USING EVOLUTIONARY ALGORITHMS TO IDENTIFY PROBLEMATIC PARAMETER SETTINGS IN SOFTWARE CONFIGURATIONS

BY

SEBASTIÁN RAMÍREZ RODRÍGUEZ

A Thesis Submitted to the Graduate Faculty of

WAKE FOREST UNIVERSITY GRADUATE SCHOOL OF ARTS AND SCIENCES

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Computer Science

May 2017

Winston-Salem, North Carolina

Approved By:

Errin W. Fulp, Ph.D., Advisor

Daniel A. Cañas, Ph.D., Chair

H. Donald Gage, Ph.D.

Acknowledgments

I would like to thank my advisor, Dr. Errin Fulp, for the continuous guidance and support he has given me throughout my research and during the process of writing this thesis. I would also like to express my gratitude to Dr. Don Gage for his insight, helpful comments, advice and occasional coffee runs. I would like to thank Dr. Daniel Cañas, not only for accepting to be part of my thesis committee, but also for being my family away from home, for his support, and for his continuous effort to provide Costa Rican students with opportunities such as this one.

I would like to thank the entire staff and faculty in the Computer Science department at Wake Forest University for creating a welcoming environment suitable for personal and professional development.

Finally I would like to express my profound gratitude to my family, specially my parents, for providing me with constant support and encouragement throughout all my years of study. This accomplishment would not have been possible without them.

Table of Contents

List of Algorithms

List of Figures

Abstract

Sebastián Ramírez Rodríguez

As software systems become more complex and configurable, failures due to misconfigurations are becoming more common. Many cyber attacks can be attributed to administrators who, unaware of insecure settings or novel attacks, expose vulnerabilities in their systems. The difficulty of diagnosing and fixing misconfigurations is primarily due to the large number of possible configuration parameter settings and the potential existence of interdependencies between them.

This thesis introduces a method for identifying security related configuration parameters by analyzing vulnerable and secure configurations obtained from a Genetic Algorithm. Given the large number of parameters in a system and the various settings to which they can be configured, a Genetic Algorithm is leveraged as a tool for exploring this search space in pursuit of diverse solutions while the correct functionality of the system is preserved. By keeping record of each setting's presence or absence among both secure and insecure configurations, it is possible to narrow down the set of parameters responsible for an exposed vulnerability; Furthermore, when combined with distinct scoring systems for the Genetic Algorithm, experimental results show that this technique is successful at identifying parameter interdependencies.

Chapter 1: Introduction

Most cyber attacks start with a reconnaissance phase (Figure 1.1) which involves an attacker gathering as much information from a system in order to identify potential vulnerabilities. Useful data includes recognizing which operating systems and services are running on the machines and how are they configured. According to a recent report by IBM, the human factor was responsible for contributing to security vulnerabilities in approximately 80% of the reported cases during 2013 [5]; 42% of these cases were related to a misconfigured system or application. For example, the `ServerTokens` setting located in the `httpd.conf` configuration file of an Apache-2.2 installation is associated to a common misconfigutation. This option defines how much information, regarding its environment, should the server display when it responds to its clients; configuring it to `FULL` provides them with information about the current running version, modules, and operating system, which can be used to determine potential weaknesses. Misconfigurations do not always introduce security related problems, they can also affect others aspects of a system, such as its performance. Following the Apache-2.2 installation example, the `KeepAliveTimeout` indicates the number of seconds the server should wait for a subsequent request before closing a connection; setting it to a high value causes performances problems when there is a large number of incoming connections as the server will wait for each one of them even if no further requests are intended [26].

Identifying a vulnerable misconfiguration in a target gives an attacker the opportunity to develop an exploit to take advantage of it and achieve their goals. In 2015, Alliance Health, a technology company that operates condition-specific social communities, was notified by a group of security researchers about a misconfiguration in

their database installation that granted access to their more than 1.5 million user's private health information [9].

Installation of new applications, upgrades, or customization of software settings are examples of changes that may inadvertently generate misconfigurations that can produce poor performance or introduce security weaknesses in a system [41]. In most cases misconfiguration failures and vulnerabilities are diagnosed by human experts such as system administrators. Once diagnosed they might proceed to perform tasks such as removing programs which are not required for the expected operation of the system, monitoring each of the running services, and updating system configurations in order to resolve any found issue. This process requires an up-to-date knowledge of disclosed vulnerabilities and constant tweaking of systems, making experts difficult to find and thus more expensive to hire [41][22].

1.1 Software Configurations

A computer's software configuration is a group of settings that dictate how a machine operates. Each individual setting belongs to either the operating system, such as file permissions or user accounts; or to installed applications, for example the `ServerSignature` or `ServerLimit` settings in an Apache server. Configurations give

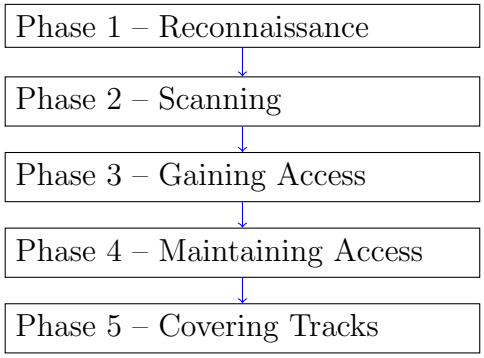

Figure 1.1: Phases of a cyber attack according to CEHv9 [3]

developers the opportunity to provide more general applications with added customization options. This gives users freedom to tailor the software's behavior to their specific needs. User mistakes, execution errors, software patches or data corruption can sometimes originate a misconfiguration in a computer. Depending on the case, a system can become unstable, unreliable or completely unavailable due to a misconfiguration [29]. For this reason it is important to be able to identify and troubleshoot these errors.

System configurations can consist of thousands of parameters. An out-of-the-box installation of Windows with no additional programs installed contains more than 200,000 entries in its configuration database (registry) [22]. Adding a MySQL database server and an Apache HTTP server contributes more than 200 new entries [45]. Some of this parameters can also be interdependent and form a *parameter chain*. For example, in an Apache-2.2 server, a high `KeepAliveTimeout` value and low `ServerLimit` could cause the server to be easily vulnerable to a Denial of Service attacks. With the low `ServerLimit`, the server can quickly spawn the maximum number of child processes, and the high `KeepAliveTimeout` value will keep idle connections open such that other connections cannot be opened.

Chains might expose a series of low-impact vulnerabilities that when combined expose a higher impact one [6]. These dependencies add a greater difficulty when identifying a misconfiguration.

1.2 Configuration Management

The search space of all possible combinations of software configuration parameters is immense and thus difficult to explore. The problem of discovering secure software configurations can be encoded as a Boolean Satisfiability (SAT) problem [30], which is a proven NP-Complete problem [34], by considering every individual setting as a

Boolean variable and the resulting combination as an expression where a *satisfiable* configuration is considered a secure one.

Previous research in the area has attempted to provide a solution to configuration management as a SAT problem. The main disadvantage of this process is that even simple configurations can result in Boolean expressions that are too large for typical *SAT-solvers*. Furthermore, SAT-solvers assume all variables involved are security related, and they will change them in search for a single solution; however, some of the variables may not be relevant to security but necessary for functionality, and multiple solutions may exist and could be advantageous to know.

Other approaches rely on a misconfiguration database. Some examples of this path include Strider [40] and PeerPressure [39]. Both programs diagnose problems by comparing a system against well-configured peers. While Strider requires manually labeled configuration cases, PeerPressure, its successor, automates the labeling using statistical methods. AutoBash [36] replaces the usual UNIX shell with one that keeps track of user typed commands, adding the ability to predict and resolve errors based on what other users from their community have done before.

Identifying and troubleshooting parameter chains is a more complicated task. EnCore [45] attempts to identify the correlation between parameters. It does this by enriching configuration values with information from their execution environment, i.e. values receive information regarding the file where they are contained, its owner, permissions, location, and others. Based on configuration rules and correlations the tool is able to identify errors more accurately than the tools mentioned before. However, it still relies on a training set of data required to understand the underlying rules.

An alternative approach for configuration management is to search for correct configurations within the space of all possible configurations. The complexity of the search space (i.e. number of configurations) and the need to identify multiple solu-

tions limits the usefulness of traditional search algorithms. Genetic algorithms (GAs) provide an efficient way to examine large search spaces. Prior research used GAs to fix misconfiguration errors [14][27][35][7] for both individual vulnerabilities and parameter chain related ones, while managing the entire configuration and without previous knowledge regarding security parameter. GAs are able to find secure configurations by exploring this search space; however they need an extension to be able to identify which parameters are causing vulnerabilities. This thesis pursues the idea of using GAs as secure software configuration generators while keeping track of the effects of different settings in order to identify vulnerable parameters.

1.2.1 Additional Configuration Management Objectives

Configuration management is not only done to resolve configuration errors, but can also be done to improve performance and provide additional security. There has been an increasing interest in how correctly setting up configuration parameters improve performance. Tuning configuration settings is usually a simple task, most applications store their values in XML or plain text files. It requires little extra cost compared with investing in new hardware. Tests show that optimal parameters can increase a distributed system's throughput by 24.5% and reduce its average response time by 28.1% compared with the default settings [44]. Other research has focused on improving performance of network services in an attempt to better utilize hardware resources and ensure the best possible service [4][43][46].

Moving Target Defense adds an extra motivation to correct software configuration management. The concept of Moving Target Defense (MTD) strategies has been introduced in cyber security as a way to add dynamism to defenders [11][12]. Relying solely on static defense methods, such as firewalls and signature based intrusion detection, gives attackers the freedom to probe a system at will and construct an attack

based on the obtained information. The main goal of MTD strategies is to keep the system in movement. This will increase uncertainty and complexity for the attacker by turning their previous knowledge of the system obsolete. It also provides the defender with resiliency through the ability to automatically recover from a successful attack by moving from its current vulnerable state to a secure one. Software configurations can be used as a MTD technique. Keeping configurations in movement by periodically changing them can nullify the reconnaissance of the attacker and/or provide the illusion that the system is actively managed (a system characteristic attackers may avoid).

A key feature of MTD strategies is the need to preserve essential properties required to maintain a correct functionality while disrupting malicious attempts. A widely used example present in most modern operating systems is address space layout randomization (ASLR). This technique is capable of disrupting exploits that depend on known absolute addresses for objects in memory by randomizing the locations of those objects, fortifying systems against buffer overflow attacks [33].

1.3 Contributions

This thesis presents an approach for identifying software misconfigurations which have been associated with cyber attacks. It also provides a scalable and resilient configuration management tool that is able to generate new secure configurations while maintaining the expected functionality of a system. Furthermore, the introduced approach is capable of correcting vulnerabilities caused by interdependent parameter settings successfully while identifying the components which have been involved in them.

1.4 Outline

The remainder of this thesis is structured as follows. Chapter 2 presents an in-depth explanation of configuration parameters, parameter interdependency, and the terminology and definitions that are going to be used in the following chapters. Chapter 3 introduces the basic concepts of Genetic Algorithms and their components. Chapter 4 explains how a Genetic Algorithm was leveraged to generated vulnerable and secure software configurations. In Chapter 5, an algorithm that uses the mentioned generated configurations for identifying vulnerable parameters is introduced. Chapter 6 demonstrates the effectiveness of the aforementioned algorithm by showing results obtained from experimental simulations. Chapter 7 reviews the results obtained from the experimental simulations and compares them to the expected contributions of the thesis. Finally, in Chapter 8, areas of future work are discussed.

Chapter 2: Software Configuration Models and Assumptions

Computer software configurations allow a single application to be more customizable. For example, modern web browsers allow their users to set which URL should be accessed when initialized (home page). If certain configuration parameters are set up incorrectly the applications may not behave as expected. For example, misspelling the URL of the new home page will result in the browser opening a different page from the expected one or no page at all. The complexity of applications, their large number of configuration parameters and the possible existing interdependencies between them, make identifying misconfigurations a challenge. This work approaches the problem of configuration management as search problem, focusing primarily on those parameters that will effect security, not functionality or performance.

2.1 Configuration Parameters

A software configuration consists of a group of parameters that are set to achieve a certain objective (software functionality, performance, and/or security). Each parameter has a type which determines its different potential settings. For example, a review of the Red Hat 5 and Apache-2.2 configuration settings revealed that, generally, parameters are one of three types: bit vectors (e.g., file permissions), numerical ranges (e.g., the `ServerAliveInterval` found in the `ssh_config` file in Apache), and discrete sets (e.g., the allowed HTTP methods on a server can be `GET`, `POST`, `OPTIONS`, `PUT`, etc). Each of the possible individual choices for a parameter is referred to as a setting. Although parameters will generally effect the performance of an application, some may also effect the security. Settings are considered *vulnerable* if they expose

a weakness which can be exploited by an attacker to compromise the security of a system.

2.1.1 Parameter Interdependency

Two or more settings are considered interdependent if they only effect the software when they are together in a configuration. This thesis considers parameter interdependency specifically in terms of their effect in the security of a configuration, i.e. two or more settings which might have been considered as secure exposed a vulnerability when configured together. This relationship is also known as a *parameter chain*.

Assume a configuration that consists of two parameters P_1 and P_2 of type "discrete set". Each parameter can be configured with one setting from the set A,B. If setting both parameters to a certain value causes a security issue, one could say that they form an *AND* chain $P_1 \wedge P_2$ (named after the Boolean AND operation). An example can be seen in Figure 2.1a where no individual setting causes the system to become vulnerable but a combination of B in both parameters does. If identified correctly, this type of chains can be fixed by changing either P_1 or P_2 to any other combination of values, e.g. in the presented example changing any parameter to A should be enough.

On the other hand, an OR chain (as in the OR logical operator) can be considered an "amplifier" where individually the chain components are security issues, but together they either increase the risk or introduce yet another vulnerability. Figure 2.1b shows the behavior of an example OR chain. Individually both parameters yield vulnerabilities, thus making it easier for an attacker to compromise a system. A single misconfigured parameter would be enough for an exploit to be developed. From the point of view of a defender, fixing an OR chain implies avoiding the vulnerable settings in every individual parameter that comprises the chain.

P_1	P_2	Vulnerable
A	A	No
A	B	No
B	A	No
B	B	Yes

(a) Parameter AND chain

P_1	P_2	Vulnerable
A	A	No
A	B	Yes
B	A	Yes
B	B	Yes

(b) Parameter OR chain

Figure 2.1: Parameter chains example

Combinations of parameter chain types can exist in software configurations. For example, assume that parameters P_1, P_2, P_3, and P_4 contain vulnerable settings and they form two different OR chains, $(P_1 \wedge P_2)$ and $(P_3 \wedge P_4)$, associated with two distinct weaknesses. Having all these parameters in the same configuration creates the chain $(P_1 \wedge P_2) \vee (P_3 \wedge P_4)$.

2.2 Terminology and Definitions

Based on the description of configurations, parameters, and settings described in the previous section, this section will provide formal definitions and important terminology for the remainder of this thesis.

Definition 1. A software *configuration* (C) dictates how a machine operates. It consists of n parameters, such that $C = \{P_0, P_1, P_2, ..., P_{n-1}\}$.

Definition 2. A *setting* (s_i) is the specific value to which a parameter P_i is configured. The set $S_i = \{s_i^0, s_i^1, ..., s_i^{k-1}\}$ represents every possible value that can be assigned to P_i. In a configuration, $|S_i|$ is not necessarily equal for every P_i.

According to their relationship to exploits, settings can be classified as follows.

Definition 3. A *vulnerable setting* or *insecure setting* is a setting that has been associated to an exploit.

Definition 4. A *secure setting* is a setting that does not expose the system to any weaknesses.

Based on these definitions, parameters can be classified as follows.

Definition 5. A *vulnerable parameter* is a parameter P_i whose settings must be either secure or insecure. In addition, these settings should have at least one secure and at least one insecure setting.

A single vulnerable setting is considered since, even when multiple settings may be responsible for an exploit (consider a numeric range where a small difference may still cause a vulnerability), it is possible to combine them and consider them as one. Furthermore, since the main objective of the algorithm is to be able to resolve the misconfiguration, it is also reasonable to assume the existence of at least one secure setting in S_i.

Definition 6. A *passive parameter* is a parameter that contains no vulnerable or secure settings.

It is reasonable to assume that these are the only two possible classifications since it would be contradictory to have a setting that causes a parameter to both yield an exploit and solve one. It is also reasonable to assume that

Definition 7. A *parameter chain* is a set of two or more security parameters that produce a vulnerability when certain combination of their settings is present in a configuration.

Definition 8. A configuration can be categorized either as a *bad configuration* or a *good configuration* depending on if they contain a vulnerable parameter or not, respectively.

Chapter 3: Evolutionary Algorithms

Evolutionary Algorithms (EAs) is an interdisciplinary research field with relationship to biology, artificial intelligence, numerical optimization, and decision support in almost any engineering discipline [1]. The original idea behind them was introduced by Holland [18] and it was inspired by how evolution works in nature. Genetic Algorithms (GAs) are one of the most interesting and popular developments of Evolutionary Algorithms. Evolutionary Strategies, Evolutionary Programming, Artificial Life, Genetic Programming, and Evolvable Hardware are other examples developed within the subject [32]. This study focuses on GAs and how their application in software configuration management can help identifying vulnerable settings while generating new secure ones.

3.1 Genetic Algorithms

Genetic Algorithms (GAs) are a search and optimization heuristic inspired by the biological process of natural selection. The basic structure required when modeling a GA is a *chromosome*. Candidate solutions to the requested optimization problem are encoded as chromosomes. Every chromosome is composed of *genes* and the set of values that can be assumed by the latter is called *alleles*; following the biological terminology. The specific location of a gene within a chromosome is called *locus*. This search approach actively manages a group of possible solutions (chromosomes) that is also called a *population* or a *pool*. Over multiple generations (successive iterations of the GA), the pool is improved by hopefully highlighting, promoting, and altering existing good solutions.

Since evolving better solutions from existing good solutions is an objective, a

fitness measure is required when modeling a problem as a GA. It is a type of objective function, as the one found in optimization problems, that assigns a score based on how well a chromosome performs as a solution to the problem [25]. Some NP-Complete problems have been proven to have a better GA representation than others [21]. For example the Boolean Satisfiability Problem can be easily modeled using a GA; thus potential *good* solutions can be readily discovered.

Algorithm 1 presents a very general implementation of a GA. It is composed by the genetic operations of *selection*, *crossover*, and *mutation* which are going to be covered in the following sections. Every generation attempts to surpass its predecessors based on fitness and the application of the fundamental GA operations. Therefore, led by evolution, they should generally take steps towards an optimal form.

Algorithm 1: Genetic Algorithm Template

```
1  create an initialPopulation of n chromosomes;
2  currentGeneration ← initialPopulation;
3  while termination condition not satisfied do
       /* Start with an empty next generation                    */
4      nextGeneration ← {};
5      for i ← 1 to n do
           /* Select two parents from the current population     */
6          parentA ← Select(currentGeneration);
7          parentB ← Select(currentGeneration);
           /* Apply the crossover operator                       */
8          offspringA, offspringB ← Crossover(parentA, parentB);
           /* Apply the mutation operator to each child          */
9          offspringA ← Mutation(offspringA);
10         offspringB ← Mutation(offspringB);
           /* Add the generated offspring to the next generation */
11         add offspringA, offspringB to nextGeneration;
12     end
       /* Set the next generation as the current one             */
13     currentGeneration ← nextGeneration;
14 end
```

3.1.1 Initial Population

The first step into a functioning GA is the creation of an initial population (see Algorithm 1 - Line 1). Each member of the population encodes a plausible answer to the problem. Choosing a large enough population size increases the possibility of identifying a chromosome containing the optimal solution in the initial state [16][15]. It can also reduce the number of generations required for a GA to *converge*, by giving it a broader overview of the search space of the problem from the beginning. However, more chromosomes per generation implies more computational power and convergence does not always mean that the optimal solution has been found.

Choosing the members of the initial population can be difficult. The most common technique of doing it implies randomly generating each chromosome. The resulting population might not cover the search space uniformly, which can make it harder for the algorithm to find a good result. It has been reported that seeding the initial generation with good or partial solutions obtained from other heuristic techniques helps the GA with finding solutions faster [32]. This method may not always be effective since there is a risk of inducing convergence to a poor solution (local minima or maxima) [23]. If there are no previous results, purposefully increasing the diversity between chromosomes is another way of obtaining a greater probability of finding solutions [8].

3.1.2 Diversity

Holland considers the very essence of good GA design to be the retention of *diversity* [19]. The maintenance of a diverse population is required to ensure that the solution space is adequately searched. A very homogeneous population, i.e. little diversity, is considered as one of the major reasons for convergence to occur on local minima or maxima (premature convergence). Premature convergence is usually attributed to a

homogeneous population where the GA is unable to evolve solutions that can outperform its parents. Genetic operations, as the ones introduced in the following sections, can both increase and decrease diversity depending on how they are implemented. The measure of diversity within a population can be calculated by evaluating the distance between chromosomes, i.e. using the Hamming distance [17].

3.1.3 Selection

The process of *selection* (Algorithm 1 – Lines 6-7) chooses a parent from the current generation based on fitness. Mimicking how natural selection tries to ensure survival of the fittest. The original method used for selection is called *fitness proportionate selection* or *roulette wheel selection* (RWS) [32][1].

Roulette Wheel Selection

Roulette Wheel Selection was developed by Holland [18] in an attempt to deal with the trade-off between exploring promising regions of the search space and new regions at the same time [1]. In it, every chromosome has a chance of being selected; However, a higher fitness value provides a higher probability of being chosen [25]. The probability of selection for each individual is proportional to the area of a sector of a roulette wheel [32]. Using the example fitness values from Table 3.1, the probability (p) of selecting each of the n chromosomes is calculated using the equation:

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j} \tag{3.1}$$

Where f_i represents the fitness evaluation for chromosome i and n the total number of chromosomes. Figure 3.1 shows an example of a "roulette wheel" created using the calculated proportions from Table 3.1; using this representation, one could select

Chromosome	Fitness	Probability
1	50	15.6%
2	30	9.4%
3	60	18.8%
4	80	31.3%
5	100	25%

Table 3.1: Selection fitness and probability values example

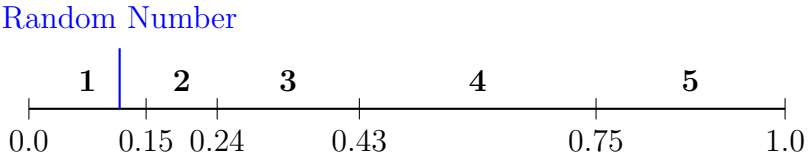

Figure 3.1: Roulette Wheel Selection example

a parent from the pool of 5 chromosomes by generating a pseudo-random number between 0 and 1. According to the location of the obtained value in the graph, a parent would be selected. For example, if the generated value is 0.12, then the selected chromosome would be number 1. This technique is done with replacement, which means that the same chromosome can be selected more than once as a parent. A policy for deciding what to do in case the algorithm inadvertently tries to mate a chromosome with itself can be implemented. This will depend on the problem being solved, but a possible solution includes implementing a "no duplicates" clause which forces both parents to be distinct.

Selection Pressure and Stochastic Universal Sampling

The *selection pressure* is the degree to which the better individuals are favored [24]. The higher the selection pressure the more the algorithm is biased towards fitter individuals. RWS has a high selection pressure, as seen in Figure 3.1, chromosome 4 has the highest probability of being chosen, which is also related to its fitness. For this reason, other strategies have been presented while exploring ways to mitigate bias and

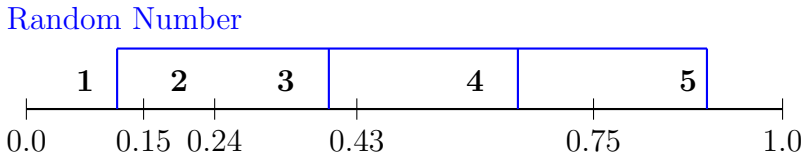

Figure 3.2: Stochastic Universal Selection example

introduce higher diversity in next generations. Stochastic Universal Selection (SUS) [2] proposes the usage of the same proportion-based roulette wheel used in RWS but instead of selecting a single parent every time, all of the required for evolving the next generation are selected at once. This is done by placing multiple equidistant marks, one per required parent, in the roulette. Starting from where the random number "landed". The distance between each mark is determined by the number of elements to be selected. This process is also done with replacement, multiple marks may land inside the area of the same chromosome but the algorithm ensures that not all of them will. Figure 3.2 uses the data from Table 3.1 to show how the selection of 4 parents would look like if the obtained random number is 0.12. In this case, the selected chromosomes would be numbers 1, 3, 4 and 5.

Tournament Selection

While some selection strategies focus on maintaining diversity, others attempt to increase the selection pressure. A higher selection pressure results in a higher convergence rate of a GA [24]. However, it doesn't ensure convergence towards an optimal solution. Tournament selection provides a high selection pressure by holding a tournament among δ competitors, with δ being referred to as the tournament size. The winner of the tournament is the chromosome with the highest fitness among all competitors. Only tournament winners will proceed to be selected as parents, increasing the overall average fitness of parents compared to the average fitness of the whole population. The GA will then be forced to attempt to improve fitness in each suc-

ceeding generation. In order to increase selection pressure even further, one could increase the tournament size. The winner of bigger tournaments will, on average, have a higher fitness than the winner of smaller ones. There is also the option to organize the tournaments with replacement, allowing individuals to participate in more than one tournament, or without replacement, either by disallowing all of them from participating again or only disallowing winners. In this case, performing the selection without replacement will maintain an extra level of diversity.

Steady State Behavior and Elitism

Steady state behavior is the preservation of the highest scoring chromosome(s) from the current generation into the next. This guarantees the existence of good solutions in the upcoming population, creating high bias towards them and leading to a high selection pressure. Tournament selection encourages a steady state behavior, but due to the possible number of participant chromosomes, it is not be enforced.

Elitism enforces a steady state behavior by selecting a constant number of the highest scoring chromosomes inside the population. Some methods solely choose chromosomes tied in the single highest score. Others, such as *Truncation selection*, allow only chromosomes in a user-defined top percentage of the population, in terms of fitness, to reproduce [32].

3.1.4 Crossover

After selecting both parents, the *crossover* operation (Algorithm 1 – Line 8) mimics the process of sexual recombination found in nature. Crossover is regarded as the driving force of a GA. It is the process where genes of two parents are exchanged to produce their offspring [37]. This introduces more diversity into the population. Assume the existence of two chromosomes A and B, where $A = (a_1, a_2, a_3, a_4, a_5)$ and

$B = (b_1, b_2, b_3, b_4, b_5)$, both represent possible solutions to a problem. A *single-point crossover* or *simple crossover* randomly selects a *crossover point* and proceeds to exchange each gene from a parent with its homologous locus. For instance, in the presented example, if the selected crossover point is 3 the resulting children from A and B would be $(a_1, a_2, a_3, b_4, b_5)$ and $(b_1, b_2, b_3, a_4, a_5)$. Depending on implementation, either one or both offspring will be added to the next generation.

Two types of biases have been identified in crossover methods: positional bias and distributional bias [10]. Single-point crossover has a high positional bias, in that it is more likely for interacting genes found farther apart in the chromosome string to be disrupted than genes that are relatively close together. The distributional bias depends on how is the crossover point being selected. If there is a specific point x in every string which is more likely to be selected, then one could say that the algorithm possesses a strong distributional bias. Simple crossover is unbiased in these terms since the crossover point is selected randomly using a uniform distribution. The lack of distributional bias does not necessarily mean a good thing for single-point crossover. This has lead to the introduction of other crossover strategies, e.g. *two-point crossover*, *multi-point crossover*, *shuffle crossover*, and *uniform crossover*.

Uniform Crossover

Uniform crossover is a simple alternative capable of removing any bias from the operation. The idea consists of generating a random binary string of length n, where n equals the length of the parent chromosomes. This string is then used as a *mask* where each value represents if a crossover must take place (1) or not (0) at a specific position. For instance, if the two chromosomes A and B introduced before, i.e. $A = (a_1, a_2, a_3, a_4, a_5)$ and $B = (b_1, b_2, b_3, b_4, b_5)$, went through the process of uniform crossover using the randomly generated mask 11001, their resulting offspring would

be $(a_1, a_2, b_3, b_4, a_5)$ and $(b_1, b_2, a_3, a_4, b_5)$.

N-Point Crossover

Increasing the number of crossover points reduces the positional bias [10]. Two-point crossover takes a step towards this idea by treating each chromosome string as a ring and selecting two unique points which split the ring into two slices. Slices are later exchanged between parents to produce their children. Two-point crossover introduces no change in terms of distributional bias when compared to simple crossover. It does, however, reduce the positional bias.

The number of crossover points can be increased further, reducing the positional bias and, unlike the move from single-point to two-point, introducing some distributional bias. Multi-point crossover maintains the idea of representing the chromosome strings as rings which the n crossover points cut into slices. The number of crossover points (n) needs to be even in order to create an even number of slices. This requirement allows the offspring to alternate slices of both parents. The increment in distributional bias comes from the change in the amount of slices children will receive from each parent. As the number of crossover points increases, the distribution of slices approaches a binomial distribution instead of the uniform found before [10].

Shuffle Crossover

The concept of shuffle crossover mitigates positional bias by randomly shuffling the chromosome strings of the parent in tandem before crossing them. After the exchange, strings are unshuffled, returning genes to their original positions [10]. This idea can be combined with the strategies of single-point, two-point, and multi-point crossover to eliminate positional bias. However, their original distributional bias would not be affected.

3.1.5 Mutation

The concept of *mutation* (Algorithm 1 – Lines 9-10) is simpler than crossover. One or a set of genes are "mutated" by changing their current value to another one from their respective alleles. The most common strategy for selecting these new values is by giving every alternative in the alleles the same probability of being selected; however, some approaches propose biasing these probabilities depending on how likely is it to find certain options or how related they are to the one being replaced (if this measure exists). Mutation allows diversity to be preserved and, by introducing chromosome strings that might have not been explored before, it can allow a GA to escape from a sub-optimal solution.

The *mutation rate* defines the probability of a given locus in a chromosome to be mutated. This probability is user defined in most cases and it is usually set to low. If it is set to a high value, every gene would be mutated, turning the GA into a random search. Some variations of the mutation process suggest some type of adaptive rate. For example, assigning different rates to different loci [13] or varying the probability depending on a diversity measure of the population [31][32].

3.1.6 Terminating

There is no single answer as to when a GA should stop evolving new generations. If no termination criterion is defined the algorithm could easily run forever (Algorithm 1 – Line 3). The most common condition used is limiting the number of generations to be created, this forces a problem independent stopping point. Other approaches include measuring the diversity of the population and stopping when it has dropped below a defined threshold or setting time restrictions as to the maximum amount of clock cycles the execution should take.

3.2 Simple Genetic Algorithms

A Simple Genetic Algorithm (SGA), originally developed by John Holland [18], is a very basic GA but it is subject to very tight constraints. Their chromosomes are limited exclusively to binary values, requiring solutions to be modeled as fixed length bit strings. The same genetic operations of crossover and mutation apply, but they are performed as bit-wise operations. Due to this limitations, SGAs are not commonly used in real applications but they do provide an easy to understand version of the algorithm for theoretical analysis. For example, properties such as the Fundamental Theorem of Genetic Algorithms (Section 3.2.2) have been originally developed for SGAs but have later been proven true for more general GAs.

3.2.1 Genetic Operators and Time Complexity Analysis

The biggest constraint regarding SGAs is chromosome modeling. Each chromosome has to be represented as a fixed length string of l bits, limiting genes to binary values. The concepts of a fitness function and the genetic operations of selection, crossover and mutation are applied in order to evolve upcoming generations just as in general GAs. The template presented in Algorithm 1 can be used for developing an SGA by writing the correct `Select`, `Crossover` and `Mutation` functions. This section introduces how an SGA can be developed while computing the time complexity of each step when evolving a new generation.

The fitness proportionate selection method, as introduced in Section 3.1.3, is used for choosing parents. An example of the `Select` (Algorithm 1 – Lines 6-7) function for an SGA is presented in Algorithm 2. In terms of complexity, one could start the analysis from the calculation of the sum of the total fitness of all n chromosomes (Algorithm 2 – Line 2). This step takes $O(n)$ time. The algorithm later proceeds to iterate through the n chromosomes moving the value of the random number around

Algorithm 2: Example fitness proportionate selection for SGA

```
1  Function Select (currentGeneration)
2      totalFitness ← sum of all Fitness values of currentGeneration;
3      len ← chromosome string length;
       /* Generate a random number between 0 and totalFitness        */
4      randomNum ← Random(0, totalFitness);
5      for i ← 0 to len do
6          if randomNum < Fitness(currentGeneration[i]) then
               /* If the random number is located in current
                  chromosome's area                                   */
7              return currentGeneration[i];
8          else
               /* If not, move to the next chromosome's area          */
9              randomNum ← (randomNum − Fitness(currentGeneration[i]));
10         end
11     end
12 end
```

the distinct areas of the created roulette wheel. This process also takes $O(n)$ time. Thus, the presented `Select` function is $O(n)$.

Algorithm 3: Example crossover function for SGA

```
1  Function Crossover (parentA, parentB)
2      p ← Random(0.0, 1.0);
3      len ← chromosome string length;
4      if p < p_c then
           /* Select a crossover point                                  */
5          crossoverPoint ← Random(1, len – 1);
           /* Yield the offspring with part of both parents             */
6          childA ← parentA[0, crossoverPoint] + parentB[crossoverPoint, len];
7          childB ← parentB[0, crossoverPoint] + parentA[crossoverPoint, len];
8      else
           /* If no recombination happened, both parents are copied
              into the next generation                                  */
9          childA ← parentA;
10         childB ← parentB;
11     end
12 end
```

The `Crossover` function (Algorithm 1 – Line 8) is implemented using a single point crossover strategy, as seen in Algorithm 3. The crossover probability, usually denoted as p_c (Algorithm 8 – Line 4), indicates how likely it is for parents to go through the recombination phase. `Crossover` exchanges at most l bits, where l is the length of the chromosome strings, thus its complexity is $O(nl)$.

Algorithm 4: Example mutation function for SGA

```
1  Function Mutation (chromosome)
2      len ← chromosome string length;
3      for i ← 0 to len do
4          p ← Random(0.0, 1.0);
5          if p < p_m then
               /* Invert the value of the current gene                  */
6              chromosome[i] ← !(chromosome[i]);
7          end
8      end
9  end
```

Since genes are restricted to binary values, the `Mutation` function (Algorithm 1 – Lines 9-10) in an SGA is relatively simple, as seen in Algorithm 4. A user defined mutation rate, denoted as p_m (Algorithm 4 – Line 5), is used to define if a bit needs to be mutated. Going through each of the n loci of the chromosome has a time complexity of $O(n)$. Mutating a specific binary gene implies inverting its value, from 0 to 1 or viceversa (Algorithm 4 – Line 6) which takes a constant amount of time. Thus the time complexity for the `Mutation` operator is $O(n)$.

The fitness function relies completely on the problem being resolved. There is no unique measure of complexity for it, but it is a fact that it needs to be computed for each of the n chromosomes in the population. Thus, the total time complexity of an SGA evolving each new generation (fitness calculations, selection, crossover and mutation) is $O(nl + nT_f)$ where T_f represents the time complexity of the fitness function. If the fitness function has a low complexity, as it usually does, then nl becomes the dominant terms. Furthermore, if there exists a stopping criterion based on a constant number of generations to be evolved, the entire algorithm can be executed in polynomial time.

3.2.2 Fundamental Theorem of Genetic Algorithms

The original theory of GAs, as introduced by Holland [18] states that, at a very general level, GAs work by discovering, emphasizing and recombining good *building blocks* of solutions in a highly parallel manner[25]. Good solutions are the result of good building blocks, i.e. combinations of bit values that result in a higher fitness of the string where they are present. This blocks are more often called *schemas* or *schemata*. The idea of parallelism, in this case, refers to the possible number of schemata that can be present in a population.

A schema is a set of bit strings which can be described by a template composed

by 1s, 0s and *s where a * represents a wild card (or a "don't care"). For example, the schema $H = 1{**}0$ represents the set of all 4 bit strings that start with 1 and end in 0, i.e. 1000, 1010, 1100, and 1110. Strings that correctly fit a schema are called instances of H. Schema H has two defined bits (elements which are not wild cards) which is equivalent to saying that H is of order 2 ($o(H)$). In this case, its defining length (distance between its outermost defined bits, $\delta(H)$) is also 2. Given a string of length l there are 2^l possible subset of strings. In an alphabet A where $|A| = k$ there are $(k+1)^l$ possible schemas of length l; therefore, in SGAs there are 3^l possible schemata.

Holland's schema theorem, better known as the fundamental theorem of genetic algorithms, shows that lower-order schemata (schemas containing a small number of defined bits) with above-average fitness should in principle increase their presence as generations evolve. The approximate dynamics of this behavior can be computed by taking into account the probability of a string x which exhibits schema H being selected as a parent, followed by the chances of it being disrupted by the genetic operations of crossover and mutation.

Let the number of instances of schema H at a time t be represented as $N(H,t)$ and its fitness, obtained from the average fitness of these instances, as $\hat{u}(H,t)$. Assume that the chances of a chromosome being selected are proportional to its fitness. The expected number of copies of a string x at time t would then be equal to $F(x)/\bar{F}(t)$, where $F(x)$ is the fitness of x and $\bar{F}(t)$ represents the average fitness of the population at time t; consequently, the fitness of schema H can be computed as $\hat{u}(H,t) = \sum_{x \epsilon H} F(x)/N(H,t)$, where $x \epsilon H$ means "x is an instance of H", Equation 3.2 denotes the expected number of instances of H to be selected in $t+1$.

$$E(N(H,t+1)) = \frac{\hat{u}(H,t)}{\bar{F}(t)} N(H,t) \tag{3.2}$$

Crossover and mutation can both destroy and create instances of a schema after selection has taken place; thus, their effects need to be included in Equation 3.2. Assuming a single-point crossover is being used and p_c is the probability of it being applied, schema H is said to survive if one of its offspring is also an instance of H. The probability of a schema being able to survive highly depends on the crossover point being selected outside of its defining length. Equation 3.3 uses $d(H)$ (defining length of H) and l (string size) to estimate a lower bound probability L_c of schema H being destroyed by crossover. A lower bound is required since there is a chance of some crossovers happening inside the defining length of the schema without destroying it, e.g., if both parents were instances of H.

$$L_c(H) \geq \left(1 - p_c \frac{d(s)}{l-1}\right) \tag{3.3}$$

Finally, the amount of disruption obtained from mutation can be calculated by using p_m as the probability any bit being mutated. The order of H represents its number of defining bits; therefore, Equation 3.4 is the total probability of not having a mutation $(1 - p_m)$ in each bit of H.

$$L_m(H) = (1 - p_m)^{o(H)} \tag{3.4}$$

Equation 3.5 extends Equation 3.2 to include all potential disruptions obtained from crossover or mutation. Thus presenting the formula for calculating the expected instances of H at a time $t + 1$.

$$E(N(H, t+1)) \geq \frac{\hat{u}(H,t)}{\bar{F}(t)} N(H,t) \left(1 - p_c \frac{d(s)}{l-1}\right) [(1 - p_m)^{o(H)}] \tag{3.5}$$

The Schema Theorem serves as the basis of the Fundamental Theorem of GAs and provides an easier understanding of how a GA performs. It is capable of expressing

what happens, on average, to a schema h as one generation evolves into another, without having any knowledge regarding any other schema. However, unless other schemas are taken into account, there is no way to extrapolate the behavior of h beyond the next generation. Some authors disregard the effectiveness of schemata, by arguing that diversity introduced due genetic operations can drive GAs into paths that no theorem can predict [38]. Nevertheless, it is proven that SGAs are capable of exploring large search spaces and even converge towards "good" solutions (an optimal solution is not always the case) more efficiently than a zero-knowledge random algorithm. This behavior provides an insight on how more general, less constrained, GAs can behave in practice.

3.2.3 No Free Lunch Theorem

Genetic Algorithms can be mistakenly claimed as a solution for any search problem, capable of identifying the global optimum in any optimization problem. There is no proof to such claim, and in fact the *No Free Lunch Theorem* (NFL) [42] has demonstrated that such claim is impossible. The NFL theorem states that there is no single search algorithm that is universally better than all others over all possible search spaces.

Let A_1 and A_2 be two different search algorithms. The NFL theorem states that if A_1 outperforms A_2 on some subset ϕ of the set of all functions $\mathcal{F}$, then A_2 should outperform A_1 on $\mathcal{F} \setminus \phi$. Thus, when averaged over all functions f in $\mathcal{F}$, the sum of performances of A_1 and A_2 should be equal. This can be formally expressed as:

$$\sum_f P(d_m^y | f, m, A_1) = \sum_f P(d_m^y | f, m, A_2) \tag{3.6}$$

Where the performance of an algorithm A iterated m times on a cost function f

is represented as $P(d_m^y|f, m, A)$.

The No Free Lunch Theorem allows to demonstrate in a simple way that an algorithm A_1 is worth pursuing for a specific problem. It is sufficient to prove that A_1 is able to outperform random search in this specific search space, as random search will, on average, perform just as well over all search spaces. It should be noted that for an algorithm to "escape" from the NFL theorem it requires to implicitly include knowledge of the domain of the specific problem in its construction, for example by using the fitness measure in a GA, this leads to random search being outperformed.

Chapter 4: Exploring Software Configurations using Genetic Algorithms

As mentioned in the Section 1.1, software configurations can consists of thousands of parameters. Each parameter can have many different settings, and as a result, there can be a very large number of possible configurations. This chapter introduces the usage of a Genetic Algorithm (Section 3.1) as a tool for exploring this search space. Using this approach, a software configuration (application and/or operating system) is modeled as a chromosome, and by mimicking the genetic operations of selection, crossover and mutation observed in nature, the algorithm is able to continually evolve system configurations. A more in-depth explanation of each component of the implementation of the algorithm is described in the following sections.

4.1 Chromosome Model

Assume the existence of a configuration C, composed by n parameters, for example, $P_0, P_1, P_2...P_{n-1}$. Each of the n parameters is configured to a specific setting s_i^j, where $S_i = (s_i^0, s_i^1, s_i^2, ..., s_i^{k-1})$ is the set of possible settings for P_i. The chromosome encoding of C is given by the string $C' = s_0 s_1 s_2 ... s_{n-1}$, and it contains every setting of configuration C as genes while their index (locus) indicates to which parameter they are associated.

Table 4.1 provides an example of five different encodings of a configuration consisting of 4 parameters. Given the number of parameters and possible settings per parameter, there are 2^4 possible encodings. In this case, each parameter could only be assigned to either value A or B, i.e. $s_i \in \{A, B\}$.

Before being added to the pool, each encoded configuration is inserted in a *2-tuple*

Chromosome	P_0	P_1	P_2	P_3
C'_0	A	B	B	A
C'_1	A	B	A	A
C'_2	B	A	B	A
C'_3	A	B	A	B
C'_4	A	A	B	A

Table 4.1: Example encoding of 5 configurations

composed by an array and a number, i.e (C'_i, λ_i) where C'_i contains the encoded configuration and λ_i tracks the number of secure settings found in C_i (further explained in Section 4.2.1).

4.2 Fitness

The fitness of a chromosome evaluates how similar its genes are to those found in an optimal solution. In terms of configurations, the value of a chromosome's fitness can be associated to the number of misconfigurations found in it. In that case, a good solutions would be represented by configurations which incur in no incidents. Not all configuration errors carry the same implications, some might be more harmful than others. For this reason, previous research [20] was performed using existing scoring mechanisms such as the Common Vulnerability Scoring System (CVSS). CVSS is a free and open industry standard that measures the severity of computer system vulnerabilities. Scores are computed based on several metrics which prioritize the effects on a system's *confidentiality*, *integrity* and *availability* (CIA triad). This approach proved to work correctly; however, further research [14] showed that a simpler scoring system, where configurations or settings received a score of 100 if classified as secure or a 1 if considered insecure, performed just as well for identifying misconfigurations. This approach was pursued not only for its simplicity but also due to how adding a recently uncovered exploit becomes an easy task when compared to waiting until it

has been correctly assessed and scored in the CVSS system.

4.2.1 Scoring Methods

Genetic algorithms attempt to evolve solutions which can outperform their parents until an optimal state is achieved. In terms of secure configuration management, an optimal state can be considered as having a configuration where every parameter is either a passive parameter or a vulnerable parameter set to a secure setting. Furthermore, when an optimal configuration is found, the GA will increase its presence in the upcoming generations (Section 3.2.2), until eventually the population becomes homogeneous and the algorithm converges.

Starting with a population that contains an example of the optimal configuration, or one that is very close, can greatly reduce the time to convergence. For example, if the implemented fitness function considers secure configurations as optimal solutions (referred to as "positive fitness"), having one in the initial population, or in an early evolved population, will reduce the number of generations required for the algorithm to converge; thus reducing the diversity of evolved populations and avoiding the exploration of insecure configurations (since their fitness value is less than their parent's). Furthermore, inverting the fitness function to score vulnerable configurations higher (referred to as "negative fitness") would express the same behavior against secure ones, avoiding them while evolving more insecure configurations which are evaluated as optimal.

There are different advantages to both possible fitness measuring mechanisms. For this reason, this thesis uses a combination of both techniques by creating two GAs whose only difference is how a fitness score is computed. When chromosomes are added to the pool, be that the initial population or one being evolved for a new generation, the count of how many secure settings have been found in its structure

is stored. Each of the GAs will later proceed to use this count accordingly for their fitness evaluation.

4.2.2 Cumulative Scoring vs Binary Scoring

Two different approaches have been developed for evaluating a configurations score based on the "secure or insecure" classification: individual parameter scoring (or cumulative scoring), used for identifying parameter OR chains; and binary scoring, which helps determining parameter AND chains.

Cumulative scoring associates a score to each parameter of a configuration, it then proceeds to sum all of these values to calculate the total fitness of the chromosome. This technique provides knowledge of the behavior of individual parameters within both vulnerable and secure configurations.

Binary scoring assigns a fixed value of either 100 or 1 to a configuration depending on whether the combination of its parameters is considered safe or not respectively. As a result, a configuration will only be given a score of 100 if it is completely secure (no vulnerabilities), otherwise the configuration is given a score of 1 regardless to how many vulnerabilities exist. This scoring system fits very well when identifying parameter AND chains since no individual setting causes the configuration to become vulnerable, but a combination of them will. Furthermore, this approach has been proven successful in previous research [27] when identifying parameter AND chains.

4.3 Selection

The process of selection identifies two parent chromosomes from the current generation's pool based on their fitness values. There are different selection methods including tournament selection, fitness proportionate selection, stochastic universal sampling, and others (Section 3.1.3). In this thesis, the implemented selection method,

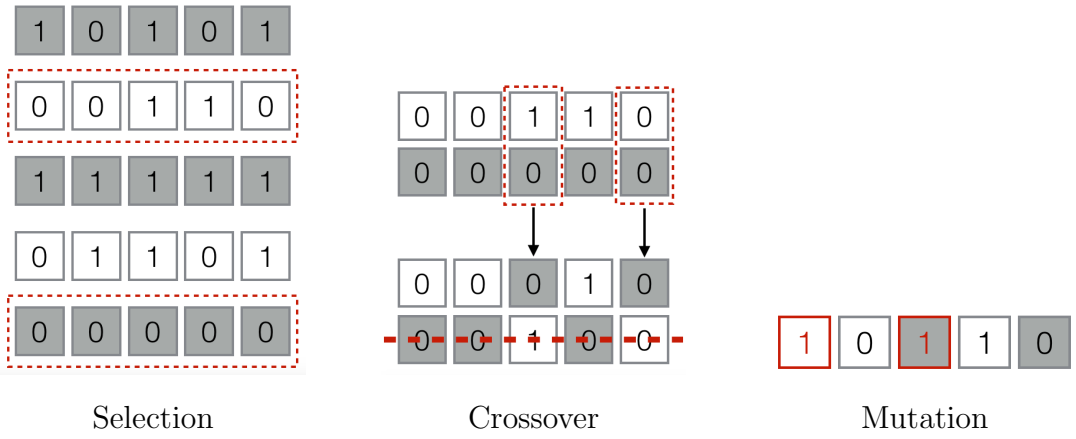

Figure 4.1: Genetic operators of selection, crossover, and mutation as performed in order to generate a single new offspring

based on the results obtained in [35], is elitism.

Selection methods, such as Roulette Wheel Selection, assign chromosomes a probability of being chosen proportional to their fitness value. This increases the likeliness of selecting more fit chromosomes but still gives lower fitness configurations an opportunity. The implemented elitism approach varies slightly within both GAs. The positive fitness GA will consider exclusively configurations with the highest count of secure settings among the population, on the other hand, the negative fitness GA will attempt to lower this count as much as possible. Since different parameter combinations can contain the same count of secure settings (λ), two distinct parents are selected randomly from among the configurations with the highest or lowest count, accordingly to the fitness measure being used, in the current population. In case there is a single configuration with the highest count, it will be duplicated and assigned as both parents. This method reduces the diversity of the population but increases its overall fitness. Algorithm 5 shows an overview of how this process has been implemented. The `SortByCount` (Algorithm 5 – Line 2) function will sort the individuals in the pool according to their associated count of secure configurations (λ) in either descending order (from highest value to lowest), for the positive fitness

GA; or descending, for the negative fitness GA,

Algorithm 5: Elitist selection implementation

```
1  Function Selection (pool)
       /* Sort the pool by the count of secure configurations       */
2      sortedPool ← SortByCount(pool);
       /* Gets the highest scoring chromosomes                      */
3      highestPool ← GetHighestScoringElements(sortedPool);
4      if Size(highestPool) == 1 then
           /* If there is a single element with the highest count   */
           /* Set the highest element as both parents               */
5          parentConfigA ← highestPool[0];
6          parentConfigB ← highestPool[0];
7      else
           /* Randomly select distinct parents from the highestPool */
8          parentConfigA = ← Random(highestPool);
9          parentConfigB = ← Random(highestPool - parentConfigA);
10     end
       /* Create a count λ which will later be updated              */
11     λ ← 0;
12     return (parentConfigA, λ), (parentConfigB, λ);
13 end
```

4.4 Crossover

After two parents have been selected, the process of crossover combines portions of both chromosomes to create new configurations. The probability of crossover (p_c) is a user defined variable which states how often should this exchange take place. In this thesis, given that the order of parameters in a configuration does not matter as long as it is consistent, a uniform crossover approach was implemented. Uniform crossover eliminates all potential bias in the operation by giving each parameter the same probability of being exchanged between the two offspring. Based on experimental results obtained by [35][47], a crossover rate, p_c, of 0.05 is applied. This rate indicates the chances each parameter has of being exchanged between children. A relatively low

rate is necessary, in this case, to balance the fairly high number of uniform crossing points. From the two obtained offspring, one of them is randomly chosen to proceed to the mutation process while the other one is discarded. Algorithm 6 shows an overview of how this process has been implemented.

Algorithm 6: Uniform crossover function implementation

```
1  Function Crossover (inputConfigA, inputConfigB)
       /* Since each inputConfig is a tuple (C'_i, λ_i)                    */
2      childConfigA ← inputConfigA[0];
3      childConfigB ← inputConfigB[0];
4      n ← total number of parameters;
5      for i ← 0 to n − 1 do
6          p ← Random(0.0, 1.0);
           /* With a crossover rate (p_c) of 0.05 in this case             */
7          if p < p_c then
               /* Exchange the current settings                            */
8              temp ← childConfigA[i];
9              childConfigA ← childConfigB[i];
10             childConfigB ← temp;
11         end
12     end
       /* Randomly select one of the generated children                    */
13     childConfig ← Random(childA, childB);
       /* Create a count λ which will later be updated                     */
14     λ ← 0;
15     return (childConfig, λ);
16 end
```

4.5 Mutation

The process of mutation provides the ability to explore new regions of the problem space by randomly changing parameters in the generated offspring. Similar to the process of crossover in the current implementation, for each parameter in the configuration there is a chance of it being mutated. When a parameter is mutated, one

of its other possible settings is randomly chosen and applied. There is no bias as to which settings are more likely to be applied; However, [35] associated a Gaussian distribution for scenarios where some settings are more likely to be found. Mutation gives the algorithm a chance to explore configurations which might not have been seen before, and provides a way to further increase diversity within the population. Algorithm 7 shows how the mutation process has been implemented.

Algorithm 7: Mutation function implementation

```
1  Function Mutation (inputConfig)
       /* Since each inputConfig is a tuple (C'_i, λ_i)                    */
2      childConfig ← inputConfig[0];
3      n ← chromosome string length;
4      for i ← 0 to n − 1 do
5          p ← Random(0.0, 1.0);
           /* With a mutation rate (p_m) of 0.05 in this case              */
6          if p < p_c then
               /* Obtain the set of possible settings S_i for P_i          */
7              S ← GetSettingsForParameter(i);
               /* Remove the current value of P_i from S_i                 */
8              S ← S − childConfig[i];
               /* Randomly assign a new value from S_i to P_i              */
9              childConfig[i] ← Random(S);
10         end
11     end
       /* Create a count λ which will later be updated                     */
12     λ ← 0;
13     return (childConfig, λ);
14 end
```

4.6 Complete Genetic Algorithm Implementation

The aforementioned GA starts its execution by analyzing a predefined pool of N configurations, all of them containing the same n parameters. They are later encoded into chromosomes, following the model presented in Section 4.1, and added to the

initial population. The processes of selection, crossover and mutation (implemented as mentioned throughout this chapter) will evolve a single offspring per iteration, thus they are repeated N times to create a new generation that matches the initial population in terms of size. These steps can be repeated indefinitely, spawning a new generation until a predefined stopping condition is met (Section 3.1.6). In this implementation, the stopping condition will be a limit on the maximum number of generations to be evolved; this value will be previously defined by the user. Algorithm 8 shows the complete implementation of the developed GA using the `Selection` (Algorithm 5), `Crossover` (Algorithm 6), `Mutation` (Algorithm 7) introduced in the previous sections.

Algorithm 8: Genetic Algorithm for Exploring Software Configurations

```
1  create an initialPopulation of N chromosomes;
2  currentGeneration ← initialPopulation;
3  generationCount ← 0;
4  while generationCount < maxGenerations do
       /* Start with an empty next generation                    */
5      nextGeneration ← {};
6      for i ← 0 to n − 1 do
           /* Select two parents from the current population     */
7          parentConfigA, parentConfigB ← Select(currentGeneration);
           /* Apply the crossover operator                       */
8          childConfig ← Crossover(parentConfigA, parentConfigB);
           /* Apply the mutation operator                        */
9          childConfig ← Mutation(childConfig);
           /* Update the number of secure settings λ             */
10         childConfig ← UpdateSecureCount(childConfig);
           /* Add the generated offspring to the next generation */
11         add childConfig to nextGeneration;
12     end
       /* Set the next generation as the current one             */
13     currentGeneration ← nextGeneration;
14     generationCount ← generationCount + 1;
15 end
```

4.6.1 Feasibility

A configuration is considered *feasible* if it maintains the necessary functionality for the system. For example, if the computer is a web server, then any configuration that blocks network access would be infeasible. For this reason, any candidate solution needs to be analyzed in terms of feasibility before being added to the new generation's pool.

Two distinct approaches were analyzed when dealing with unfeasible configurations. The first one considered preserving them, since they might contain useful information, but lowering their fitness value. This strategy ensures the chromosome's presence in the immediate future generation; however, given the implemented elitist selection method, it will most likely be ignored when evolving new offspring.

The second approach proposed discarding infeasible configurations before adding them to the upcoming generation's pool. This strategy provides the algorithm with a chance to repeat the processes of selection, crossover, and mutation to generate a feasible candidate solution. Given the opportunity to obtain a better performing offspring, this was the approach used for the implementation.

Chapter 5: Identifying Vulnerable Parameters using Genetic Algorithms

Genetic Algorithms are able to explore the solution space efficiently, discovering secure and insecure configurations effectively thanks to the fitness function; however they lack the ability to extract information regarding each individual parameter settings. Identifying vulnerable settings and their interdependencies is a crucial to task in system security. If the poorly configured parameters can be correctly identified then the administrator is able to set the remaining parameters to ensure functionality and improve performance. In this chapter, an approach for identifying parameter issues is introduced as an extension to the genetic algorithm presented in Chapter 4. Using the evolved configurations from the GA in conjunction with a histogram designed to keep track of each setting's effects in a configuration, the algorithm is able to classify parameters as either passive of vulnerable. The remainder of this chapter will refer to the implementation of the algorithm in terms of the positive fitness GA, i.e. the GA will attempt to explore new secure settings while the histogram will assign positive counts to settings considered secure and negative counts to vulnerable ones. This logic can be inverted for the negative fitness GA.

5.1 Algorithm for Identifying Vulnerable Parameters

Given sets of unique configurations, it is possible to compare the parameter settings to determine what they have in common. Likewise, it is possible to compare unique vulnerable configurations to identify which parameter settings are also in common. A simple approach to this is to maintain counts for each possible parameter setting

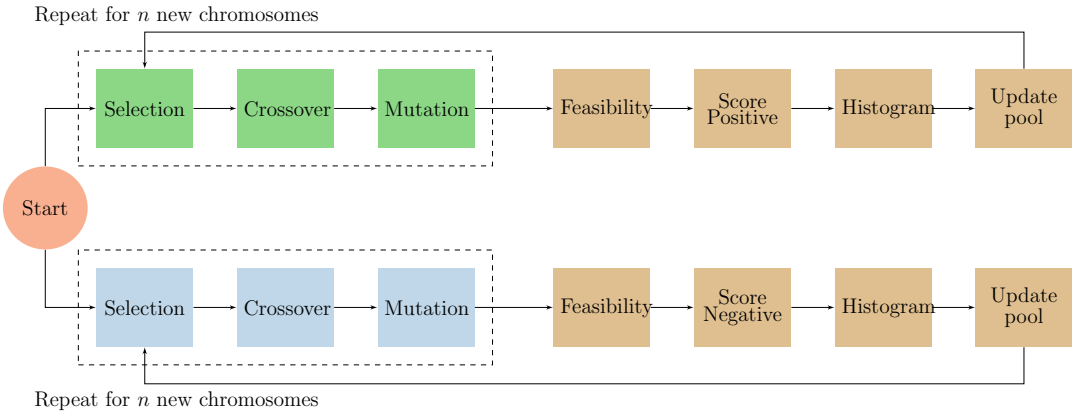

Figure 5.1: Flow chart of how both genetic algorithms are extended in order to identify vulnerable parameter. Note, this chart is repeated with every new generation

observed throughout the evolved generations. If a setting has only been associated with vulnerable configurations, it can be considered as an insecure setting. On the other hand, if it has only been observed in secure configurations, there is a possibility of it being a secure setting. Any parameter containing at least one setting which is being classified as vulnerable, would be added to a suspect list.

A histogram has been implemented in order to perform the task of recording the behavior of each setting as new generations are evolved. Once the GA creates a new population, each of the N configurations is used to update this histogram. Assuming that a configuration C is considered secure, meaning that every parameter $(P_0, P_1, ..., P_{n-1})$ is set to a secure setting s_i, each of its settings s_i^j will receive an increment of 1 in their histogram counts (represented as M_i^j where M represents the histogram, i the parameter number and j the setting's index within S_i); on the other hand, if the configuration is classified as vulnerable, they will receive decrements of -1. Figure 5.2 shows an example of this algorithm being used to update a histogram. It should be noted that a matrix is being presented for the sake of the example, the actual implementation of this histogram is not actually an $n \times k$ matrix since not all sets S_i contain k elements.

	P_0	P_1	P_2	P_3
C_0	s_0^2	s_1^0	s_2^3	s_3^3
C_1	s_0^3	s_1^0	s_2^1	s_3^3
C_2*	s_0^0	s_1^2	s_2^0	s_3^0
C_3*	s_0^0	s_1^1	s_2^2	s_3^1

Population

$M =$

	P_0	P_1	P_2	P_3
s_i^0	-2	2	-1	-1
s_i^1	1	-1	1	-1
s_i^2	1	-1	-1	0
s_i^3	0	0	1	2

Histogram

Figure 5.2: Example histogram generated from a population of 4 chromosomes, 2 of them vulnerable (C_2* and C_3*)

After the histogram has been updated, it is explored in order to identify potential vulnerable settings. This process is done by dividing the stored value in the histogram by the number of configurations in which the setting appears. In a more formal representation:

$$ratio(s_i^j) = \frac{M_{i,j}}{\#\ of\ configurations\ where\ s_i^j\ is\ observed} \tag{5.1}$$

Using this equation, for example, one could compute the ratio for s_0^0 (from Figure 5.2), obtaining $\frac{-2}{2} = -1$. In order to obtain a value of -1, a setting must have been associated exclusively with vulnerable configurations; this suggest that its parameter should be considered as a suspect. Furthermore, when using individual parameter scoring (Section 4.2.1), if a setting's ratio shows that it has been found in both secure and vulnerable configurations, it is safe to classify it as secure. This strategy is not replicable when using binary scoring, since parameter AND chains are its main target. If a group of settings form an AND chain, there exists a possibility of observing them individually in both secure and vulnerable configurations but strictly in vulnerable configurations when together.

The algorithm stops once all settings have been evaluated. Depending on the existing pool of configurations used, one run of this algorithm might not fully classify all parameters. Parameters currently in the suspect list may not be vulnerable but the

algorithm has not evaluated enough information to discard it. For this reason, this algorithm is paired with the GA and executed after each newly evolved generation.

5.1.1 Configuration Uniqueness

Ideally any configuration used to update the histogram contributes a new combination of settings, this will provide it with more information to assist with their classification; however, this is not always the case. The GA may evolve configurations that have already been recorded in the histogram. For this reason, a hash table was incorporated as a uniqueness verifier. Configurations which have not been seen yet, according to the hash table, are processed by the histogram and added to the hash table consequently.

Figure 5.3 shows the difference in histogram counts before and after applying the uniqueness verification. This results are obtained from an execution where every configuration was composed by 4 parameters ($|C| = 4$), each parameter's set S_i contained $|S_i| = 4$ settings, and the population size $N = 4$. Results were obtained after 50 evolved generations and assuming there was a single parameter (P_1) involved in an attack. Figure 5.4 shows this same execution in terms of the obtained ratios for each setting, obtained using Equation 5.1. As mentioned before, a value of -1 is assigned strictly when a setting has been associated exclusively with vulnerable configurations. After execution has finished, P_1 is considered the prime suspect for the attack since three of its settings (s_1^0, s_1^2, s_1^3) have been declared vulnerable. Furthermore, s_1^1 has been identified as a secure setting for P_1.

	s_i^0	s_i^1	s_i^2	s_i^3
P_0	50	48	46	56
P_1	55	44	49	52
P_2	51	48	52	49
P_3	54	56	50	40

Without uniqueness verification

	s_i^0	s_i^1	s_i^2	s_i^3
P_0	32	30	37	38
P_1	38	25	40	34
P_2	30	36	35	36
P_3	39	31	38	29

With uniqueness verification

Figure 5.3: Histogram counts obtained from an execution, before and after applying the uniqueness verifier

	s_i^0	s_i^1	s_i^2	s_i^3
P_0	-0.44	-0.58	-0.56	-0.64
P_1	-1.0	1.0	-1.0	-1.0
P_2	-0.52	-0.33	-0.69	-0.67
P_3	-0.48	-0.53	-0.6	-0.65

Without uniqueness verification

	s_i^0	s_i^1	s_i^2	s_i^3
P_0	-0.56	-0.6	-0.78	-0.57
P_1	-1.0	1.0	-1.0	-1.0
P_2	-0.66	-0.61	-0.71	-0.55
P_3	-0.79	-0.35	-0.52	-0.86

With uniqueness verification

Figure 5.4: Computed ratios obtained from an execution, before and after applying the uniqueness verifier

Chapter 6: Experimental Results

In this chapter, the performance of the parameter classification algorithm is measured experimentally. Experiments testing the effect of population size, configuration size, and chain size demonstrate the effectiveness and scalability of this algorithm. Experiments were also performed where this algorithm was paired with a random configuration generator instead of the introduced Genetic Algorithm in order to compare their effectiveness.

The following experiments are based on simulated configurations generated to be similar to those found in Red Hat 5 and Apache-2.2 installations. The type and percentage of occurrence of their parameters is based on the checklists published by the National Institute of Standards and Technology (NIST) in their National Checklist Program Repository [28]. This probability densities are presented in Table 6.1. It is worth highlighting that more than 50% of the parameters seen can only be set to binary settings (either true or false), while 35% can be configured to one setting of a set of five possible, and lower percentages include sets of 3,4 and 6. For existing parameters that can be set to a value within a predefined range, it is assumed that values close to one another have generally the same effect. For example, there may be little difference in setting a timer to 5 or 6 seconds given the possible range is from 0 to 65535 seconds and 1 second is the smallest unit. For this reason, numbers in ranges are grouped into 5 settings according to their effects. Furthermore, bit vectors are also converted into sets of 5, as there is a limited number combinations of bits that should be taken into account, for example, when dealing with file permissions. Configurations are generated using these parameter statistics instead of actual configurations in order to be able to perform experiments with variable configuration sizes and measure the

Parameter type	Observed ratio
Binary	0.575
Set of 3	0.0375
Set of 4	0.0125
Set of 5	0.35
Set of 6	0.025

Table 6.1: Parameter type densities seen in National Checklist Program Repository

algorithm's scalability.

Unless stated otherwise, results presented in the remainder of this chapter were obtained from averaging the results obtained from 100 repetitions of the same experiment, using the generated pool of 100 configurations, all set to the same vulnerable status, where each parameter contains a single secure setting and the rest being vulnerable, and assuming every possible configuration is feasible. Furthermore, the number of generations required to determine a list of suspect parameters (set of parameters considered vulnerable at a specific generation) is shown, results are labeled as "positive" or "negative" according to the fitness measure being used where a positive GA assigns better scores to secure configurations while the negative does the same to vulnerable ones (Section 4.2.1), and the probability of crossover and mutation, p_c and p_m respectively, have been set to 0.05; values which were determined experimentally, as explained in Sections 4.4 and 4.5.

6.1 Search Comparison

The GA introduced in Chapter 4 was implemented with the intention of exploring the immense search space of possible configurations for a system. The algorithm for identifying vulnerable parameters (Chapter 5) was later added as an extension to the GA, in order to harness evolved configurations from each generation as a source of information. Theoretically, any search algorithm able to provide distinct

configurations could be matched with the implemented histogram in order to identify the components of a vulnerability. Figure 6.1 shows the improvement provided by using this GA compared to using a random search algorithm. Furthermore, this experiment also shows how, by including domain specific knowledge, the implemented GA is able to outperform random search for this specific problem, thus "escaping" the No Free Lunch theorem (Section 3.2.3).

It is worth mentioning that this is only one of the advantages of using the GA over random search (or other search algorithms), one of them being the ability to maintain core system settings required for the expected functionality of the system and generate new secure configurations around them.

The graph's behavior can be attributed to how the initial population is setup. Every chromosome in the initial population encodes a vulnerable configuration. For this reason, the positive GA considers every single parameter as a suspect. Meanwhile, the negative GA, which recognizes an insecure configuration as an optimal solution, starts the execution with zero suspects. Once diversity gets included via the genetic operators of crossover and mutation, the positive GA is able to narrow down the list of suspects. Furthermore, the negative GA finds secure configurations (which in this case imply a lower fitness value) causing the small "spike" in the beginning and later being narrowed down as more configurations are analyzed.

6.2 Effect of Settings Distribution

The settings distribution is defined by how many secure and vulnerable settings are found in the set S_i for every parameter P_i. As seen in Figure 6.2, the performance of the different fitness measuring GAs highly depends on this distribution. In this experiment, both approaches attempted to solve the same 6 setting parameter AND chain using the binary scoring method. Results show that their behavior is very

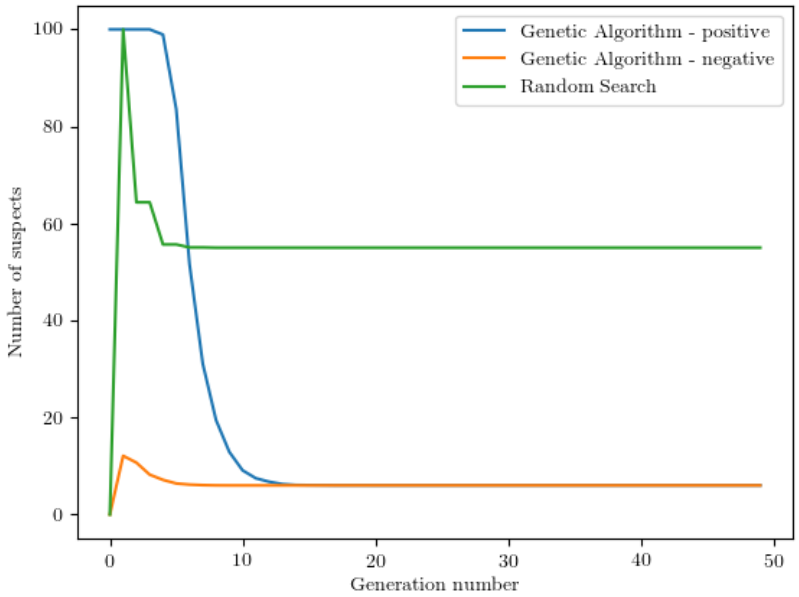

Figure 6.1: Comparison of the average number of generations required to solve a 6 parameter OR chain with a population of 100 and a configuration size of 100, using the developed GA (positive and negative) versus using Random Search

similar in reversed cases. For example, when only one insecure setting is present in every S_i the positive fitness GA (one vulnerable - positive) performs as good as the negative fitness GA does when a single secure setting exists in each S_i (one secure - negative). Similarly, the poor performance of the positive fitness GA when there exists a single secure setting per S_i (one secure - positive) is homologous to the poor performance of the negative fitness GA when a single insecure setting per S_i exists (one vulnerable - negative).

This mirrored behaviour can be attributed to the GA's attempts to find the optimal settings for each given fitness measure. While the positive fitness attempts to explore exclusively secure configurations, the negative fitness does the opposite. Through the remainder of this chapter, experiments were performed assuming the existence of a single secure setting, and the remaining insecure, in each S_i for all P_i. As demonstrated in this experiment, if one of the GAs performs better than the other one on certain experiment, it is more than likely that changing the setup to a single vulnerable configuration will cause its counterpart to outperform it.

6.3 Effects of Uniqueness

This set of experiments were designed to demonstrate the effect of processing only unique configurations when updating the histogram. Figures 6.3 and 6.4 shows the effects of adding a hash table for allowing only configurations that have not been seen before to be recorded in the histogram, solving a six parameter and an eight parameter OR chain respectively. Given the large number of possible combinations of settings in a configuration of 100 parameters (1.48×10^{52} according to the generated configurations) the probability of finding a repeated configuration is very low. In fact, an analysis of the final counts obtained in both histograms revealed that throughout 50 generations, the hash table registered 4898 unique configurations in comparison

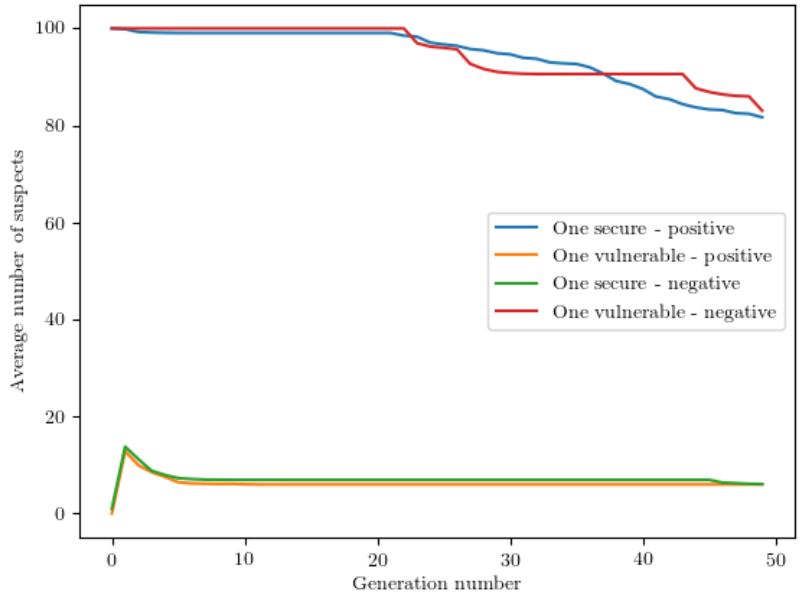

Figure 6.2: Comparison of the average number of generations required to solve a 6 parameter AND chain where one setting per parameter is secure and the rest is vulnerable against one insecure setting and the rest secure

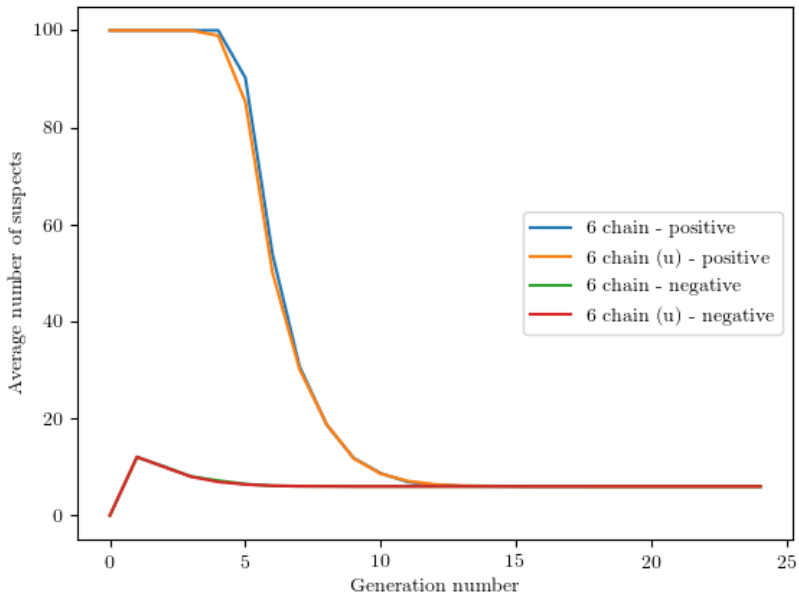

Figure 6.3: Average number of generations required to solve a 6 parameter OR chain allowing the implemented histogram to track any configuration in comparison to only tracking uniques (u)

to 5000 that were recorded without the uniqueness verification. Taking into account that the GA starts its execution with a homogeneous pool (every chromosome is equal) of 100 configurations, the total number of repetitions found in the totality of the aforementioned execution was 2; hence the similarity in the obtained results of these executions.

6.4 Effects of Population Size

Figure 6.5 shows how the total number of configurations per generation (population size) affect the performance of the GA, and thus the parameter identification algorithm, when solving a 6 parameter OR chain. As a larger population results in more configurations being observed at each generation, the algorithm is able to narrow down the set of suspect parameters (parameters which might be related to the discovered

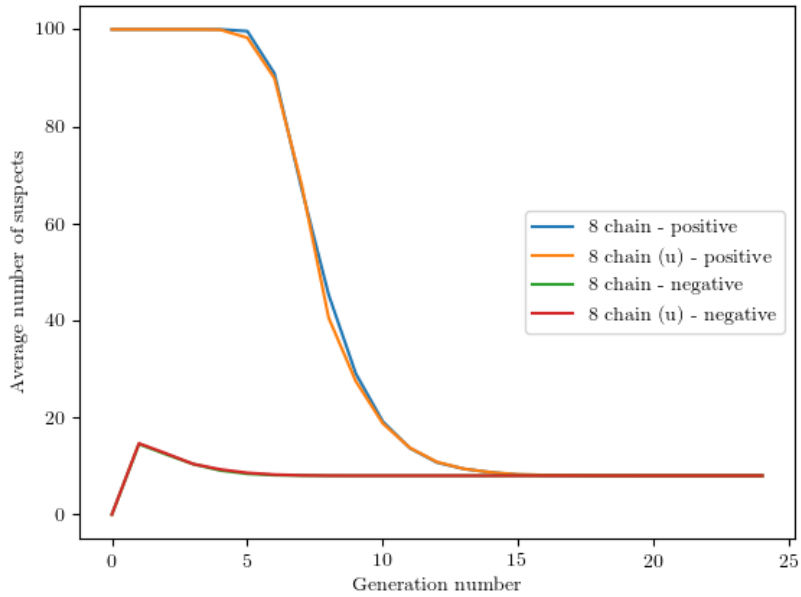

Figure 6.4: Average number of generations required to solve a 8 parameter OR chain allowing the implemented histogram to track any configuration in comparison to only tracking uniques (u)

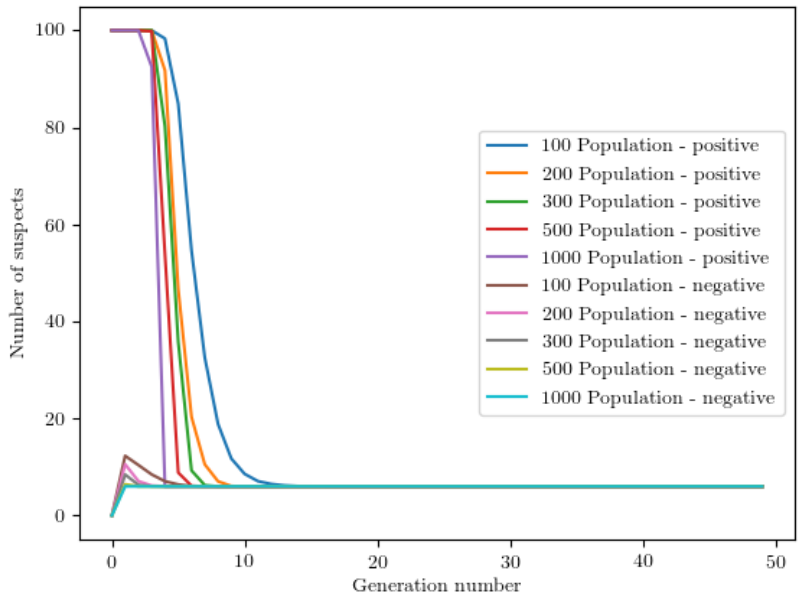

Figure 6.5: Comparison of the average number of generations required to solve a 6 parameter OR chain with a variable sized population and a configuration size of 100, using the developed GA (positive and negative)

vulnerability) in less number of generations. Although increasing the population size reduces the number of generations required to determine the components of a parameter chain, it increases the actual execution time (wall clock) of the algorithm; since there are more configurations to be processed as each generation is evolved. It is fair to say that, regardless of the population size, there is a certain number of configurations needed to determine the suspects. So in reality, it will take the same amount of time (a few large populations = large number of small populations).

6.5 Effect of the Number of Vulnerabilities

This section presents the experimental results obtained when performing simulations with various numbers of interdependent parameters. In addition, results from both fitness measures (Section 4.2.1), positive and negative, are compared as a way to

demonstrate their conduct in different scenarios.

6.5.1 Effect in a Parameter OR Chain

Figure 6.6 shows the average number of generations required for the algorithm to successfully identify parameter OR chains (vulnerable issues that can amplify each other) of various sizes using the distinct fitness functions. In this case the initial population of 100 chromosomes contained exclusively vulnerable configurations, therefore the positive fitness GA considers every possible setting as a suspect and narrows down the list with every generation until it eventually identifies the correct issues. Accordingly, the negative fitness GA starts generation zero with exclusively vulnerable configurations, thus an optimal overall fitness among the population. For this reason, there are initially no suspects. As soon as generation one is evolved and new configurations are introduced via crossover and mutation, some vulnerable parameters are set to their respective secure setting turning them into potential suspects.

6.5.2 Effect in a Parameter AND Chain

Figure 6.7 shows the average number of generations required for the algorithm to successfully diagnose parameter AND chains (a specific combination of settings yields the vulnerability) of various sizes using both fitness functions (positive and negative). It is worth to mentioning that this particular experiment was executed for 100 generations instead of 50 in order to provide a better view of the positive GA's performance as chains get bigger. To determine this type of chains, the binary scoring strategy is used; therefore, a configuration will be scored as either a 100 or a 1. In this case, the negative fitness GA has a similar behavior to the one seen when identifying the components of OR chains. It starts by finding no suspects in the initial all vulnerable population and, as crossover and mutation randomly introduce configurations that fix

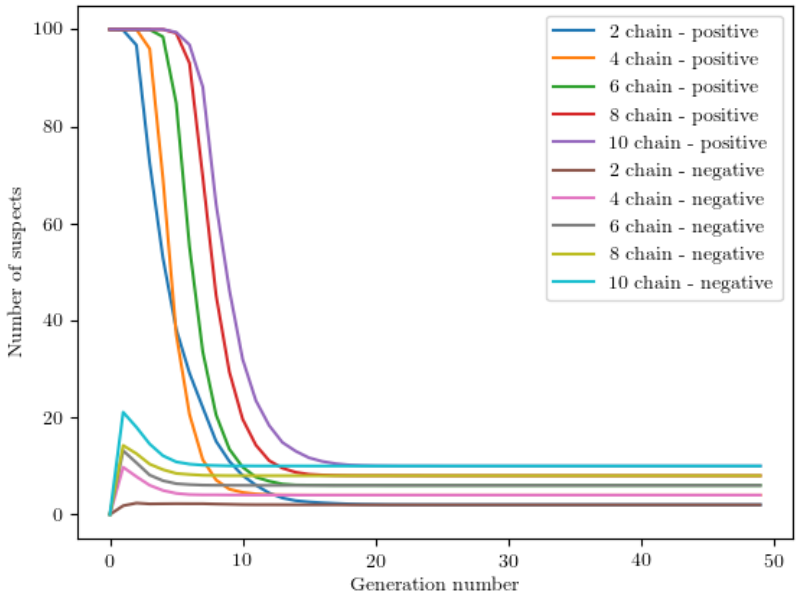

Figure 6.6: Average number of generations required to solve an OR chain of varying number of issues with a population of 100 and a configuration size of 100, using both fitness methods (positive and negative) and individual parameter scoring

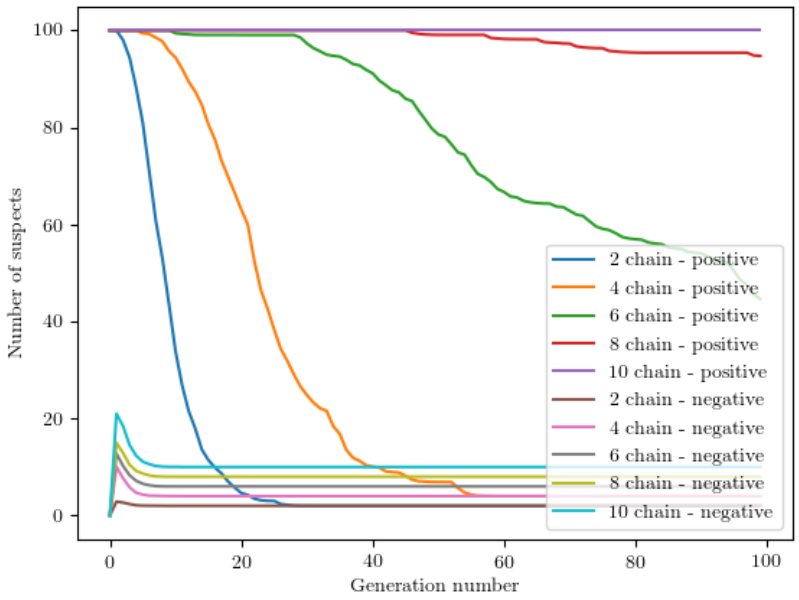

Figure 6.7: Average number of generations required to solve an AND chain of varying number of issues with a population of 100 and a configuration size of 100, using both fitness methods (positive and negative) and binary scoring

the chain, it quickly establishes a set which is narrowed down in further generations. On the other hand, the positive fitness GA struggles when diagnosing this type of chain. Since it can easily fix the misconfiguration (since its an AND chain, changing any of the settings should suffice), the GA will attempt to continue generating secure configurations. However, once the vulnerable combination of settings is introduced coincidentally, by the means of mutation and/or crossover, the GA is able to slowly narrow down this suspect list; as the chain size increases, there is a lower probability of this combination being introduced, which explains the poor performance of this GA when identifying AND chains with a greater number of components.

6.6 Diversity

Given the large search space of possible configurations that can exist in a system with 100 different parameters ($n = 100$), the average number of new configurations seen per generation, is shown in Figure 6.8, is constantly the size of the total population. However, as shown in Section 6.1 this does not necessarily mean that any algorithm that provides generates valid configurations will yield the same results as the GA. As the GA has knowledge over the domain, it can provide distinct combinations of vulnerable and secure configurations by altering the values of parameters which are not directly related to the discovered vulnerability.

Figure 6.9 shows the diversity measure (Hamming distance), of the same experiment, present in the configuration pool after each generation is evolved. These experiments were performed using both fitness scoring mechanisms, where the initial population was homogeneous (thus explaining the initial spike shown in generation 0) and three different distributions of settings within each set S_i; one secure setting among only vulnerable settings, one vulnerable setting among secure ones, and half secure/half vulnerable settings.

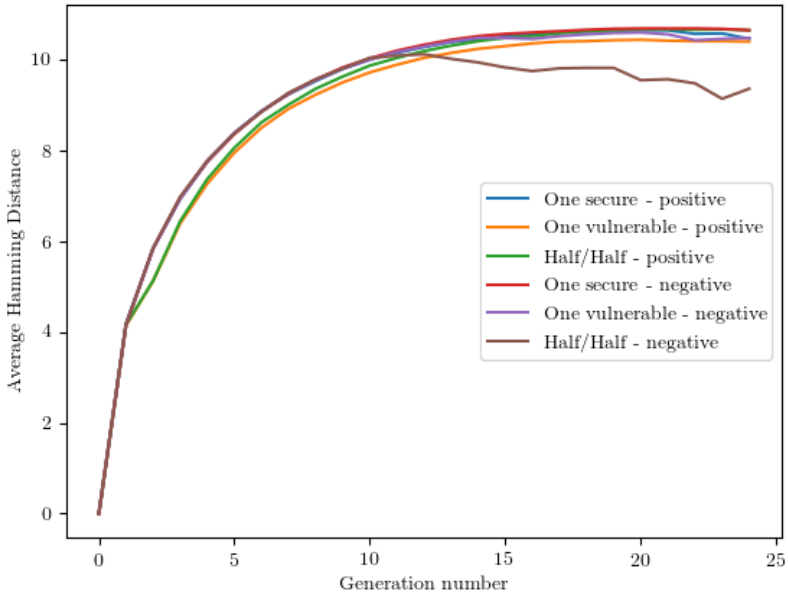

Figure 6.8: Average Hamming distance measurement among 100 chromosomes in each generation using a configuration size of 100, both fitness methods (positive and negative), and parameters with only one secure setting and the rest vulnerable; one vulnerable and the rest secure; and half vulnerable/half secure settings

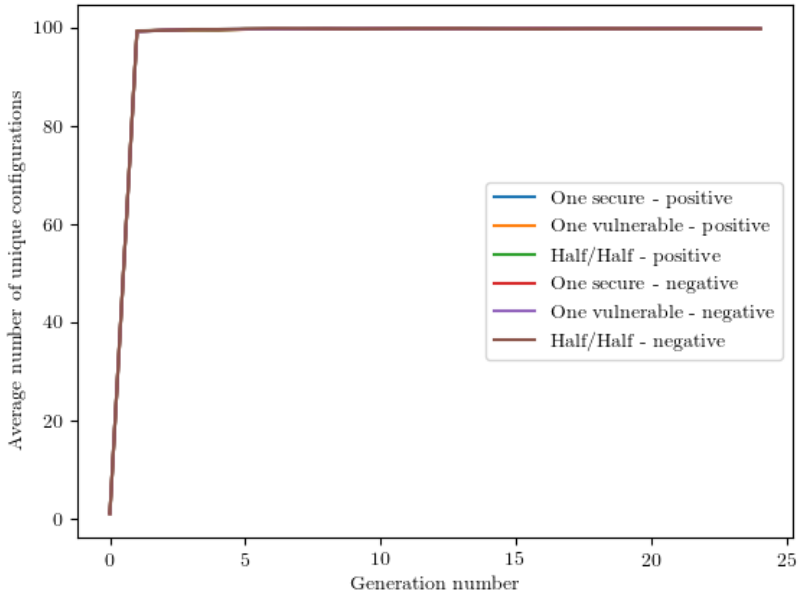

Figure 6.9: Average number of unique configurations seen in each generation using a population size of 100, both fitness methods (positive and negative), and parameters with only one secure setting and the rest vulnerable; one vulnerable and the rest secure; and half vulnerable/half secure settings

Chapter 7: Conclusions

Software configuration management becomes a more difficult task as systems become more complex. Many cyber attacks can be attributed to misconfigurations that provided a malicious entity with exploitable vulnerabilities. Due to the large number of configuration parameters coexisting within a system, the potential interdependencies between them, and the number of security exposures being discovered each day, maintaining a secure software configuration manually is a challenging process. This thesis introduced an automated approach for secure software configuration management, able to identify distinct types of chain related vulnerabilities and their components. Furthermore, by using a Genetic Algorithm, this procedure is capable of generating multiple secure configurations which maintain the expected usability of a system, providing a tool for resilient cyber defense.

Experimental results show that, by expanding this Genetic Algorithm with a customized implementation of a histogram, capable of identifying commonalities between insecure configurations based on their setting's frequencies, and two distinct fitness measuring systems, the introduced approach is able to successfully identify interdependent configuration parameters and resolve vulnerabilities containing parameter chains. In addition, results also demonstrated the scalability of this technique by measuring its performance within large numbers of configurations, settings and various quantities of misconfigurations.

This research successfully achieved the objective of introducing a scalable and resilient configuration management tool capable of fixing misconfiguration problems caused by interdependent parameter settings, while identifying the relationship between their components.

Chapter 8: Future Work

Although the provided experimental results are promising, there are several areas where further research could provide improvements to this work.

8.1 Termination Conditions

The current termination condition for the implemented Genetic Algorithm is a user defined number of generations. Further research may be able to provide a more self-aware GA implementation which may be capable of making this decision based on different measures.

Although the hash table of unique configurations didn't provide an improvement in terms of the number of required generations for identifying vulnerable parameters (Section 6.3), it does contribute a total sample size of explored configurations. This could be paired with further statistical analysis to determine a confidence level in terms of suspects.

A customized diversity measurement system, the current implementation uses Hamming distance, could provide a better insight on the similarity within the population. Based on this evaluation, the GA would be able to decide if evolving a new generation is of value.

8.2 Parallelism

There is a wide range of alternatives when attempting to parallize a Genetic Algorithm. This could represent a potential reduction in execution time. However, the introduced algorithm for identifying vulnerable configuration parameters relies on a

shared histogram which would represent a bottleneck when attempting to implement a parallel solution. A possible solution would be implementing a parallel reduction strategy using partial histograms and counts of the number of settings that have been encountered, and later computing the required ratios in a parallel manner. Keeping track of uniqueness would not be a problem since, as shown in Section 6.3, it has little to no influence on big configuration sizes.

8.3 Parameter Chain Combinations

This thesis provides an algorithm for identifying both AND and OR type parameter chains attributed to a single vulnerability. However, in real situations there exists a possibility of having multiple vulnerabilities being exposed, producing a combination of both parameter chains. This problem adds a layer of difficulty by requiring the identification of separate chains, where interdependencies only exist within components of the same chain. An extension to this research, or a combination of the introduced strategies, could provide a technique capable of dealing with these cases.

Bibliography

[1] Thomas Bäck. *Evolutionary Algorithms: In Theory and Practice*. Oxford University Press, New York, NY, USA, 1996.

[2] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 14–21, 1987.

[3] Raymond Blockmon. *CEH V9: Certified Ethical Hacker Version 9 Practice Tests*. SYBEX Inc., Alameda, CA, USA, 1st edition, 2016.

[4] Haifeng Chen, Guofei Jiang, Hui Zhang, and Kenji Yoshihira. Boosting the performance of computing systems through adaptive configuration tuning. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 1045–1049, 2009.

[5] International Business Machines Corporation. *2013 IBM Cyber Security Intelligence Index*, 2013 (accessed Mar, 2017). `http://www-935.ibm.com/services/us/en/security/infographic/cybersecurityindex.html`.

[6] National Infrastructure Advisory Council. *Common Vulnerability Scoring System (CVSS-SIG)*, 2015 (accessed Apr, 2017). `https://www.first.org/cvss`.

[7] Michael Crouse and Errin W. Fulp. A moving target environment for computer configurations using genetic algorithms. In *Configuration Analytics and Automation (SAFECONFIG), 2011 4th Symposium on*, pages 1–7. IEEE, 2011.

[8] Pedro A. Diaz-Gomez and Dean F. Hougen. Initial population for genetic algorithms: A metric approach. In *Proceedings of the International Conference on Genetic and Evolutionary Methods*, pages 43–49, Norman, OK, USA, 2007.

[9] Dissent. *Misconfigured database may have exposed 1.5 million individuals PHI: researcher (UPDATE2)*, 2015 (accessed Mar, 2017). `https://www.databreaches.net/misconfigured-database-may-have-exposed-1-5-million-individuals-phi-researcher-2/`.

[10] Larry J. Eshelman, Richard A. Caruana, and J. David Schaffer. Biases in the crossover landscape. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 10–19, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[11] David Evans, Anh Nguyen-Tuong, and John Knight. Effectiveness of moving target defenses. In Sushil Jajodia, editor, *Moving Target Defense: An Asymmetric Approach to Cyber Security*, chapter 2, pages 29–48. Springer, 2011.

[12] Glenn A. Fink, Jereme N. Haack, A. David McKinnon, and Errin W. Fulp. Defense on the move: Ant-based cyber defense. *IEEE Security Privacy*, 12(2):36–43, Mar 2014.

[13] Terence C. Fogarty. Varying the probability of mutation in the genetic algorithm. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 104–109, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[14] Errin W. Fulp, Howard D. Gage, David J. John, Matt R. McNiece, William H. Turkett, and Xin Zhou. An evolutionary strategy for resilient cyber defense. In

2015 IEEE Global Communications Conference (GLOBECOM), pages 1–6, Dec 2015.

[15] David E. Goldberg. Sizing populations for serial and parallel genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 70–79, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[16] Stanley Gotshall and Bart Rylander. Optimal population size and the genetic algorithm. 2000.

[17] Deepti Gupta and Shabina Ghafir. An overview of methods maintaining diversity in genetic algorithms. *International Journal of Emerging Technology and Advanced Engineering*, 2(5):56–60, May 2012.

[18] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 2 edition, 1992.

[19] John H. Holland. Building blocks, cohort genetic algorithms, and hyperplane-defined functions. *Evolutionary Computation*, 8(4):373–391, December 2000.

[20] David J. John, Robert W. Smith, William H. Turkett, Daniel A. Cañas, and Errin W. Fulp. Evolutionary based moving target cyber defense. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 1261–1268, New York, NY, USA, 2014. ACM.

[21] Kenneth A. De Jong and William M. Spears. Using genetic algorithms to solve np-complete problems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 124–132, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[22] Emre Kiciman and Yi-Min Wang. Discovering correctness constraints for self-management of system configuration. In *Proceedings of the IEEE International Conference on Autonomic Computing (ICAC)*, 2004.

[23] David Levine. Commentarygenetic algorithms: A practitioner's view. *INFORMS Journal on Computing*, 9(3):256–259, 1997.

[24] Brad L. Miller and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.

[25] Melanie Mitchell. Genetic algorithms: An overview. *Complexity*, 1(1):31–39, 1995.

[26] Jeffrey C. Mogul. The case for persistent-connection http. *SIGCOMM Comput. Commun. Rev.*, 25(4):299–313, October 1995.

[27] Caroline A. Odell. Using genetic algorithms to detect security related software parameter chains. Master's thesis, Wake Forest University School of Arts and Sciences, Winston-Salem, NC, USA, May 2016.

[28] National Institute of Standards and Technology. *National Checklist Program Repository*, (accessed Apr, 2017). `https://nvd.nist.gov/ncp/repository`.

[29] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems*, volume 4 of *USITS'03*, Berkeley, CA, USA, 2003. USENIX Association.

[30] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. Mulval: A logic-based network security analyzer. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, Berkeley, CA, USA, 2005. USENIX Association.

[31] Colin R. Reeves. A genetic algorithm for flowshop sequencing. *Computers and Operations Research*, 22(1):5–13, January 1995.

[32] Colin R. Reeves and Jonathan E. Rowe. *Genetic Algorithms - Principles and Perspectives*. Kluwer Academic Publishers, 2003.

[33] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[34] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3 edition, June 2012.

[35] Robert W. Smith. Evolutionary strategies for secure moving target configuration discovery. Master's thesis, Wake Forest University School of Arts and Sciences, Winston-Salem, NC, USA, May 2014.

[36] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. Autobash: Improving configuration management with operating system causality analysis. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 237–250. ACM, 2007.

[37] Kanta Vekaria and Chris Clack. Selective crossover in genetic algorithms: An empirical study. In *Proceedings of the 5th Conference on Parallel Problem Solving from Nature*, pages 438–447. Springer-Verlag, 1998.

[38] Michael D. Vose. A closer look at mutation in genetic algorithms. *Annals of Mathematics and Artificial Intelligence*, 10(4):423–434, 1994.

[39] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with peerpressure. In *Proceedings of the*

6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[40] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang. Strider: a black-box, state-based approach to change and configuration management and support. *Science of Computer Programming*, 53(2):143–164, 2004. Topics in System Administration.

[41] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, pages 77 – 90, Berkeley, CA, USA, 2004.

[42] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

[43] Tao Ye and Shivkumar Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 03)*, pages 196–205, 2003.

[44] Fan Zhang, Junwei Cao, Lianchen Lu, and Cheng Wu. Performance improvement of distributed systems by autotuning of the configuration parameters. *Tsinghua Science and Technology*, 16(4):440–448, 2012.

[45] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. *SIGPLAN Not.*, 49(4):687–700, February 2014.

[46] Wei Zheng, Ricardo Bianchini, and Thu D. Nguyen. Automatic configuration of internet services. In *EuroSys '07: Proceedings of the 2007 conference on EuroSys*, pages 219–229, 2007.

[47] Xin Zhou. Measurements associated with learning more secure computer configuration parameters. Master's thesis, Wake Forest University School of Arts and Sciences, Winston-Salem, NC, USA, 2015.

Curriculum Vitae

Sebastián Ramírez Rodríguez

EDUCATION

Wake Forest University 2015-2017
MSc. Computer Science *Winston-Salem, North Carolina*

Instituto Tecnológico de Costa Rica (ITCR) 2010-2014
B.Sc. Computer Science *Cartago, Costa Rica*

LEADERSHIP ACTIVITIES

Computer Science Student Association October 2012-October 2013
President *ITCR*

· Led an executive board of 9 people representing all computer science students at ITCR.
· Organized events for companies to approach students in different ways.
· Motivated new students to pursue a computer science major.

AIESEC Costa Rica May 2012 - September 2012
Local Committee Vice President - Marketing *ITCR*

· Adapted attractive proposals for students to be part of exchange programs with the organization.
· Executed campaigns that helped position and promote the organization within campus.

WORK EXPERIENCE

Konrad Group April 2013 - July 2014, July 2014 - July 2015
Junior Web Developer - Senior Web Developer *Cartago, Costa Rica*

· Engineered software solutions for company clients.
· Utilized technologies like ASP.NET, MVC 5, JavaScript, NodeJS, JQuery, AngularJS, HTML and CSS.

Wake Forest University August 2015 - May 2017
Teaching Assistant - Research Assistant *Winston-Salem, NC*

· Assisted professor with student evaluations and laboratory lessons.
· Tutored students from the department through the Computer Science Center.

OTHER

Upsilon Pi Epsilon Honor Society 2016
Member *Wake Forest University*

Association for Computing Machinery 2016
Member *Wake Forest University*