensure that the optimal parameters and values would have been used, all parameters and values would have to be tested at the same time for all possible combinations. Due to the computational complexity of the GA, evaluating all possible combinations of parameters was not feasible during this study. This makes the result of the parameter test less reliable since if the parameter tests were to be done again with different initial parameter values, a different combination of parameters could have been selected.

Source Criticism

The fitness function for the GA in this study was largely inspired by GenProg [11], where the test cases were mainly used to ensure the functionality in the program remained intact during the repair. Le Goues et al. [11] discussed that this concept was built upon the fact that the test suites for the programs were extensive enough not to introduce other bugs and vulnerabilities during the repair. This risk would then also be present in this study, where the GA managed to reintroduce a vulnerability with all passing tests. However, it could have also introduced unintended bugs that were not covered by the test suite. The risk of this should be lower for this study since the already existing code changes are being kept or removed, instead of editing existing code.

Finally, the literature about GAs was well established, and similar information could be found in papers from many different authors, making the information about GAs well-grounded.

6.3 The Work in a Wider Context

This benchmark test suite generator, while being able to produce powerful test suites with automatic vulnerability verification through ground truth, is still many times slower than synthetic benchmark test suites, such as LAVA-M [7], at generating test suites. Compared to benchmark test suites similar to MAGMA [6], the benchmark test suite generator is faster, or at least less costly, since it requires minimal manual work to generate test suites, which is the most attractive aspect from a user perspective. But a significant downside would be the time taken to generate the test suite, as the risk of vulnerabilities might interfere with each other, resulting in a smaller test suite than intended or no viable test suite at all. Additionally, the generated code changes are not guaranteed to be optimal changes, which could make it harder to determine which parts of the code would be contributing to the vulnerabilities when used to benchmark fuzzers.

There are potentially many severe vulnerabilities in the complex systems that run the world and could be exploited for nefarious purposes. By enabling a way to generate benchmark test suites for fuzzers with code based on these complex systems, it could increase the

performance of fuzzers to catch more of these vulnerabilities before they can be exploited. This, however, comes with an increased energy consumption due to the computational complexity of the benchmark test suite generator. Making the benchmark test suite generator more sustainable is something to aim for, but compared to the much higher energy consumption of fuzzers, it is not the most important aspect to make it a viable competitor among other fuzzer benchmark test suites.



7

Conclusion

This section presents the answers to the research questions and discusses future work for this study.

7.1 Conclusions

The main goal of this thesis was to create a benchmark test suite that could automatically generate test suites for fuzzers with automatic ground truth for real-world bugs to reduce the manual work required during fuzzing evaluations. The created benchmark test suite achieved this by minimizing code changes required to reintroduce vulnerabilities into programs. The benchmark test suite managed to reintroduce all vulnerabilities chosen for this thesis with a quite consistent minimization ratio, except for a few vulnerabilities where the minimization ratio was less consistent. The average time taken to reintroduce a vulnerability was around 1 hour, and the generated test suites included 76% of the vulnerabilities on average.

This thesis shows that the approach to automatically reintroduce vulnerabilities into programs to evaluate fuzzers is a possible and viable option to other fuzzing evaluation methods. The groundwork for further advancements within this approach of fuzzing evaluations has been laid out in this thesis.

7.2 How Can GAs Be Used To Minimize Diffs?

To minimize diffs using GAs, the diffs have to be transformed into a format that can easily be modified by a GA and guarantees a unique mapping between the format used in the GA and the resulting diff. This is crucial in order to accurately determine the objective value of the diff, which is needed to compare different diffs and drive the GA towards diffs with better objective values.

How Can a Diff Be Represented For Use In a GA?

A Diff can be represented as a string of ones and zeros, a bit-vector, where each symbol corresponds to whether a specific file, hunk, or line in the diff would be kept or removed. If the symbol was a "1" the file, hunk, or line was included in the output diff, if the symbol

7.3. How Can Several Vulnerable Versions of a Program With Only One Known Vulnerability Be Merged Into One Program With the Same Vulnerabilities?

was a "0" it was not. This made it possible for the GA to "switch off" a certain file, hunk, or line in the diff by changing the symbol from "1" to "0" regardless of whether the symbol corresponded to a file, hunk, or line.

How Can Diffs Be Evaluated To Drive a GA Towards Better Diffs?

In order to drive a GA to produce better diffs, a number of different evaluations had to be used. The diffs had to be generated based on the bit-vector produced by the GA, compiled, and pass all tests from the diff's test suite, and ensure that the reintroduced vulnerability can still be triggered. The set of bit-vectors that compiled, passed all tests, and reintroduced the vulnerability was combined together using the and operator to get the maximum bit-vector length currently required to reintroduce the vulnerability, and would drive the GA to produce bit-vectors smaller and better diffs in the next generations.

7.3 How Can Several Vulnerable Versions of a Program With Only One Known Vulnerability Be Merged Into One Program With the Same Vulnerabilities?

The set of vulnerable versions of a program could be merged into a single program with the same vulnerabilities through a heuristic. Whenever a vulnerable version was merged into the program with a set of vulnerabilities, the vulnerabilities in the merged program were checked to ensure that they could still be triggered with ground truth, with the added vulnerability.

7.4 Future Work

This work was limited to show that the reintroduced vulnerabilities could be triggered by a fuzzer using the reproducible test case in a program with several known vulnerabilities. Further research needs to be done to explore the actual extendability of this benchmark test suite generator and how well these test suites perform during a fuzz testing session compared to other fuzzing benchmark test suites, such as LAVA-M [7], MAGMA [6], and Darpa CGC [42].

It became clear in this thesis that some adjustments to the behavior of existing functions in GAs were needed to effectively minimize a diff, and there are likely more adjustments to be made here to increase the effectiveness even further.

Over the course of this thesis, code generation with the use of machine learning models (e.g., ChatGPT) has become much better. It would be interesting to see if these models could be used to automate some of the manual parts of this test suite generator, for example, the collection of relevant vulnerabilities and the parsing of the compilation logs and project test suite results. Furthermore, it would be interesting to explore if synthetic vulnerabilities generated through these models could be more similar to real-world vulnerabilities compared to the synthetic vulnerabilities used in LAVA-M [7].



Bibliography

- [1] Herbert H Thompson. “Why security testing is hard”. In: *IEEE Security & Privacy* 1.4 (2003), pp. 83–86.
- [2] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. “Software Vulnerability Discovery Techniques: A Survey”. In: *2012 Fourth International Conference on Multimedia Information Networking and Security*. 2012, pp. 152–156. DOI: 10.1109/MINES.2012.202.
- [3] Patrice Godefroid. “Fuzzing: Hack, art, and science”. In: *Communications of the ACM* 63.2 (2020), pp. 70–76.
- [4] Jun Li, Bodong Zhao, and Chao Zhang. “Fuzzing: a survey”. In: *Cybersecurity* 1.1 (2018), pp. 1–13.
- [5] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. “Evaluating fuzz testing”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 2123–2138.
- [6] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. “Magma: A ground-truth fuzzing benchmark”. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4.3 (2020), pp. 1–29.
- [7] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. “Lava: Large-scale automated vulnerability addition”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 110–121.
- [8] Google. *Google OSS-Fuzz Database Index*. Accessed 2023-08-13. URL: <https://bugs.chromium.org/p/oss-fuzz/issues/list?sort=-reported&q=type%5C%3DBug-Security>.
- [9] Google. *Google OSS-Fuzz*. Accessed 2023-08-13. URL: <https://google.github.io/oss-fuzz/>.
- [10] Darrell Whitley. “A genetic algorithm tutorial”. In: *Statistics and computing* 4.2 (1994), pp. 65–85.
- [11] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. “Genprog: A generic method for automatic software repair”. In: *Ieee transactions on software engineering* 38.1 (2011), pp. 54–72.

- [12] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. "A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each". In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 3–13.
- [13] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. "The art, science, and engineering of fuzzing: A survey". In: *IEEE Transactions on Software Engineering* (2019).
- [14] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. "{UNIFUZZ}: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers". In: *30th {USENIX} Security Symposium ({USENIX} Security 21)*. 2021.
- [15] Brian Caswell. *Cyber Grand Challenge Corpus*. Ed. by Lunge Technology. URL: <http://www.lungetech.com/cgc-corporus/>.
- [16] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. "A review on genetic algorithm: past, present, and future". In: *Multimedia Tools and Applications* 80.5 (2021), pp. 8091–8126.
- [17] Anupriya Shukla, Hari Mohan Pandey, and Deepti Mehrotra. "Comparative review of selection techniques in genetic algorithm". In: *2015 international conference on futuristic trends on computational analysis and knowledge management (ABLAZE)*. IEEE. 2015, pp. 515–519.
- [18] David E Goldberg and Kalyanmoy Deb. "A comparative analysis of selection schemes used in genetic algorithms". In: *Foundations of genetic algorithms*. Vol. 1. Elsevier, 1991, pp. 69–93.
- [19] Brad L Miller and David E Goldberg. "Genetic algorithms, selection schemes, and the varying effects of noise". In: *Evolutionary computation* 4.2 (1996), pp. 113–131.
- [20] Abid Hussain, Yousaf Shad Muhammad, and Muhammad Nauman Sajid. "Performance evaluation of best–worst selection criteria for genetic algorithm". In: *Math Comput Sci* 2.6 (2017), pp. 89–97.
- [21] Tobias Blickle and Lothar Thiele. "A comparison of selection schemes used in evolutionary algorithms". In: *Evolutionary Computation* 4.4 (1996), pp. 361–394.
- [22] Ahmad Hassanat, Khalid Almohammadi, Esra'a Alkafaween, Eman Abunawas, Awni Hammouri, and VB Surya Prasath. "Choosing mutation and crossover ratios for genetic algorithms—a review with a new dynamic approach". In: *Information* 10.12 (2019), p. 390.
- [23] Siew Mooi Lim, Abu Bakar Md Sultan, Md Nasir Sulaiman, Aida Mustapha, and Kuan Yew Leong. "Crossover and mutation operators of genetic algorithms". In: *International journal of machine learning and computing* 7.1 (2017), pp. 9–12.
- [24] Anant J Umbarkar and Pranali D Sheth. "Crossover operators in genetic algorithms: a review." In: *ICTACT journal on soft computing* 6.1 (2015).
- [25] Padmavathi Kora and Priyanka Yadlapalli. "Crossover operators in genetic algorithms: A review". In: *International Journal of Computer Applications* 162.10 (2017).
- [26] Dirk Thierens and David Goldberg. "Elitist recombination: An integrated selection recombination GA". In: *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*. IEEE. 1994, pp. 508–512.
- [27] Congrui Yang, Qian Qian, Feng Wang, and Minghui Sun. "An improved adaptive genetic algorithm for function optimization". In: *2016 IEEE International Conference on Information and Automation (ICIA)*. IEEE. 2016, pp. 675–680.
- [28] Il-Seok Oh, Jin-Seon Lee, and Byung-Ro Moon. "Hybrid genetic algorithms for feature selection". In: *IEEE Transactions on pattern analysis and machine intelligence* 26.11 (2004), pp. 1424–1437.

-
- [29] Martin Safe, Jessica Carballido, Ignacio Ponzoni, and Nelida Brignole. “On stopping criteria for genetic algorithms”. In: *Brazilian Symposium on Artificial Intelligence*. Springer. 2004, pp. 405–413.
- [30] Git. *Official Git Documentation*. Accessed 2026-05-13. URL: <https://git-scm.com/docs/git>.
- [31] Git. *Official Git Documentation*. Accessed 2026-05-13. URL: <https://git-scm.com/docs/git-commit>.
- [32] Git. *Official Git Diff Documentation*. Accessed 2023-08-13. URL: <https://git-scm.com/docs/git-diff>.
- [33] GNU. *Official GNU Diffutils Documentation*. Accessed 2023-08-13. URL: https://www.gnu.org/software/diffutils/manual/html_node/.
- [34] Git. *Official Git Patch Documentation*. Accessed 2023-08-13. URL: <https://git-scm.com/docs/git-apply>.
- [35] Eric Schulte, Stephanie Forrest, and Westley Weimer. “Automated program repair through the evolution of assembly code”. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 2010, pp. 313–316.
- [36] ntop.org. *nDPI*. Accessed 2023-12-28. URL: <https://github.com/ntop/nDPI>.
- [37] mruby.org. *mruby*. Accessed 2023-12-28. URL: <https://github.com/ntop/nDPI>.
- [38] GNOME. *libXML2*. Accessed 2023-12-28. URL: <https://github.com/GNOME/libxml2>.
- [39] David Anderson, Alberto Carlos Enciso, and Vincent Torri. *libdwarf*. Accessed 2023-12-28. URL: <https://github.com/davea42/libdwarf-code>.
- [40] László Németh. *Hunspell*. Accessed 2023-12-28. URL: <https://github.com/hunspell/hunspell>.
- [41] Google. *OSS-Fuzz Issue Tracker*. Accessed 2023-05-01. URL: <https://bugs.chromium.org/p/oss-fuzz/issues/list?sort=-id%20-reported&q=type%3DBug-Security%20status%3DVerified&can=1>.
- [42] DARPA. *DARPA Cyber Grand Challenge*. Accessed 2023-08-13. URL: <https://github.com/CyberGrandChallenge/>.



A Appendix

A.1 Detailed Stats

| Vulnerability | Input Diff Size | Output Diff Size | Size Reduction | Time Taken (s) |
|----------------|-----------------|------------------|----------------|-----------------|
| nDPI 57448 | 4 581 | 13 | 99.7% | 693 s |
| nDPI 57369 | 4 639 | 13 | 99.7% | 680 s |
| nDPI 57317 | 4 649 | 23 | 99.5% | 1 003(±9) s |
| nDPI 56272 | 7 270 | 13 | 99.8% | 1 105 s |
| nDPI 55218 | 73 295 | 13 | 99.9% | 9 864(±1 194) s |
| mruby 57037 | 4 561 | 22 | 89% | 1 574(±11) s |
| mruby 56991 | 4 689 | 52 | 99.5% | 3 390(±122) s |
| mruby 56889 | 4 774 | 17(±4) | 99.7(±0.1)% | 2 036(±24) s |
| mruby 56406 | 7 698 | 291(±438) | 96.2(±5.7)% | 9 858(±2 185) s |
| mruby 53183 | 25 917 | 13 | 99.9% | 1 730(±7) s |
| libxml2 57521 | 10 490 | 17(±3) | 99.8% | 4 046(±10) s |
| libxml2 57469 | 10 512 | 32 | 99.7% | 4 448(±41) s |
| libxml2 57410 | 10 526 | 12 | 99.9% | 3 906(±2) s |
| libxml2 57304 | 11 347 | 77 | 99.3% | 8 703(±201) s |
| libxml2 57294 | 11 347 | 84(±7) | 99.3(±0.1)% | 9 735(±498) s |
| libdwarf 57527 | 7 157 | 12 | 99.8% | 744(±6) s |
| libdwarf 57442 | 9 576 | 24 | 99.7% | 1 946(±153) s |
| libdwarf 57437 | 9 837 | 12 | 99.9% | 941(±16) s |
| libdwarf 57429 | 9 837 | 24 | 99.8% | 1 329(±60) s |
| libdwarf 56906 | 14 522 | 134(±39) | 99.1(±0.3)% | 2 094(±219) s |
| hunspell 56737 | 157 | 26 | 83.4% | 1 169(±9) s |
| hunspell 55818 | 619 | 13 | 97.9% | 495(±27) s |
| hunspell 55191 | 705 | 12 | 98.3% | 798(±46) s |
| hunspell 54672 | 1 090 | 12 | 98.9% | 1 056(±82) s |
| hunspell 54244 | 2 307 | 13 | 99.4% | 1 196(±40) s |

Table A.1: Result of the GA from the initial diffs to the final diffs.

| Vulnerability | Input Diff Size | Output Diff Size | Size Reduction | Time Taken (s) |
|----------------|-----------------|-------------------|-------------------|---------------------|
| nDPI 57448 | 4 581 | 52 | 98.9% | 506 s |
| nDPI 57369 | 4 639 | 145 | 96.9% | 488 s |
| nDPI 57317 | 4 649 | 62 | 98.7% | 487 s |
| nDPI 56272 | 7 270 | 66 | 99.1% | 594 s |
| nDPI 55218 | 73 295 | 44 631(\pm 23) | 39.1% | 5 035(\pm 174) s |
| mruby 57037 | 4 561 | 503 | 89% | 1 574(\pm 11) s |
| mruby 56991 | 4 689 | 373(\pm 238) | 92(\pm 5.1)% | 2 245(\pm 64) s |
| mruby 56889 | 4 774 | 159 | 96.7% | 1 050 s |
| mruby 56406 | 7 698 | 620(\pm 747) | 91.9(\pm 9.7)% | 3 777(\pm 335) s |
| mruby 53183 | 25 917 | 69 | 99.7% | 1 455(\pm 7) s |
| libxml2 57521 | 10 490 | 956 | 90.9% | 3 401(\pm 3) s |
| libxml2 57469 | 10 512 | 1 034 | 90.2% | 2 849(\pm 2) s |
| libxml2 57410 | 10 526 | 970 | 90.8% | 3 413(\pm 2) s |
| libxml2 57304 | 11 347 | 3 147 | 72.3% | 2 968(\pm 11) s |
| libxml2 57294 | 11 347 | 3 016 | 73.4% | 2 861(\pm 8) s |
| libdwarf 57527 | 7 157 | 1 461 | 79.6% | 385(\pm 6) s |
| libdwarf 57442 | 9 576 | 684 | 92.9% | 697(\pm 60) s |
| libdwarf 57437 | 9 837 | 1 502 | 84.7% | 556(\pm 15) s |
| libdwarf 57429 | 9 837 | 488 | 95% | 690(\pm 28) s |
| libdwarf 56906 | 14 522 | 410 | 97.2% | 220 s |
| hunspell 56737 | 157 | 77 | 51% | 231(\pm 1) s |
| hunspell 55818 | 619 | 15 | 97.6% | 179 s |
| hunspell 55191 | 705 | 101 | 85.7% | 561(\pm 46) s |
| hunspell 54672 | 1 090 | 235 | 78.4% | 687(\pm 69) s |
| hunspell 54244 | 2 307 | 162 | 93% | 713(\pm 40) s |

Table A.2: Result of the GA from the initial diffs to the file output diffs.

| Vulnerability | Input Diff Size | Output Diff Size | Size Reduction | Time Taken (s) |
|----------------------|------------------------|-------------------------|-----------------------|------------------------|
| nDPI 57448 | 52 | 13 | 75% | 121 s |
| nDPI 57369 | 145 | 13 | 91% | 121 s |
| nDPI 57317 | 62 | 23 | 62.9% | 289(± 1) s |
| nDPI 56272 | 66 | 29 | 56.1% | 125 s |
| nDPI 55218 | 44 631(± 23) | 30 | 99.9% | 4 567($\pm 1 119$) s |
| mruby 57037 | 503 | 32 | 93.6% | 740(± 23) s |
| mruby 56991 | 373(± 238) | 63 | 77.1(± 9.4)% | 765(± 552) s |
| mruby 56889 | 159 | 61 | 61.6% | 348(± 5) s |
| mruby 56406 | 620(± 747) | 320(± 453) | 58.3(± 8.1)% | 1 546($\pm 1 856$) s |
| mruby 53183 | 69 | 13 | 81.2% | 177 s |
| libxml2 57521 | 956 | 30 | 96.9% | 367 s |
| libxml2 57469 | 1 034 | 41 | 96% | 1 215(± 39) s |
| libxml2 57410 | 970 | 18 | 98.1% | 365 s |
| libxml2 57304 | 3 147 | 86 | 97.3% | 3 593(± 97) s |
| libxml2 57294 | 3 016 | 100(± 14) | 96.7(± 0.5)% | 4 242(± 167) s |
| libdwarf 57527 | 1 461 | 20 | 98.6% | 308 s |
| libdwarf 57442 | 684 | 55 | 92% | 454(± 17) s |
| libdwarf 57437 | 1 502 | 27 | 98.2% | 325(± 3) s |
| libdwarf 57429 | 488 | 43 | 91.2% | 248(± 1) s |
| libdwarf 56906 | 410 | 176 | 56.3% | 171 s |
| hunspell 56737 | 77 | 68 | 11.7% | 186(± 3) s |
| hunspell 55818 | 15 | 15 | 0% | 54(± 3) s |
| hunspell 55191 | 101 | 13 | 87.1% | 120 s |
| hunspell 54672 | 235 | 14 | 94% | 239 s |
| hunspell 54244 | 162 | 19 | 88.3% | 229 s |

Table A.3: Result of the GA from the file output diffs to the coarse output diffs.

| Vulnerability | Input Diff Size | Output Diff Size | Size Reduction | Time Taken (s) |
|----------------|------------------|------------------|---------------------|----------------------|
| nDPI 57448 | 13 | 13 | 0% | 66 s |
| nDPI 57369 | 13 | 13 | 0% | 71 s |
| nDPI 57317 | 23 | 23 | 0% | 227(± 10) s |
| nDPI 56272 | 29 | 13 | 55.2% | 386 s |
| nDPI 55218 | 30 | 13 | 56.7% | 262(± 2) s |
| mruby 57037 | 32 | 22 | 31.3% | 1 076(± 108) s |
| mruby 56991 | 63 | 52 | 17.5% | 3 913(± 538) s |
| mruby 56889 | 61 | 17(± 4) | 72.6(± 6.8)% | 638(± 25) s |
| mruby 56406 | 320(± 453) | 291(± 438) | 18.7(± 7)% | 4 534(± 574) s |
| mruby 53183 | 13 | 13 | 0% | 100 s |
| libxml2 57521 | 30 | 17(± 3) | 43.7(± 10.7)% | 278(± 6) s |
| libxml2 57469 | 41 | 32 | 22% | 385(± 12) s |
| libxml2 57410 | 18 | 12 | 33.3% | 128 s |
| libxml2 57304 | 86 | 77 | 10.5% | 2 141(± 148) s |
| libxml2 57294 | 100(± 14) | 84(± 7) | 14.9(± 11)% | 2 632(± 422) s |
| libdwarf 57527 | 20 | 12 | 40% | 51 s |
| libdwarf 57442 | 55 | 24 | 56.4% | 796(± 90) s |
| libdwarf 57437 | 27 | 12 | 55.6% | 60(± 5) s |
| libdwarf 57429 | 43 | 24 | 44.2% | 391(± 38) s |
| libdwarf 56906 | 179 | 134(± 39) | 25(± 21.7)% | 1 703(± 219) s |
| hunspell 56737 | 68 | 26 | 61.8% | 752(± 11) s |
| hunspell 55818 | 15 | 13 | 13.3% | 263(± 28) s |
| hunspell 55191 | 13 | 12 | 7.7% | 117 s |
| hunspell 54672 | 14 | 12 | 14.3% | 131(± 18) s |
| hunspell 54244 | 19 | 13 | 31.6% | 254(± 1) s |

Table A.4: Result of the GA from the coarse output diffs to the final diffs.

| Project | Vulnerability | Rejection Reason |
|----------|---------------|---------------------------------------|
| nDPI | 57317 | Conflicting changes |
| mruby | 56406 | Conflicting changes |
| libxml2 | 57304 | Failed to trigger all vulnerabilities |
| libxml2 | 57294 | Failed to trigger all vulnerabilities |
| libdwarf | 56906 | Conflicting changes |
| hunspell | 54244 | Conflicting changes |

Table A.5: Diffs which failed to be merged into their test suite.