

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**Esa Jääskelä**

**Genetic Algorithm in Code Coverage Guided Fuzz Testing**

Master’s Thesis

Degree Programme in Computer Science and Engineering December 2015

**Jääskelä E. (2015) Genetic Algorithm in Code Coverage Guided Fuzz Testing.** University of Oulu, Department of Computer Science and Engineering. Master’s Thesis, 41 p.

**ABSTRACT**

**The security of computers is a growing concern when the amount of devices increases. New and more comprehensive testing methods need to be done to avoid damages to the users and their computers. Fuzzing is a testing method that inserts semi-valid input to the tested system and has before been considered as a good method for the security testing. However, it usually either does not get high code coverage or it requires a long set-up process or a source code analysis to achieve better code coverage. This work presents a genetic algorithm that automatically balances the probabilities of multiple mutators in a fuzzing program. This balancing aims to maximize the code coverage fuzz testing. After fuzzing two different open source libraries it was found that the grey-box approach in fuzzing gives better results than pure black-box fuzzing.**

**Keywords: security testing, optimization,**

**Jääskelä E. (2015) Geneettinen algoritmi koodin kattavuuden perusteella ohjatussa fuzzauksessa.** Oulun yliopisto, tietotekniikan osasto. Diplomityö, 41 s.

**TIIVISTELMÄ**

**Tietokoneiden tietoturva on kasvava huolenaihe, kun laitteiden määrä lisääntyy. Uusia ja kattavampia testauksia täytyy suorittaa, jotta voidaan estää käyttäjille ja heidän laitteilleen tapahtuvat vahingot. Fuzzausta on pidetty hyvänä testausmetodina, mutta yleensä se ei saavuta hyvää koodikattavuutta tai vaatii joko monimutkaisen asennuksen tai lähdekoodianalyysin. Tämä työ esittelee geneettisen algoritmin, joka automaattisesti tasapainottaa fuzzerin eri mutaatiofunktioiden todennäköisyydet. Tämä tasapainotus pyrkii maksimoimaan saavutetun koodikattavuuden ja parantamaan fuzzaamisen tehokkuutta. Kahden avoimen lähdekoodin kirjaston testaamisen perusteella mutatorit koodikattavuuden perusteella tasapainottava työkalu pärjäsi paremmin kuin perinteinen, lisätietoa hyödyntämätön black-box fuzzaus.**

**Avainsanat: tietoturvatestaus, optimointi**

**TABLE OF CONTENTS**

[ABSTRACT](#_bookmark0) [TIIVISTELMÄ](#_bookmark1)

[TABLE OF CONTENTS](#_bookmark2) [FOREWORD](#_bookmark3)

[LIST OF ABBREVIATIONS AND SYMBOLS](#_bookmark4)

1. [INTRODUCTION 7](#_bookmark5)
2. [BACKGROUND AND EARLIER WORK 8](#_bookmark6)
   1. [Fuzz Testing 8](#_bookmark7)
      1. [Vulnerabilities discovered with fuzzing 9](#_bookmark8)
   2. [Dynamic analysis methods 11](#_bookmark9)
      1. [Code coverage 11](#_bookmark10)
      2. [Other dynamic analysis methods 12](#_bookmark11)
      3. [Dynamic analysis methods in fuzz testing 13](#_bookmark12)
   3. [Static analysis methods 14](#_bookmark13)
   4. [Genetic Algorithms 15](#_bookmark14)
      1. [Multi Objective Genetic Algorithms 15](#_bookmark15)
      2. [Genetic algorithms in fuzz testing 16](#_bookmark16)
3. [IMPLEMENTATION 17](#_bookmark17)
   1. [Description of the system 19](#_bookmark18)
   2. [Input files 20](#_bookmark19)
   3. [Fuzzer 20](#_bookmark20)
   4. [Instrumentation of tested systems 22](#_bookmark21)
      1. [SanitizerCoverage 22](#_bookmark22)
      2. [AddressSanitizer, MemorySanitizer and LeakSanitizer 23](#_bookmark23)
   5. [Genetic Algorithm 24](#_bookmark24)
4. [EXPERIMENTS 27](#_bookmark25)
   1. [Black-box fuzzing 28](#_bookmark26)
   2. [Tested applications 28](#_bookmark27)
      1. [Librsvg 29](#_bookmark28)
      2. [Libav 29](#_bookmark29)
   3. [Experiment and AMGA preparations 30](#_bookmark30)
5. [RESULTS 31](#_bookmark31)
   1. [Effect of the mutator balancing on code coverage 32](#_bookmark32)
   2. [Effect of the variables in the genetic algorithm on code coverage 34](#_bookmark33)
   3. [Effect of the mutator balancing on the mutator probabilities 35](#_bookmark34)
6. [CONCLUSION 37](#_bookmark35)
7. [REFERENCES 38](#_bookmark36)

**FOREWORD**

This thesis was written at OUSPG as one of their Google Research Award theses. During these 6 months of writing, I met multiple people from both Google and OUSPG and I would like to thank all of them who have helped and supported me during this time. Especially I would like to thank Professor Juha Röning, Thomas Schaberreiter and Christian Wieser for supervising this work. I would also like to thank Atte Kettunen for his support in background research and especially with the technical implementation. Finally, I would like to thank my fiancé Sanna Mäkelä and our dog Sulo who have managed to keep me going during this long project.

Oulu, 18.12.2015

Esa Jääskelä

**LIST OF ABBREVIATIONS AND SYMBOLS**

AFL American-fuzzy-lop

AMGA Automatic Mutator-balancing Genetic Algorithm AVI Audio Video Interleave

BTS Branch Trace Store

CVE Common Vulnerabilities and Exposures CPU Central Processing Unit

JPEG Joint Photographic Experts Group LOC Lines Of Code

OUSPG Oulu University Secure Programming Group SAGE Scalable Automated Guided Execution

SUT System Under Test

# INTRODUCTION

During these times when computers surround us and constantly become more widely used, the security of these devices is an important issue. When devices start to be smarter and more interconnected, it is crucial that malicious entities cannot attack and control these computers as they will. In 2014, more CVEs, warnings about vulnerabilities and exposures on computers, were reported than any year before [1]. This, combined with the fact that the amount of computers connected to the Internet is in a steady rise, [2] means that the amount of exploitable software and devices is constantly increasing.

To create more secure software, many aspects need to be considered, ranging from the initial design and programming all the way to the testing and deployment of the completed system. Some of the possible vulnerabilities can be caught before the actual testing with good coding practices and auditing, but testing is mandatory for a functional and secure system as it should reveal unnoticed vulnerabilities. However, writing tests for software can be a time consuming task and may be overlooked during a tight development schedule.

For this reason additional automatable and comprehensive testing should be performed to the developed products. No single testing method can completely cover all possible failures, but more efficient and easier methods are constantly researched. Fuzzing is a useful testing method that can be used in many different fields of software to find security issues [3]. Not only computer programs, but for example also network protocols and embedded systems can be tested with fuzzing. Fuzzing is a negative test, which inserts edited input data into the system under test (SUT). This data can be completely random, mutated from an existing valid data samples or self- generated under certain rules [4].

Although fuzzing sounds like a simple testing method, it often uncovers serious vulnerabilities that could lead to a situation where an attacker can execute arbitrary code. One example of such situation is a stack smash attack [5]. To uncover these serious faults faster it is usually not enough to use pure black-box fuzzing that only feeds random input without understanding what happens in the tested system. Instead of this simple approach some information of the application state should be used to generate more intelligent fuzzing systems.

For creating such intelligent fuzzing environment, AMGA (Automatic Mutator- balancing Genetic Algorithm) was created. In AMGA a genetic algorithm balances the mutators of a fuzzer in a grey-box testing environment. This genetic algorithm uses information on the code coverage obtained during the execution of the tested system to guide the fuzzer to automatically use the best possible mutators. A mutator in this context means one type of mutation that the fuzzer performs to sample files. Fuzzers usually have different kinds of mutation operations, for example bitflips, byte copies and byte removals. Earlier work focuses on the optimization of the input files instead of the mutators when maximizing test coverage.

The designed system is used to answer a research question “What is the effect of the mutator balancing on the amount of code covered with fuzzing”. To answer this question, a test system was created where the results of AMGA were compared to the results of simple black-box fuzzing. These test runs were made to two open source programs, librsvg and libav, and to three different filetypes; SVG, MP4, and H.263. After fuzzing it was found that the AMGA achieved more code coverage than the pure black-box fuzzing with 8.7% slowdown.

# BACKGROUND AND EARLIER WORK

This thesis presents a fuzzing tool that uses a genetic algorithm to improve the code coverage of fuzzing. Fuzzers often provide multiple mutation operators, such as bitflips, byte removals and string copies. Different operations can be applied in a single test case with different probabilities. For example, one fuzzer output can contain a bit flip with 50% probability, byte removal with 21% probability and so on. To improve the efficiency of the fuzzing these different probabilities should be balanced according to the tested system. The tool presented in this thesis automatically balances the mutator probabilities by using code coverage information in an attempt to discover new blocks to be tested. Maximization is done with an applied genetic algorithm, where fitness function a set of probabilities is controlled by the code coverage, crashes and hangs.

Earlier work considering the genetic algorithm with fuzzing has had a different approach, considering the function paths and optimizing the coverage of different paths [6] [7]. Commonly, instead of code coverage, these researches use taint analysis. This analysis analyzes what inputs affect different variables during the execution of the program. These researches also often use static analysis, that analyses source code instead of the executions of the program, as additional information. On the other hand, code coverage is used with fuzzing guiding approaches of other kind. However, this earlier code coverage guiding is not related to balancing the mutators, instead it focuses on balancing and ordering of the input files to increase coverage.

## Fuzz Testing

First mention of fuzzing used in research is from 1990 [8], when Miller et al. created a tool named “fuzz” that created random input strings for their reliability tests. The interest towards fuzzing has been rising during the past few years when its potential has been improved, and it is still a rather new field of research. The primary idea of the fuzz testing is to create semi-valid test data either by mutating existing, valid data or by generating data with certain rules [4]. Fuzzers that modify existing test cases to create new ones are called mutation-based fuzzers. On the other hand, fuzzers that create the test cases themselves are called generation-based fuzzers. Both generation- and mutation-based fuzzers have to fuzz the test cases they have. They can do the fuzzing either randomly or with pre-defined rules that have been given to them [9].

During fuzz testing multiple steps are taken. First thing to do is deciding the attack surface of the tested system. This attack surface is some input point to the system, usually the one that can be affected with user input. After choosing this input point, the initial samples need to be gathered. In case of a generation-based fuzzer, the generation rules for the samples need to be defined. Next step is instrumenting the source code with a tool that can detect wanted faults in the system. After the system is instrumented and built, the actual fuzzing can begin. When the fuzzing is finished the error messages given by the instrumentation are analyzed to see if there are actual faults or security vulnerabilities in the system.

Black-box fuzzing is a form of fuzz testing that does not utilize any information about the system it is fuzzing. A black-box fuzzer instead blindly mutates or generates data and feeds it as an input to the system without considering what

happens in the SUT [10]. Total black-box fuzzing without any knowledge about the state of the tested application can rarely reach high code coverage and is considered rather inefficient [11].

Because black-box fuzzers often get poor coverage, the fuzzers usually utilize some information about the program status. The fuzzers that use additional information are called grey-box fuzzers. This external information can be any interesting information the SUT produces. Commonly used information is code coverage, which tells what pieces of code are executed in the SUT, or memory and CPU usage, which tell how the SUT uses given resources [9]. Depending on the collected information fuzzer behaves differently: in case of code coverage the fuzzer may attempt to maximize the code coverage and in the case of CPU usage the fuzzer may attempt to investigate what kind of fuzzing causes the abnormal resource usage.

White-box fuzzers on the other hand utilize the actual source code of the system [11]. This source code is used to infer the detailed behavior and constraints of the SUT and this information is then used to create test-cases. This source code can be either in some programming language source files or machine code inferred from binaries [11]. This way a white-box fuzzer can for example ensure that all paths are tested, because it has the information about all the paths from the source code. Grey- box fuzzer on the other hand can use the code coverage to see what paths have been explored already, but it cannot know if all the paths have been explored because it does not use the source code of the SUT for that information. Figure 1 illustrates these differences between different approaches to the fuzzing. An arrow in the diagram displays information flow from one entity to another.

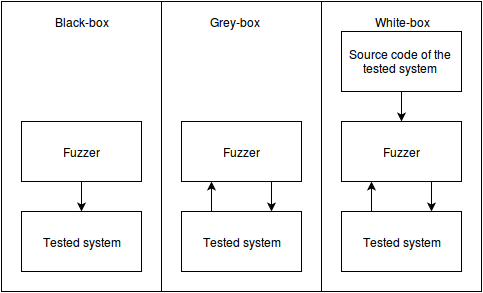


Figure 1: Comparison of different types of fuzzing.

### *2.1.1. Vulnerabilities discovered with fuzzing*

Fuzzing focuses on feeding tested application with semi-valid fuzzed input. This is why common fuzzing targets include “files, registry entries, APIs, user interfaces, network interfaces, database entries, and command line arguments” [9] that are common inputs for applications. When searching for the possible weak input points in the system that a malicious user could try to exploit, it is also important to

consider reachability [6]. Reachability means that only some pieces of code can be reached from one input. When looking for exploitable vulnerabilities, it is more important to focus on the sections of code that a user can control directly with his input, and leave the unreachable sections and sections where user has only indirect access to lesser focus. Fuzzing covers both directly and indirectly reachable sections, because fuzzing focuses on creating inputs to the system.

Table 1 presents a categorization for different kinds of errors. From there, it can be seen that fuzzing finds resource errors, which are common sources for security vulnerabilities [3]. Vulnerabilities that fuzzers usually find are buffer overflows [12]. This makes fuzz testing a good choice for security testing, since overflows are the third most common type of CVEs [1]. The efficiency of the fuzzer is high when the fuzzer is capable of exploiting edge and corner cases, for example by changing integers in the sample data to other numbers known to cause buffer overflows and crashes. Another way of improving the fuzzer performance is reading control statements and directing application to rarely used blocks by satisfying different constraints in the control statements [11].

Table 1. Categorization of errors

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Examples** | **How discovered** | **Problems created** |
| **Syntax error** | Missing semicolons, typing errors | IDE and compiler errors | Unbuildable program |
| **Arithmetic error** | Incorrect calculations, loss of accuracy with decimals | Unit testing, static code analysis | Unpredictable behavior |
| **Resource errors** | Null references, use-after-frees, use-before-inits, buffer overflows | Unit testing, fuzz testing, static code analysis, IDE and compiler warnings | Security vulnerabilities, crashing and unpredictable behavior |
| **Logic errors** | Off-by-one errors, wrong if- statements | Unit testing, static code analysis | Unpredictable behavior |
| **Multithreading errors** | Race-conditions, deadlocks | Thread testing tools, unit testing | Hanging or crashing programs, unpredictable behavior |

The buffer related errors can be divided into these categories: double frees, use before inits, out-of-bounds accesses and buffer overflows. Use before init uses the variable before its memory has been allocated, which causes vulnerabilities in C and C++ languages where uninitialized variables contain previous information. This fault occurs also on other variable types than buffers. Out-of-bounds error reads data outside the allocated space of the buffer. Double free fault frees the same memory location twice causing corrupt memory data management. Buffer overflow writes outside the allocated space of the buffer. Reading outside the intended variable either by use before init or out-of-bounds read is not considered automatically an

exploitable vulnerability. However, it still provides information of the return addresses or other variables for the attacker, and this information can help crafting a larger attack. Buffer overflow or double-free on the other hand possibly allows the attacker to execute arbitrary code in the program, which is always a serious threat. This can be performed for example by creating a stack smash attack [5]. In this attack attacker overwrites the return address to point to some other code he has written in the memory, for example to launch a root shell.

## Dynamic analysis methods

Code analysis methods are used to collect information about what happens in the application during its execution. This information can be collected either statically from source code of an application or dynamically at run-time. In static methods, the source code is analyzed without actually compiling and running it. Run-time methods instrument code with specific commands, compile and collect the information about the commands executed when the application is run. The weakness of static methods is that it is quite difficult to obtain accurate information on how the program behaves without actually running the program [7]. At the same time dynamic run-time methods tend to suffer a slowdown caused by the instrumentation commands and the overhead of storing the information from the commands [13].

Dynamic analysis methods are commonly used in security testing because they provide actual and real-time information on what happens in the system during testing [14]. Code coverage is the simplest form of dynamic analysis; it only collects the information about the sections of code that were executed during the application run time. Two other most commonly used dynamic analysis methods are dynamic taint analysis, which analyses how the user input affects the variables in the system, and forward symbolic execution, which analyses the branches that a certain input encounters in the system [15]. Also more commonly used tests, such as unit tests, integration tests and acceptance tests, alongside with assertions can be considered as one form of dynamic code analysis. Usually multiple methods are used in conjunction to create analysis systems where one method supports another [14].

### *Code coverage*

Code coverage is a simple dynamic analysis and is commonly used measuring how comprehensive the testing has been. Code coverage consists of splitting the source code to “blocks” with certain granularity and tracking what blocks were encountered during run time. Sometimes the amount a certain block was encountered during testing is also recorded. Granularity of the measured coverage can vary, and multiple definitions exist for the names of different granularities, but three commonly used definitions are block coverage, decision coverage and condition coverage [16] [17]. In block coverage, a block means a piece of code that has no if-statements or other control statements that would direct execution away from the block. During run time the coverage of these blocks is calculated. Decision coverage on the other hand looks for places where application makes decisions, mainly if-statements, and then analyzes how well all the possible decisions are covered. Condition coverage also searches for the if-statements and tries to track if all the different boolean values in

these branches have been tested. However, the condition coverage will not ensure the decision coverage if application has if-statements that have multiple conditions to check.

In the context of security research, the code coverage will not provide information if the program contains vulnerabilities or not. This is because code coverage information only tells which sections of the code are run during execution. Code coverage does not perform actual analysis of what occurs in the system. This analysis needs to be done outside the coverage instrumentation, for example by additional instrumentation or by analyzing other actions system performs. However, the code coverage provides accurate information about how well a certain test actually tests the system. By utilizing this information, the code coverage can be used to create better testing tools.

Researchers working on the code coverage have been studying if the code coverage is a good measurement of the fault coverage. Fault coverage is a measurement that indicates how big part of the faults in the program have been uncovered and fixed. Del Frate et al. [17] suggested that there is a simple correlation between the code coverage obtained in testing and software reliability. However, Briand and Pfahl [16] suggested in more recent work that there is not always a clear correlation between the code coverage and the fault coverage, and the relationship between these two variables would be a more complex correlation. Cai and Lyu [18] arrived at a similar conclusion, suggesting that code coverage is “moderate” indicator of fault coverage. This would mean that if the code coverage is, for example, 80% the fault coverage can be significantly higher or lower than this code coverage.

The amount of code coverage that can be reached with different testing and fuzzing methods has also been a subject of research. In his text Marick [19] suggested that good code coverage for custom written tests is around 90%-95%, and during software development this level should be reached. Godefroid et al. compared white-box, black-box and custom tests in their research [20]. Their custom test suite reached 58.8% coverage. Mutation-based black-box and white-box fuzzers got coverages of 14.2% and 14.7%. Generation-based white-box fuzzing was the most successful fuzzing method, gaining 20% coverage. This result is similar to results presented in [4], where it was suggested that generation based methods can reach up to 76% better coverage than mutation based methods.

A comparison between different mutation based fuzzing methods was done by Neystadt [21]. His results showed that black-box fuzzers that did not have any information on what they were fuzzing got the coverage of 50%, which was significantly lower than the 80% achieved when the black-box fuzzer had certain rules that it used for fuzzing. Similarly the “dumb” white-box fuzzer mutating data without any constraints got the code coverage of 80% and the smart white-box fuzzer got as much as 99%.

### *Other dynamic analysis methods*

Dynamic taint analysis analyses the program flow during runtime to analyze what inputs affect which variables [14]. The effects of inputs can be either implicit or explicit. The explicit effects affect variables directly, for example by adding a value of from input to a variable. Implicit effects on the other hand are indirect. One example of such effect is for example taking a branch with certain input, where a specific value is set in a variable. Examples of both of these cases are displayed in

the Figures 2 and 3. Tainted variable will further taint other variables it affects [14]. In security testing, dynamic taint analysis can be used for example to check if untrusted user input is able to overwrite return addresses [13].

int example (int input) { return 5 + input;

}

Figure 2. Example of explicit effect of input.

int example (int input){

if(input > 5)

return 5;

else

return 0;

}

Figure 3. Example of implicit effect of input.

A problem with dynamic taint analysis is over-tainting and under-tainting [22]. Over-tainting propagates taint to variables that are not actually affected by the input and under-tainting does not propagate taint when it should. This in turn can lead to false negatives or positives in security testing. This problem can be mitigated with the usage of different taint policies or taint sanitization [15]. Different taint policies apply taint differently, for example tainted jump policy tracks the status of variable address and variable value separately, whereas tainted address policy tracks the status of both of them together. Sanitization on the other hand cleans taints periodically to avoid having too large taint spread, which often leads to inaccurate tainting [15].

The forward symbolic execution takes a different approach to the dynamic code analysis. It analyzes the branches a certain input of the system faces during its execution. From this information, analysis tries to understand the behavior of the program by creating a logical formula of the program flow [15]. For the smart fuzzers, this can be very useful, because they can get information about how to fuzz samples to reach new paths. Similarly for generation based fuzzers this information is useful for guiding the creation of new samples.

The problem with forward symbolic execution is that it requires significant performance. The forward symbolic approach gets exponentially complex in each branch it needs to analyze [15]. In TaintCheck [13] Newsome and Song used dynamic taint analysis, and in worst-case scenarios the taint analysis slowed the application 24 times. On average the application was slowed 6 times.

### *Dynamic analysis methods in fuzz testing*

SAGE is a fuzzing tool that utilizes information about program constraints to generate better test cases for fuzzing [11]. It executes a program with valid input and

by using a machine code based approach collects constraints the input receives during its handing. After constraint generation SAGE modifies the input file so that it matches different constraints, thus getting different parts of code covered.

AFL is a feedback-driven fuzzing tool that utilizes code coverage [23]. The tested program is compiled with AFL instrumentation that tracks which code blocks were executed and approximately how often they were executed. It loads samples, fuzzes them and inputs them to the SUT. After running the program, it analyses what parts of the program were covered. If new parts were covered, AFL marks the input file as “interesting”. This means it is more likely to be selected for further test cases. However, AFL does not take into account mutators used and their effect on coverage. LibFuzzer is similarly a feedback-driven code coverage utilizing fuzzer [24]. The main difference between AFL and LibFuzzer is that LibFuzzer utilizes in-process fuzzing. Motivation for this choice is potentially faster fuzzing. On the downside, in- process fuzzing is more unstable and restrictive than regular fuzzing. LibFuzzer uses compile-time instrumentation and input files which are mutated to get the samples

for fuzzing.

HongFuzz is a fuzzing tool that utilizes the perf subsystem of kernel and the BTS available in the Intel processors as feedback to improve the fuzzing efficiency [25]. User can choose between instruction count, branch count and unique branch pair count as feedback value that guides the fuzzing. During fuzzing, the chosen counter will be attempted to be maximized.

## Static analysis methods

Static analysis methods differ from the dynamic methods in that they analyze the application and its source code without actually running the application. The depth of the analysis can range from simply going through the source code line by line looking for warnings and errors to creating control flow graphs and analyzing larger units to create approximations how the system behaves during an actual execution. Usually static analysis methods are similar to dynamic analysis methods; for example, the static data flow analysis is similar to the dynamic taint analysis.

The earliest example of static analysis is from the 1970s when Johnson wrote Lint [26], an application that analyzed sources code before compilation. In theory, static analysis can capture faults that dynamic analysis or fuzzing misses because it can analyze all of the code. Dynamic analysis and fuzzing can only analyze sections they encounter during run time. However, the problem with static analysis methods is that they tend to produce either false negatives or false positives because they do not actually run the application [27]. The problem of the lack of dynamic execution information is mitigated with a control flow graph. This graph displays the flow of the functions in the system and can be used to analyze restraints for different values in the system. Approach like this is similar to the forward symbolic execution explained in the Section 2.2.2.

Static analysis testing is not usually used to verify the functionality of the system. Static analysis is more suitable for ensuring that coding practices are followed, calculating the complexities of algorithms and analyzing the structure of the code. In the context of security research, static analysis provides information about null references, buffer overflows, and uninitialized variables which can be considered security risks as described in Section 2.1.1.

Also static analysis can be useful for making sure that the guidelines for secure programming are followed during the development. For example, it can be used for enforcing print formatting, one cause for the security vulnerabilities [28]. Printf- function in C language can be unformatted, which leads to possible security vulnerabilities. Figure 4 displays an example of both formatted and unformatted prints. In the unsafe case it is possible that the printf-function prints values from the memory instead of the value of the variable.

printf(“%s”, buffer\_pointer)

//Safer

printf(buffer\_pointer)

//Unsafe

Figure 4. Example of formatted (above) and unformatted (below) prints.

## Genetic Algorithms

Genetic algorithms are evolutionary algorithms that can be used in different maximization and optimization problems [29]. They have been a subject of research since 1960s-1970s and have applications in multiple fields, including economics [30], scheduling [31] and bug search simulations [32]. Genetic algorithm picks individual from the pool of individuals, and with certain probabilities executes mutation or crossover with another individual to the selected individual. After creating the test individual, the algorithm runs this subject on fitness function to test how good the individual is. The genetic algorithm then proceeds to pick next individual. At certain intervals, weaker individuals are removed from the individual pool.

By using this method of utilizing the best and removing the weakest individuals it is hoped that the correct answer or good approximation can be found. Caution should be exercised when removing old individuals because it may lead to situation where only local maximum will be found [29]. The popularity of genetic algorithms is partly due to the fact that they can solve arbitrarily complex functions without requiring accurate mathematical description or constraints of the problem [33].

### *Multi Objective Genetic Algorithms*

Multi objective genetic algorithms tackle problems that are more complex and require more than one value to be optimized. One example of such a multi objective genetic algorithm is a situation where the lines of code (LOC) covered should be maximized, but the time taken to run the program should be minimized in one test case. In situation like this there are two options: turn the problem into a single objective maximization with a linear combination of the variables or to search for the Pareto-optimal solutions and pick one from those solutions [34]. In the first case of the linear combination the fitness of a test case could be analyzed with for example with Equation (1)

*f = LOC + 1/t* (1)

where LOC is the lines of code covered during a test case, and t is the time used to finish the test case. After formulating this function, the system would proceed to maximize the value of *f* as in a normal genetic algorithm.

However, creating such functions can be difficult and it is hard to find a linear algorithm that would optimize the problem as desired. This is why the search of Pareto-optimal solutions can be useful. Pareto-optimal solution is a solution where one objective cannot be improved without degrading the other one [35]. When searching for these solutions the algorithm usually creates a Pareto front, since it is often hard to find the only true solution with a multi objective genetic algorithm.

### *Genetic algorithms in fuzz testing*

The idea of genetic algorithms in fuzzing is not new. KameleonFuzz [36] utilized a genetic algorithm for discovering cross site scripting vulnerabilities with fuzzing. Individual in the genetic algorithm was an input sequence that targeted some certain vulnerable state in the SUT and the fitness function considered how close to that desired state the input got. These vulnerable states and the input’s path through the application were found with the taint flow inference.

Liu et al. [7] implemented a fuzzer that also used a genetic algorithm. Like KameleonFuzz, GAFuzz used the internal status of application as the fitness function. In the beginning GAFuzz disassembles code, searches for vulnerable points and calculates a path to that vulnerability. After the static analysis, it feeds inputs to the program. The fitness function again analyzes how close to the desired state input got by checking how many points in the path of the executed input are the same as in the path that was being looked for.

In their research Sparks et al. [6] implemented a genetic algorithm that focused on exploring less common blocks in code. In the beginning, the nodes of the application and probabilities of the transitions between blocks were analyzed. After this, input was inserted into the application and the path taken during the execution was analyzed by fitness function. This function gave higher scores to the inputs that reached more rarely executed branches in the system. As expected, this grey-box fuzzer gained much better coverage and got into deeper nodes faster than a pure black-box fuzzer.

# IMPLEMENTATION

To test the research question about the effect of the mutator balancing on the code coverage obtained with fuzzing, a test system was created. This system, named AMGA (Automatic Mutator-balancing Genetic Algorithm), performed the required actions of a testing suite: test case generation, test case injection and test case analysis. Test cases were made with mutation-based fuzzing by fuzzing valid sample files to create semi-valid files. These sample files were injected to the tested program by starting the program and using the fuzzed file as input. After application had finished it outputted the code coverage and possible crashes, which were then analyzed by the system. After analysis genetic operations were applied with certain probabilities. Figure 5 displays this system in graph form.

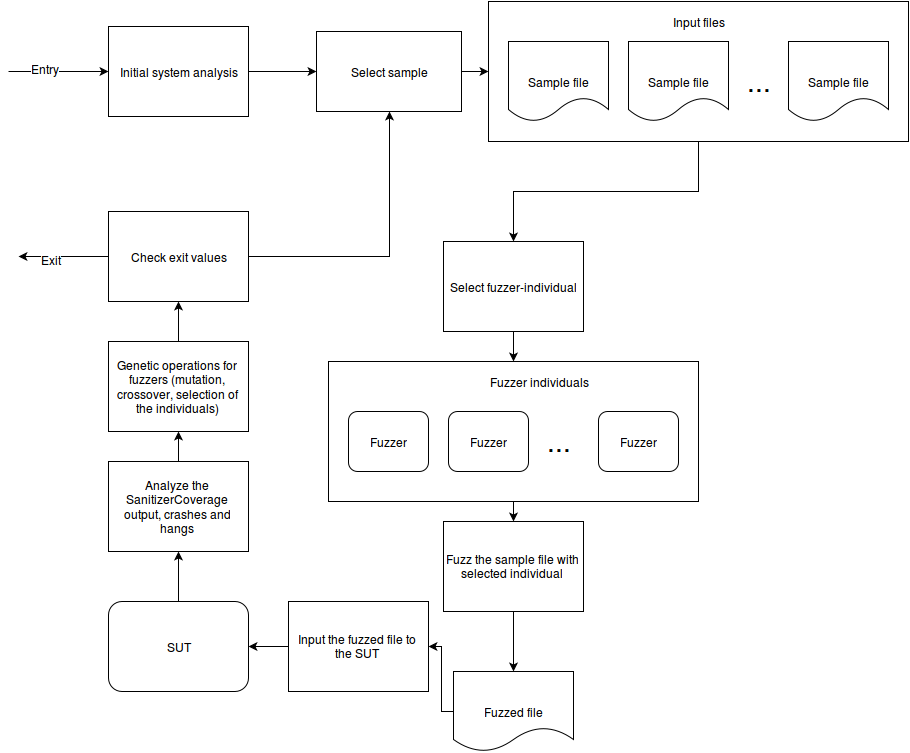


Figure 5. The designed fuzzing framework.

Fuzzing was chosen for the test case generation method of the system because black-box fuzzing is easy to automatize and does not require a complicated initial setup [37]. Fuzzing is also considered as a good method of discovering security vulnerabilities [3]. However, the weakness of the black-box fuzzing is that it gets worse results than white-box fuzzing. To mitigate this problem AMGA uses code coverage information to create a grey-box fuzzing tool.

For the mutator balancing the genetic algorithm was used for the same reason as fuzzing was used for test case generation: it requires little setting up [33]. Also

genetic algorithm is well suited for the optimization problems like this where the exact solution is not required, or in some cases it does not even necessarily exist, and the problem is difficult to define mathematically.

For the feedback, that was used to evaluate the quality of mutators, the dynamic analysis methods presented in the section 2.2 were considered: code coverage, dynamic taint analysis and forward symbolic execution. Code coverage was chosen because it has better accuracy and smaller slowdown impact on the SUT than the other dynamic analysis such as dynamic taint analysis and forward symbolic execution. Also it provides good information on what occurred in the SUT during the execution. Over- and under-tainting could be problematic when deciding the fitness for the mutators, which would lead to problems with a genetic algorithm. From the different granularities of the code coverage, the edge coverage (synonymous for branch coverage) was used. It is the most accurate granularity and its slowdown was considered low enough. Additional caller-callee coverage which records caller and callee for the function calls was considered. The idea was however discarded, because the feature had limitations in SanitizerCoverage that was the code coverage instrumentation used in AMGA [38].

The system was implemented with NodeJS, that is a Javascript-environment. This platform choice was made to make the system available on a wide variety of platforms. Also by using Javascript, the application does not need to be compiled. One preference file needs to be set up to define the fuzzer, SUT and instrumentation settings, which are then loaded into the fuzzing system. The system also features a graphical user interface suitable for terminal emulators for displaying the status of the fuzzing.

A screenshot of the user interface is displayed in the Figure 6. The user interface consists of three main sections. The biggest section is the leftmost area, which prints overall information on what is going on in the system. This area can be used like a normal debugging console. The information displayed here includes exit codes, starts and ends of iterations, errors that occur in the AMGA and the crashes and hangs that occur in the tested system. On top-right is the overview of the fuzzing that shows the general statistics of the fuzzing so far. This includes the amount of executions, hangs and crashes, the start time and the current time and the amount of blocks covered and the amount of blocks in total. The area in the bottom right displays the individuals currently being used and their current scores.

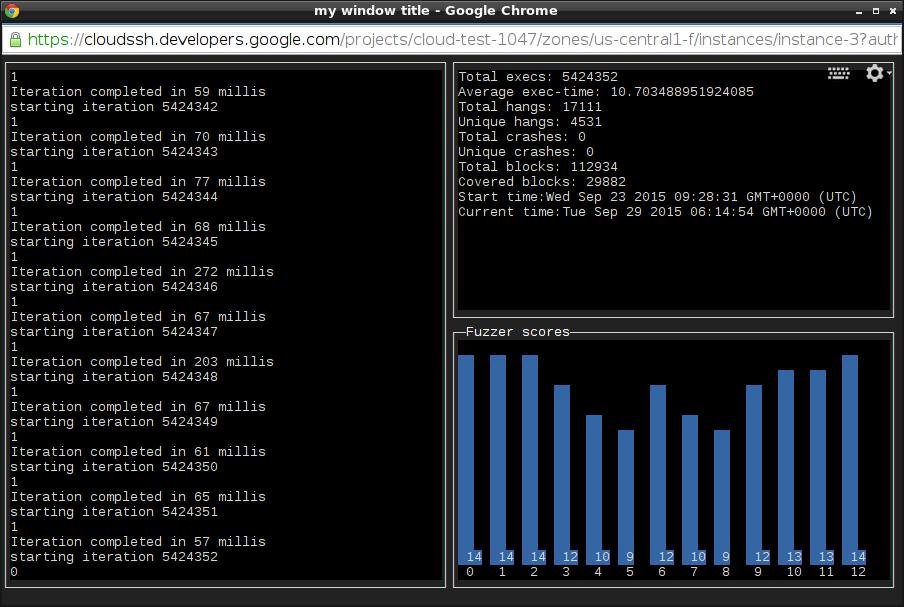


Figure 6. Screenshot of AMGA running in Google Cloud Computing environment.

## Description of the system

In the beginning AMGA performs an initial analysis. In this phase, the target binary and the shared libraries it uses are analyzed for the instrumentation. This way the system can also analyze the shared libraries at the same time with the actual SUT. External libraries are added as fuzzing targets if they contain instrumentation, because it increases the scope of fuzzing. Also they provide more blocks to be analyzed. Table 2 shows a few example applications and the amount of blocks with and without instrumented libraries. For example, rsvg-convert utility in librsvg has only 239 blocks, but with all the possible external libraries it has 112934 blocks. On the other hand avconv from libav has only 6632 blocks in the external libraries.

Table 2. Overview of librsvg utility properties

|  |  |  |
| --- | --- | --- |
| **Name** | Rsvg-convert | avconv |
| **Total blocks** | 239 | 252 169 |
| **Total blocks with libraries** | 112 934 | 258 801 |

During this initial phase, the system also loads preferences to set up the fuzzing environment and initializes the sample and individual arrays. Item in the sample array contains the path to sample and a number indicating how often it has been used as a sample in fuzzing. An item in the individual array contains the weights of the fuzzer’s mutators and the fitness of these weights.

Once the initial phase has been completed the actual fuzzing begins. First AMGA picks the individual from the individual array to be used. The fuzzer used in the AMGA is Surku and it is explained more in depth in the section 3.3. The selection of the individual is done with the roulette wheel method, where the proportion in the wheel for each individual is their current score. When the individual is chosen the weights of mutators are set to the fuzzer.

After this, the sample file is picked from the possible samples randomly. The fuzzer then proceeds with fuzzing the chosen sample file. The next step is running the SUT with the fuzzed sample. AMGA then waits for the application to finish. If the application does not finish on time, AMGA kills the application with SIGSEGV- signal to obtain the coverage. If the program does not halt after SIGSEGV in time, a SIGTERM-signal is sent until the application stops. However in this case no coverage is obtained.

When the SUT has stopped the coverage data is analyzed to see if new blocks were covered, either in the SUT or in any of the shared libraries. Coverage data is produced by SanitizerCoverage and its usage is described more in depth in the section 3.4.1. If new coverage was found, the score of the individual will increase. Similarly, found hangs and crashes are analyzed. If a hang or a crash occurred, the individual’s score would be increased. If new coverage or a new crash occurred, the sample was added to the input file array for further fuzzing. On the other hand if the sample did not provide any new coverage or crashes the sample was removed.

After the analysis genetic operations are applied to the individual. Two types of genetic operations are in the system: mutation and crossover. Mutation means a creation of a new fuzzer by mutating the existing one by randomly increasing or decreasing the weights of mutators. Cross-over on the other hand is an operation where the probabilities from two different individuals are mixed randomly with each other. After the genetic operations are applied AMGA checks time and test case

amount limits to see if either of them have reached their limit. If not, the system starts the next iteration of fuzzing. Otherwise fuzzing ends.

## Input files

A folder to the initial input files is provided in a preference file that needs to be set up by the user. AMGA did not impose limitations on the input sample files regarding their size, format or other qualities. Interesting fuzzed samples were also copied to the input files for further analysis and fuzzing. A file that found new coverage was considered as interesting. Fuzzed samples that lead to hangs or crashes were not added to the input file list because their addition makes the fuzzing slower. If new samples that cause hangs are added to the list of input files, the amount of test cases to use the maximum execution time increases and the fuzzing becomes slower. Also these new hangs and crashes often just repeat the same crash with a slightly different path, meaning that they do not increase the unique hangs and crashes found.

Unique hang or crash meant that the edges covered until the end of execution had to be unique. For unique crashes, also the last line of execution had to be unique, meaning that it was not encountered in any previous crash. This was done to prevent adding fuzzed samples that lead to a previously seen crash from a different path. The order of the edges covered did not matter when comparing the uniqueness. Crashes that were not considered unique had their crash log and fuzzed sample stored in a separate place for further analysis.

No special queue or ordering was made for the input files but instead they were chosen purely at random. The main purpose of this was to make the effect of the mutator balancing more visible and that way get a better answer to the research question. The samples could have been for example given a score according to how interesting they were in terms of crashes and new coverage and then ordered into a queue. However, this would have led to a situation where the order of the samples would also have contributed to the effectiveness of the fuzzing. Another purpose of not having a special input file order is to ensure that the samples added later on to the sample list would get an equal chance to be selected and that newly added samples would not prevent older samples from being run. These issues may arise if many new samples are added to the initial set or fuzzing runs are shorter when all the samples cannot be examined.

## Fuzzer

Few different fuzzers were thought to be used for the implementation. The fuzzer needed to have optimizable variables, that affected the fuzzing, to be considered. Radamsa is a general purpose black-box fuzzing tool created by Helin [39]. This tool can be used to create semi-valid test cases by mutating existing valid samples. Radamsa features multiple mutators with editable probabilities. Surku is similar type of black-box fuzzer created by Kettunen [40]. It is programmed with the JavaScript using Node.JS platform, which extends JavaScript with features such as file editing that are needed in a fuzzer. Like Radamsa, Surku provides multiple mutators for both binary and string file formats with editable mutator probabilities. This gives better control over how the fuzzer mutates the sample it has been given. Zzuf is a black-box fuzzer made by Caca Labs [41]. It differs from the previous two with the fact that it

does not use mutators for fuzzing. Instead Zzuf provides a fuzzing ratio which simply tells how many bits should be flipped from the input.

From these fuzzers, Surku was chosen as the default fuzzer for the system. It features multiple mutators that can be applied during fuzzing with varying probabilities. This feature is required to be able to answer the research question. Surku is programmed in Javascript, allowing seamless integration with AMGA. This close integration meant that the usage of AMGA is faster because the fuzzer does not have to be edited using command line commands. Instead the fuzzer can be edited directly in the AMGA with Javascript.

When fuzzing a new test case, Surku applies multiple mutations with different mutators into the sample file being fuzzed. In this system a minimum of one and a maximum of ten mutations were used. Range between mutations is high, because one mutation can have slight to none impact to the fuzzed output, whereas ten mutations most likely causes serious deformations to a file. For each of these mutations, Surku picks one of the mutators randomly by the probabilities they have been assigned with, meaning that a mutator with higher probability is more likely to be picked. All of the available mutators in Surku are presented in the Table 3. String-based mutators in the table are mutators that do not operate by editing bits directly; instead they read the contents of the input and operate by editing the strings and characters the bits in the file represent.

Table 3. Different mutators in the version of Surku used

|  |  |  |
| --- | --- | --- |
| **Name** | **String-based** | **Action** |
| FreqString | No | Searches and repeats common patterns in data |
| ChunkSwapper | No | Switch places of two data chunks |
| ChunkCollateTrigram | No | Uses trigrams to mix up the structure of the data |
| RegExpTrick | No | Uses regExp to replace values in the data |
| StrStuttr | No | Repeat a chunk of data |
| DelimiterMutator | No | Performs move, swap and repeat operations to chunks split with delimiters |
| ReplaceXMLValuePair | Yes | Replaces XML-values in a file |
| StrCopyShort | No | Copy a chunk of data to different place |
| StrCopyLong | No | Copy larger chunk of data to different place |
| StrRemove | No | Remove a chunk of data |
| InsertSingleChar | No | Insert one random character to data |
| InsertMultipleChar | No | Insert multiple characters to one location in data |
| ReplaceSingleChar | No | Replace one random character to data |
| ReplaceMultipleChar | No | Replace multiple characters to one location in data |
| RepeatChar | No | Repeat one character in the data |
| RepeatChars | No | Repeat sequence of characters in data |
| BitFlip | No | Flip bits in data |
| PdfObjectMangle | Yes | Inserts values to a PDF-file |
| XMLMutator | Yes | Edits structure of XML-file various ways |

## Instrumentation of tested systems

To get feedback about the application status, AMGA utilizes information about the code coverage. For gathering code coverage SanitizerCoverage [38] was used. To discover security vulnerabilities three different instrumentations for several different fault types were used. Even though the usage of these vulnerability instrumentations is not necessary to answer the research question on the mutator balancing, it is still important to see if AMGA can find vulnerabilities to test the real-world applicability of the system.

### *SanitizerCoverage*

SanitizerCoverage [38] is a tool developed by Google and is part LLVM since version 3.7. It is a tool for measuring function-level, block-level or edge-level coverage of code. Function-level coverage measures which functions were covered during run time and it is the broadest measurement of code coverage. Block-level coverage measures the basic blocks inside the system. A basic block means a block of code that has no control statements in it to direct program flow away from the block. Block-level coverage can also be considered as statement-level coverage. Edge-level coverage is the most accurate one, measuring the transitions between these basic blocks. SanitizerCoverage was chosen as the code coverage provider over the other coverage tools because it provided the required accuracy with edge-level code coverage measurement. Also it was easily available as an open source application in LLVM. The slowdown introduced by SanitizerCoverage with edge- level coverage is tolerable, approximately up to 40% [38]. The other instrumentation tools described in the section 3.4.2 that were compatible with SanitizerCoverage were considered as an advantage.

In the tests, the edge-level instrumentation was used. Edge-level coverage gives more accurate information than the block-level coverage. For example, consider Figure 7. It contains one branch, the if-sentence, and three blocks: if-sentence (block A), the contents of the if-sentence (block B), and the exit (block C). If the function in the example receives a value of 1 in the parameter, it will go through edges A-B and B-C, achieving the full block-level coverage. To achieve the full edge-coverage the tests still need to test the edge A-C. However, this edge is critical, meaning that it is not the only edge leaving the block (A-B also leaves block A) nor the only edge arriving at its destination (B-C also arrives at block C). A new block D needs to be inserted during instrumentation so that A-C is split into two edges, A-D and D-C, to be sure that also A-C edge has been tested.

int example (int i){

if ( i > 0 ){

i = 0;

}

return i;

}

//A

//B

//C

Figure 7. Example for edge-level coverage.

### *AddressSanitizer, MemorySanitizer and LeakSanitizer*

AddressSanitizer [42] is a tool commonly used in security testing. Like SanitizerCoverage, it is developed by Google and is part of LLVM since version 3.1. It is also part of GCC since version 4.8 [43]. AddressSanitizer instruments the code and looks mainly for the use-after-free errors and buffer overflows in the heap, stack and global buffers. It is also capable of finding use-after-returns, memory leaks, double frees and invalid frees. The invalid free search option is still under development. These memory related issues are possible causes of vulnerabilities (as discussed in section 2.1.1). AddressSanitizer introduces a slowdown of 2x [42]. For uncovering memory leaks LeakSanitizer can be used. It is integrated into AddressSanitizer although it can also be used without AddressSanitizer to avoid the slowdown effect. MemorySanitizer was developed for detecting uninitialized memory reads. MemorySanitizer is available in LLVM 3.3. Memory reads can also be security vulnerabilities as explained more in depth in section 2.1.1.

During testing , AddressSanitizer was used for extra instrumentation, however the usage of MemorySanitizer is also supported in the system. Additional instrumentation serves two purposes: firstly, it tests the real world applicability of the AMGA to see if it can find real failures and vulnerabilities like other fuzzers. Secondly it provides additional information about the crashes for further analysis. This information can be used to see if the crashes would actually be security vulnerabilities or not. This instrumentation however creates a slowdown which needs to be taken into account during analysis. An example of AddressSanitizer output in case of error is shown in Figure 8. This example presents a case where a variable in heap memory area is read after its memory has already been freed, as can be seen on three first rows. Beginning from the line four is the stack trace of the program until the error occurred.

==9442== ERROR: AddressSanitizer heap-use-after-free on address 0x7f7ddab8c084 at pc 0x403c8c bp 0x7fff87fb82d0 sp 0x7fff87fb82c8

READ of size 4 at 0x7f7ddab8c084 thread T0

#0 0x403c8c in main example\_UseAfterFree.cc:4

#1 0x7f7ddabcac4d in libc\_start\_main ??:0

0x7f7ddab8c084 is located 4 bytes inside of 400-byte region [0x7f7ddab8c080,0x7f7ddab8c210)

freed by thread T0 here:

#0 0x404704 in operator delete[](void\*) ??:0

#1 0x403c53 in main example\_UseAfterFree.cc:4

#2 0x7f7ddabcac4d in libc\_start\_main ??:0 previously allocated by thread T0 here:

#0 0x404544 in operator new[](unsigned long) ??:0

#1 0x403c43 in main example\_UseAfterFree.cc:2

#2 0x7f7ddabcac4d in libc\_start\_main ??:0

==9442== ABORTING

Figure 8. Example of an AddressSanitizer crash report.

## 3.5. Genetic Algorithm

As discussed in the section 2.3.3, genetic algorithms have been used in optimizing the fuzzing performance before. However, these previous fuzzers focused on utilizing a genetic algorithm with the optimization of the input files or paths inside the application instead of the individual mutators in the fuzzer. However, mutators are something that should be considered when optimizing the fuzzers, especially if the fuzzer gives an option to edit the probabilities of the different mutators. Optimizing these probabilities is important because not all mutators are suitable for all file types. The genetic algorithm in this implementation focuses on maximizing the code coverage, crashes and hangs by optimizing the probabilities of the mutators. Figure 9 visualizes the control flow through the genetic algorithm. The implemented genetic algorithm is not a classical genetic algorithm described in section 2.4, but instead a version of it applied for this problem. As usual, the genetic algorithm has candidate solutions for the optimization problem, called individuals. Individual in this genetic algorithm contains the probabilities for the different mutators. System first picks one sample file to be fuzzed. One individual is picked after the sample and the system fuzzes the sample using the probabilities from the picked individual. After this, the fuzzed file is input to the tested system. If new code is covered or a unique hang or crash is discovered, the fitness for the individual that fuzzed the file will be increased. Otherwise the fitness for that individual will stay

the same.

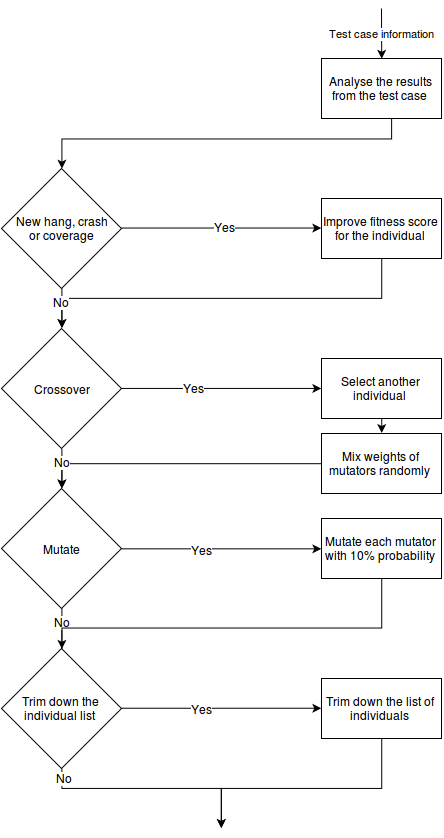


Figure 9: Overall view of the genetic algorithm.

As in the traditional genetic algorithms, the system performs crossovers and mutations to the individuals. Crossover is an operation where two different individuals are combined. In AMGA, during the genetic crossover operation the probabilities for two different individuals are mixed up. This is done by going through each mutation function of the individuals and with 50% chance swapping the probabilities for that function in the new individuals. This produces two new individuals to the system where some of the probabilities are swapped with each other. Old individuals are also kept intact in the individual pool.

In genetic algorithms, the mutation operation is the operation that alters an existing individual so that algorithm gets different values from time to time. The mutation operation of the genetic algorithm in AMGA selects one individual from the system, goes through each mutator function of the individual and modifies its probability with 10% probability. This 10% probability was chosen because it approximately performs two mutations during one genetic mutation.

These reproduction actions were performed after every test run of the system. It is possible to apply both operations on one run. In mutation, the individual mutated is the individual used in fuzzing. In crossover, one individual is the one used in fuzzing and the other individual is picked from the individual list. Because the individuals with the best fitness are more likely to be picked for fuzzing, the best fit individuals have a higher chance of creating offspring with the genetic operations. This guides the genetic algorithm towards an optimal solution.

Genetic algorithms perform a “natural selection” where the poor individuals are removed. The individual list was trimmed down with partial elitism after certain rounds of fuzzing. A portion of the individuals with the best score were always chosen for the new generation. The rest of the individuals for the new generation were chosen with the roulette wheel method that avoids killing poorer individuals too soon but still favoring the better individuals [29]. The roulette wheel method is a method where all the individuals get a chance to be selected. However, the individuals with a higher score have a higher chance of being selected. This kind of genetic algorithm creates a constantly evolving individual list. This list favors the good individuals in all the sections of the genetic algorithm, but still gives a chance to the poorer performing individuals. Table 4 presents initial values of different values in the genetic algorithm. The effects on the changes of these variables were researched to see if they have significant effect on the code coverage.

Table 4: Variables in the system

|  |  |
| --- | --- |
| **Variable** | **Initial value** |
| Selection frequency | 100 iterations |
| Genetic operation probability | 2.5% |
| Maximum change in the probability of mutation | 50% |
| Amount of individuals | 100 individuals |
| Amount of individuals chosen with elitism | 40% |

These initial values were used to create a system that evolves slowly towards the optimal point. This is done to ensure that the system behaves in a stable way when searching for optimal values and to prevent sudden, large changes that may make the system act unwanted ways. When the interval between selections is 100 iterations in

the system where there are two genetic operations with 2.5% probability, about eight new individuals are created between selections. This slow increase keeps the individual pool stable. Also fewer new individuals give a better possibility for all the individuals to earn new coverage between the selections.

The amount of the elite individuals is below 50%, because with 40% elitism the system gives a good advantage to the best individuals while the less fit individuals still have a considerable chance to be used in the future. By also using the less fit individuals, the gene pool does not become stagnant. The maximum change speed in the mutators was set to 50%, meaning that the probability of the mutator in an individual can change at most 50% from its current probability. 50% was used because it gives a chance to significant changes in the probabilities. However, it still also gives a reasonable possibility to smaller, more accurate changes. Using 100% change speed, where the mutator could be in the best case doubled or in the worst case zeroed, was considered as too extreme.

# EXPERIMENTS

To test the effect of the mutator balancing with a genetic algorithm on the total code coverage, the system was run for 1 000 000 iterations with two different open source programs. Microsoft’s Security Development Lifecycle recommends the minimum of 500 000 iterations with at least 250 000 iterations after the last crash [44]. The minimum amount of iterations was reached and exceeded, but the rule of 250 000 iterations after last crash was not applied. This rule was ignored because it is more reliable to compare results where the amount of iterations is a constant, not a variable.

The results from these test runs were compared with the regular black-box fuzzing without the genetic algorithm. Besides not doing the mutator balancing the black-box testing was identical. The time it took to complete these test runs was recorded to calculate how significant the slowdown effect introduced by the mutator balancing was. These test runs were run four times to mitigate the effect of the randomness caused by fuzzing.

Similarly, the different variables of the genetic algorithm were tested. This was done to see how they affect the total code coverage during the fuzzing. These tests were done again by running two open source applications with 1 000 000 iterations. The initial benchmark run was made with the default values shown in Table 4. After this run, new runs were made where only one variable at a time was changed to see how its changes affect the resulting code coverage.

Table 5 shows the different test values for the variables that were tested. For each variable, a smaller and larger value was tested. Smaller value was about a half of the initial value, which is significantly smaller than the original value and should show some differences between variables. An exception to this was in the “maximum change speed in mutation” variable where an even smaller value was tested. The change speed was considered a special variable since it affects mutator probabilities directly. More drastic changes were wanted to be seen from this variable and thus a smaller value was used. The larger test value is about the double from the initial value. Doubled value is significantly different from original and by using this kind of value more extreme results are searched.

Table 5: Different test values used to see the variable’s effect on code coverage

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Smaller test value** | **Initial value** | **Larger test value** |
| Selection frequency | 50 iterations | 100 iterations | 200 iterations |
| Genetic operation probability | 1% | 2.5% | 5% |
| Maximum change speed in mutation | 10% | 50% | 100% |
| Amount of individuals | 50 individuals | 100 individuals | 200 individuals |
| Amount of individuals chosen with elitism | 20% | 40% | 60% |

These tests were done to two open source libraries that featured a few testable applications, librsvg [45] and libav [46]. Tested file types were SVG that was tested with librsvg and MP4 and H.263 that were tested with libav. SVG is an XML-based

vector graphic format. SVG-files consist of strings that describe 2D shapes that create the desired graphic. MP4 and H.263 are both binary video compression formats.

As the testing platform in the tests a Google Cloud computing virtual machine instances were used. These instances were running Ubuntu 14.04 64-bit version. Hardware specifications for the virtual machine instances can be seen in Table 6. These machines provided a command line access to the instances via SSH. After connecting to the instance the necessary applications could be set up and fuzzing session started. Each machine had only one fuzzing session running at a time to avoid problems with sharing the resources in the machine.

Table 6. Virtual machine properties

|  |  |
| --- | --- |
| **Base memory** | 7.5 GB |
| **Processors** | 2 x 2.5GHz Intel Haswell processors |

## Black-box fuzzing

For locating the differences in average fuzzing times and obtained code coverage, a black-box version of AMGA was created. This black-box version was implemented for comparing the results reliably between the grey-box approach and the black-box approach. The black-box version performed most of the same actions as AMGA, but instead of having multiple individuals it only used one fuzzer that had equal probabilities for all the mutators. Also the genetic algorithm and the operations related to it were completely removed.

The approach of using AMGA for black-box fuzzing introduces a small slowdown compared with, for example, using only command line scripting for fuzzing. However, using the modified version of the same tool makes the results more comparable. Also, by using the same tool the slowdown of the actual mutator balancing process is easier to calculate, because the tools work practically the same way. The results from command line script black-box fuzzers and JavaScript grey- box fuzzers would not be as easily comparable, because JavaScript itself adds a small overhead.

During the black-box fuzzing the same SanitizerCoverage and AddressSanitizer instrumentations as with the regular AMGA were used. This was done to better evaluate the effect of the slowdown generated by the genetic algorithm in AMGA by having the same slowdown generated by instrumentation. With this instrumentation, the code coverage and uniqueness of the crashes and hangs could be analyzed also in the black-box fuzzer. This makes the comparison of the end results between different types of fuzzers easier. However, the information gathered during the black-box fuzzing was not used to guide the fuzzer by adding interesting samples or by scoring mutators depending from their findings.

## Tested applications

Few important features were required from the tested applications. Firstly, the source code had to be available. This was required for building the applications with the instrumentation. Secondly, the applications had to have utilities that took file inputs. These input files were used as the targets for the fuzzing. Also it was important that

the tested utilities worked without a display, which meant that view and display utilities were not considered from the libraries.

These requirements for the applications do not significantly limit available applications. Software testers generally have the source code available for instrumentation during testing and for the performed tests open source software was used. File inputs are commonly present in the software and AMGA in theory can also be used for other types of inputs as well.

What can be generalized from the results are the differences between the two different approaches, grey- and black-box fuzzing. However, the improvements in the amounts of code coverage, crashes and hangs cannot be generalized, because they depend on the tested application. This means that it can be stated which approach for fuzzing is better, but how large the change is depends on the application itself and cannot be generalized to apply for all the applications.

Two chosen open source libraries, librsvg and libav, satisfied the requirements. Both libraries have a conversion tool that converts a file from one format to another. These types of conversion tools fulfill all the requirements for tested software: they are open source, take file inputs and do not require a graphical interface to function. The libraries were chosen because they use different types of file formats in processing: librsvg uses string-based XML-files, whereas libav uses binary-based video files. This kind of testing covers both string and binary file types to see if there is difference in their results.

### *Librsvg*

Librsvg is a SVG rendering library that is a part of the GNOME project [45]. The version 2.40.10 was used for testing. Librsvg has one application that was tested. Rsvg-convert is an application that converts SVG files to PNG raster images and PDF or Photoshop vector images. It is also possible to create Cairo XML-dumps. During testing the conversion from SVG-format to PNG-format was tested.

As the initial collection for input samples a collection of SVG-files from Mozilla Security [47] was used. The sample set was used because it provides large initial set, giving a good starting point to see how the total code coverage increases. The test set consisted of 2353 files, so it was minimized with the afl-cmin tool that is a part of the AFL fuzzing tool [23]. Version 1.84b of the AFL was used for minimization. This tool minimized the initial test set to remove test cases which did not provide any new coverage in the initial phase to the smallest possible set. This initial set had 541 files and it covered 18793 blocks of the rsvg-convert utility. During fuzzing the test set was not minimized further.

### *Libav*

Libav provides utilities for playing, converting, editing and streaming of video files. The release version 11.4 was used for testing. Avconv is the converting utility of the libav and it is capable of converting various media formats. It was fuzzed for two different conversions: H.263 file to AVI file and MP4 file to AVI file.

As the initial input sample set from the fate-suite [48] was used. These files in the suite were used as the initial sample set because they are used for testing purposes in the actual libav development. The automatic reporting system of Fate was not used,

only the provided sample files. Table 7 shows the different input and output combinations used and the total amount of initial samples for each conversion.

Table 7. Overview of different conversions tested with avconv

|  |  |  |  |
| --- | --- | --- | --- |
| **Input format** | **Output format** | **Amount of samples** | **Initial coverage** |
| H.263 | AVI | 6 | 10120 blocks |
| MP4 | AVI | 18 | 11175 blocks |

## Experiment and AMGA preparations

Before starting the actual fuzzing, a few things need to be done.

1. Obtain the latest source code for the library that is going to be fuzzed.
2. Instrument, configure and build the dependent libraries.
3. Instrument, configure and build the library for fuzzing.
4. Set up the preferences file of AMGA.

The first step is usually simple, because the code for the open source applications is commonly available. Second step requires getting the list of the dependencies for the application that is going to be tested. After this, the source codes for these additional libraries need to be obtained and they need to be built with the SanitizerCoverage and AddressSanitizer instrumentations. This second step is optional but recommended, because it can help by analyzing multiple targets at once. The third step is building the actual library for testing. During the configuration of the library, the SanitizerCoverage and AddressSanitizer instrumentations are defined. After the configuration is done the library is built.

Final step is setting up the preference file for AMGA. This preference file contains important paths and variables required for the fuzzing.

* Path to the Surku fuzzer
* Path to the input sample files
* Path to the shared libraries in the system (if also analyzing the dependencies)
* Path where samples causing the crash will be stored
* Path where the fuzzed files will temporarily be stored
* Command to start the tested application
* Flags used during the tested application start command
* Removable output of the tested application (e.g. converted file from a conversion tool)
* Path to the folder where the coverage information is stored
* The file format of the fuzzed file

This information gives all the required information for fuzzing. After this, the user defines a time limit or a test case amount limit and begins the actual fuzzing process. When starting the application from a command line, three options can be set: a test case amount limit, a time limit in seconds and the black-box mode.

# RESULTS

The data gathered with the methods described in the section 4 is used to answer the research question “What is the effect of the mutator balancing on the amount of code covered with fuzzing”. Also the data is used to determine how different variables of the genetic algorithm affect the obtained code coverage. Table 8 shows the comparison between the amounts of unique hangs, crashes and coverage of two different approaches. The presented numbers are the sums of the averages for three different fuzzed applications: rsvg-convert and two different avconv conversions. Each of these applications was tested four times and the average was calculated for each application from these four fuzzing runs. Although hangs and crashes are presented in the results, the main focus in the analysis is in the code coverage. Compared with hangs and crashes, code coverage is the most stable metric.

Table 8: Overview of the results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Fuzzer** | **Hangs** | **Crashes** | **Coverage** | **Time** |
| AMGA | 5944 | 354 | 59170 blocks | 23078 minutes |
| Black-box | 1367 | 86 | 47466 blocks | 20899 minutes |

As it can be seen, the mutator balancing in AMGA increases the obtained coverage on average by 19.7%. The coverage values for the two approaches were tested with Mann-Whitney U test. This test is used for comparing means and proving if they are statistically significantly different. The Mann-Whitney U test is similar to the t-test, however the U test can be used for results with unknown distributions whereas the t-test always requires normal distribution from the results. Because the distribution of the results was unknown the U test was used. The U test gives p=0.018 when testing if the code coverage is significantly higher in the results of AMGA than in the black-box results. This p-value is smaller than 0.05, meaning that the difference is statistically significant.

This performance gain however introduces a slowdown. Table 9 shows slowdown rates in the different fuzzed applications. On average the slowdown of the mutator balancing is 8.2%. On the other hand Table 10 shows the average times between the unique crashes, hangs and code blocks covered. From this table it can be seen that although AMGA has some slowdown compared with the black-box fuzzing, the time between each of the measured values is shorter, suggesting that the mutator balancing is a better approach than the black-box approach also in the terms of used time.

Table 9: Summary of AMGA slowdowns in different fuzzing runs

|  |  |
| --- | --- |
| **Tested application** | **Slowdown to black-box** |
| librsvg-convert | 0.96 |
| avconv; H.263 -> AVI | 0.92 |
| avconv; MP4 -> AVI | 0.87 |

Table 10: Average time between each occurrence of a hang, crash or new coverage

|  |  |  |  |
| --- | --- | --- | --- |
| **Fuzzer** | **Time per hang** | **Time per crash** | **Time per unique covered block** |
| AMGA | 3.7 minutes | 65.2 minutes | 22 seconds |
| Black-box | 15.3 minutes | 243.0 minutes | 26 seconds |

## Effect of the mutator balancing on code coverage

The mutator balancing grey-box approach of the AMGA was compared with the black-box approach of the fuzzing. This comparison reveals information on how the mutator balancing affects the resulting code coverage in different applications. The default values shown in the Table 4 were used in the system during the fuzzing. An overview of the mutator balancing and black-box fuzzing used with librsvg can be seen in the Table 11, where the average increase in coverage was 623 blocks. This table presents average results from the four performed fuzzing rounds. Tables 12 and 13 present results from the libav fuzzing in the same way. In H.263-conversion, the average increase in coverage was 6041 and in MP4-conversion 4977 blocks. The significantly larger increase in the coverage obtained in libav fuzzing compared with the coverage obtained in librsvg fuzzing is most likely caused by smaller initial sample set. Librsvg fuzzing had larger sample set and the set initially covered a large amount of the application. On the other hand, libav had smaller sample sets, which meant that it was the task of a fuzzer to get more of the code covered.

Table 11: Result of grey- and black-box fuzzing of convert utility

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Approach** | **Hangs** | **Crashes** | **Coverage obtained** | **Time** |
| AMGA | 2710 | 335 | 21589 | 2805 minutes |
| Black-box | 930 | 80 | 20966 | 2688 minutes |

Table 12: Results of grey- and black-box fuzzing of avconv with MP4-conversion

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Approach** | **Hangs** | **Crashes** | **Coverage obtained** | **Time** |
| AMGA | 329 | 9 | 18789 | 9238 minutes |
| Black-box | 66 | 5 | 13812 | 8059 minutes |

Table 13: Results of grey- and black-box fuzzing of avconv with H.263-conversion

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Approach** | **Hangs** | **Crashes** | **Coverage obtained** | **Time** |
| AMGA | 2905 | 10 | 18729 | 11035 minutes |
| Black-box | 341 | 1 | 12688 | 10152 minutes |

It can be seen that on average the AMGA performs better than the black-box fuzzing. This improvement applies to all the result values of the fuzzing: hangs, crashes and coverage. These total values were again tested with Mann-Whitney U test to see if the differences were statistically significant.

The results from the librsvg fuzzing give p=0.248 when testing with the U test if an approach had significant impact on the obtained code coverage. This large p-value suggests that the difference in the coverage between the two approaches happened by chance. However, the U test for the libav MP4 conversion gives p=0.048 and for

H.263 conversion p=0.021, which would suggest that the AMGA would be statistically better. This irregularity between resulting significances in different applications is most likely caused by a small sample size (N=4), the larger initial coverage of the librsvg sample set and the randomness of fuzz testing. However, it can be assumed that the mutator balancing would be better also in the librsvg fuzzing if the fuzzing rounds were performed few more times.

Figure 10 presents the changes in the total coverage in one fuzzing round for AMGA and black-box fuzzing. As it can be seen, in the initial phase both approaches gain new coverage rapidly, but as the fuzzing goes on, the growth slows down. However, AMGA keeps finding new coverage at constantly faster rate. Also during the fuzzing AMGA uncovers more “gates”, which are blocks that lead to significant amounts of new coverage and can be seen as sharp increases in the coverage found. A clearer example of a gate is in Figure 11, which displays a situation where one test case discovered nearly 4000 new blocks.

16000

15000

14000

13000

12000

11000

10000

9000

8000

Black-box

AMGA

**Iteration of the fuzzing**

**Total amount of covered blocks**

Figure 10: Total coverage over iterations in MP4-conversion of libav.

24000

22000

20000

18000

16000

14000

12000

10000

8000

AMGA

**Iteration of the fuzzing**

**Total amount of covered blocks**

0

50000

100000

150000

200000

250000

300000

350000

400000

450000

500000

550000

600000

650000

700000

750000

800000

850000

900000

950000

1000000

Figure 11: Example of a “gate” found with fuzzing.

0

50000

100000

150000

200000

250000

300000

350000

400000

450000

500000

550000

600000

650000

700000

750000

800000

850000

900000

950000

1000000

**Covered blocks**

## Effect of the variables in the genetic algorithm on code coverage

The effect of the changes made to the variables of the genetic algorithm presented in the Table 4 was tested. The resulting changes from varying the different variables are presented in Table 14. This table presents information about the runs made with the avconv tool of libav when converting MP4 files to avi-files. The differences between the code coverages obtained with different values are also illustrated in the Figure 12.

Table 14: Overview of the effects caused by changing fuzzing variables

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Variable** | **Value** | **Hangs** | **Crashes** | **Coverage** |
| **Individual selection frequency** | 50 iterations | 32 | 6 | 16295 |
| 100 iterations | 329 | 9 | 18729 |
| 200 iterations | 71 | 4 | 14675 |
| **Genetic operation probability** | 1% | 59 | 5 | 17867 |
| 2.5% | 329 | 9 | 18729 |
| 5% | 96 | 4 | 20832 |
| **Elite individual amount** | 20% | 141 | 8 | 14701 |
| 40% | 329 | 9 | 18729 |
| 60% | 9 | 6 | 16053 |
| **Maximum mutator probability change** | 100% | 30 | 6 | 15285 |
| 50% | 329 | 9 | 18729 |
| 10% | 172 | 6 | 15134 |
| **Individual amount** | 50 individuals | 47 | 5 | 14699 |
| 100 individuals | 329 | 9 | 18729 |
| 200 individuals | 441 | 4 | 15042 |

Figure 12: Effects of variable changes to code coverage.

25000

20000

15000

10000

5000

Lower value

Default value Higher value

0

Selection Genetic Amount of Maximum Amount of

frequency operation individuals change in individuals probability chosen with the

elitism probability

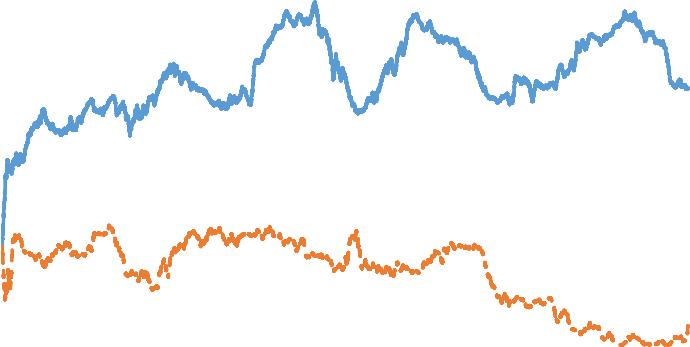
of mutation

From this information it can be seen that the default values resulted in the largest coverage in the majority of cases. Higher genetic operation probability on the individuals however gives larger coverage on average. However, performing U test comparing the mean coverages between the default and high genetic operation probabilities values gives p=0.648, which suggests that this difference occurred by chance.

However, the default values and tested comparison values are not necessarily the best values for the variables. It is possible that in between the default values and test values there are untested values that would perform better. Also it is possible that these optimal values are different for different programs.

## Effect of the mutator balancing on the mutator probabilities

The mutator balancing had an effect on the probabilities of the different mutators. Figure 13 illustrates such example. In the diagram are the total probabilities of two mutators in all the individuals of the genetic algorithm: xmlMutate which is a string based mutator and strStuttr which is not. The results in the diagram come from the librsvg fuzzing, meaning that the fuzzed file format was string based. This means that the string based mutators should have an advantage, and from Figure 13 it can be seen that the xmlMutate mutator gets more likely over time, whereas the probability of strStuttr declines over time.



18%

16%

14%

12%

10%

8%

6%

4%

2%

0%

xmlMutate

strStuttr

**Iteration of the fuzzing**

**Overall probability of the mutator**

Figure 13: Changes of the probabilities of two mutators during librsvg fuzzing.

0

50000

100000

150000

200000

250000

300000

350000

400000

450000

500000

550000

600000

650000

700000

750000

800000

850000

900000

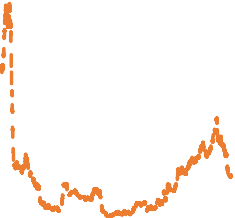
950000

1000000

Figure 13 also shows how the probability seems to go up and down in a wave motion for the xmlMutate. This occurs because AMGA does not keep track on how the probabilities of the mutators have changed over time. Optimal probability for the xmlMutate mutator seems to be around 14%-15%, and after reaching this value the probability drops momentarily. Since the AMGA does not lock the probability in the

optimal value, the value keeps fluctuating. On the other hand, this keeps fuzzing constantly ready and evolving. For example, in a case where new coverage is found rapidly with a different mutator it is important to be able to adapt quickly to the new situation.

Effect of the changes in the mutator probability change speed is illustrated in the Figure 14. This figure displays the changes in the probability of the freqString mutator over time in librsvg fuzzing with two different mutator change speeds. As it can be seen, the higher speed of the probability change leads to significantly bigger changes in the probability in a shorter time. It is also worth noting how both change speeds in the beginning increase the probability of the mutator, after which the probability drops. In the end, the probability is around 3%-4% with both change speed values. This similarity would suggest that the change speed in the probability of the mutator does not affect the time it takes to find the optimal probability.



25%

20%

15%

10%

Change speed 10%

Change speed 50%

5%

0%

**Iteration of the fuzzing**

**Overall probability of the mutator**

Figure 14: Changes of the percentage of freqString-mutator.

0

50000

100000

150000

200000

250000

300000

350000

400000

450000

500000

550000

600000

650000

700000

750000

800000

850000

900000

950000

1000000

# CONCLUSION

Fuzzing is a good tool for the security testing of software. However, fuzzing tools usually require some setup and additional information to perform better. This thesis presented a new approach for fuzzer optimization by using a genetic algorithm to automatically balance and re-adjust the mutators of a fuzzer. The implementation of this approach was named AMGA. Compared with the black-box fuzzing, the results showed an average increase of 19.7% in the coverage with about 8.2% slowdown when using the mutator balancing. The slowdown of the mutator balancing is however justified with the larger coverage found and shorter time between the unique blocks covered.

In general, the idea of the mutator balancing to optimize a fuzzer seems to have potential. Most of the time it performs better than the unoptimized fuzzing with a minor tradeoff in the speed. Mutator balancing has its place among the fuzzing optimization techniques. It can be introduced as an additional fuzzing optimization method alongside for example input sample set optimization like in the AFL or function path search optimization like in the KameleonFuzz. An addition of mutator balancing provides more robust and comprehensive testing. This better testing in turn provides safer software, which is a requirement in the world where the amount of computers is in steady rise.

Future work in the field of testing and security research holds many possibilities, and the importance of this research should not be forgotten. The rise in the amount of the users of the Internet and the ever growing Internet-of-things mean that the importance of the security testing is only increasing. Most likely two things will be more important in the future: complexity and automatization of the testing. Complexity means that the testing tools will become more and more complex, searching for harder to find vulnerabilities. This trend is already visible in fuzzing, where it is not enough to create and use simple black-box fuzzers and hope for the good results. Automatization of the tests is important so that the testing can be performed constantly in all the stages of development. In the fuzzing, this means automating the tests, sample sets and the other aspects of the fuzzing process.

# REFERENCES

[1] “CVE Security Vulnerability Database.” [Online]. Available: [http://www.cvedetails.com/.](http://www.cvedetails.com/) [Accessed: 24-May-2015]

[2] “World Internet Users Statistics and 2015 World Population Stats” [Online]. Available: [http://www.internetworldstats.com/stats.htm.](http://www.internetworldstats.com/stats.htm) [Accessed: 10-June-2015]

[3] Kim HC, Choi YH & Lee DH (2011) Efficient file fuzz testing using automated analysis of binary file format. Journal of Systems Architecture 57(3): 259-268.

[4] Miller C & Peterson ZN (2007) Analysis of mutation and generation-based fuzzing. Independent Security Evaluators. URL: ftp://fre- mirror.picosecond.org/defcon/defcon15- cd/Speakers/Miller/Whitepaper/dc-15-miller-WP.pdf. Accessed 10.6.2015

[5] Cowan C, Pu C, Maier D, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q & Hinton H (1998) StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. Proc. Usenix Security. 98. San Antonio, Texas, 63-78.

[6] Sparks S, Embleton S, Cunningham R & Zou C (2007) Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. Proc. Computer Security Applications Conference, 2007. ACSAC 2007. Miami Beach, Florida, 477-486.

[7] Guang-Hong L, Gang W, Tao Z, Jian-Mei S & Zhuo-Chun T (2008) Vulnerability analysis for x86 executables using genetic algorithm and fuzzing. Proc. Third International Conference on Convergence and Hybrid Information Technology, 2008. ICCIT'08. Busan, South Korea, 491-497.

[8] Miller BP, Fredriksen L & So B (1990) An empirical study of the reliability of UNIX utilities. Communications of the ACM 33(12): 32-44.

[9] Oehlert P (2005) Violating assumptions with fuzzing. IEEE Security & Privacy 3(2): 58-62.

[10] Sutton M, Greene A & Amini P (2007) Fuzzing: brute force vulnerability discovery. Indiana, Pearson Education.

[11] Godefroid P, Levin MY & Molnar DA (2008) Automated Whitebox Fuzz Testing. Proc. NDSS 2008. San Diego, California, 151-166.

[12] Vuagnoux M (2005) Autodafe: An act of software torture. Proc. 22nd Chaos Communications Congress. Berlin, Germany.

[13] Newsome J & Song D (2005) Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. Proc. NDSS 2005. San Diego, California.

[14] Clause J, Li W & Orso A (2007) Dytan: a generic dynamic taint analysis framework. Proc. 2007 international symposium on Software testing and analysis. New York, New York, 196-206.

[15] Schwartz EJ, Avgerinos T & Brumley D (2010) All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). Proc. IEEE Symposium on Security and Privacy. Oakland, California, 317-331.

[16] Briand L & Pfahl D (1999) Using simulation for assessing the real impact of test coverage on defect coverage. Proc. 10th International Symposium on Software Reliability Engineering. Boca Raton, Florida, 148-157.

[17] Del Frate F, Garg P, Mathur AP & Pasquini A (1995) On the correlation between code coverage and software reliability. Proc. Sixth International Symposium on Software Reliability Engineering. Toulouse, France, 124- 132.

[18] Cai X & Lyu MR (2005) The effect of code coverage on fault detection under different testing profiles. Proceedings of the 1st international workshop on Advances in model-based testing. New York, New York, 1-7.

[19] Marick B (1999) How to misuse code coverage. Testing Foundations. URL: https:/[/www.dami](http://www.damianbrunold.ch/files/bbst/Marick_coverage.pdf)a[nbrunold.ch/files/bbst/Marick\_coverage.pdf.](http://www.damianbrunold.ch/files/bbst/Marick_coverage.pdf)

Accessed 27.6.2015

[20] Godefroid P, Kiezun A & Levin MY (2008) Grammar-based whitebox fuzzing. Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, New York, 206-215

[21] Neystadt J (2008) Automated penetration testing with white-box fuzzing. MSDN Library. URL: https://msdn.microsoft.com/en- us/library/cc162782.aspx. Accessed: 5.9.2015

[22] Kang MG, McCamant S, Poosankam P & Song D (2011) DTA : Dynamic Taint Analysis with Targeted Control-Flow Propagation. Proc. NDSS 2011. San Diego, California.

[23] “American Fuzzy Lop.” [Online]. Available: [http://lcamtuf.coredump.cx/afl/.](http://lcamtuf.coredump.cx/afl/) [Accessed: 24-May-2015].

[24] “LibFuzzer.” [Online]. Available: [http://llvm.org/docs/LibFuzzer.html.](http://llvm.org/docs/LibFuzzer.html) [Accessed: 24-May-2015].

[25] “Honggfuzz” [Online]. Available: https://github.com/google/honggfuzz [Accessed: 7-June-2015]

[26] Bardas AG (2010) Static Code Analysis. Journal of Information Systems & Operations Management 4(2): 99-107.

[27] Zitser M, Lippmann R & Leek T (2004) Testing static analysis tools using exploitable buffer overflows from open source code. Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering. New York, New York, 97-106.

[28] Cowan C, Barringer M, Beattie S, Kroah-Hartman G, Frantzen M & Lokier J (2001) FormatGuard: Automatic Protection From printf Format String Vulnerabilities. Proceedings of the 10th conference on USENIX Security Symposium. Washington, DC.

[29] Russell S, & Norvig (2003) Artificial Intelligence: A modern approach.

USA, Pearson Education.

[30] Arifovic J (1994) Genetic algorithm learning and the cobweb model.

Journal of Economic Dynamics and Control 18(1): 3-28.

[31] Bakirtzis A, Petridis V & Kazarlis S (1994) Genetic algorithm solution to the economic dispatch problem. IEE Proceedings-Generation, Transmission and Distribution 141(4): 377-382.

[32] Iba H, Higuchi T, De Garis H & Sato T (1993) Evolutionary learning strategy using bug-based search. Proceedings of the 13th international joint conference on Artifical intelligence. Beijing, China, 960-966.

[33] Branke, J (2012) Multi-Objective Evolutionary Algorithms and MCDA. European Working Group “Multiple Criteria Decision Aiding”. URL: [http://www.cs.put.poznan.pl/ewgmcda/newsletter/Spr12.pdf.](http://www.cs.put.poznan.pl/ewgmcda/newsletter/Spr12.pdf) Accessed: 19.5.2015

[34] Horn J, Nafpliotis N & Goldberg DE (1994) A niched Pareto genetic algorithm for multiobjective optimization. Proceedings of the First IEEE Conference on Evolutionary Computation. Orlando, Florida, 82-87.

[35] Abraham A & Jain L (2005) Evolutionary multiobjective optimization.

London, Springer.

[36] Duchene F, Rawat S, Richier J & Groz R (2014) KameleonFuzz: evolutionary fuzzing for black-box XSS detection. Proceedings of the 4th ACM conference on Data and application security and privacy. San Antonio, Texas , 37-48.

[37] Gorbunov S & Rosenbloom A (2010) Autofuzz: Automated network protocol fuzzing framework. International Journal of Computer Science and Network Security 10(8): 239-245.

[38] “SanitizerCoverage” [Online]. Available: <http://clang.llvm.org/docs/SanitizerCoverage.html> [Accessed: 11-June- 2015]

[39] “Radamsa” [Online]. Available: https://github.com/aoh/radamsa. [Accessed: 10-June-2015]

[40] “Surku” [Online]. Available: https://github.com/attekett/Surku [Accessed: 11-June-2015]

[41] “Zzuf” [Online]. Available: [http://caca.zoy.org/wiki/zzuf.](http://caca.zoy.org/wiki/zzuf) [Accessed: 10- June-2015]

[42] “address-sanitizer” [Online]. Available: https://code.google.com/p/address- sanitizer/ [Accessed: 11-June-2015]

[43] “GCC 4.8 Release Series” [Online]. Available: https://gcc.gnu.org/gcc- 4.8/changes.html [Accessed: 11-June-2015]

[44] “SDL Process: Phase 4: Verification” [Online] Available: https://msdn.microsoft.com/en-us/library/cc307418.aspx [Accessed: 27- Novmber-2015]

[45] “Projects/LibRsvg” [Online]. Available: https://wiki.gnome.org/action/show/Projects/LibRsvg [Accessed: 20-July- 2015]

[46] “Libav” [Online]. Available: https://libav.org/ [Accessed: 5-October-2015] [47] “MozillaSecurity/fuzzdata” [Online]. Available:

https://github.com/MozillaSecurity/fuzzdata [Accessed: 8-August-2015]

[48] “LibAv documentation : FATE automated testing environment” [Online].

Available: https:/[/www.libav.or](http://www.libav.org/fate.html)g[/fate.html](http://www.libav.org/fate.html) [Accessed: 7-October-2015]