

Final Year Dissertation

# Test Case Generation for Software Fuzzing using Genetic Programming to Detect Vulnerabilities

Department of Computer Science

School of Mathematical and Computer Sciences

Heriot Watt University Dubai

Author: Cassandra Ann Fernandes

Supervisor: Dr. Mohammed Hamdan

BSc Computer Systems Hons.

April 23, 2018



## Abstract

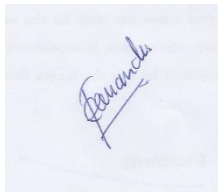
With the growing demand of solutions by stakeholders are on the rise, there is an increasing demand for ready to use software to suit the client's needs. Such software developed in a small amount of time can lead to unanticipated behavior from the software. By focusing on repeated testing of the software, vulnerabilities can be identified and losses can be prevented. This paper looks into researching the process of the generation of better test cases using genetic algorithms to improve fuzzing results.

**Key Words:** Fuzzing, Genetic Programming, Genetic Algorithms, Test Cases, QuickSort

## Declaration

I, Cassandra Ann Fernandes, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed:

A photograph of a handwritten signature in blue ink on a white piece of paper. The signature is written in a cursive style and appears to read 'Cassandra Fernandes'.

Date: April 23, 2018

## Table of Contents

<b>Chapter 1. Introduction</b>	<b>1</b>
1.1 Aims	1
1.2 Objectives	1
1.3 Research Question	1
1.4 Document Structure	2
<b>Chapter 2. Background</b>	<b>3</b>
2.1 Vulnerabilities	3
2.2 Software Fuzzing	4
2.3 Classification of Fuzzers	4
2.4 Components of a Fuzzer	5
2.5. System under test - QuickSort and it's time complexity	5
2.6 Introduction to Genetic programming	6
2.7 Genetic algorithms	7
2.7.1 Components of the genetic algorithm	7
2.7.2 Stages of the genetic algorithm	12
2.8 Related Works	14
2.9 Conclusions from Related Works	21
<b>Chapter 3. Methodology</b>	<b>23</b>
3.1 Requirements	23
3.2 Implementation Design	25
3.3 Program Components	27
3.4 Technologies Implemented	34
3.4.1 Languages	34
3.4.2 Environments	34
<b>Chapter 4. Evaluation</b>	<b>35</b>
4.1 Evaluation Overview	35
4.2 Measures for Evaluation	36
4.2.1 Fitness of Population	37
4.2.2 Comparison of Initial and Final Individuals	39
4.3 Discussion	41

<b>Chapter 5. Conclusion .....</b>	<b>43</b>
5.1 Summary .....	43
5.2 Limitations and Future Work .....	44
<b>References .....</b>	<b>45</b>

## Chapter 1.

### Introduction

Software fuzzing can be used to test a system in which, malformed inputs are run against it. By observing this behavior, we can identify under what conditions specified in a test case, that causes this behavior. When doing so, the process can expose possible vulnerabilities present in the system under test under those conditions. In our research we present an optimized approach to finding these vulnerabilities. The system that is being tested is the quicksort algorithm that takes the test cases generated by the genetic algorithm to trigger the unexpected behavior. This unexpected behavior is considered a vulnerability.

#### **1.1 Aims**

The aim of this project is to conduct research into existing approaches to software fuzzing. We implement genetic programming to optimize the generation of inputs to cause unexpected behavior of a system.

#### **1.2 Objectives**

To generate test cases that are valid but anomalous inputs for the system under test.

To identify and distinguish between expected and unexpected behavior of the system under test.

To evaluate the results of the genetic algorithm through analysis of the unexpected behavior of the system under test.

#### **1.3 Research question**

This paper focuses on test case generation in respect to software fuzzing. We conduct research on how to generate test cases as input to cause unexpected behaviors in the target system under test.

## **1.4 Document Structure**

Our research is first structured into a brief background into key areas of our research and the related works that helped inspire this project. This is followed by an initial design of the program that was developed and how the implementation of our approach was carried out. We then evaluate our research report the analysis behind the results and finally conclude what has been achieved and future work regarding our research.

## Chapter 2.

### Background

In this chapter we shall look into existing literature about genetic programming and software fuzzing. The structure of this chapter will aim to provide the reader on how genetic programming and software fuzzing work. We shall also explore how our implementation focuses on the genetic programming in software fuzzing in terms of test case generation.

#### **2.1 Vulnerabilities**

“A security flaw is a defect in a software application or component that, when combined with the necessary conditions, can lead to a software vulnerability” [1].

A vulnerability can be classified as a set of conditions that permit violations of security policies put in place.[1]

In the context of software, these vulnerabilities can be indications of a flaw or defect found in a software program or system. The defects can be found in the system design or implementation where they are overlooked or hidden. If appropriate action is not taken, these flaws can be exploited by an attacker (i.e) the vulnerability can be taken advantage of by a person with unlawful intentions.

There are numerous vulnerabilities in code that can be exploited due to design or implementation errors. Since software development is not a foolproof process, there will always exist a vulnerability in that code. Vulnerabilities that are known or unknown will always pose a risk to those that use the computer software. They would be open to attack or malfunction due to these gaps in the code.

In our research, we will be focusing on how the time taken to process an input is seen as a vulnerability in our system. The input is the test cases generated by the genetic algorithm.



## 2.2 Software Fuzzing

Software fuzzing is a systematic approach [2] to software testing where a series of structured but anomalous test cases are used to test a system. This approach can be used on any system that takes an input to process across a system boundary. Normally, fuzzers are utilized to test programs that take structured inputs. The inputs of the fuzzer are the test cases that can be used to generate an unexpected behavior. This is achieved if the test case that is being processed, either passes or fails against the expected behavior of the system. The main purpose of fuzzing is to detect vulnerabilities that have not yet been discovered within the system. Furthermore, the behavior can be analyzed to improve these systems based on the test cases that are used. Any deviation from expected behavior of the system can indicate a failure or vulnerability within the system.

## 2.3 Classification of Fuzzers

To further analyze what software fuzzers are, we shall take a look into the types of fuzzers.

Software fuzzers can be separated based on test cases that are processed and fuzzing methods that are implemented. [3] They can further be broadly classified into the based on the test cases that are generated. They are:

**1. Mutative or dumb fuzzers:** A mutative fuzzer uses an existing model of a test case and alters or mutates some of the features to produce a new test case. An example of a mutative fuzzer is ZZUF.[4] ZZUF employed bit flipping where some of the bits in a sequence are mutated randomly. The bit flipping process takes an executable file, and using the shared library file LibFuzz, changes the bits randomly in the input format of the file.

**2. Generative or intelligent fuzzers:** A generative fuzzer creates a new test case for input depending on what input is required by the system to be fuzzed and at what depth. A good example of a generative fuzzer is Peach Fuzzer. [5] Peach fuzzer can use mutation as well as generative fuzzing through the use of data and state modelling of a test case.

## **2.4 Components of a fuzzer**

A fuzzer is composed of three components [6]:

1. The poet: For testing the system, a series of malformed inputs need to be created to test the it's behavior. This component is responsible for generating these inputs.
2. The courier: The courier is responsible for the delivery mechanism that will deliver the test cases generated by the poet to the system that is to be fuzzed.
3. Oracle: The oracle is responsible for monitoring the behavior of the system that is under test to see if it deviates from expected behavior.

For the purpose of this research, we shall be focusing on the area of the poet, where the malformed inputs are to be generated by the use of a genetic algorithm.

## **2.5 System under test - QuickSort and it's time complexity**

Quicksort is a sorting algorithm that uses divide and conquer strategy to be able to sort elements. The strategy partitions a structure of elements around a decided pivot. The pivot is the index that is used to sort the elements in a structure. The main process in the quicksort is the partition where the pivot splits the structure of elements into smaller parts to be sorted into order. For example, let us consider an array of elements such that, for an element  $x$  as the pivot, put  $x$  at the correct position in the sorted

array. Next, put all smaller elements (smaller than  $x$ ) before  $x$ , and put all greater elements (greater than  $x$ ) after  $x$ . [7] The quicksort algorithm uses different pivots to sort elements. They are as listed below: [7]

1. First element as a pivot: The first element is selected as the index to sort the elements in the structure.
2. Last element as a pivot: The last element in the structure is used to sort the elements in the structure.
3. Random element: A random element is chosen from the structure to act as an index to sort the elements in the structure.
4. Median element as pivot. The difference between the first and last element in the size of the structure gives the median element as a pivot for the structure to sort the elements.

The average quicksort time complexity is  $O(n \log n)$ , where the average number of comparisons is dependent on the size of the structure with elements being sorted. The time complexity deteriorates as the number of elements in the structure increases. In this case the worst time complexity of the algorithm is  $O(n^2)$ . For our implementation, the vulnerability here is the worst case time complexity that will be triggered due to the test cases generated by the genetic algorithm.

## **2.6 Introduction to Genetic programming**

Genetic programming is an automated method for problem solving in computer programs. It is based on the Darwinian principle of the survival of the fittest and draws its ideals from natural selection. Darwin's principle of natural selection describes the survival of a species that can adapt to a changing environment. These individuals are selected to reproduce and evolve into a newer generation. Using this ideology, operations such as selection of individuals, crossover and mutation take place. The individual that comes closest to solving the problem can be called the fittest individual. The fittest individual is best

described as the individual that has the highest reproductive success to be able to get the optimal solution to the problem.

The Genetic algorithm technique is mainly concerned with the problems of optimization. It selects the best test case that can create an unexpected behavior that could lead to an exposed vulnerability. [8]

## **2.7 Genetic algorithms**

Genetic algorithms build from existing machine learning and search techniques to solve a problem. [9]

They were developed by Professor John Holland and his team of students at the University of Michigan during the period of 1960's and 1970's. [10]

The genetic algorithm is seen as a way of evolving and generating the optimal solution to a particular problem using the process of evolution of the fittest individuals.

The genetic algorithm selects the optimal solution to solving a problem by computing the fitness values. Fitness values are the best scores of test cases that can trigger an unexpected behavior. This unexpected behavior can reveal new vulnerabilities in the code. [1]

### **2.7.1 Components of the genetic algorithm: [8]**

**Initialization:** Initialization is necessary to create an initial population. The population can be randomly generated and the size of the population can be defined in this component. The parameter values are selected either randomly or chosen from previous generations or from some existing knowledge of protocol.

**Individual:** Individuals are the elements to which a fitness function is applied. An individual can consist of even smaller components called genes. A collection of genes make an individual. Figure 2.1 shows an individual consisting of four genes.

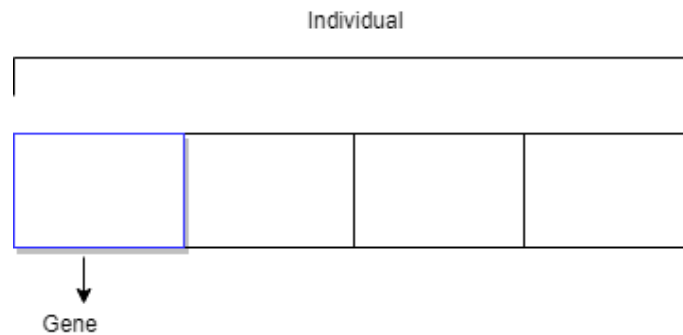


Figure 2.1 illustrating genes in an individual

In the context of the genetic algorithm, individuals are seen as the genes that make up the population. However, in the view of our implementation, the initial test case is considered as the individual.

**Population:** A population is a collection of individuals after the process of fitness determination takes place. The population should have a large set of individuals to be able to maintain a healthy offspring of generations. The figure 2.2 shows the entire population in a genetic algorithm consisting of an individual and its genes.

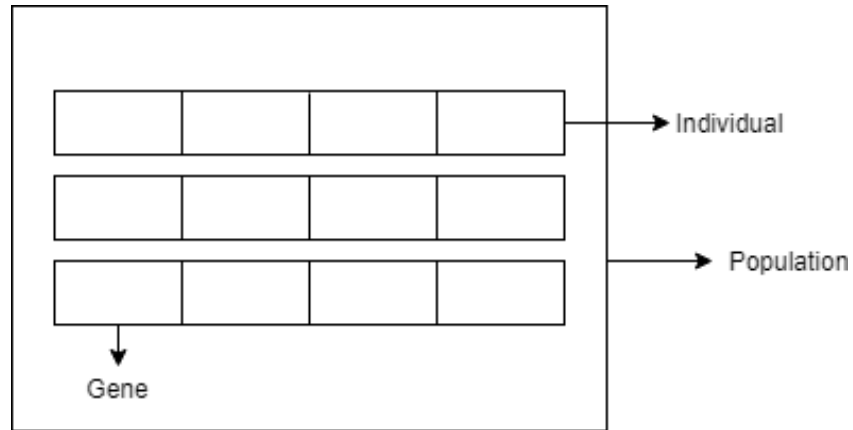


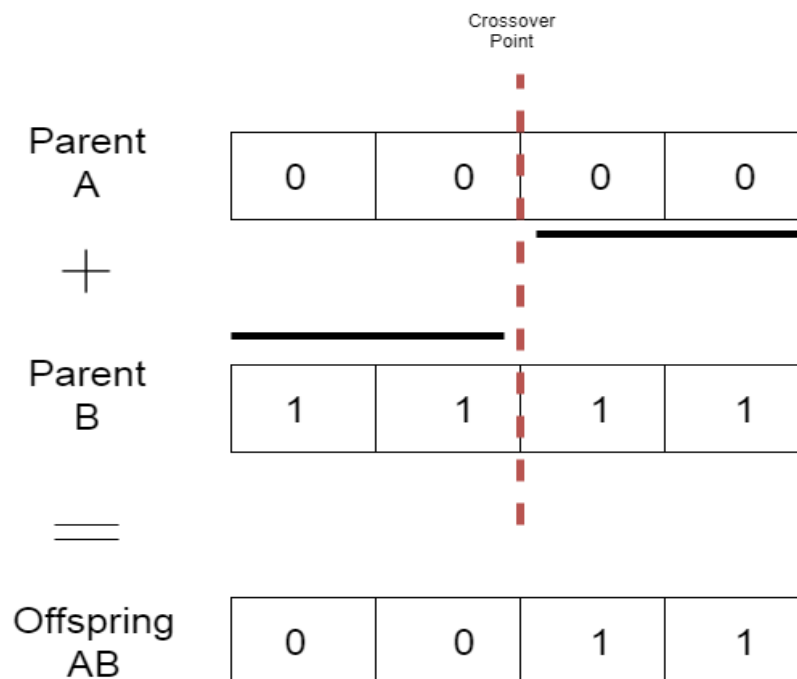
Figure 2.2 illustrating a population

**Representation:** The representation of the individuals acts as a model for generating new individuals. In order to have a representation, it is required to optimize the problem and increase the search space while preventing redundancy of the same individuals. This should be done at minimal computational cost. [11]

**Fitness Function:** The fitness function is at the core of the algorithm where the problem-solving ability of the individual is determined. The fitness function is the most challenging to write as any discrepancy results in the optimal solution being outside the area of the search. The area of search is the population from which we would like to get the results for. The fitness function assesses the appropriateness of a test case given the input parameters. For our implementation, we have decided to use the time taken from the quicksort of an individual as the fitness for the generation of a new population.

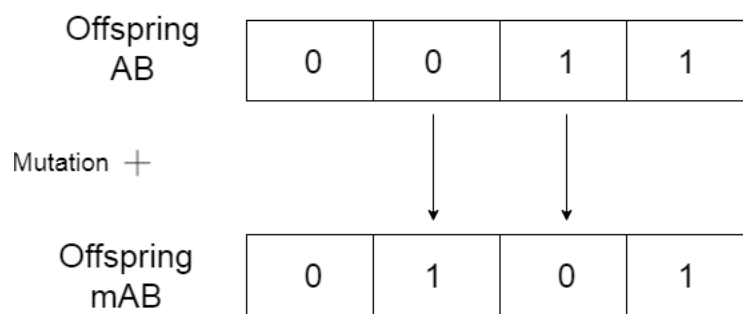
**Selection:** The selection process decides which individuals to be used for reproduction of new individuals in a new generation. The chosen individuals need to be closest to the fitness function to improve the quality of the problem-solving ability of the newer population. Usually selection is done randomly in a roulette wheel fashion where two individuals are picked randomly to reproduce. They are decided based on a random index within the population.

**Crossover:** Crossover is the reproduction phase of the individuals to produce the best fit offspring. During this process, the parent individuals exchange substring data to be able to generate offspring. Below the figure 2.3 shows the process of crossover of the two parent individuals. The crossover point that is decided becomes the index where both parents are combined to form an offspring. At the crossover point, the second part of Parent A is combined with the first part of parent B to give the offspring AB.



**Figure 2.3 illustrating the crossover of two parent individuals**

**Mutation:** In order to generate diversity in the population of individuals, mutation is required. The operation is usually performed after the crossover. For our research, the mutation function alters a random bit in the individual. For example, figure 2.4 shows the alteration of the offspring where the bits of the gene are flipped. In our implementation, the bits are flipped randomly according to the mutation rate given as a parameter.



**Figure 2.4 depicts the mutation function on an individual**

**Evaluation function:** The evaluation function works similarly to the selection function. After performing the selection and mutation function, the resulting offspring is evaluated to see if they meet the optimal solution to the problem.

**Generation of new population:** When the genetic algorithm creates a new set of populations, a series of genetic operations take place where the individuals are evaluated to find which of these are a best fit to the fitness function. This process is the generation of a new population.



### 2.7.2 Stages of Genetic Algorithm [8]

The figure 2.5 shows the stages in the genetic algorithm that will achieve an optimal solution to the problem.

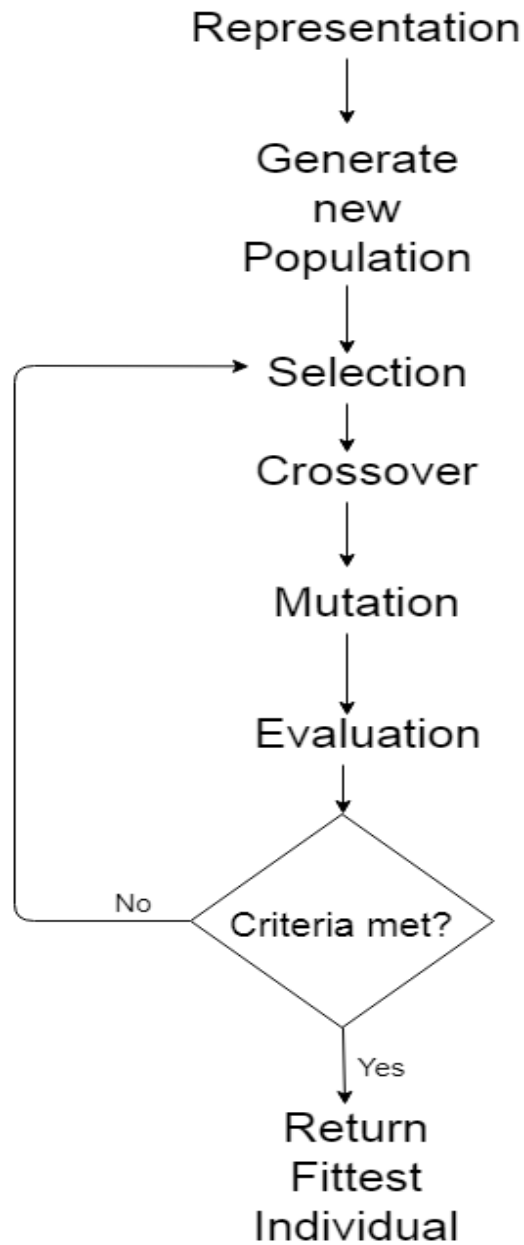


Figure 2.5 showing the stages of the genetic algorithm

1. Define the individuals representation format: Defining how we are to represent the individuals in the population is the first step in the stages of development of the genetic algorithm.
2. Generate the initial population: This stage involves Initializing the population of individuals to solve the problem.
3. Use the genetic selection operator to find the best suitable fitness value: Selection of the chromosome to be passed into the new generation.
4. Select the fittest parent individuals for reproduction: The individuals need to be selected based on how close they are in terms of the fitness values to find the solution.
5. Reproduce the selected parent individuals using crossover and mutations: After selection of the fittest parents, the genetic operators which are the crossover and mutation are performed to get the fittest offspring.
6. Output the offspring into the population: Evaluate the individual to be reintroduced back into the population.
7. If criteria is met, return the fittest chromosomes, else loop back to step 3:

## 2.8 Related Works

**McNally, Yiu, Grove, et al. [12]** aim to create a basis for fuzzing. They acknowledge the identification of different types of fuzzers. The beginning of fuzzer development is said to have begun at the university of Wisconsin as a part of a class project led by Professor Barton Miller where the official term "fuzzer" was coined.

The paper discusses the analysis of fuzzers in general and how progression of fuzzers takes place.

The description on fuzzer types explores the different approaches to fuzzing where mutative and generative fuzzers are described. Further classification considers template based fuzzing where the fuzzer has some understanding of the input structure. The block based approach is discussed where the technique makes it easier for data representation. Grammar based fuzzers use the grammar to fuzz the input language of the system under test. The research helps to create a basic understanding of a fuzzer, how it works and what kind of approaches already exist while building a fuzzer. Their report gives a brief look into the future works of fuzzers.

For our implementation, we have focused on building a generative type fuzzer that uses a sorting algorithm as the system under test.

**Reber, Cha, Avgerinos, et al. [13]** focus on how the effect on best seed files can maximize the total number of bugs that are found during a fuzz campaign. They assess 6 different algorithms during a period of 650 CPU days on Amazon's Elastic Compute Cloud (EC2). Their results show that they found 240 bugs in 8 applications during this campaign and analyze the effect of the choice of the algorithm used on the number of bugs found. In this paper, the authors assume that increasing the number of bugs found is the main goal while ignoring the type of fuzzer used. The purpose of reviewing these algorithms is to select a minimum set (minset) of input seed files from a larger set of files. The following seed selection algorithms are chosen: The Peach set cover algorithm that has no minimum set cover, A

minimal set cover, a minimal set cover weighted by its size and a hotset algorithm. These were evaluated to see which algorithm yielded the best results were given a set of seed files to calculate the data reduction.

This research question is not proven since the authors focus on file parsing applications. Their target applications do not use command line to get their input. They suggest extending the fuzzing framework to generate inputs randomly and conclude that support for more input types as future work.

**Goyal and Kaur [14]** propose a new genetic algorithm that prioritizes test cases based on the complete code coverage of that test case. The code coverage of the test case shows how much of the code the test case was able to reach. The genetic algorithm automates the process of prioritization of test cases. Through this approach a suitable population is pulled where it is made to be more adaptable and efficient.

The two fitness criteria that need to be satisfied in order to select the suitable population are maximum fault covered in the execution time and the total code coverage by the algorithm. The genetic operations are performed where the crossover and mutations combine and randomly swaps the entities.

A minimization function was employed where the individuals which were redundant had to be removed. Next, the optimization process takes place where if the fitness criteria are not matched, repeated reproduction and genetic operators take place. The two individuals selected are a test suite of nine cases. This is the test suite where the optimized solution did not work and another test suite of eight were selected to perform the genetic operations instead. Evaluation of the approach is done using Average Percentage of Condition Coverage (APCC) values where it assesses the code coverage testing and effectiveness. Their research show the potential of the approach to save time and effort to prioritize test cases but also identified files the risks since the genetic algorithm is limited to a small number of

programs. It concludes more analysis is required. The future works suggest having a larger number of programs to run the proposed algorithm on.

**Godefroid, Peleg, and Singh [15]** propose an approach where they focus on automating the generation of input grammar for fuzzing using sample inputs. Their Problem area is of automatically generating input grammars for grammar-based fuzzing by using machine-learning techniques and sample inputs. This was achieved using neural network based statistical machine learning techniques. In addition, they implement a new algorithm to decide where to fuzz using input probability distribution. The main idea is to learn a generative language model over the set of PDF object characters given a large corpus of objects. To establish baseline coverage, they choose 1000 PDF objects from the training objects of 63000 training objects. Their coverage on the Microsoft EDGE parser is measured over 3 hosts. Their results conclude that even though the baseline coverage varied over the different hosts, 90 per cent of the host includes coverage regardless of what new objects appends i.e. the “re-combined” PDF file is always perceived as well formed by the EDGE pdf parser. While learning new PDF objects, comparing sample and sample space algorithms, the results show that the pass rate is between 70-90 per cent meaning the learning is of good quality.

**Pargas, Harrold, and Peck [16]** propose a technique where a genetic algorithm is used for automatic test case generation. The test data generation approach executes a given statement in the system that is being tested. Their technique is applied in the algorithm that they develop called Tgen. Tgen was developed to perform parallel processing. Parallel processing is used to improve the performance of the search goal of the optimal solution. The algorithm evaluates the input which is the test data. When the program is tested, it records the predicates that are found in the program that execute with the test data. The approach uses multiple search goals instead of one to maximize the variety of test case generation.

**Kim, Choi, and Lee [17]** develop a methodology that is concerned with the buffer overflow vulnerability of software applications. The methodology proposes to analyze the parsing system mechanism of the software system in real time. By doing so, the number of fault-inserted files reduces for fuzz testing. Their research is mainly concerned with the analysis of the file format and generation of the minimum number of fault inserted files and its test case coverage. They devise an algorithm to divide the binary file format into fields that can insert fault data into the file considering the file format. The analysis algorithm is used to get maximum test case coverage with only a small number of fault-inserted files. These are compared with another file fuzzer that does not consider file format. The methodology works by tracing the file execution and analyzing it at low level. For evaluating the efficacy of the methodology, BFAFI is made to analyze the parsing of WMF file format. The BFAFI analyzes the field position in the WMF file and the field type such as NT and BAT. The BFAFI is consisted of the Binary File Format Analyzer (BFA): traces parsing process of the WMF file and analyzes the field position and field type of the WMF file format. This is broken down into:

- Software Monitor (SM):

the role of the SM is to trace execution of the software system, write a log file about execution about values such as instruction, registers etc. It also monitors all parsing instructions and CPU related registers.

- Log Analyzer (LA): the LA analyzes the log file and extracts field information from the WMF file records. This is done because the it takes a long time for the BFA to trace and analyze the WMF format. Then the LA generates the relevant information of the file format to transfer to the Fault injection tool (FIT).

- **Fault Injection Tool (FIT):** The fault injection tool inserts faults into the fields of the WMF file while taking into consideration the file format and then executes the fault inserted files. While inserting faults, the FIT tool takes into consideration the NT and BAT fields of the WMF format. This is done so that the number of fault inserted files are reduced with maximum test case coverage.

Next the exception monitor executed the application using the fault inserted file and observes any exception event that may occur. If such an even does arise, the EM holds and analyzes it.

The BFAFI can find more exceptions when compared to a general fuzzer. The exception analysis shows that the BFAFI catches more exceptions when compared to results from FileFuzz. One of the limitations show the parsing functions needed to be identified in advance because of the extraction of field information needed by the BFAFI. It is also evident that the BFAFI cannot analyze some errors since they do not take into account some of the information in the BAT files.

In our implementation, the vulnerability we will be looking at is the time taken to sort an array of lists. The longer the time taken, the more effective are the test cases being sorted. These are the test cases that will be generated by the genetic algorithm to prove that the optimize test cases produce input that causes a vulnerability.

**Katerina Goseva-Popstojanova [18]** evaluate the effectiveness of detecting vulnerabilities of static code analysis tool to understand their strengths and weaknesses. They address the lack of quantitative results and small samples that have been observed in similar works. They conduct experiments consisting of

bench marking the test suite Juliet which is run against three commercial tools used for static code analysis. The results show that not all tools were able to detect all the vulnerabilities and produced false positives.

What can be observed here is that through the use of the genetic algorithm in our implementation, we have increased the size of the individuals and the population for the implementation.

**Patrice Godefroid [19]** create a different approach to whitebox fuzz testing. White box testing is a technique to identify vulnerabilities in code segments [3]. Their approach makes a record of the well-formed input of the system that is being tested and checks how the system uses these inputs. Optimization strategies are made by dynamically generating test cases as input files. This approach is similar to **Goyal and Kaur [14]** where redundancy is avoided in the search using their algorithm by maximizing the new tests generated.

In our implementation, to avoid redundancy, we have used an effective mutation technique to counter this issue.

**Iozzo [20]** proposes a system that shows how static and dynamic analysis can be used together in the view of security testing issues. Static Analysis determines which functions to fuzz. Requirements include at least one loop and high cyclomatic complexity scores. The metric calculates the number of independent paths that are present in a code block. Cyclomatic complexity is the complexity of a function in terms of code path that is computed. This is based on how many number of edges and nodes a function has in the code.

There is a direct relationship where more complicated the structure, higher the complexity. Here,



the dynamic analysis is done through a form of dynamic analysis which is through DYTAN. But it cannot explore the program paths at the time of runtime execution. It does not have enough data to compare the results computed by the fuzzer. A comparison of the fuzzer to the methodologies that are in evolutionary and mutation and shows that it has better results. But with generation based fuzzers, this technique will have better results if the input is more complex. Figure 2.6 showing the implementation of the 0 knowledge fuzzer.

The phases have been listed in the figure 2.6.

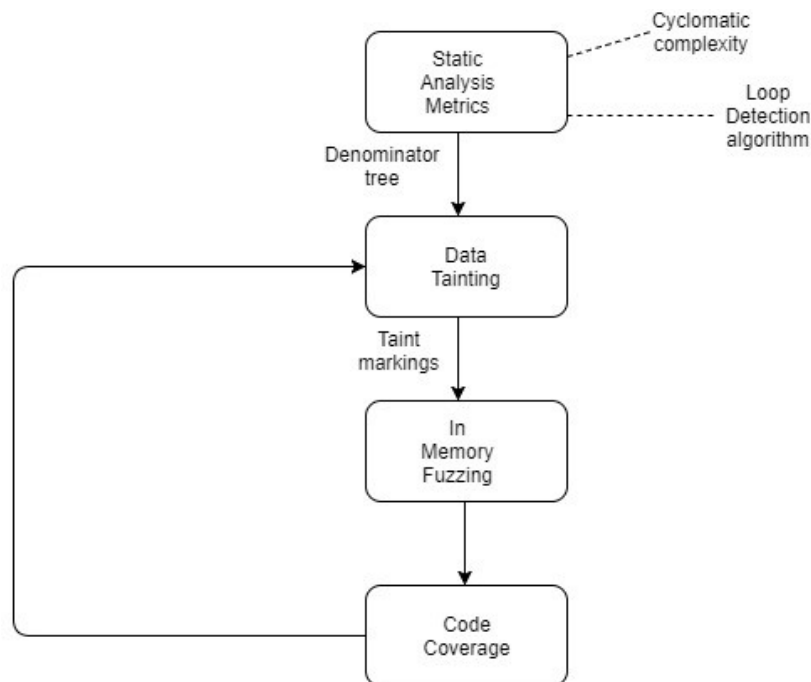


Figure 2.6 showing the phases of the 0 knowledge fuzzer

The Static analysis metrics take the Cyclomatic complexity and Loop detection algorithm to output a denomination tree. The tree data is tainted and with these markings move on to the next step which is in memory fuzzing using these markings. The overall implementation is checked for how much code was tested using the test case. This is the code coverage referenced in the figure 2.6 of the research.

**Sugihara [21]** define three measures for the performance of genetic algorithms. The measures are based on the observations from experiments by simulation. The three measures are the likelihood of optimality, the average fitness value and the likelihood of evolution leap. The likelihood of optimality is defined by the author as the estimated probability of reaching the optimal solution given that, the genetic algorithm was executed for  $k$  generations in  $n$  number of runs. Out of these runs,  $m$  number of runs generated an optimal solution within those  $k$  runs. The likelihood is defined as the probability  $m/n$  given the  $k$ th generation.

The average fitness values are those values of the best fitness obtained through execution of  $k$  generations in  $n$  runs.

Likelihood of evolution leap is defined as if there is an increase between the fitness of one generation over the other in succession. Based on these measures the best cut off generation  $K$  was also defined.

The best cut off generation is the number of generations should be executed by the GA in each run.

Their implementation that the experiments had been conducted on were of genetic algorithms that had Selections of different strategies, crossover points and mutation strategies. Their results show that the genetic algorithm strongly depends on the choices of parameters.

## **2.9 Conclusions from Related Works**

For the purpose of our implementation, we chose to evaluate the fitness values of the time being sorted. The best fit individuals are seen as the optimal solution to the problem and the implementation terminates either after the best individuals are found or before the number of generations terminate.

The optimal solution here is the worst case time complexity of the Quicksort algorithm. We generate individuals that will trigger  $O(n^2)$ . The trigger for this behavior are the test cases generated by the

genetic algorithm. The test cases will be decided based on the operations performed by the genetic algorithm which are selection of the fittest individuals, crossover of these and mutation of the fittest individuals here. The output of individuals that are generated from this approach gives the worst time complexity of the quicksort.

## Chapter 3.

### Methodology

This chapter takes a look into how the genetic algorithm and its operators were implemented to carry out our research. The requirements will give a brief outline of what was achieved and how the implementation helps to form a basis for our research.

#### **3.1 Requirements**

FR1: The system under test should be able to receive input test cases through command line interface.

The requirement has been modified to automate the process of the receipt of the test cases. The initial individuals in the population where the quicksort algorithm sorts the individuals. The quicksort in this case receives the individuals from the application of the genetic operators that find the best fitness after they have been sorted.

FR2: The algorithm should be able to generate optimized test cases.

The algorithm produces new generations up until a point where the optimal solution has been reached. In our implementation, the test cases that are generated are the best fit where the time taken is the optimal solution.

FR3: The Test cases generated should be valid but anomalous to test against the system under test.

This has been achieved where the test cases that are generated are sorted in the Quicksort Algorithm.

The individuals that are produced through the genetic algorithm are accepted by the Quicksort algorithm and the individuals in the population are sorted.

FR4: The output of the program should be displayed in readable format.

The generation of the new individuals are produced in the console of the environment.

FR5: The results of the genetic algorithm process should be stored for analysis and evaluation.

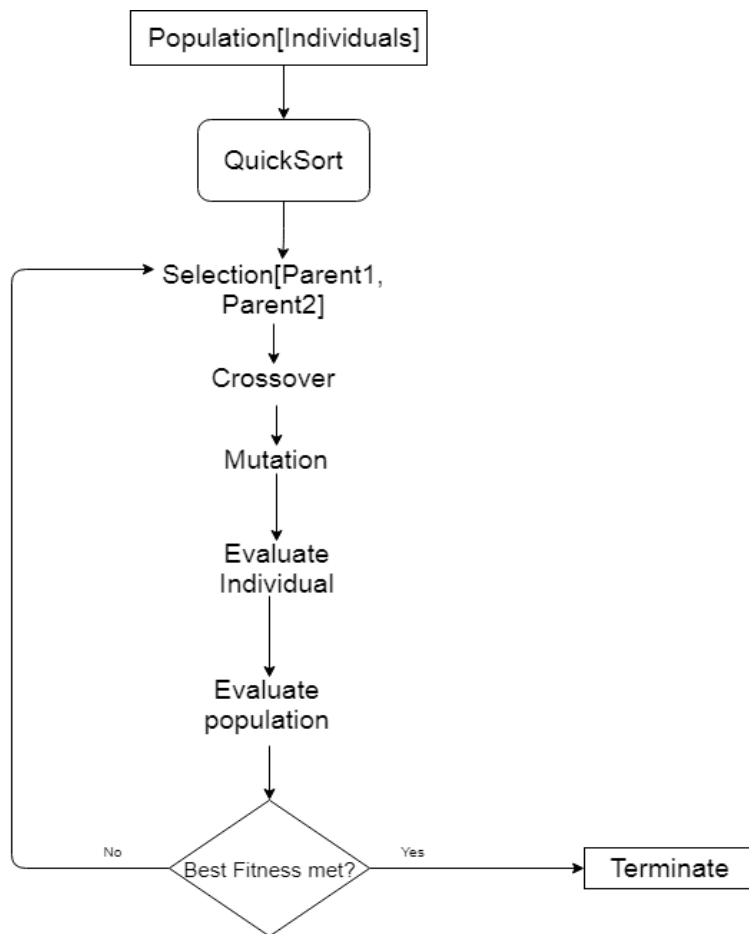
The results of the genetic algorithm process stores the best fit before and after sorting in a CSV file format for further evaluation.

FR7: The test cases should be able to be delivered from the software fuzz to the system under test.

The test cases generated from the genetic algorithm are sent to the sort function by continuously evaluating the population and evaluating the individual based on its fitness values. If the maximum generation is reached, the program terminates.

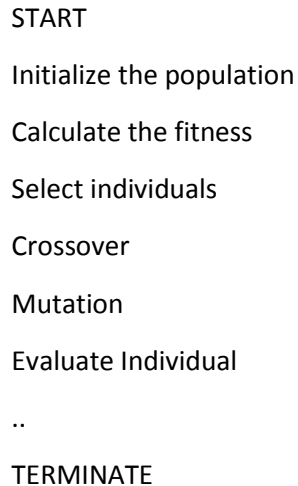
### 3.2 Implementation Design

The Initial design of the program was planned to be according to each stage of development of the algorithm. The system under test is the sorting algorithm where each test case generated by the genetic algorithm is the input for the quicksort. The genetic algorithm is then used to generate the test cases for the Quicksort algorithm to give the best fitness values for the next iteration of test cases to be produced. Figure 3.1 shows the initial program design using the Quick sort.



**Figure 3.1 showing the initial design for our approach.**

From the related works and background, a general approach to a simple genetic algorithm can be described in the following pseudo code figure 3.2:



```
graph TD; START([START]) --> Init[Initialize the population]; Init --> Calc[Calculate the fitness]; Calc --> Sel[Select individuals]; Sel --> Cross[Crossover]; Cross --> Mut[Mutation]; Mut --> Eval[Evaluate Individual]; Eval --> Dots[..]; Dots --> Terminate([TERMINATE]);
```

START

Initialize the population

Calculate the fitness

Select individuals

Crossover

Mutation

Evaluate Individual

..

TERMINATE

**Figure 3.2 showing general process of the genetic algorithm**

This varies with our program pseudocode as shown in figure 3.3:

```
create initial population
apply Quicksort to get fitness value (time taken to sort each individual)
select best fit individuals to reproduce
apply crossover operator using two crossover points
apply mutation operator using random point
evaluate the fitness(time taken to sort) of each individual
determine the fitness of the entire population
recombine the population and replace old population with fittest one
repeat the process from selection until the terminating condition (
number of generations reached)
```

**Figure 3.3 showing the pseudocode of the implementation**

Our implementation takes into account the system under test; the quicksort where the time taken to sort is the fitness function. The selection is done on the basis of the time taken to sort each individual in the population. The best fit individuals are those with the best fitness value. These individuals are selected to reproduce using the crossover and mutation operations. Next, the crossover is done on the two individuals using two crossover points and the mutation uses a random index to swap. Next and

evaluation is done on the basis of the fitness values of the individuals and a final evaluation of the new population is done to get the best fitness of the new population. This process repeats till the termination of the program where either the maximum number of generation has reached.

### 3.3 Program components:

To better understand the stages of the implementation of the program, we have discussed in detail about each component. Each are listed below for the reader's understanding of how it executes.

#### Creation of the individuals:

Initial creation of the individuals is done randomly. The individual is defined as an array with N random integers. There exists N number of elements in each list that is contained. For the purpose of our research and for analysis of the results of the quicksort, the dataset for the N elements are defined as random integers from 0 to N. Each of the individuals have random integers generated by using the random library in python. These have random integers from 1 to N integers in the individual. This is illustrated below in figure 3.4 where each individual consists of random integers from 1 to N.

```
def initializeIndividual():  
    ind = []  
    for j in range(0, N):  
        ind.append(random.randint(1, N))  
    return ind
```

**Figure 3.4 showing how the Individual is initialized**



### Population:

The population in our implementation is defined as an array of the individuals i.e. P is the population array that holds lists of N individuals. In our generative type model, the offspring that are generated by the genetic algorithm replace the new one after each iteration. The following figure shows how this is implemented in the code. The new population is replaced by the individuals that have been generated out of the genetic algorithm. Figure 3.5 shows how we have implemented the generation of each new population being updated according to the fitness over each generation.

```
for i in range(I,P):
    nextp.append(population[chosenOnes[i-I]])
population = nextp
```

Figure 3.5 showing the population being replaced by newer population

### QuickSort:

The implementation of the Quicksort uses a partition function that takes the middle element of the array as the pivot. In the figure 3.6, we check if the current element being sorted is smaller or equal to the pivot element and increments the index of the smaller element.

```
def partition(arr,low,high):
    i = ( low-1 )
    pivot = arr[(high+low)/2]
    for j in range(low , high):
        if arr[j] <= pivot:
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]
    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )
```

Figure 3.6 shows the partition of the quicksort

Next in figure 3.7 the quick sort function is called where the partitioning index pi is placed at the right place in the array arr. The elements are then sorted separately before the partition and after the partitioning of the array.

```
def quickSort(arr, low, high):  
    if low < high:  
        pi = partition(arr, low, high)  
        quickSort(arr, low, pi-1)  
        quickSort(arr, pi+1, high)
```

**Figure 3.7 demonstrating the quicksort function**

### **Fitness function:**

The quicksort algorithm takes the input test cases to get the fitness value defined which is the time taken to sort each individual in the population. The quicksort algorithm is the system under test. For our research, the fitness values should show to increase with every new individual generated by the genetic algorithm. The fitness function is a maximizing one where the time taken to sort should maximize or increase the fitness value. This means the time taken to sort increases with every iteration within the program. In the figure 3.8, we have used a timer to record the time taken to sort. The timer starts before the Quicksort sorts the individual and stops after the individual is sorted. The fitness value of the individual is defined as the difference between the two timers. The fitness value is then returned as fit.

```
def cal_fitness(self):  
    c1 = time.clock()  
  
    quickSort(self.ind, 0, len(self.ind)-1)  
    c2 = time.clock()  
    fit = c2 - c1  
  
    return fit
```

**Figure 3.8 showing the calculation of the fitness values for each individual**

### Selection:

The selection of the individuals for reproduction is a vital part of the program design. The selection function helps drive the population for every generation closer to the optimal solution. The selection operator of the genetic algorithm was implemented by using the best fit individuals from the population. The best fitness values are those with the time taken to sort the individuals in the population. Figure 3.9 shows the implementation using the selection operator.

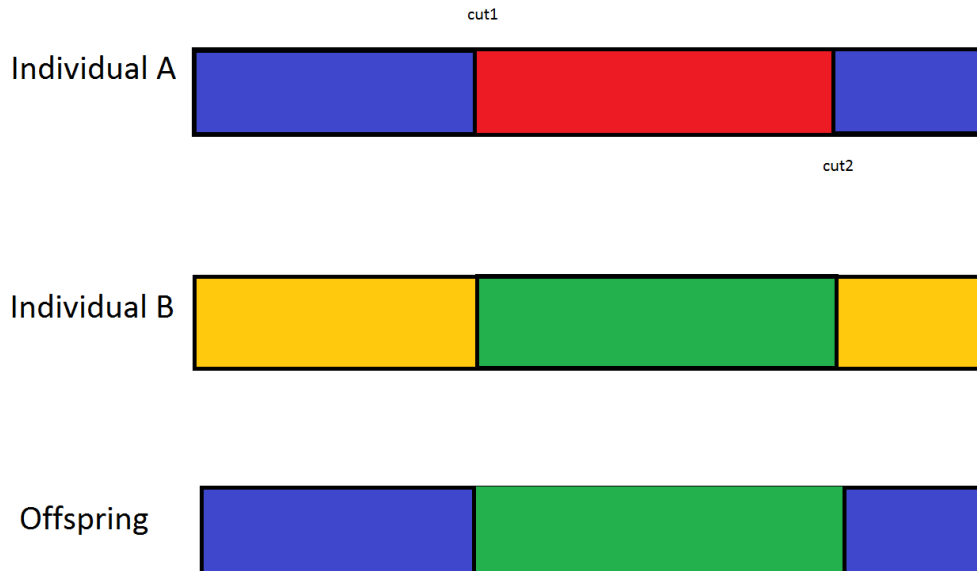
```
def generateFromDistribution(p, d, g):
    sum = p[0]
    prob = []
    returned = []
    for i in range(0, d):
        sum = sum + p[i]
        prob.append(sum)
    for i in range(0, g):
        proba = random.random()
        for j in range(0, d):
            if(proba <= prob[j]):
                break
        returned.append(j)
    return returned
```

Figure 3.9 showing the implementation of the selection operator

### Crossover:

The cross over of the selected individuals used a two point crossover. This strategy was implemented to ensure that the algorithm would have ample individuals to search for the optimal solution. The strategy uses two crossover points to segment both individuals. This is done keeping in mind the length of each individual. The crossover probability rate has been defined as 0.8. The crossover function takes two cut points in our implementation which are cut1 and cut2. First, the integers in the from the beginning of individual is copied till the first crossover point cut1. Next the part from the first crossover point from cut1 is copied till the second crossover point cut2 in the second individual. These two parts are appended with the remainder of the first individual to give the offspring individual. Figure 3.10 shows

the how the two crossover points were implemented in this function. The figure 3.3 shows how the code executes the crossover.



**Figure 3.10 illustrating the two point crossover function implemented.**

The following is the figure 3.11 of the python code to implement this crossover strategy.

```
def crossOver(i1, i2, d, nPoints):
    if (nPoints==2):
        cut1 = random.randint(0, d-1)
        cut2 = random.randint(0, d-1)
        while cut1==cut2:
            cut2 = random.randint(0, d-1)
        if (cut1 > cut2):
            i=cut1
            cut1=cut2
            cut2 = i
        is1 = []
        is2 = []
        for i in range(0, cut1+1):
            is1.append(i1.ind[i])
            is2.append(i2.ind[i])
        for i in range(cut1+1, cut2+1):
            t = i1.ind[i]
            i1.ind[i] = i2.ind[i]
            i2.ind[i] = t
            is1.append(i1.ind[i])
            is2.append(i2.ind[i])
        for i in range(cut2+1, d):
            is1.append(i1.ind[i])
            is2.append(i2.ind[i])
```

```

    IS1 = Individual(is1)
    IS2 = Individual(is2)

    p = []
    p.append(IS1)
    p.append(IS2)
    return p

```

**Listing 3.11 illustrating how the crossover operator uses two crossover points.**

### **Mutation:**

The mutation function of the algorithm that was implemented, swaps the bits of each selected individual. This is employed by flipping the bits randomly of each individual. This mutation occurs using a defined value for mutation. The probability has been kept to a minimum to ensure that the algorithm search is still close to the optimal solution but not too far so as to break out of that search. Figure 3.12 illustrates how the mutation was implemented in the python code.

```

def mutation (ind2, d):
    m1=random.randint(0, d-1)
    m2=random.randint(0, d-1)
    x = ind2.ind[m1]
    ind2.ind[m1] = ind2.ind[m2]
    ind2.ind[m2] = x
    return ind2

```

**Figure 3.12 showing the mutation function implemented**

### **Best fitness of population:**

The best fitness of the population returns the evaluation of the population after applying the genetic operators. The best fitness of the population ensures that the next iteration of the selection has a higher fitness value. This results in the quality of the individuals increasing to meet the optimal solution to the problem. The figure 3.13 shows how we have implemented the evaluation of the best fitness of the entire population in the program.

```
def bestfit(p):
    fit = -1
    for i in p:
        if i.fit > fit :
            fit = i.fit;
    return fit
```

**Figure 3.13 showing the best fitness of the entire population**

### Best Fitness of individual

The fitness value of each individual needs to be evaluated to make sure that each of the individuals in the new population have a better fitness value from the previous individuals in the last generation.

Figure 3.14 shows how each individual in the population is compared to find the those individuals with the best fitness value.

```
def bestind(p):
    best = 0
    fit = -1
    for i in p:
        if i.fit > fit :
            fit = i.fit;
            best = i
    return best.ind
```

**Figure 3.14 showing the evaluation of the individual with best fitness value**

### Termination

The algorithm terminates after the  $i$ th generation defined in the program or when it reaches the best fitness of an individual. The termination of the program implies the stopping criteria has been reached. This could be that the maximum iterations of generations has been reached or an optimal solution has been found for the problem. For example, if the defined number of generations is 500, program will terminate after the 500<sup>th</sup> generation of the population with best fitness.

### **3.4 Technologies implemented**

This section shall take the reader through the technologies that were used to complete the development of the implementation.

#### **3.4.1 Language**

The language selected for the development of the algorithm was Python. The author found that the syntax is easier to learn and the code is more readable as it helps to program at a faster rate. Python also has a range of extensible libraries that can be put to use to help develop the prototype genetic algorithm. It has wide support and runs on cross platform for Windows, Linux and Mac OS X. It is free and open source for anyone to use.

#### **3.4.2 Environments**

For development purpose JetBrains Pycharm was used. Pycharm is free for open source and for students that has suitable interface for development of the algorithm. This includes package management being easier to install to use as libraries for development purposes.

## Chapter 4.

### Evaluation

This chapter take the reader through the evaluation measures found in the related works and how it contrasts from our evaluation approach. This is followed by the analysis of the results obtained from the algorithm implemented in our approach.

#### 4.1 Evaluation Overview

To be able to evaluate the results of our implementation, it would be essential to understand the problem that we have to find the optimal solution for. The Quicksort time performance in the worst case is the problem was aimed to be the optimal solution. The individuals that are generated through the genetic algorithm generate the fitness of the worst-case scenario where the time complexity of the quick sort should be the worst case.

For example, to sort  $n$  number of items, the quicksort algorithm takes  $O(n \log n)$ . Our implementation generates individuals that give the worst-case time complexity where for  $n$  number of items, the quicksort algorithm should exhibit a time complexity of  $O(n^2)$ . In the view of software fuzzing, this worst case simulated by the implementation will indicate a vulnerability in the system under test. The genetic algorithm produces those individuals that are the tests cases that trigger the worst-case time complexity in the quick sort algorithm.

The experiments were conducted on the author's own workstation. All the results presented are the values obtained from several executions.



## 4.2 Measures for evaluation

To evaluate the efficiency and the effectiveness of the implemented genetic algorithm, a set of attributes must be defined to measure its performance.

From related works, it can be seen that the most common evaluation techniques focus on:

**1.Pass Rate:** Whether the test cases pass against the expected behavior of the target system under test.

**2.Vulnerabilities found:** The variety and number of vulnerabilities found by the fuzzing tool using the generated test cases. In the context of our research however, we will be focusing on the time taken to sort as a vulnerability.

**3.Code Coverage:** How much of the test case is covered during a fuzz campaign. The code that is tested through the use of the test case would be the code coverage of the test case.

**4.Validity:** If the test cases are valid but are mutated enough to cause unexpected behavior in the system under test.

For our implementation, we will be focusing on the analysis of the fitness of the entire population and the comparison of the individuals before and after the implementation of our approach.

#### 4.2.1 Fitness of the population:

Datasets used: During these experiments, the population size was kept at 100000 and the number of generations was kept at 500. During this, the mutation and crossover rates were kept at 0.2 and 0.8 respectively. With every iteration of the quicksort, the fitness shows an increase with the newer individuals being sorted being generated in the program. Figure 4.1 shows the dataset that was used to run this experiment.

Individual size	Population size	Generations	Crossover Rate	Mutation Rate
100000	20	500	0.8	0.2

**Figure 4.1 illustrating the dataset used.**

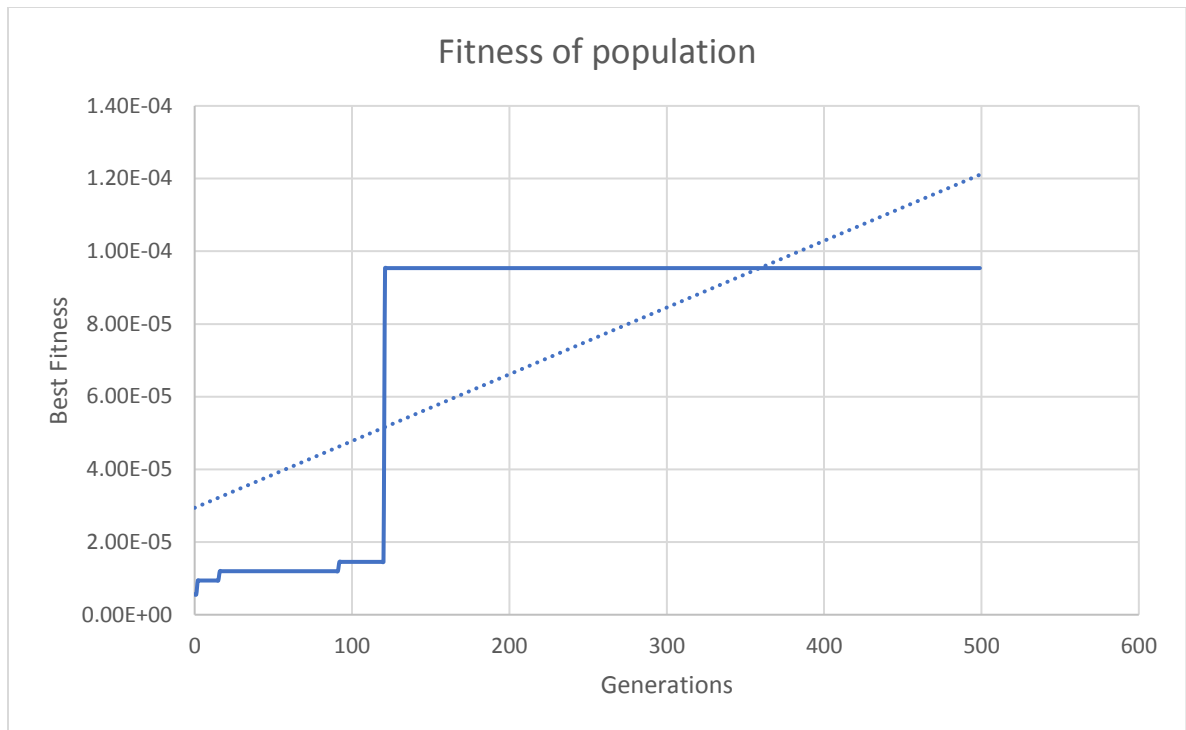
**The individual size:** The individual size for the purpose of this experiment was kept to 100000 random integers in each list.

**Population size:** The size of the population being sorted was set to be an array of 20 lists.

**Generation size:** The program was set to have 500 iterations where 500 new generations were produced during this experiment.

**Crossover rate:** From related works, the crossover rate was set to be at 0.8 to allow the search space for the optimal solution to be expanded.

**Mutation rate:** To have perform the mutation operation on the individuals, the mutation rate was set to be at 0.2.



**Figure 4.2 showing the best fitness over 500 generations**

#### **Observations:**

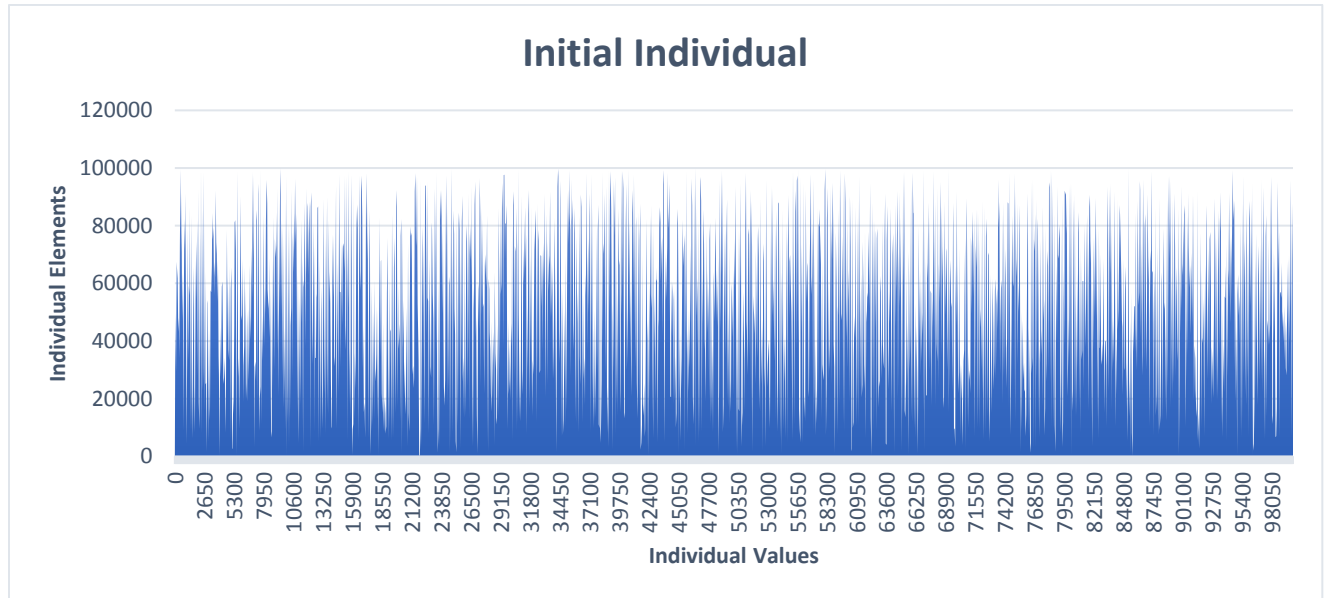
The figure 4.2 is the resulting graph from the fitness of each generation that is generated out from the genetic algorithm. The results show that the fitness of the population increases up until a point till the best fitness of the population has been reached. The increase is a linear where the time taken to sort increases till the best fit generation has been reached.

This should indicate that the newer individuals being generated in each population have better fitness values where the time taken to sort the individuals is longer than that of the previous generation. The search for the optimal solution remains at a plateau until the generation has been terminated.

It is worth mentioning that this is the optimal solution for the quick sort of upto 500 generations. If the number of generations were to increase, the subsequent optimal solution space would also increase.

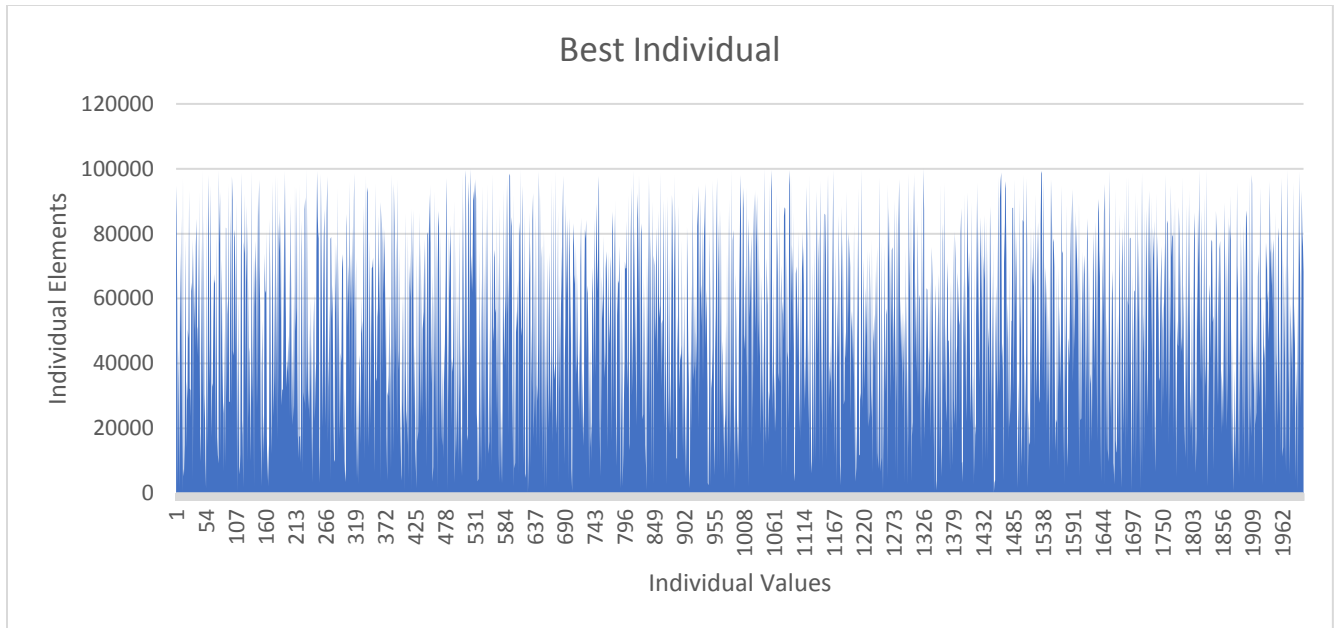
#### 4.2.2 Comparison of the initial individuals and the final individuals:

From the above dataset, the initial and final fittest individuals were compared to find the best individual from the first iteration and the last iteration of the program as well. The change in the shape of the data would indicate that the genetic algorithm searched for the best fitness of the individuals in the population and all subsequent generations were successful in creating the worst time complexity in the quicksort. Here, figure 4.3 is the initial individual that is going to be sorted before the genetic algorithm operators have been implemented.



**Figure 4.3 showing the initial individual array**

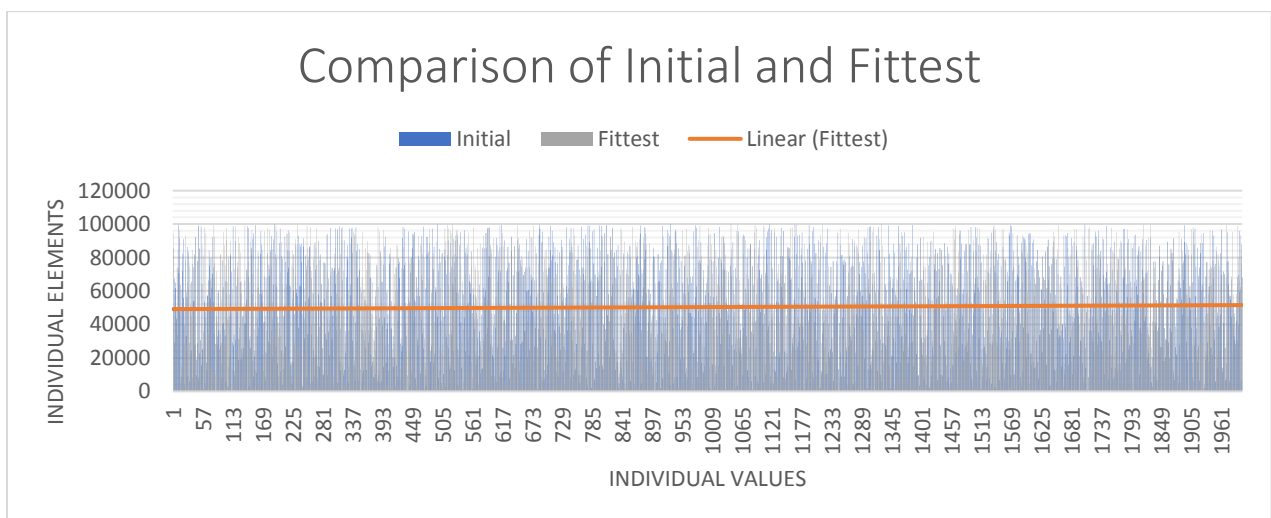
The fittest individual after application of the genetic operations is shown in figure 4.4.



**Figure 4.4 showing the fittest individual in the population after the genetic algorithm operations.**

The fittest individual is obtained after the maximum number of iterations through the genetic algorithm have been reached.

The figure 4.5 shows the comparison of both individuals from the initial to the last iteration of the genetic algorithm.



**Figure 4.5 showing the comparison of the initial and fittest individuals**

**Observations:**

The initial and the fitness individuals show a contrast in their values where the increase of the variation shows that the genetic algorithm slowly changes the values of the individuals after every iteration of the program. The trend line increases slowly as each individual value is mutated using the genetic operations.

**4.3 Discussion**

For the purpose of the experiments the best fitness of the total population increase as the number of generations is increased. Here, the best fitness of the entire population increases while fluctuates at certain points to reach the optimal solution. Since the genetic algorithm depends on the size and length of the resulting individual, in the context of the comparison between the initial and fittest individuals, the size and population individuals can be increased to a point. But due to the amount of memory being allocation for the operations on the author's workstation, the sizes of the population and individuals had to be decreased to meet the computing needs. The fitness of the population shows a linear increase where the after each iteration, the fitness value i.e. the time taken to sort the best fittest individual increases linearly over every iteration in each generation produced by the implemented program. The program reaches a plateau at where the best fitness of 500 generations has been reached. This indicates that the vulnerability here, is the time taken to sort where after 500 generations, gives the best individual that causes the sort to take a longer time to sort the best individual. The optimal solution is the best individual as a test case that gives the best fitness value. The best fitness value is the longest time taken to sort.

In comparison to the related works, **Sugihara [21]** defines three measures for performance of the genetic algorithm. If we apply these measures to our implementation, optimality of the genetic algorithm would be the probability of our implementation reaching the optimal solution given the number of runs to the number of generations. However, since the number of generations had been optimized for performance on the workstation, the probability that the implementation would have reached the optimal solution could have been higher given the sizes of the individual and the generations that could have been increased. This is made evident where **[21]** the performance of the genetic algorithm relies heavily on its parameters. The next measure that was used was the likelihood of an evolution leap. In relation to our implementation, there is a linear increase in the fitness of each successive generation over the other. The evolution leap toward the end of the execution stalls at a plateau but can be improved by increasing the number of individuals and size of the population.

## Chapter 5:

## Conclusions

### 5.1 Summary

To review the work that has been done in this dissertation, our approach successfully implements a genetic algorithm to produce test cases for a system under test. The system under test is the sorting algorithm where the worst case scenario is to be triggered by the generated test cases from the genetic algorithm. These are optimized to find the best fitness value in the quick sort. The best fit individuals are to invoke the time complexity of the worst case scenario of the quicksort.

To conclude, the research provided a simulation of a larger subset of software fuzzing. This is evident where the fitness of each of the populations generated increases with every iteration of the implementation. The performance of the genetic algorithm however depended highly on the parameters set for the execution of the implementation. From related works, we can infer that genetic algorithms may not always find the exact solution to the problem, rather the most favorable or optimal solution to the problem.

When complexity is higher, it may be resource expensive as seen when allocating memory to calculate the fittest individuals and population at every iteration. The implementation of the fitness function also plays an important role here where if the wrong choice of fitness value is returned, subsequent genetic operations are affected. This can occur where the selection of the fittest individuals would be of the wrong fitness value.



## 5.2 Limitations and Future work

Since one of the limitations to this research was the diversity of pivots used to generate the input, further work would be to test the genetic algorithm on how the pivot selection affects the individuals being generated using our approach. Our approach only addressed the middle pivot as the sorting algorithm as the test where the test cases were generated. This could be improved by extending the implementation to use all four pivots for a better evaluation of the performance of our approach.

For the evaluation of the performance of our approach, the dataset selected was initially planned to be of a larger scale. The individuals were planned to be of a much bigger size and the population size was to be increased as well. Due to the author's workstation being of lower RAM space, additional memory would have been required to run a bigger dataset of individuals and population sizes. Due to the low memory space and language used, since the memory was dynamically allocated during execution, this would have resulted in a better outcome of test cases to trigger the worst case in the quicksort which is under test.

The genetic algorithm also looks for an approximate of the best fitness solution. By randomly generating the population at every execution, this would mean that the search space for the solution would change with every iteration. The fitness values will also differ depending on what random value has spawned for the individuals in the population.

The final results from the analysis of the fittest population and the comparison of the initial individuals and the final individuals show that it is possible to implement a genetic algorithm to optimize the test case generation.

## References

1. R. Seacord , A. Householder, A Structured Approach to Classifying Security Vulnerabilities, Technical Report, CMU/SEI, January 2005
2. J. Seitz, Gray Hat Python: Python Programming for Hackers and Reverse Engineers. 2009.
3. M. Sutton, A. Greene, and P. Amini, Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley, 2007.
4. ZZUF internals, website: <http://caca.zoy.org/wiki/zzuf/internals> , Date accessed: 10 April 2018,
5. Peach Fuzzer, website: <http://community.peachfuzzer.com/Introduction.html> , Date accessed: 11 April 2018,
6. Synopsys, What is fuzzing: The poet, the courier, and the oracle, 2017,
7. GeeksforGeeks, website: <https://www.geeksforgeeks.org/quick-sort/> , date accessed: 12 April 2018,
8. R. Mohsen, Automated vulnerability analysis using advanced fuzzing: Generation based and evolutionary fuzzers," MSc in Information Security, Royal Holloway, University of London, 2010,
9. Developing New Fitness Functions in Genetic Programming for Classification With Unbalanced Data, 26 September 2011, Urvesh Bhowan ; Mark Johnston ; Mengjie Zhang
10. Holland, J. H. (1992), "Genetic algorithms", Scientific American July 1992:66-72
11. M. Mitchell, An introduction to Genetic Algorithm. MIT Press, 1996.
12. R. McNally, K. Yiu, D. Grove, and D. Gerhardy, Fuzzing: The state of the art, 2012.
13. A. Reber, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, Optimizing seed selection for fuzzing," in 23rd USENIX Security Symposium, 2014.

14. S. Goyal and A. Kaur, "A genetic algorithm for regression test case prioritization using code coverage," International Journal on Computer Science and Engineering (IJCSE)
15. P. Godefroid, H. Peleg, and R. Singh, "Learn and fuzz: Machine learning for input fuzzing," Microsoft Research, The Technion, Tech. Rep.
16. R. P. Pargas, M. J. Harrold, and R. R. Peck, Test data generation using genetic algorithms," Journal of software testing, verification and reliability,
17. H. C. Kim, Y. H. Choi, and D. H. Lee, Efficient file fuzz testing using automated analysis of binary file format monitoring test execution," Journal of Systems Architecture,
18. A. P. Katerina Goseva-Popstojanova, "On the capability of static code analysis to detect security vulnerabilities," Elsevier Information and Software Technology,
19. D. M. Patrice Godefroid Michael Y. Levin, "Automated whitebox fuzz testing," 2018.
20. V. Iozzo, "0-knowledge fuzzing," 2010.
21. K. Sugihara, "Measures for Performance Evaluation of Genetic Algorithms", 1997