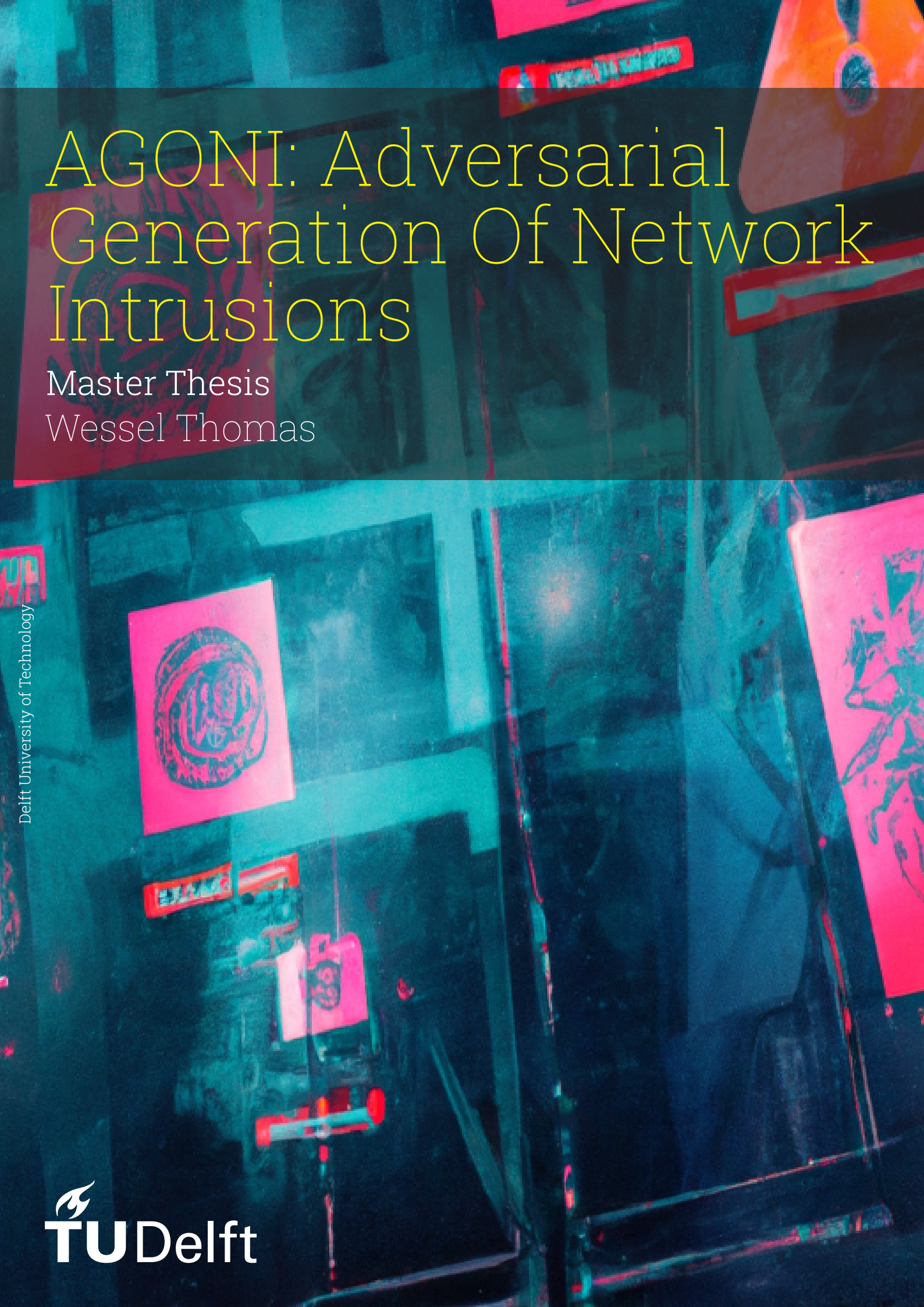


AGONI: Adversarial Generation Of Network Intrusions

Master Thesis

Wessel Thomas

Delft University of Technology

TU Delft

AGONI: Adversarial Generation Of Network Intrusions

by

Wessel Thomas

Student Name	Student Number
Wessel Thomas	4656296

Thesis Committee:	Dr. Ir. S. Verwer	TU Delft, supervisor
	Dr. A. Panichella	TU Delft
	Ir. D. Vos	TU Delft
Project Duration:	September, 2022 - June, 2023	
Faculty:	Faculty of Electrical Engineering, Mathematics & Computer Science, Delft	

TU Delft

Preface

In an era where network security plays an important role in safeguarding sensitive information and critical systems, the development of effective Network Intrusion Detection Systems is of utmost importance. This master thesis presents a novel contribution in the form of a product capable of generating adversarial examples specifically for Network Intrusion Detection Systems. These adversarial examples serve as valuable tools for evaluating the robustness of defense systems against potential threats and attacks.

I would like to express my deepest gratitude to my supervisors, Sicco Verwer and Daniël Vos, for their invaluable guidance and expert knowledge throughout my research. Their mentorship has been an amazing help in shaping the direction and scope of this thesis. Their encouragement and insightful feedback have helped me to learn an incredible amount.

I would also like to show my heartfelt appreciation to all the people of the cybersecurity group, whose welcoming environment with lunches and never-ending coffee-breaks have made this journey an enjoyable experience. The discussions of various topics have brought me both the necessary distractions and helpful ideas for my thesis.

All of my friends, whose company and distractions have brought me great joy and relief, brought me the necessary balance amidst the demanding times during this thesis. To every single on of you I am incredibly grateful. Your support and understanding have been a constant source of encouragement, reminding me that there was more to this life than just my thesis.

Last but certainly not least, I would like to give my biggest thanks to my family. Their unwavering support, love, and belief in my abilities have been the cornerstone of my academic pursuits. Their encouragement and understanding have been an amazing source of strength throughout this journey. I am forever indebted to them for their continuous faith in me.

This thesis would not have been possible without the collective effort, encouragement, and support of all those mentioned above, as well as many others who have played a role, however small, in my journey as a student and as a person. I am very proud to be able to present this thesis, hoping that it contributes to the field of network security and inspires further research in this domain.

I hope you enjoy reading my thesis!

Wessel Thomas

Abstract

Network Intrusion Detection Systems (NIDSs) defend our computer networks against malicious network attacks. Anomaly-based NIDSs use machine learning classifiers to categorise incoming traffic. Research has shown that classifiers are vulnerable to adversarial examples, perturbed inputs that lead the classifier into misclassifying the input. Existing work has shown weaknesses in classifiers for classifying network traffic, but none have shown the possibility of automatically recreating network attacks that can bypass existing anomaly-based NIDSs. Regular methods for generating black-box adversarial examples create packets that are invalid. We present AGONI, a Multi-Objective Genetic Algorithm for generating network packets that are both valid packets and adversarial examples for NIDSs. AGONI can successfully generate valid adversarial examples in multiple attack scenarios. Against the NIDS Suricata, 99.93% of the generated adversarial examples can successfully bypass the defence. We compare the performance of AGONI against randomly generating network packets, the Boundary Attack and an adjusted version of the Boundary Attack which can better create valid adversarial examples. Only AGONI consistently generates valid adversarial examples when compared to the Random Attack (82%), the Boundary Attack (0%) and the Networking Boundary Attack (74%).

Contents

1 Introduction

The increasing reliance on computer networks has led to a significant rise in cyber threats that can compromise the confidentiality, integrity, and availability of network resources. These threats can include malware, network intrusions, denial of service attacks, and other forms of cyber attacks. As a result, organizations have invested in Network Intrusion Detection System (NIDS) solutions to help mitigate these threats and protect their network infrastructure.

One type of NIDS is anomaly-based NIDS. An anomaly-based NIDS relies on Machine Learning (ML) models to categorize incoming traffic into different classes. However, research has demonstrated that ML models are susceptible to adversarial examples, which are instances of a certain class that are slightly altered to cause misclassifications by the ML model. In the context of network traffic, it is a malicious internet packet that looks slightly different from the original, causing an anomaly-based NIDS to perceive it as a non-malicious packet. Attackers can use adversarial examples to exploit anomaly-based NIDSs. This possibly exposes the entire network infrastructure to various network attacks. Attackers could construct adversarial examples for malicious network attacks that a NIDS would no longer be able to stop. To prove the weaknesses of anomaly-based NIDSs, previous studies have concentrated on identifying adversarial examples of malicious network traffic in diverse settings. These studies aim to show the vulnerabilities and incite new research aimed at increasing the NIDSs. Nevertheless, current research has not yet demonstrated how to reproduce network attacks from these adversarial examples and verify that the identified vulnerabilities are practically exploitable.

Genetic Algorithms (GA) find high-quality solutions in a large search space of solutions. GA's evolve their solutions based on a fitness metric to find the best solution. Finding adversarial examples in the search space of all possible network packets is a suitable job for a GA, due to the complexity of the problem and the number of features involved. While existing work has used GAs for generating adversarial examples, they as well did not show how to use the results to reproduce network attacks.

1.1. Research objectives & contributions

The ultimate objective of this line of research is to generate a sequence of packets, which is an adversarial example of a network attack. Not only must this adversarial example evade detection by the system, but the attack functionality must also remain successful. Existing research does not yet show how to generate adversarial examples that represent valid network packets. We perform research focused on the generation of adversarial examples for individual packets that enforces packet validity and have an increased probability to bypass detection systems. The main research question is:

> *How can Genetic Algorithms be used to generate valid adversarial network traffic to effectively evade Machine Learning-based anomaly detection?*

To attempt to construct a complete answer, the thesis is divided into the following research sub-questions:

RQ1. *How do you design a Genetic Algorithm for valid internet packets?*

RQ2. *What effect do different Genetic Algorithms have on generating network packets aimed at anomaly-detection evasion?*

RQ3. *Do attack constraints affect the quality of adversarial examples generated by Genetic Algorithms?*

RQ4. *Do existing NIDSs detect adversarial internet packets?*

RQ5. *How does AGONI perform compared to existing black-box methods for generating adversarial examples?*

1.2. Contributions

In this work, we study the generation of valid adversarial network packets to bypass NIDSs. We conclude that the generation of packet sequences requires in-depth knowledge of network traffic, increasing the difficulty beyond the scope of this research. We propose AGONI: a Multi-Objective Genetic Algorithm designed to produce singular valid internet packets that serve as adversarial examples for anomaly-based NIDSs. AGONI evolves internet packets based on their adversarial distance and their similarity to benign traffic, while continuously applying validity constraints. We study the effect of different operators on the quality of generated packets, showing that a wide set of operators are suitable for generating valid adversarial examples. We test AGONI against the Suricata NIDS where 99.93% of all generated packets bypass Suricata. Finally, we compare AGONI with the black-box methods Random Attack and the Boundary Attack, where we implement the Networking Boundary Attack to better incorporate the domain constraints of network traffic. Among the 4 methods, only AGONI is able to consistently generate adversarial examples representing valid network packets, whereas the other methods have success rates of 82% (Random Attack), 0% (Boundary Attack) and 74% (Networking Boundary Attack).

Our main contributions are:

- A new algorithm for generating valid network packets that bypass NIDSs.
- A study on the influence of different Genetic Algorithm operators on the quality of generated solutions.
- A quantification of AGONI's performance against the Suricata NIDS.
- A version of the Boundary Attack improved for the domain constraints of network traffic, called the Networking Boundary Attack.
- A study on the performance of AGONI compared to other black-box methods.

1.3. Outline

This thesis is structured as follows: In Chapter 2, we provide background information and discuss related work on generating adversarial examples for network traffic. In Chapter 3, we introduce AGONI, the algorithm for generating adversarial examples that represent valid network packets. In Chapter 4, we study the effect of different Genetic Algorithm operators on the solution quality. We evaluate the performance on AGONI with different sets of constraints in Chapter 5. In Chapter 6, we deploy the NIDS Suricata to inspect solutions generated by AGONI. In Chapter 7, we compare AGONI with black-box methods for generating adversarial examples. We discuss remaining topics in Chapter 8 before we conclude in Chapter 9.

2

Literature Review

In this chapter, we explain the relevant topics for this thesis such that any reader can proceed to read the performed research with sufficient knowledge to grasp the nature of the research as well as the implications of the results. Section 2.1 covers the structure of internet traffic and intrusion detection systems. Section 2.2 explains adversarial examples and their distance cost functions. Then we describe what fuzzing is and how evolutionary algorithms work in Section 2.3. Section 2.4 explains how different crossover, mutation and selection operators achieve their results. Finally in 2.5, we discuss the existing works in the field of generating adversarial examples in settings both with and without domain constraints.

2.1. Network Traffic

This section covers the traffic that travels within a computer network. A computer network is a collection of computers that share resources and information. The communication involved in this process generates network traffic. Network traffic is all the data and information that moves around a network during any given time. This traffic consists of many *network packets* with the information needed to deliver (also referred to as *payload*) and the metadata that provides the network with the instructions regarding what to do with one particular network packet. Figure 2.1 displays the structure of a singular internet packet, where the data in the upper layer is the payload that needs to arrive at the other party. All the headers contain fields that describe the state of the packet or add security to ensure packet integrity.

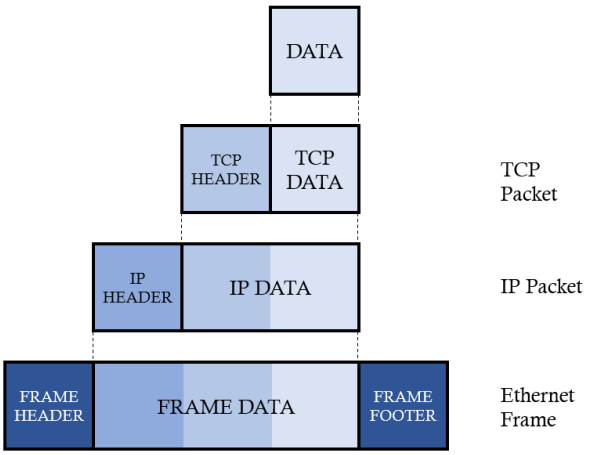


Figure 2.1: A visual representation of the structure of a network packet. The data that needs to be delivered is encapsuled with 3 layers; the Transmission Control Protocol (TCP) header, the Internet Protocol (IP) header and finally the Ethernet frame.

The headers contain information headers vital for the network to deliver the data. The network uses the Ethernet frame for local communication. The Internet Protocol (IP) layer is responsible for the payload

arriving at the correct address globally. Once the payload arrives at the destination, the Transmission Control Protocol (TCP) headers are responsible for proper data delivery.

2.1.1. Ethernet Frame

The Ethernet frame is the outer layer of an internet packet. It is a data unit used for communications within the Data Link layer [1]. The essential properties of an Ethernet frame are the MAC addresses of the sender and receiver, highlighted in Table 2.1. The Data Link layer uses those addresses to identify the devices locally.

The preamble, start frame delimiter (SFD) and frame check sequence (FCS) are indications for the start and end of the Ethernet frame. The FCS value also checks for corrupted data. Additionally, the 'EtherType' indicates the size or protocol type of the encapsulated payload.

Preamble	SFC	**Destination MAC**	**Source MAC**	EtherType	Payload	FCS

Table 2.1: The structure from an Ethernet frame [2]. The destination and source MAC addresses are crucial for delivery, while the other headers help with the packet's security.

2.1.2. IP Packet

Within the Ethernet frame, the IP packet locates devices on a global scale. The internet uses this data unit for communication in the Network layer [1]. This thesis addresses only IPv4 packets and not IPv6 packets due to the limited adoption of the IPv6 protocol. The main features of importance are the IP addresses for the source and destination. These addresses help the network deliver traffic on a global scale. Many more headers are present in the structure of an IP packet, mainly for security-related purposes. We show the packet structure in Table 2.2. We now briefly cover the remaining non-highlighted headers from the IP header.

- **Version**:
  IPv4 or IPv6.
- **IHL**:
  Internet Header Length.
- **DSF**:
  Differentiated Services Field. A field used to identify the level of service a packet receives in the network.
- **Total Length**:
  The length of the IHL and payload length combined.
- **Identification**:
  A value unique to the packet used to re-assemble fragmented packets.
- **Flags**:
  The values that indicate to the network if the packet is fragmented and has multiple parts that will follow.
- **Fragment Offset**:
  The offset of a particular fragment relative to the beginning of the original unfragmented IP payload.
- **Time To Live**:
  An indicator how long the packet should be kept within the network. A use-case for this is preventing packets getting stuck in infinite loops within the network.
- **Protocol**: The protocol used by the Transport Layer [1]. Frequent values are those for the TCP and UDP protocols.
- **Header Checksum**: A value used for error-checking in the packet. Every router recalculates this value and compares it to the checksum in the packet. If the values differ, that indicates there has been an error during the transmission.
- **Options**: Fields that can be used to warrant special treatment of the package or pass extra information.

[1] Find the full list of possible protocol values here: `https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml`

Version	IHL	DSF	Total Length	
Identification			Flags	Fragment Offset
Time To Live		Protocol	Header Checksum	
Source IP Address				
Destination IP Address				
Options				

Table 2.2: The structure of an IP packet header [3]. The source IP address and destination IP address are essential for communication between the correct parties.

2.1.3. TCP Packet

Once the IP protocol finds the destination device, TCP is responsible for a secure data exchange. Table 2.3 highlights the source and destination port where the packet should be delivered. Each port is associated with a specific process or service. Ports allow computers to differentiate between different kinds of traffic. We show the structure of the TCP header in Table 2.3 and briefly discuss the other features within the said table.

- **Sequence Number**:
  This is a value that helps to keep track of how much data has been transferred and received. If a packet has sequence number X and the length of that packet is Y, then the sequence number of the next packet is $X + Y$.
- **Acknowledgement Number**:
  This value is related to the sequence number. The acknowledgement number represents the expected sequence number of the next packet. Following the earlier example, the acknowledgement number would be $X + Y$ of the initial packet.
- **Data Offset**:
  This feature specifies the length of the TCP header.
- **Reserved**:
  These bits of the packet are reserved bit values for future use and should be set to zero.
- **Flags**:
  This feature contains 9 bits that all represent a flag that indicates how the header (or certain features within it) should be treated. The flag names are CWR, ECE, URG, ACK, PSH, RST, SYN and FIN.
- **Window Size**:
  The size of the *receive window*. This specifies the amount of data the sender is currently willing to receive.
- **Checksum**:
  Similar to the checksum in the IP header, this checksum enables checking for errors. There are two different checksums in the entire internet packet to ensure integrity in multiple layers of the network.
- **Urgent Pointer**:
  It indicates how much data in the current segment counting from the first byte is urgent.
- **Options**: Fields that can be used to warrant special treatment of the package or pass extra information.

Source Port			**Destination Port**
Sequence Number			
Acknowledgement Number			
Data Offset	Reserved	Flags	Window Size
Checksum			Urgent Pointer
Options			

Table 2.3: The structure of a TCP packet header [4]. The source and destination port are used for delivery and indicate what kind of service is receiving the packets.

2.1.4. Network-based Intrusion Detection Systems

The majority of network packets is harmless and gets delivered as intended. However, malicious parties try to corrupt internet packages or send packages with altered payloads in order to gain access to network devices. In 1980, John Anderson proposed the idea of an intrusion detection system [5]. Since then, researchers have invented various ways to detect traffic to the network. Such an approach is called a Network-based Intrusion Detection System (NIDS).

NIDS exist in two categories of detection systems: signature-based NIDS and anomaly-based NIDSs [6]. A *signature-based* NIDS takes known network attacks and stores their 'signature' in a database. It then compares all incoming internet traffic to all known signatures looking for a match. If internet traffic matches a malicious signature, the NIDS intercepts that traffic. This approach is effective against known attacks but is powerless against new attack patterns embedded in internet traffic. Because the database has no signature that matches the new attack, the NIDS will not detect it.

A widely used signature-based NIDS is Suricata[2], an open-source signature-based network intrusion-detection engine published in 2009. It has a default set of signatures that is immediately usable. It is also possible to extend this set, or create a new one with custom rules specifically designed for the purposes of one's network. Suricata can operate on an active basis, where it monitors all incoming traffic in real-time, but also passively, where packet captures are analysed after they entered the network. Below, we show two examples of Suricata rules that are part of the default set. The first rule partially verifies the packet validity of all the network traffic. The second rule alerts the network when IP addresses with a poor reputation occur. There is a numerous amount of Suricata rules that check for several different aspects of network traffic to optimise the quality, efficiency and safety of the network.

```
alert pkthdr any any -> any any (
    msg:"SURICATA IPv4 packet too small";
    decode-event:ipv4.pkt_too_small;
    classtype:protocol-commands-decode;
    sid:2200000;
    rev:2;
)

alert ip [1.15.103.197, ..., 103.186.1.55] any -> \$HOME_NET any (
    msg:"ET 3CORESec Poor Reputation IP group 1";
    reference:url,blacklist.3coresec.net/lists/et-opne.txt;
    threshold: type limit, track by_src, seconds 3600, count 1;
    classtype:misc-attack; sid:2525000;
    rev:620;
    metadata:affected_product Any, attack_target Any, deployment Perimeter, tag 3CORESec,
        signature_severity Major, created_at 2020_07_20, updated_at 2022_12_05;
)
```

Anomaly-based NIDSs try to compensate for the weakness in checking only for signatures. The idea is to train a Machine Learning model that tries to identify malicious traffic from the benign. This model should not be a classifier that focuses on distinguishing benign and malicious traffic. It should be an anomaly detector, where the goal is to separate the frequent from the less-frequent data. This anomaly detector inspects all new incoming traffic and intercepts all traffic perceived as malicious. Anomaly-based NIDS are more efficient against unknown internet attacks than the signature-based NIDS, since the anomaly-based systems can recognise regular traffic. Attacks with novel approaches that try to breach the network will often have different traits in their network packets than benign traffic. Therefore, even if those attacks are unknown to the system, the anomaly-based NIDS might still detect them. The weakness of anomaly-based detection is the risk of false positives, meaning that a model classifies benign traffic as malicious. Possible causes are poorly trained models or the occurrence of irregular benign traffic. A high false positive rate hinders the functionality of the network and should always be minimal.

[2] Information and download links are available on `https://suricata.io/`

2.2. Adversarial Examples

The core idea of Machine Learning (ML) is to predict a property of new data with a model created with training data. Modern ML research aims to test the robustness of ML classifiers by attempting to mislead those classifiers by generating *adversarial examples*. To create such an adversarial example, one takes a data point classified as a class *a* by a particular classifier and then applies an alteration to its features such that the classifier classifies it as another class *b*. The alteration is dependent on the data features, the contextual constraints and the goal of the adversarial example. One can make alterations to cause the classifier to classify the adversarial example as anything but the original class *a* or alter the data point to change the classification to a specific class from the data set.

A well-known example of an adversarial example is one generated by Goodfellow, Shlens and Szegedy using their Fast Gradient Sign Method [7], which we show in Figure 2.2. They altered a correctly classified picture of a panda with noise not visible to the human eye. Suddenly, the classifier in question changed its classification for the picture to "gibbon" with a very high confidence percentage. This example shows the importance of testing all classifiers against robustness, to prevent malicious parties from attacking the classifiers and reducing their accuracy and effectiveness for their respective purposes. Such an attack is called an **adversarial attack.**

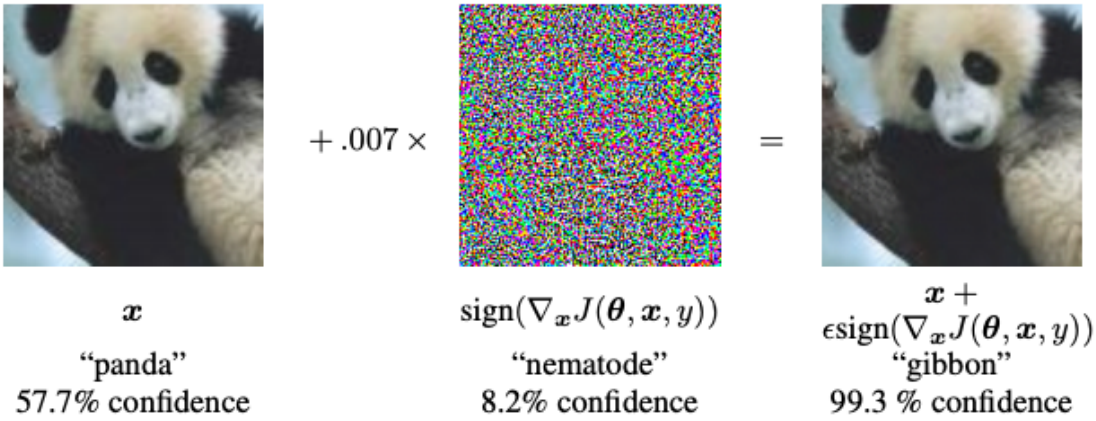


Figure 2.2: An adversarial example generated by adding a little noise to the picture of a panda. The used classifier now perceives the picture as a different class with higher confidence, even though the picture looks no different to the eyes of a human.

2.2.1. Distance cost-functions

In theory, every instance is mutable in such a way it turns into an adversarial example. If we want an instance of class *a* to look like class *b*, we change the original instance until it has all the characteristics of class *b*. This would not be a fair adversarial example since we are giving an instance of class *b* as input. The goal is to input an instance of class *a* that still is very similar to other instances of class *a*. Therefore, we only want to allow limited adjustment to our instance to obtain an adversarial example. We quantify the adjustments made with *cost functions*, which turns generating adversarial examples into an optimisation function. The best adversarial examples are those with the lowest cost. The following functions are the most applied techniques of defining distance (or cost) between an instance with *n* features and its adversarial example [8] (with x_i being the i^{th} feature of an instance and x'_i the i^{th} feature of the altered version):

- L_0 distance: the sum of changed features (also known as the Hamming distance). $\sum_{i=1}^{n} \mathbb{I}_{x_i \neq x'_i}$
- L_1 distance: the sum of the difference between features. $\sum_{i=1}^{n} |x_i - x'_i|$
- L_2 distance: the euclidean distance. $\sqrt{\sum_{i=1}^{n} (x_i - x'_i)^2}$
- L_∞ distance: the maximum feature difference. $max_i |x_i - x'_i|$

The resulting adversarial examples will change depending on the choice of distance metric. The L_0 distance minimises the number of changes made to create an adversarial example while the L_∞ allows more changes as long as they are not too different from the original feature value, since only the maximum feature difference defines the L_∞ distance. There is no universally best metric. The context of the problem changes what metric is most suitable.

2.3. Genetic Algorithms

A **Genetic Algorithm (GA)** is an algorithm that can produce highly optimised solutions in a wide range of problem settings. A GA tries to this result with a methodological approach, where it tries to evolve its data towards the best version. GAs can also find an optimised answer within a search space that is too large to search exhaustively. A simple example would be automatically finding a patch for a program containing software bugs. There are several ways to change a piece of software, but only a few changes will alter the code so that the program behaves as intended.

2.3.1. Fuzzing

Fuzzing is an automated software testing process that provides randomised invalid or unexpected data as inputs to a software program. It provides this randomized data in the hopes of triggering faulty or unexpected behaviour within the program. Researchers make a distinction between *dumb* and *smart* fuzzing. *Dumb* fuzzing relies on randomness. It changes existing input values on an arbitrary basis without an understanding of the input data structure. *Smart* fuzzing uses domain-specific knowledge of the targeted program. Understanding the program design and generating the new inputs takes more time than dumb fuzzing, but smart fuzzing provides greater coverage of the solution search area compared to random fuzzing as it is likely to adhere to the programs' data structure requirements.

A dumb fuzzer generates completely random input for a program. Three types of fuzzing techniques improve upon this, called *mutation-based* fuzzing, *generation-based* fuzzing and *evolutionary* fuzzing [9]. Mutation-based fuzzing mutates existing input to create the one, with no awareness of input constraint the program might enforce. Generation-based fuzzing creates input from scratch and does know these input constraints. Therefore, it requires a-priori knowledge of the program, but the generated input gets validated quicker. Finally, evolutionary fuzzing is explained more in-depth in the remainder of this section.

A final distinction is the degree of available knowledge and use of static analysis on the targeted program's source code [10, 11]. Fuzzing with full knowledge of the program's source code is called *white-box* fuzzing. *Black-box* fuzzing has zero knowledge of program internals and *grey-box* fuzzing uses some knowledge for generating input.

2.3.2. The concept of GAs

GAs follow the concept of the theory of evolution that states that given a population with limited resources, the conflict for those resources results in 'survival of the fittest', meaning that only the individuals from the population that are most fit to survive get to live [12]. So given a problem that has an enormous set of possible solutions, we want to find the best option from that set by evolving our solution towards the best 'fitness'. For the earlier example, the fittest solution would pass all available tests written for that piece of code.

To design a GA, it needs to determine what solutions from the population (also referred to as *individuals*) are 'fitter' than the others within that population. So, one must have a well-defined fitness function that gives a score to each individual indicating how well it solves the problem at hand. According to that fitness function, there is a space that maps every possible individual to a score. Displayed beneath in Figure 2.3, a search space is illustrated for a problem with only two variables where the fitness score represents a function of those two variables. If such a search space would be flat with a lot of solutions with the same fitness score, a GA has difficulties finding the correct direction to evolve towards since all directions have the same fitness. The GA then has no indication of what solutions are better and must randomly guess what direction is better.

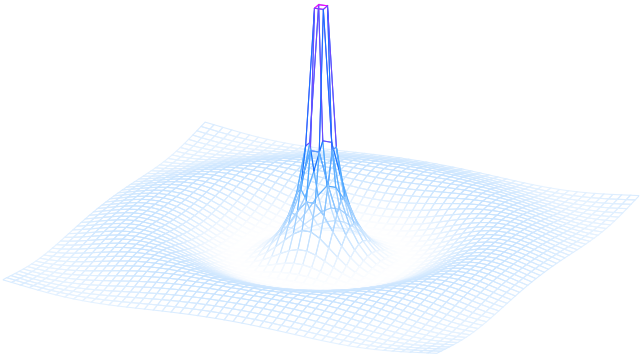


Figure 2.3: Example of a possible search space where the higher areas represent better solutions to the problem. With a non-flat search space like this, a GA can easier evolve towards the direction that increases its fitness and find the best solution.

2.3.3. How do GAs evolve their individuals?

With a properly defined fitness function, a GA is able to evolve its individuals to improve the final solution. The main idea consists of 4 steps. These steps can be altered and tailored to fit the exact problem one is dealing with. The steps are as follows:

- Initialisation
- Selection
- Mutation
- Crossover

For the **initialisation**, the GA creates a randomly sampled set of individuals, called the initialisation of the population. The GA samples from the search space of all possible solutions. If one increases the size of their population, the algorithm will be more likely to find individuals with better fitness. The downside of a larger population is an increase in computing time.

Next is **selection**, where the algorithm tries to find the fittest individuals of the current generation. These individuals evolve into the following generations during the next steps. Popular approaches for selection are 'Tournament selection' [13] and 'Roulette wheel' [14]. Tournament selection samples n individuals and lets them 'compete' in a tournament against each other, where the winner becomes part of the next generation. The algorithm repeats this process until it has sampled enough individuals to fill the population for the next generation. The roulette wheel gives each individual from the current population a percentage which represents the chance that the wheel picks them for the next generation. Better fitness for an individual will result in a higher chance for that individual. The algorithm then samples the new generation according to the percentages for all individuals of the current one. There are numerous other ways to perform selection [15], with all available methods having their own pros and cons that are of great significance when designing an evolutionary algorithm.

After the selection phase, the algorithm will start evolving its current solutions. It does so by first applying **mutation** to all individuals. A mutation is a small change to an individual, with the goal of creating a better solution. The exact mutation operations vary depending on the context of the problem. Often people default to either adding pieces to the solution, deleting them or changing the value of a subset of the solution. The mutation process performs these operations randomly, so the quantity of mutations one individual receives varies. In Table 2.4, we provide examples of how mutation changes a solution. After applying mutations to the current individuals, the last step of evolving the current generation is the **crossover** operation. A crossover takes two individuals and combines them into two new individuals who have one part from each of the original individuals. There are various ways to perform crossover, with popular methods being one-point crossover [16], multi-point crossover [17] and uniform crossover [17]. In Table 2.5, we show examples of how these methods work. A one-point crossover takes a point (for the first example, the middle), splits the individuals at that point and recombines the resulting parts into two new individuals with the same length as the originals. Multi-point follows the same idea but slices the individuals at multiple points (for the second example, two points). Finally, for uniform crossover, the parts from the two individuals are distributed among the two new individuals according

Table 2.4: Examples of mutation, slightly altering examples in various ways (alteration, addition, deletion).

Pre-mutation	Post-mutation
[A-A-A-A]	[A-B-A-A]
[A-A-A-A]	[A-A-A-A-A]
[A-A-A-A]	[A-A-A]

Table 2.5: Examples of crossover, combining the solutions according to various patterns (one-point, multi-point, uniform).

Pre-crossover	Post-crossover
[A-A-A-A] [B-B-B-B]	[A-A-B-B] [B-B-A-A]
[A-A-A-A] [B-B-B-B]	[A-B-B-A] [B-A-A-B]
[A-A-A-A] [B-B-B-B]	[A-B-A-B] [B-A-B-A]

to a percentage. To clarify, a section from an individual has a chance of x% to be assigned to the first new individual and a $(100 - x)$% chance to be assigned to the second.

These four steps have created the next generation. From this new generation, the fittest are again selected for the mutation and crossover process, to again create a generation. The algorithm repeats these steps until it reaches an exit condition. Examples of such conditions are that the algorithm stops after *n* generations or that it stops once it has *converged*. The algorithm has converged when all individuals in the population have the same fitness with similar solutions and no improvement happens in all the fitness scores for several generations. The type of operators chosen for selection, crossover and mutation influence the *convergence rate*, that speed that the GA closes in on a converged state.

2.3.4. Multi-Objective Genetic Algorithms

For certain problems, the quality of the solutions gets measured by a singular metric (e.g., how many tests succeed for a piece of software). However, other problems are not as simple and multiple metrics exist. A problem example happens when buying a car. You want to buy the fastest car you can get, but on the other hand you want to minimise the cost of the car. It is hard to prioritise solutions (or in this case 'cars') when the available fitness metrics indicate different solutions as the best (e.g., a car with higher speed but higher cost versus a car with lower speed but also decreased cost). A Multi-Objective Genetic Algorithm (MO-GA) searches for optimal solutions accounting for multiple metrics. Per generation, the MO-GA evolves its individuals to create solutions that *dominate* the previously acquired solutions. A solution *dominates* another solution when all their fitness metrics outperforms the metrics of the other. Looking back at the car example, one would prefer the cheaper and faster Car A over the more expensive and slower Car B. Car A dominates Car B in this setting.

A MO-GA wants to keep track of all non-dominated solutions, also known as the *Pareto-efficient* solutions. The set of all Pareto-efficient solutions is called the *Pareto front*. Below in Figure 2.4, we show an example of such a Pareto-front for the car-acquisition problem. Every datapoint represents a car with a certain speed and cost that is not dominated by any other points. To pick a final solution, one must decide which metric bares more importance in the specific use case for the car. The same goes for any other multi-objective optimisation problem, where one must pick the final solution based on the context of the problem.

2.4. GA Operators

The process of creating a new generation consists of the selection phase, followed by the crossover and mutation phase. For every phase, there exist various approaches. This section covers the operators for every phase included in this thesis to generate valid adversarial examples. This selection is not an exhaustive representation of all operators designed within the research field of GAs. It is a selection based on existing literature and review studies indicating what operators are frequently used and acknowledged.

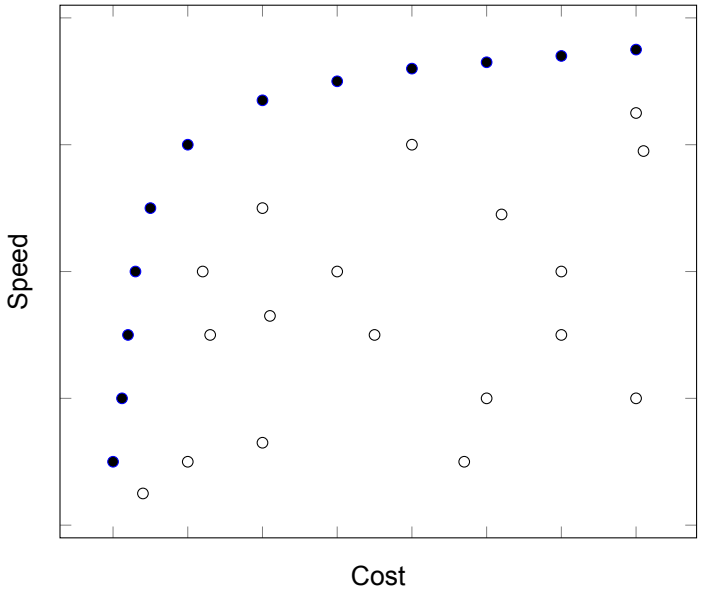


Figure 2.4: An example of a Pareto-front corresponding to the car example. The blue dots create the Pareto-front and represent the cars that are non-dominated in speed and cost.

2.4.1. Crossover Operators

There are various ways to perform a crossover operation on individuals. We selected a subset of popular crossover operators [18], fitting for the generation of network packets. The selected operators are the *Single Point* crossover operator, the *Multi-Point* operator, the *Uniform* operator, the *Arithmetical* operator and the *Wright* operator [19].

Section 2.3 covered the steps behind the Single Point, Multi-Point and Uniform operators. Here we will briefly cover how the Arithmetical and Wright crossover operators function. The *Arithmetical* crossover operator works only on individuals with values that exceed those of one bit. Based on the values of every feature, Arithmetical crossover calculates two new individuals with feature values that lie in between the values of the old individuals. The parameter α determines where in between the new individuals lie. With feature x_1 and x_2 from two individuals from the previous generation, we determine the new features x_1' and x_2' for the individuals of the new generation with the following formulas:

$$x_1' = x_1 + \alpha * (x_2 - x_1)$$
$$x_2' = x_1 + (1 - \alpha) * (x_2 - x_1)$$

In the algorithm, the value of α is randomly set in a range of 0 and 1 per iteration of applying crossover. To illustrate the Arithmetical crossover operator, Table 2.6 shows the results of applying the operator with an α value of 0.25. The added value of this operator over the previous two, is that it does not simply copy the values from the old features, but creates new values which increases the search power of the algorithm. The downside that comes with it, is the difficulty maintaining the values that heavily increase the fitness of individuals.

Table 2.6: Example of the Arithmetical crossover operator, creating 2 new individuals with $\alpha = 0.25$.

Individual	Pre-mutation	Post-mutation
x_1	[10-14-20-0]	[12-11-25-6]
x_2	[18-2-40-24]	[16-5-35-18]

The *Wright* crossover operator uses the fitness of the old individuals to determine what values are more likely to be passed on to future generations. When performing crossover on individuals I_1 and I_2, with I_1 having the superior fitness score, the following applies while creating new individuals I_1' and I_2':

- Step 1: Every feature from I_1 is assigned **once** to either I'_1 or I'_2.
- Step 2: The remaining missing features for I'_1 and I'_2 are randomly assigned from the features of I_1 and I_2. The greater the difference is between the fitness of I'_1 and I'_2, the larger the chance is that the features from I_1 are chosen again.

In Table 2.7, a possible scenario is depicted when using the Wright operator on two individuals that have features $I_1 : [A - B - C - D]$ and $I_2 : [1 - 2 - 3 - 4]$. Step 1 shows the features of the better individual I_1 being divided between x'_1 and x'_2, with a _ representing an unfilled feature. Step 2 fills the empty slot with randomly selected features from either I_1 or I_2, causing the final versions of x'_1 and x'_2 to consist mostly of features from the individual with the better fitness, I_1. The added value of the Wright operator is its capability to maintain the feature values that increase the fitness. Its potential weak side is that this operator heavily increases the convergence rate, due to it constantly picking the majority of the superior features.

Crossover should not be applied all of the time, since a 100% crossover risk discards all the best-performing solutions and creates an entirely new generation. Researchers have tried to determine the preferred crossover rate with various results. For using one-point crossover, the recommended crossover rate is 0.6 [20], while for two-point crossover it is 0.95 [21]. Ideally, we would have liked to perform extensive experiments to determine the optimal crossover rate per operator, but due to time constraints we adapted the recommendation from De Jong [20] of a 0.6 crossover rate.

Table 2.7: Example the Wright crossover operator creating 2 new individuals.

Individual	Step 1	Step 2
x'_1	[A-_-_-D]	[A-2-C-D]
x'_2	[_-B-C-_]	[1-B-C-D]

2.4.2. Mutation Operators

Similar to crossover, there are various mutation operators to choose from [22]. The selected operators are the *1&0 exchange* operator, the *swapping* operator, the *reversion* operator, the *one-point* operator and finally the *uniform* crossover. Since we are bound by the constraints of a network packet, it is not possible for the mutation operators to delete or add features. Every operator is implemented to only perform alteration mutations.

The *1&0 exchange* operator, further simplified to *exchange* operator, exchanges all the bits of a certain feature to their inverted value. It is a very aggressive mutation operator that changes the features entirely. The *swapping* operator takes two bits of the feature value in random places and swaps their positions. This operator is significantly less aggressive than the exchange operator. Third is the *reversion* operator, which takes a random substring of the binary representation of a value and reverses the bits. The aggressiveness of reversion is dependent on the size of the selected substring, with bigger sizes resulting in bigger changes in the value. Then the *one-point* operator, that takes 1 randomly selected bit from a feature value and flips it, which mutates less aggressively compared to the other operators. Finally, the *uniform operator* which iterates over all the bits and probabilistically flips them. A similar approach to the exchange operator, but significantly less aggressive. Table 2.8 shows worked out examples of the way all the operators change the values of features.

Table 2.8: Examples for all the stated mutation operators. Blue highlighted bits represent the bits affected by the operator.

	Pre-mutation	Post-mutation
Exchange	**00001111**	**11110000**
Swapping	0**0**001**1**1	0**1**001**0**1
Reversion	00**001**111	00**1100**11
One-point	0000**1**111	00**1**01111
Uniform	**0**000**11**11	**1**00**100**11

An important decision for mutation operators is the value of the mutation rate. Numerous studies exist recommending various values for the mutation rate. Recommendations range between set values in between 0.001 and 0.1 [20, 23, 24, 21] to values that scale with the population size [25] or adapt during the execution of the algorithm [26]. A common consensus is that the mutation rate should not be too high, since it alters the algorithm to resemble random search which defeats the purpose of evolving solutions towards an optimum [20]. Similar to the crossover rate, we would have preferred to determine the optimal mutation rate per mutation operator, but this plan was deserted due to the afore-mentioned time constraints. The mutation rate for this algorithm is set to 0.001, again following the recommendation of De Jong [20] due to it being a commonly used value showing results of high quality.

2.4.3. Selection Operators

Finally, we define the set of selection operators that pick the set of individuals to be used for creating the next generation. The selected selection operators are called *roulette wheel*, *stochastic universal sampling*, *tournament*, *linear rank*, *exponential rank* and *truncation* selection. These selection operators are taken from the study of Jebari and Madiafi [27], stating that they are well-known and commonly used.

The roulette wheel operator selects the individuals in a probabilistic fashion. Every individual receives a probability to be picked based on its fitness, resulting in better solutions having a higher chance to be selected for the next generation. All these probabilities together create a 'roulette wheel' that gets spinned until enough individuals are selected. Figure 2.5 shows such a roulette wheel with dummy percentages, that give Individual 1, 2 and 6 a high percentage to be selected, whereas Individual 3, 4 and 5 have significantly less change to be picked.

The stochastic universal sampling method is very similar to the roulette wheel. It assigns the same probabilities to the individuals but takes a different approach to sampling the new generation. Instead of spinning the wheel repeatedly, it spins it once and then determines the rest of the sampled features by universally sampling over the roulette wheel. To better illustrate this, Figure 2.6 shows the stochastic universal sampling of 6 individuals over a roulette wheel. One line is determined stochastically and the others are evenly split over the rest of the wheel to sample the desired number of individuals.

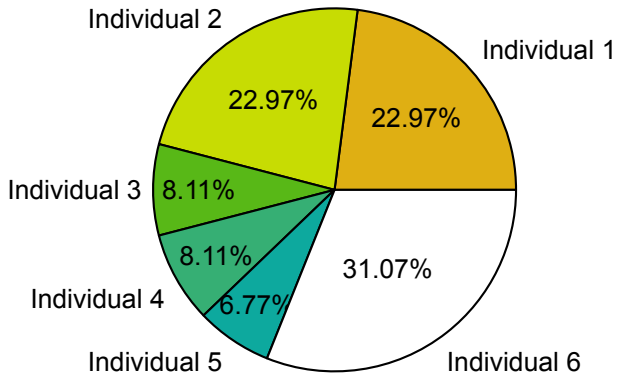


Figure 2.5: An example representation for a roulette wheel used to select new individuals, with every part representing the likelihood a particular individual will get selected for the next phase.

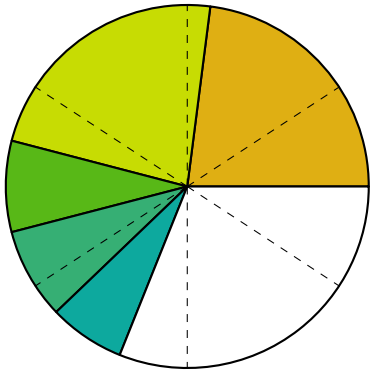


Figure 2.6: An example representation for a roulette wheel with stochastic universal intervals where 6 new individuals needed to be sampled.

Tournament sampling makes the individuals compete for the right to be sampled. It repeatedly samples a percentage of the individuals and then selects the individual with the best fitness as the 'winner of the tournament'. This continues until enough individuals have been sampled for the next generation. The size of the tournament determines the convergence rate of the algorithm. With a large tournament size, the new generation for a majority will consist of the fittest individual. With smaller tournament sizes, the sampling starts to resemble random sampling. Historically, research has preferred small tournament

sizes [28, 29, 30]. While these small values were popular, usually works take these values without much justification [31]. Unfortunately experiments for determining the optimal population size are out of scope, so we set the tournament size to 2 based on existing works stating it to be the most popular setting for tournament selection [31, 32].

2.5. Related Work

This section highlights all recent existing works for generating adversarial examples and their use in attacking NIDSs. We show all the various domains that suffer from a decreased performance in an adversarial setting, specifically how the domain of network intrusion detection is evaded with adversarial examples crafted both with and without domain constraints.

2.5.1. Black-box adversarial examples

For generating adversarial examples in a black-box setting, we cover the Random Attack and the Boundary Attack.

A simple and efficient way of generating adversarial examples is the randomised fuzzing of network packets, which we call the **Random Attack**. This approach has no structure but takes little computational time to generate solutions, allowing for a large number of solutions in little time in hopes of finding a successful adversarial example. To apply the Random Attack on the context of network packets, we generate the features within the valid range of each feature. Generating the values outside of this range has no merit and only increases the likelihood that the resulting packet is invalid.

The **Boundary Attack** creates a bad adversarial example with a high adversarial distance and then gradually moves it into the direction of the original instance. The attack checks for every step if the resulting position is still an adversarial example before taking another step. After a certain amount of steps, the attack reaches a point that is no longer an adversarial example, meaning it has found the border. After finding the border, the attack tries to randomly move along this border to find adversarial examples that are even closer to the original instance. If the Boundary Attack gets stuck, it adjusts the step size to get closer to the border. To get a better understanding, Figure 2.7 illustrates the process of the Boundary Attack.

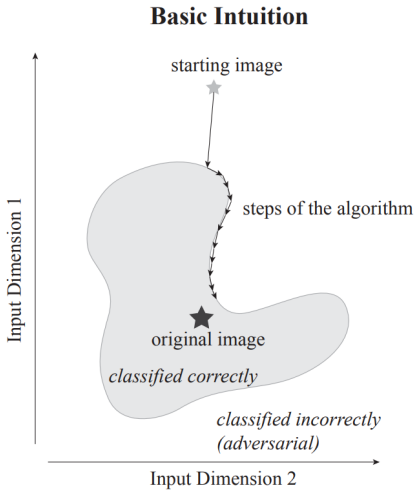


Figure 2.7: The Boundary Attack decreasing the adversarial distance step by step by moving along the classification border [33].

2.5.2. Adversarial examples for various domains

Biggio et al. [34] and Szegedy et al. [35] first showed in their respective papers showing the vulnerabilities of deep neural networks for adversarial examples. Since then, researchers have exposed the vulnerabilities of many other ML models. As stated, deep neural networks are vulnerable [7, 35], as are linear models, decision trees and support vector machines [36, 37]. The scope of the issue persists across various domains. Image processing has been repetitively shown to contain vulnerabilities [7, 38, 39, 40], which happened as well in malware detection [41, 42], text [43, 44] and speech [45].

2.5.3. Unconstrained adversarial examples for NIDS

Studies show that NIDSs are no exception and are just as vulnerable to adversarial examples as the aforementioned domains. The approach of the following studies allows all perturbations for adversarial examples, even if they would result in invalid traffic.

Warzyński and Kołaczek [46] use the Fast Gradient Sign Method (FGSM) from Goodfellow et al. [7] to attack neural networks trained on the NLS-KDD dataset[3] in a white-box setting. They show complete misclassification for malicious instances in an adversarial setting.

Rigaki [47] also uses FGSM as well as the Jacobian Saliency Map Attack (JSMA) [40] to attack linear classifiers. A set of targeted classifiers showed a substantial decrease in performance when faced with an adversarial setting. Wang extended this paper with attacks based on DeepFool [39] and C&W [38] on a neural network, which confirmed the vulnerabilities for these attacks. He conducted analysis of the relation between adversarial attacks and features of traffic records. In both these studies, the authors performed the experiments in a white-box setting.

Yang et al. [48] target deep neural networks with several attacks and do so in a black-box setting. They generate adversarial examples with C&W used on a substitute model, zeroth order optimisation [49] (ZOO) and a generative adversarial network [50] (GAN). The resulting performance implies that adversarial attacks are possible without knowing the internal state.

Aiken and Scott-Hayward launch adversarial attacks on a self-developed NIDS in a black-box setting. This system uses a k-means clustering algorithm. Since the adversary does not know the features the NIDS uses, the authors state that adversaries can approximate a limited set of features to adjust, depending on the attack the adversary aims to hide from the NIDS.

2.5.4. Constrained adversarial examples for NIDS

While the previous studies show vulnerabilities in existing techniques that NIDS utilise, they do not take constraints into account. They can generate various adversarial examples to bypass their respective NIDS. However, there is no guarantee that those adversarial examples could practically attack a network. The following studies aim to improve realism by either constraining the actions of the adversary or the values an adversarial example can take. These studies will match the closest to the content of this research, so we will go more into depth and discuss the knowledge gaps these papers leave that we aim to fill.

Hashemi (2019)

Hashemi et al. generate adversarial examples for packet-based NIDS by severely restricting the actions of the adversary [51]. They can split packets into multiple ones, as long as relevant headers like the acknowledgement number are updated accordingly. Delaying the time between outgoing packets is also allowed, which is a simple operation that still guarantees the validity of the traffic. Finally, the adversary can inject packets into the traffic that get ignored by the victim but processed by the NIDS. An example would be a packet with an invalid acknowledgement number. For all three operations, the proposed method uses a copy model to evaluate different parameter values. For the delay operation, they find the length of the delay by performing a binary search between 0 and 15 seconds to see if any value is able to fool the copy model.

[3] NLS-KDD dataset, available on `https://www.unb.ca/cic/datasets/nsl.html`

In their experiments they show that the detection rate of NIDS can be lowered by up to 70% in a white-box setting and that the constraints imposed on the adversary do not make it impossible to craft adversarial examples. As mentioned in their paper, the computational complexity for this approach is high. Therefore, the authors used a subset of their full dataset for their experiments.

While the authors do ensure that the adversarial examples represent valid traffic that does not break underlying network protocols, the white-box setting decreases the realism of this attack. An adversary having an exact copy model of an NIDS is unlikely. The paper also states that all the packets must carry out their malicious intent effectively to be considered a successful adversarial example, yet there is no coverage on if the functionality from the perturbed packets was preserved.

Lin (2022)
Lin et al. developed IDSGAN [52] to generate adversarial malicious traffic to launch an evasion attack against an IDS. They use a neural network as a generator to create the adversarial examples. A second neural network acts as a discriminator, classifying traffic as malicious or benign. The authors created a black-box setting by training the discriminator network with queries to the defense NIDS that they want to evade. Not only can the IDSGAN be used in a black-box setting, they can also update it for modern traffic by performing additional queries at a later time to retrain the discriminator.

To preserve the functionality of the generated traffic, they use a mechanism that restricts the modification to features of the original malicious traffic records that do not alter the intended functionality. They use the dataset NLS-KDD, that distinguishes between 4 different sets of features. Per attack in the dataset the authors indicate which feature set cannot be changed. The generator is then free to alter the features from the other feature sets in order to create an adversarial example.

While the authors claim that the generated traffic should be usable for a real network attack, the paper does not show how to construct traffic based on their results. So the question regarding how one should construct a real attack remains unanswered. From the approach, some inconsistencies stand out. They alter features independently, meaning that features whose values are dependent on each other might receive faulty values from IDSGAN. The modified features are normalised to values between 0 and 1, implying a minimum and maximum value for features when creating real packets.

Sheatsley (2022)
Sheatsley et al. created the algorithm Adaptive-JSMA [53] that adheres to the domain constraints of network packets. They extract a set of constraints from a network dataset that guarantees validity to all instances that obey those constraints. They differ between primary features and secondary features. A primary feature dictates the values of the other features based on its own value. A secondary value has no influence on the values of other features. For all these constraints, they do not only consider what the range of values is but also what the likelihood of every value is.

The authors crafted all the constraints in this paper manually. They used their understanding of the domain and observations of the data to identify the constraints of the generation of adversarial examples. They also argued that primary features could likely be inferred by the ranking of most correlated features. If directly correlated features are filtered (e.g. two features that are each other's inverses), the primary features should be able to be found systematically.

Adaptive-JSMA generates adversarial examples that fall within the said constraints. With this set of constraints, the expectation is that NIDSs are more robust since they reject all perturbations that exceed those constraints. However, the results from this study show they can generate an adversarial example with a success rate of 95%.

In real scenarios, the adversary cannot always alter every feature. The paper shows experiments for generating adversarial examples with a continuously decreasing size for the set of features that the adversary could change. With only 5 features available, creating an adversarial example still had a 50% succes rate.

3

Creating a Genetic Algorithm

In this chapter, we answer RQ1; *How do you design a Genetic Algorithm for valid internet packets?* We present a black-box Multi-Objective Genetic Algorithm called AGONI, i.e., an evolutionary smart fuzzing algorithm used to generate valid adversarial examples for malicious network packets to bypass an NIDS. AGONI is designed to generate individual network packets, enforcing the validity of the packets.

This chapter is structured as follows: we explain the high-level design of the algorithm and how it generates valid network packets that are adversarial examples in Section 3.1. Section 3.2 describes the structure of the individuals that represent the network packet. Then, we define the criteria that the GA uses as a fitness function in Section 3.3. Section 3.4 explains how the algorithm ensures the validity of the solutions. Finally, Section 3.5 discusses an example run of AGONI to show how the algorithm reaches its answers.

3.1. Design of the GA

As stated in Section 2.3, there are various approaches towards fuzzing for designing an algorithm that generates network packets. While dumb fuzzing can be very powerful, we need individuals to represent valid network packets, meaning we have to consider a substantial set of constraints. Therefore, the choice fell on a smart fuzzing approach. Mutation-based and generation-based fuzzers, unlike evolutionary fuzzers, require no feedback per generated individual, which requires fewer computations. However, said feedback is a good metric for evaluating the overall performance of the particular fuzzer. We want to understand what kind of generated packets receive better fitness functions, not just to circumvent the defences of NIDSs, but to showcase possible weaknesses of the current state-of-the-art intrusion detection systems. Therefore, we opted for evolutionary fuzzing. The final decision was whether to develop the algorithm in a white-box, grey-box or black-box setting. While white-box and grey-box models might yield better results due to their increased knowledge and use of static analysis, they require knowledge about the model that can be hard to acquire. To circumvent the NIDS on any network to the best of our capabilities, we created a black-box model.

Figure 3.1 shows the high-level overview of the algorithm. In the beginning, a malicious network packet is selected, for which the algorithm needs to create an adversarial example. AGONI creates and evaluates the initial population, starting a loop of evolving individuals into new generations. The loop consists of applying selection, crossover and mutation operators on the population before evaluating the validity of the individuals. The last step is deciding whether to continue the algorithm or to terminate after re-evaluating the population fitness.

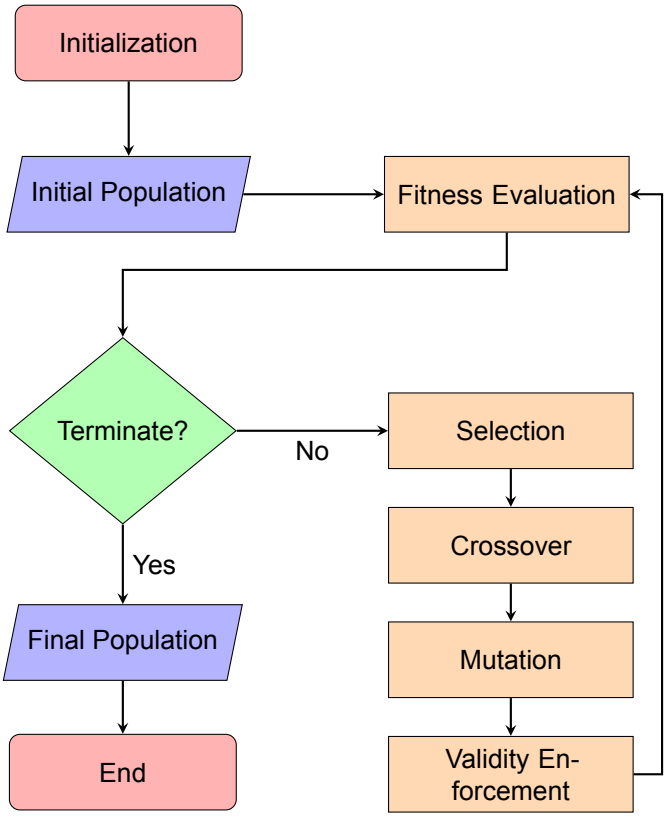


Figure 3.1: The structure of AGONI displaying its order of operations. At the start, AGONI randomly samples an initial population that enters the loop of creating new generations. The loop consists of creating new individuals using selection, crossover and mutation, before enforcing the validity of every individual at the end of a loop. Once a termination criterion is hit (e.g., hitting the maximum number of generations), the algorithm terminates and returns the final population.

3.2. Individuals

The individuals that AGONI uses represent the solutions that exist for the respective problem. The selected solutions represent network packets existing of an IPv4 and TCP layer. Several other protocols exist that are used within the transport and network layer (e.g., UDP and IPv6 respectively), meaning there are various combinations of protocols that we can base the structure of our individuals on. We decide to use one of the most common protocol combinations; IPv4 together with TCP. The research objective is to bypass the defence of an NIDS, regardless of the protocols used in the generated packets. We can apply the same research on different protocol combinations in future work with a different representation of the network packets.

In Section 2.1, all the headers were shown along with their meanings for both the IP and TCP protocol. Not all values are part of an individual due to them requiring a locked value for our purposes or because their values are inferable from other headers. We also opted to exclude the Ethernet Frame, since all of the headers within it are either locked or inferable. In Table 3.1, we show the IP and TCP features that have been excluded from the features of the individuals.

- **IP - Version**: We are using IPv4, so the version value is locked at 4.
- **IP - IHL**: The options field is the only header that influences the IHL and since the options are locked to contain no additional information, the IHL defaults to 5.
- **IP - Flags**: The IP only has 3 flags, from which only 2 are used just to indicate fragmentation. Since we generate only singular packets, we can lock the flags on indicating non-fragmented packets.
- **IP - Fragment Offset**: This offset is used to reorganise fragmented packets. Since we are generating singular packets, this is not applicable so the value is locked.
- **IP - Protocol**: We use the TCP protocol in this research.
- **IP - Header Checksum**: The checksum is calculated based on the values and length of the rest of the IP header. If fuzzing changes the value of the header checksum, we need to alter all the other values such that the checksum is correct to ensure the packet is still valid. This is non-trivial and heavily limits the fuzzing on the other header values. Therefore, we opt to not fuzz over this value and calculate the correct value at the end of the algorithm.
- **IP - Source IP Address**: In an attack, all the packets come from the IP address of the attacking device(s). We acknowledge that spoofing IP addresses is possible, but still exclude it from the individual features.
- **IP - Destination IP Address**: The argumentation here is similar for the source IP, since in the attack scenario the target's IP address is already defined.
- **IP - Options**: The firewall recommendations for handling IPv4 Options from the Internet Engineering Task Force [54] state that a packet should be dropped almost always if the options field is non-empty. Based on that, we opt to lock the options field as empty.
- **TCP - Data Offset**: This value represents the TCP header length and is a locked value since the TCP Options is the only header that influences this value. Since we locked the TCP Options to be empty, the data offset value remains unchanged as well.
- **TCP - Checksum**: This is inferable and calculated at the end of the algorithm, following the same reasoning as the IP Header Checksum.
- **TCP - Reserved**: These values are locked almost by definition. Fuzzing values different than zero is only a give-away for a NIDS that our packets are suspicious.
- **TCP - Options**: The majority of the TCP Options are historical, obsolete, experimental, not yet standardized, or unassigned [55]. Therefore, we lock the options as empty as they barely add value and only significantly increase the difficulty of maintaining the packet validity.

IP Feature	Cause
Version	Locked
IHL	Locked
Flags	Locked
Fragment Offset	Locked
Protocol	Locked
Checksum	Inferable
Source IP Address	Locked
Destination IP Address	Locked
Options	Locked

(a) The excluded IP features.

TCP Feature	Cause
Data Offset	Locked
Checksum	Inferable
Reserved	Locked
Options	Locked

(b) The excluded TCP features.

Table 3.1: The IP and TCP features that were excluded from an individual for either having a locked or inferable feature.

The remaining features from the IP and TCP headers that are not locked or inferable make up the individuals to be used in the Genetic Algorithm. When generating packets, this is the representation for network packets that is used, answering RQ1. For all of these features, we need to be able to ensure the validity so that the packet made from all these features is valid. The most important step is ensuring that the algorithm generates values that are allowed within a packet. For example, the source port can never be $100,000$ since 16 bits are available in a network packet for the source port, meaning the highest valid value is only $2^{16} - 1 = 65,535$. In Table 3.2, all the features for the individuals are

shown along with the number of bits that a particular feature has within a network packet. From this table, we infer a set of hard-constraints that ensure that whatever the packet looks like, a certain feature has to be between zero and $2^{bits} - 1$. Based on the meanings of the TCP flags, we extend this set of constraints with the following:

- If the SYN flag is set, then the sequence number is the initial sequence number and should be set to 1.
- If the ACK flag is not set, then the sender is not expecting an ACK and the acknowledgement number should be set to 0.
- If the URG flag is not set, no urgent pointer can be given and should be set.

Individual Feature	Available bits
IP - Total length	16
IP - Time To Live	8
IP - Type of Service	8
IP - Identification	16
TCP - Source Port	16
TCP - Destination Port	16
TCP - Sequence Number	32
TCP - Acknowledgement Number	32
TCP - Urgent Pointer	16

Individual Feature	Available bits
TCP - Window Size	16
TCP - CRW Flag	1
TCP - ECN Flag	1
TCP - URG Flag	1
TCP - ACK Flag	1
TCP - PSH Flag	1
TCP - RST Flag	1
TCP - SYN Flag	1
TCP - FIN Flag	1

Table 3.2: All the 18 features used in the individuals and the number of bits available for each feature.

3.3. Fitness function

The genetic algorithm needs a fitness function to indicate the quality of generated individuals. As stated in Section 1.1, the goal is to generate adversarial examples for malicious packets in order to bypass an NIDS. This translates into 2 separate metrics: how suspicious the packets are when compared to regular traffic and the distance between the adversarial example and the original. Therefore, we design a Multi-Objective Genetic Algorithm (MO-GA) with 2 fitness functions.

3.3.1. Objective 1: Similarity score

We use an anomaly detector to measure the similarity to benign traffic, which returns a *similarity score* that represents the degree to which traffic is an anomaly compared to regular traffic. We choose to use an Isolation Forest (IF) [56] as the anomaly detector, due to it performing well in various settings [57]. A challenge we encountered was acquiring data that the algorithm could use to train the IF model. Well-known packet captures (PCAPs) often combine benign network traffic along with attack traffic, which is undesirable for anomaly detection since the ML model learns that attack traffic is non-anomalous. Researchers rarely publish benign PCAP files used in their research, due to privacy concerns (e.g., published IP addresses). A possibility would be to personally capture a PCAP that we would use to train an IF, but we would be unable to show that such a capture is a realistic representation of regular network traffic. When one traines an IF with an unrealistic dataset, it results in a model with unreliable classifications. We remedied this issue by training the IF on a PCAP filled with malicious traffic. The IF now no longer gives scores that represented the similarity to benign traffic but the similarity to malicious traffic. The scores range from -0.5 (least similar) to 0.5 (most similar), which means that a -0.5 not only represents a packet that was least similar to malicious traffic but also one that was most similar to benign traffic. Consequently, we inverted the scores to achieve an IF that acted similar to one trained on benign traffic. Therefore, the first fitness function is:

$$FF_{similarity}(individual) = -IF\_Similarity\_Score(individual)$$

A drawback of this approach is that no malicious network capture can be an exhaustive representation of malicious traffic. Network traffic coming from network attack A might look vastly different than that from network attack B. So, an IF trained on a PCAP containing traffic resulting from attack A is likely to

classify traffic from attack B as benign since its traffic might look nothing like the traffic from attack A. Therefore, our product should not replicate packets from any attack, but only from attacks used in the training data for the IF. We used PCAPs from the CTU-13 dataset [58], a frequently used, established dataset containing malicious botnet traffic. After training the IF on CTU-13, we inspected the similarity scores from the packets of CTU-13 itself. Figure 3.2 shows the distribution of similarity scores given to CTU-13 packets. Since no labels are available, the IF uses the *contamination* parameter to determine how many of the packets are benign or malicious. For example, with the contamination parameter set to 0.5, half of the packets will be deemed as inliers and the other half as outliers. Since we have no basis for determining what percentage of CTU-13 should be treated as outliers, we opted to use the default value of the IF, resulting in 85% of the packets receiving a negative similarity score.

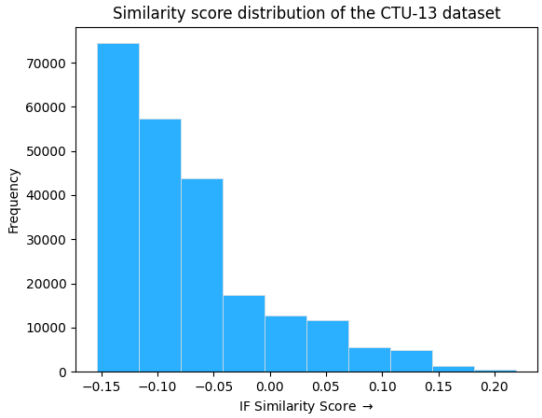


Figure 3.2: Distribution of the similarity scores of the CTU-13 dataset, consisting of around 230,000 entries. The default value of the IF contamination parameter resulted in 85% contaminated packets.

3.3.2. Objective 2: Adversarial distance

For measuring the quality of the adversarial example, we determine the degree to which the new instance differs from the original. We calculate the distance between the feature vector of the original sample and the one from the adversarial example. The calculated distance between an adversarial example and the original instance is a fitness metric we need to minimise to create an adversarial example that is as close to the original instance as possible. This difference needs to be a quantified representation that includes the differences of all features; therefore, we do not use the L_0 distance norm that does not quantify the differences well, nor the L_∞ distance norm that does not include all the features. The L_1 and L_2 distance norms represent the difference between all of the features in a quantified way. Both these distance norms are suitable to use for calculating the difference between instances. For this research, we use the L_2 norm. An expected consequence of this choice is that the difference between two instances will be more easily built on small changes to multiple features, whereas the L_1 norm can generate solutions where the distance is based on the difference of a small set of features [59]. Concludingly, the second fitness function is:

$$FF_{distance}(individual) = \sqrt{\sum_{i=1}^{n}(original_i - individual_i)^2}$$

When calculating the distance between two feature vectors, one issue occurs, namely the presence of categorical data. The headers for the IP Type of Service, TCP Source Port, TCP Destination Port and TCP Flags are all categorical data. For categorical data, values x and $x+1$ are not necessarily more similar than x and $x+100$. However, a distance metric indicates that $x+1$ is way more similar to x, due to the smaller distance in between. The usual solution for this is applying one-hot encoding to the categorical data. This is feasible for the TCP flags since it only resulted in 8 new features. However, for the IP Type of Service, TCP Source Port and Destination Port, it would mean adding $2^8 + 2^{32} + 2^{32}$ new features. This is unfeasible and therefore we refrained from applying one-hot encoding to these

features. An alternative solution would have been to apply one-hot encoding to the most popular n values of the categorical features and adding 1 feature in case the original feature was not part of the most popular n values. This solution would require a small literature or field study to determine the most popular values for the categorical features. For this research, such a study was out of scope and therefore the one-hot encoding for the most popular n values was not applied. Consequently, our distance metric treats categorical features such that values x and $x + 1$ are more similar than x and $x + 100$.

3.3.3. Balancing metric weights

A Multi-Objective Genetic Algorithm needs to carefully handle its objectives to ensure both objectives receive equal weight. If one objective receives too much priority over other objective(s), the MO-GA is more likely to find improvements in that objective. The other objective is then neglected meaning the final generated solutions are not optimised for both objectives. In the current definitions for our fitness objectives, the range for all similarity scores is based on the output of the Isolation Forest (from -0.5 to 0.5) which does not match up with the range of possible distance values. The minimum for the distance metric occurs when the adversarial example is equal to the original sample, resulting in a distance of 0. The maximum distance value occurs between one vector with 18 features that all have the lowest possible value and the other vector that has the maximum value for all 18 features. The resulting Euclidean distance then roughly becomes $6.074 \cdot 10^9$. So there is a big difference in the range of values for both objectives. The similarity score has a range of 1 and the adversarial distance has a range of $6.074 \cdot 10^9$. It is easier to find bigger improvements in the adversarial distance due to the large range of possible values, and those solutions with bigger improvements get picked more for future generations. This means that the objective of adversarial distance has more importance in the process of generating solutions.

To give both metrics the same importance, the ranges need to become equal. If the ranges are equal, then the algorithm is less likely to pick solutions that only find solutions in one objective. We use min-max normalisation to fit all values in the range 0-1. This would give both metrics the same priority. A 0.2 difference in the similarity score would have the same impact as a 0.2 difference in the adversarial distance. However, we make one more adjustment. The distance is calculated based on 18 different features, however, the range per feature varies greatly. Features like flags only consist of 1 bit, meaning they can only take the values 0 and 1. However, values like the sequence and acknowledgement number are made up of 32 bits, meaning that there are 2^{32} different values for those features. Consequently, features with increased ranges have more influence on the distance value, both standard and when normalised. In Table 3.3, we show the (normalised) distances from feature vectors to the feature vector with only zeros. Changing a single flag from 0 to 1 has a small effect on the calculated distance. However, when setting a feature with a bigger range of values (e.g., the acknowledgement number) to zero, the impact on the distance is significant. While the similarity and score now have equal weight, several features become irrelevant due to their limited range even though they could be very impactful in actual traffic.

	All features maximised	**Single Flag at 0**	**Ack. Number at 0**
Real distance	$6.074 \cdot 10^9$	$6.074 \cdot 10^9$	$4.294 \cdot 10^9$
Normalised distance	1	1	0.707

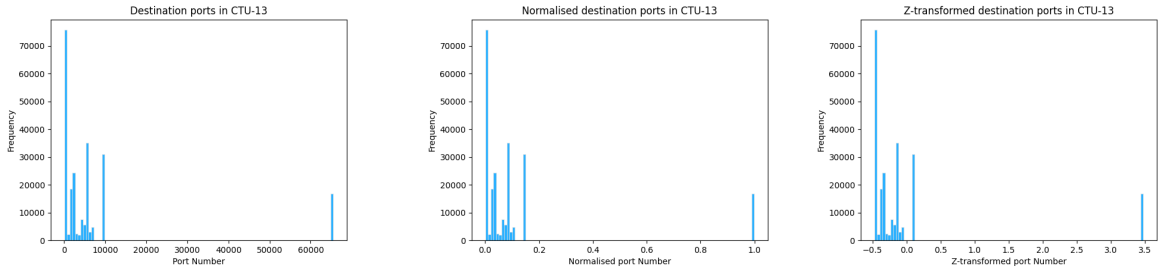
Table 3.3: The (normalised) distances for feature vector from the minimal feature vector. Changing a single flag is insignificant to the degree that the distance does not change. The acknowledgement number has more influence causing the distance to drop by 29.3% when set to zero.

To guarantee equal influence for all features, we perform min-max normalisation on the features before normalising the distance fitness. Now every feature is represented with a number between 0 and 1 and carries the same weight. Table 3.4 shows the updated distance values for the real and normalised distance between feature vectors and the minimal normalised feature vector. With this adjustment, both fitness metrics possess the same priority with every individual feature having the same importance.

The main weakness of min-max normalisation is outliers that cause the other instances to become cluttered to keep the exact same scale. *Z*-score normalisation solves this problem by adjusting the scale to incorporate outliers. However, with adjusted scales, it becomes harder to justify the differences between solutions. Judging whether a particular solution is a better adversarial example, or looks that way due to the scale adjustment is undesired. We compared the effect that both min-max and *z*-score normalisation have on the features and present the result for one of the features, the destination port. We see that both methods of normalisation create the same scale of values and the adjusted scale from the *z*-score normalisation is not noticeable. Since the different normalisation methods did not cause a difference, we simply opted to use the min-max normalisation.

	All features maximised	**Single Flag at 0**	**Ack. Number at 0**
Real distance	4.243	4.123	4.123
Normalised distance	1	0.972	0.972

Table 3.4: The (normalised) distances for the feature vector from the minimal feature vector when the features are normalised in advance. Now a flag and the acknowledgement number have equal influence on the distance, individually causing it to drop by 2.8% when set to zero.



(a) Distribution of the values of destination ports in CTU-13.

(b) Distribution of min-max normalised destination ports in CTU-13.

(c) Distribution of *z*-score normalised destination ports in CTU-13.

Figure 3.3: The distribution of the destination port values in the CTU-13 dataset. We show the effect of min-max normalisation and *z*-score normalisation on the data. *Z*-score normalisation changes the scale of the data, however this is not visible.

3.4. Validity Enforcement

After AGONI created the new generation, we enter the phase of validity enforcement as was shown in Figure 3.1. AGONI pushes generated packets through a network packet parser that validates the packets. Since we use ML in this research, we chose Python to implement the solutions due to its available libraries for ML purposes. We require a solution for creating and validating network packets in Python. To create packets in Python, two common methods exist: crafting the packet data frame manually, or using *Scapy*.

3.4.1. Manual crafting of internet packet

Crafting an internet packet manually is possible by creating a sequence of bytes that represents the packet data frame, as shown in Figure 3.4. This method works, but has zero means of validating the packet. Packet headers not only have a range of valid values but also contextual meaning, influencing what values other headers can take. These contextual meanings result in the following constraints:

- If the URG flag is not set, the urgent offset has to be 0.
- If the SYN flag is set, the sequence number should be 1.
- If the ACK flag is set, the acknowledgement number should be 0.
- The total length of the packet should be larger than 40.

The only manner to validate a packet is to use a set of hard constraints that enforce correct header values and the scenarios listed above. However, this approach gives no guarantee of validity. In practice, network devices use the same approach with a set of constraints. However, devices contain

vulnerabilities when encountering unexpected header values in internet packets (e.g., the IPv4 header Time To Live set to 0) [60]. So regardless of the quality of the set of constraints, the possibility remains that there exist internet packets that would still pass undetected.

```
ip_header  = b'\x45\x00\x00\x28'  # Version, IHL, Type of Service | Total Length
ip_header += b'\xab\xcd\x00\x00'  # Identification | Flags, Fragment Offset
ip_header += b'\x40\x06\xa6\xec'  # TTL, Protocol | Header Checksum
ip_header += b'\x0a\x0a\x0a\x02'  # Source Address
ip_header += b'\x0a\x0a\x0a\x01'  # Destination Address

tcp_header  = b'\x30\x39\x00\x50' # Source Port | Destination Port
tcp_header += b'\x00\x00\x00\x00' # Sequence Number
tcp_header += b'\x00\x00\x00\x00' # Acknowledgement Number
tcp_header += b'\x50\x02\x71\x10' # Data Offset, Reserved, Flags | Window Size
tcp_header += b'\xe6\x32\x00\x00' # Checksum | Urgent Pointer
```

Figure 3.4: Python code showing how to manually produce a TCP/IP packet by creating a sequence of bytes [61]. The comments give some indication as to what bytes should be changed if the value of particular header should be changed, but still the code is not very intuitive.

3.4.2. Scapy

Scapy is a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols, send them on the wire and capture them. Figure 3.5 shows a code snippet that shows the creation of a TCP header within an IP header with a customised destination IP address and destination port. All the other fields automatically get filled by Scapy, reducing effort for determining the remaining header values. However, Scapy also allows packet creation with non-valid values and does not perform a validity check on the packet. In Figure 3.6, the created TCP package is successfully created with a negative port number, which is not a valid value. Therefore, while Scapy is very convenient for crafting packages, the used values still require manual validation.

```
        >>> packet = IP(src=dummyIP1, dst=dummyIP2)/TCP(dport=dummyport)
        >>> packet.show()

        ###[ IP ]###
          version   = 4
          ihl       = None
          tos       = 0x0
          len       = None
          id        = 1
          flags     =
          frag      = 0
          ttl       = 64
          proto     = tcp
          chksum    = None
          src       = dummyIP1
          dst       = dummyIP2
          \options   \
                  ###[ TCP ]###
             sport     = 20
             dport     = dummyport
             seq       = 0
             ack       = 0
             dataofs   = None
             reserved  = 0
             flags     = S
             window    = 8192
             chksum    = None
             urgptr    = 0
             options   = ''
```

Figure 3.5: Python code showing how to create a packet with an IP and TCP layer using Scapy. Only the features that require certain values need to be coded, the rest of the features get generated by Scapy.

```
>>> packet = TCP(dport=-10)
>>> packet.show()

###[ TCP ]###
  sport     = 20
  dport     = -10
  seq       = 0
  ack       = 0
  dataofs   = None
  reserved  = 0
  flags     = S
  window    = 8192
  chksum    = None
  urgptr    = 0
  options   = ''
```

Figure 3.6: Python code showing how Scapy allows faulty header values (e.g., a negative destination port number).

Regardless of how we create the packet, validation remains a manual task. A guaranteed way to discover the validity of a packet is by transmitting it through a network and monitoring if it reaches the specified destination. Although this approach ensures success, validating numerous packets can be a time-demanding process and requires extensive preparation, as it necessitates a functional and observable network for testing.

3.4.3. Applying set of constraints

To enforce validity, we created a set of constraints that would correct header values if they caused the packet to be invalid. Such constraints will detect the majority of non-valid internet packets, but we can not deliver proof that the set of constraints detects all non-valid packets. However, for the purposes of this research, we deemed 'near-perfect' sufficient to demonstrate the findings of this study.

There are two types of constraints for the validity of internet packets. The first type uses lower and upper bounds for header values so that invalid features do not occur (e.g., negative values). The second type is contextual constraints that use the meaning of a header, which indicates what values other features can take. During validity enforcement, we check if every feature has a valid value. If this is not the case, we alter the feature to a valid one.

For the constraints of lower and upper bounds, we treat the feature values as unsigned bit strings. This rules out the possibility of negative values, which is beneficial since negative values are invalid for all packet headers. During the crossover and mutation phases, any alteration now never results in a negative value. Since all alterations to an individual happen at the bit level, every resulting value is always an integer. This avoids dealing with float values that are invalid and require to be rounded to be valid. For the contextual constraints, it is still possible that AGONI generates packets that violate these constraints. Therefore, we change the features that cause the violation back to their valid range.

An alternative way to apply this set of constraints is to not forcefully change the values of the packets, but to design a penalty for the fitness function(s) that reduces the fitness of non-valid internet packets. The GA should then notice that values that cause invalid packets receive worse fitness values and evolve its solutions towards those with valid feature values. We opted against this for a few reasons. The first reason is that this allows for solutions with a high fitness score, but with a penalty. These solutions would remain in the population over the generations due to their high fitness, but in the end would not be viable solutions. A packet must be valid, else it is not usable as an adversarial example. A possible solution for this problem is to set the fitness to the lowest possible value whenever it is non-valid. This would remove any invalid packets with high fitness scores. However, this changes the search space to have various areas that only consist of invalid packets with the lowest possible fitness. This causes the GA to lose its effectiveness since the search space is flat and does not tell the GA in which direction it should evolve its solutions. Ultimately, one could experiment on different penalty weights to find a penalty weight that does not cause invalid packets to appear in the final population but also does not flatten the search space in such a way the GA is negatively affected.

However, we considered these experiments to be out of scope and opted for applying the constraints on the population to enforce packet validity.

3.5. AGONI run visualisation

To show the behaviour of AGONI, we execute a single run and visualise the process and the results as an example. We generate one solution with Single Point crossover, Exchange mutation and Roulette Selection using a population size of 100 and 500 generations. We analyse both the fitness metrics and the feature values over the generations.

To properly illustrate how both fitness metrics behave over the generations, we need 2 graphs, which we will refer to as *generation graphs*. The generation graphs are shown in 3.7, displaying the fitness scores of the individual with the best overall fitness metric and the average fitness score of the entire population. The distance graph decreases with more generations before stabilising after around 100 generations. However, the similarity score behaves counterintuitively. Even though the similarity score of individuals is a problem where higher scores represent better solutions, the generation graph for the similarity score shows the value decreasing over the generations. This is unexpected, yet not impossible behaviour. We can conclude from this that AGONI found solutions that improve more on the adversarial distance and worsen less on the similarity score. This means the solutions decrease in similarity score, but decrease even more on the adversarial distance. This increases the overall quality of the solutions. The average score in the generation graphs starts being far off from the score of the best individual, but with more generations the average score and the best score get closer together. This is due to the entire population evolving towards the best individual, changing the feature values to be like the feature values from the best individual. For the average score, spikes appear that diverge from the best individual. This is caused by the mutation that occasionally changes the individuals in an attempt to find undiscovered solutions. Since the average and best score are so close together before the spikes happen, it means that (almost) the entire population consists of multiple copies of the best individual or individuals that are similar. A change caused by mutation changes the population to have some new individuals that do not consist of features from the best individual, causing a change in the average scores. However, since still the majority of the population consists of the best solution, the few mutations quickly get flushed out of the population by the selection and crossover phases while building the next generation. Therefore the changes in the average score disappear quickly, causing the change in the average score to only be a spike.

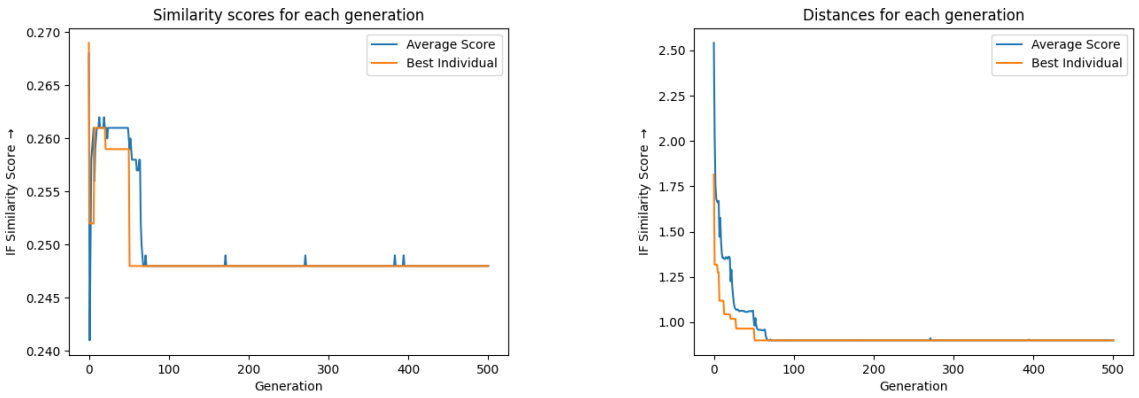


Figure 3.7: The similarity and distance of of the best scoring individual and the average of the entire population. The adversarial distance gradually goes down over the generations. The similarity should be maximised, but also decreases.

In Table 3.5, we show a summary of the values of every feature during the generations. We compare it with the values of the original instance, to show whether the features are converging towards the correct value. The difference in generations between columns is smaller in the first few columns to show that the most changes happen in the first 100 generations and that no changes occur to the feature values in the remaining 400 generations. We included the exact process across the generations for every feature in appendix E.

Feature	Original	Gen. 0	Gen. 10	Gen. 50	Gen. 100	Gen. 500
Total Length	65	27809	5184	5184	5184	5184
Time to Live	117	254	253	253	253	253
ToS	0	223	54	54	54	54
Identification	24490	51872	60661	4874	4874	4874
Src. Port	5296	19291	37443	37443	28092	28092
Dst. Port	2343	58959	6213	6213	6213	6213
Seq. Number	74984	3868742852	1093025678	1093025678	1093025678	1093025678
Ack. Number	316	430201660	2359635432	1893826515	1893826515	1893826515
Urgent Pointer	0	13445	0	0	0	0
Window	63925	32142	60689	60689	60689	60689
CRW Flag	0	0	0	0	0	0
ECN Flag	0	0	0	0	0	0
URG Flag	0	0	0	0	0	0
ACK Flag	1	1	1	1	1	1
PSH Flag	1	1	1	1	1	1
RST Flag	0	0	0	0	0	0
SYN Flag	0	0	0	0	0	0
FIN Flag	0	0	0	0	0	0
Similarity	-	0.26859	0.26102	0.25870	0.24843	0.24843
Distance	0	0.42803	0.26382	0.22735	0.21224	0.21224

Table 3.5: The changes in all the features across several generations in the best individual. From generation 100 to 500, no changes occurred.

4

Comparing MO-GA operators

In Chapter 2, we presented the design of AGONI and the various operators available for the selection, crossover and mutation phases. For GAs, there is no set of operators that universally performs better than the rest. Depending on the context of the problem, some operators might be more suitable than others. This chapter discusses the process of answering RQ2; *What effect do different GA versions have on the generation of network packets aimed for anomaly-detection evasion?* We perform experiments to determine the preferred configuration of available operators for AGONI. We explain the experiment setup in Section 4.1, where we justify hyperparameter choices and the application of crossover and mutation. Section 4.2 shows the results and statistics before we summarise and conclude in Section 4.3.

4.1. Experiment setup

To find the preferred configuration, we compare all possible combinations between the crossover, mutation and selection operators. Every configuration is run 10 times, where for each iteration a new instance of the MO-GA is created. Every instance of the MO-GA selects an internet packet known to be malicious and uses that as the instance to generate adversarial examples for. This means all individuals receive their distance fitness by calculating the distance with that particular malicious instance. Since we want to evaluate the overall performance of the GA with certain operators and not just on one particular instance, the experiment re-initialises the MO-GA every run to create various scenarios that the algorithm has to create adversarial examples for. Per run we store the fitness per generation, the individuals from the final population and the Pareto-front accumulated across the execution.

4.1.1. Parameter tuning

This experiment has 3 design choices that can be tuned: the population size, the number of generations and the amount of experiments performed per configuration. Determining the population size and number of generations is a trade-off between computational complexity and result quality. Setting a higher population size gives the algorithm more points to look at in the search space, but requires more time per generation to compute the fitness and generate the following generation. This is similar for the number of generations; more generations give the algorithm more time to perform its search across the search space. Less generations require less computations, yet when set too low, the algorithm could terminate too early without having had a chance to converge the individuals to an optimised solution.

A review study done by Hassanat et al. [62] on the tuning of GA parameters showed various recommendations for the population size in the range of 2-600. Experimentation on the preferred population size for the context of this problem falls outside the research scope. Therefore, we again adapted to the recommendation of De Jong [20] and selected a population size of 100. Determining the optimal number of generations is a problem highly dependent on the problem type and complexity [62, 63], but the choice is usually not well motivated. Therefore, we selected a number with the priority of allowing the algorithm to converge its individuals, namely 500 generations. The focus of the research is finding high-quality solutions for the problem and not minimising the computation complexity of the process.

Finally, for determining the quantity of the experiments, we chose to run each configuration 10 times. Every operator is included in many different configurations, creating a substantial set of results that can be compared per operator. Every crossover operator is part of 300 experiments, with 5 mutation operators and 6 selection operators adding up to 30 different configurations that get repeated 10 times each. Similarly, every selection operator is part of 200 experiments and every mutation operator is part of 240 experiments. Therefore, 10 repetitions per configuration results in a set of data that is large enough for investigating different GA operator configurations.

4.1.2. Application of Crossover and Mutation

There are 2 ways to apply crossover and mutation on the individuals of AGONI. In Figure 4.1, we show a minimised representation of 2 individuals before applying crossover or mutation. Every individual has two features, A and B, and every feature exists out of bits. The first option is to apply the operators on the feature level, meaning that per feature it is decided if crossover or mutation is applied. This is shown in Figure 4.2, where a simple crossover is applied on the feature level. This approach swaps out entire features with all their corresponding bits, meaning that the features do not change their values, but only belong to different individuals. The second option is to apply the operators on the bit level, as visualised in Figure 4.3. This figure again shows the result of a crossover, but only the bits are moved. Each feature still belongs to the same individual, but since all the bits are shuffled, each feature now has a different value. Theoretically, applying mutation on bit-level can cause the same changes as the mutation on feature-level. However with network packets having several features and mutation being applied during every generations, the results are quite different with mutation on bit-level. When applying mutation on feature-level, entire features are adjusted, leaving all other features untouched. Mutation on bit-level can affect the value of multiple features. For the problem of generating network packets, features can take on a large number of values. Therefore, we chose to apply the operators on bit-level, so features can obtain different values more easily, exploring more of the search space.

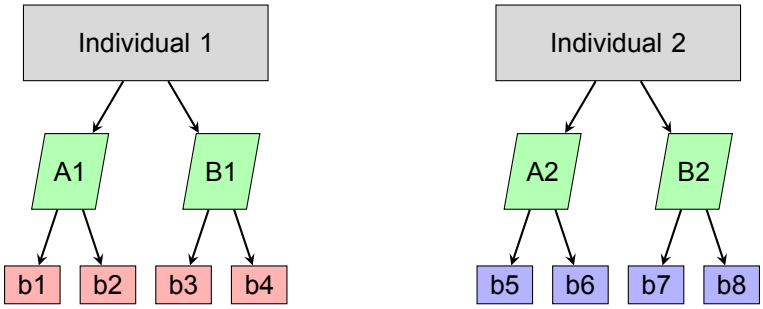


Figure 4.1: Illustration of 2 individuals before applying crossover or mutation. The individuals exist of 3 layers. Each individual (gray) has features (green), and every feature value exists out of bits (red).

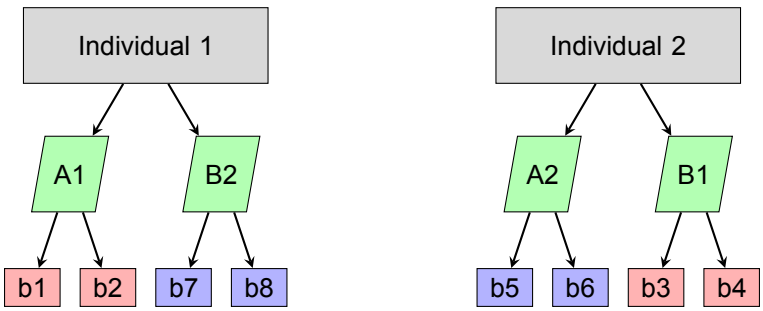


Figure 4.2: The 2 individuals after crossover is applied on the feature level. Crossover swapped feature B1 and B2 with all their corresponding bits. Feature B1 and B2 still have the same value, but belong to different individuals.

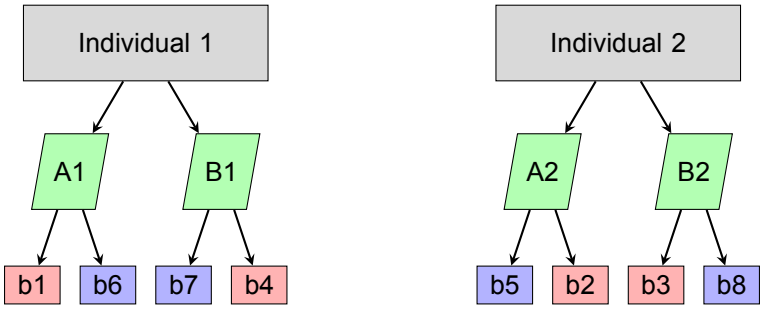


Figure 4.3: The 2 individuals after a crossover is applied on the bit level. All the features still belong to the same individual, but the bits have been swapped by crossover, meaning the values of the features have changed.

4.2. Results

From every execution of a configuration, we store both fitness objectives of the best generated solutions per generation, the population of the final generation and the fitness of the individuals that made up the Pareto-front that formed during execution.

4.2.1. Top-individuals

With 5 crossover operators, 5 mutation operators and 6 selection operators, there exist 150 different possible configurations that we need to compare. Individually analysing the data per configuration and comparing it with all the other possibilities is not efficient and does not result in a clear overview that shows how different operators affect the process of generating network packets. To get this overview, we gathered the fitness objectives and created 3 different sets based on the operator type. To clarify, we took 3 copies of the original results and sorted 1 of them on their operator for crossover, mutation and selection respectively. The results are plotted in Figure 4.4.

From these plots, we see that the found solutions are overlapping, regardless of the operators used. Some operators generate more diverse solutions, creating individuals that have an increased adversarial distance, but compensate that with a better similarity fitness (e.g., the crossover operator 'wright'). Due to all the operators having overlapping results, it is hard to judge the diversity of some operators within the plots. To individually inspect all operators in more detail, we refer to Appendix A. From the collection of crossover operators, the quality of solutions is similar with every operator, with the only noticeable difference being the variety of solutions generated. The Wright operator generates more diverse solutions, while the Arithmetical and Uniform operators generate more consistent solutions. When looking at the mutation operators, the Swapping and Reversion operators are more diverse while the other operators are more consistent. The Uniform operator appears to be better at finding solutions with decreased adversarial distance over solutions with increased similarity score. Finally for the selection operators, all of them show a similar performance with a diverse set of generated solutions. None of the operators stand out to be universally more effective for crossover, mutation or selection. To get a better insight, we show the mean and standard deviation of both fitness metrics for all operators in Table 4.1, 4.2 and 4.3 for crossover, mutation and selection respectively.
All operators perform similarly when looking at the similarity score. Every operator is able to consistently generate adversarial examples. There is more difference in the performance for the distance fitness, where there is an apparent trade-off for operators between the quality of their solutions (a lower distance) and the diversity of their solutions (a higher standard deviation). Operators such as Wright Crossover and Swapping Mutation have larger distance means, but also larger standard deviations. While the average quality of the distance metric might be lower, a large standard deviation implies that among the generated solutions there are some with better distance metrics. So not only does a high diversity still result in solutions with high quality, it also covers more of the solution search space. This increases the chance of the algorithm finding a solution with a very high quality.

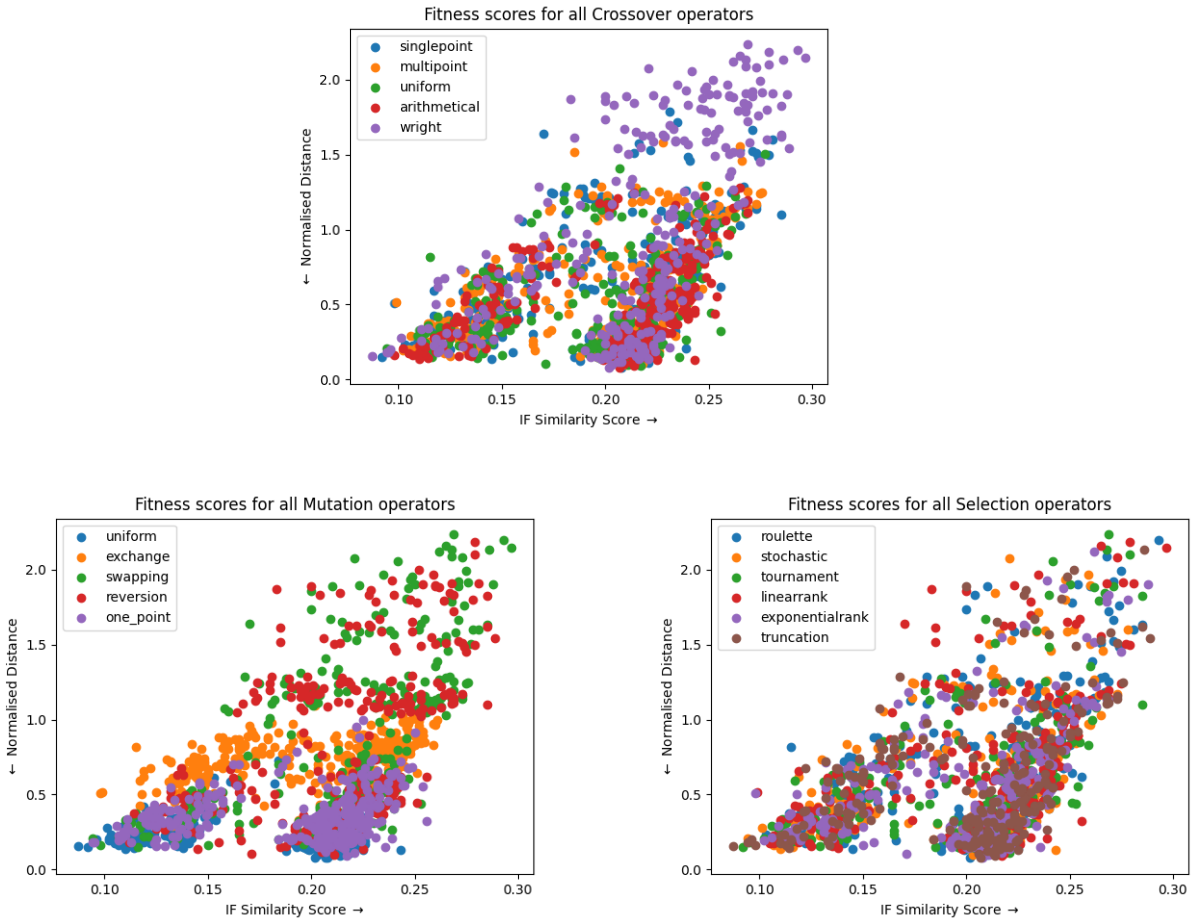


Figure 4.4: The fitness scores from the best generated individuals per operator. The distance fitness is the distance between the original instance and the adversarial example, both with normalised features. The similarity score is produced by the Isolation Forest. Better solutions have a lower distance fitness and a higher similarity fitness.

4.2.2. Pareto-fronts

Showing the overall performance and statistics is not the only aspect we need to inspect. While it is important for the algorithm to generate good solutions that are diverse and cover a lot of the search space, the end goal is to find a network packet that can bypass a NIDS. This means that we want the best possible solution with no regard whether the rest of the generated solutions is below average or not diverse. If a particular configuration of operators has a poor performance overall, but manages to find a few solutions that significantly improves upon other discovered solutions, that configuration is preferable over one that performs well overall, but does not generate solutions that are of significant higher quality. Therefore, we visualise the fitness of the non-dominated individuals gathered across all the experiment executions to form the Pareto-fronts. These fronts help to inspect the quality of the best generated solutions for each operator. The fronts can be seen in Figure 4.5, for both the crossover, mutation and selection operators respectively. These graphs produce similar results as the previous experiments for the overall performance, namely that the points for all operators are close together and do not indicate a clear result for a better performing operator. Again, we include all individual scatter plots per operator with their respective Pareto-front, shown in Appendix B.

We recalculate the mean and standard deviation of both fitness metrics for each operator based on the Pareto-fronts, shown in Table 4.4, 4.5 and 4.6 for crossover, mutation and selection respectively. Contradictory, the mean for the adversarial distance and similarity score is worse for the non-dominated solutions of the Pareto-front, than the means calculated for the data of Figure 4.4. The first cause for this is a decreased number of points, since only new discoveries for a Pareto-front are added. This

Crossover	All points	Single Point	Multi Point
Similarity: Mean	0.201 ± 0.001	0.198 ± 0.002	0.195 ± 0.002
Distance: Mean	0.616 ± 0.012	0.607 ± 0.024	0.564 ± 0.002
Similarity: STD	0.043	0.043	0.043
Distance: STD	0.450	0.408	0.350
	Uniform	**Arithmetical**	**Wright**
Similarity: Mean	0.195 ± 0.002	0.202 ± 0.002	**0.212 ± 0.003**
Distance: Mean	0.492 ± 0.018	**0.482 ± 0.017**	0.936 ± 0.037
Similarity: STD	0.040	0.042	**0.045**
Distance: STD	0.310	0.289	**0.637**

Table 4.1: The mean, standard error of the mean and standard deviation ($n = 300$) for all crossover operators for both fitness metrics. The operators with best-performing statistics are highlighted in green. The Wright operator performs best on 3 of the 4 statistics.

Mutation	All points	Uniform	Exchange
Similarity: Mean	0.201 ± 0.001	0.181 ± 0.002	0.199 ± 0.002
Distance: Mean	0.616 ± 0.012	**0.218 ± 0.006**	0.742 ± 0.008
Similarity: STD	0.043	0.042	0.042
Distance: STD	0.450	0.096	0.141
	Swapping	**Reversion**	**One Point**
Similarity: Mean	**0.215 ± 0.002**	**0.215 ± 0.002**	0.193 ± 0.002
Distance: Mean	0.900 ± 0.032	0.854 ± 0.031	0.366 ± 0.009
Similarity: STD	**0.043**	0.038	0.040
Distance: STD	**0.548**	0.533	0.161

Table 4.2: The mean, standard error of the mean and standard deviations ($n = 300$) for all mutation operators for both fitness metrics. The operators with best-performing statistics are highlighted in green. The Swapping operator outperforms on 3 of the 4 statistics and the Uniform operator is outstanding on finding solutions with a low adversarial distance.

causes outliers to have a larger impact on the statistics. The second cause is the diversity of a Pareto-front. The common entries of a front are solutions where both fitness metrics have decent scores. However, it also includes solutions with an outstanding good score for one of the fitness metrics, but a below average score for the other metric. These types of solutions are outliers that affect the statistics of the results.

Judging on the Pareto-front statistics, we re-evaluate the performances of the operators. Among the crossover operators, Wright still has the highest mean for the similarity score, but does not have the most diverse solutions as shown by the standard deviation in the fitness generated by Wright. Single Point crossover now has the lowest mean for the distance and the most diverse solutions for the similarity score, with the Arithmetical operator having the most diverse scores for adversarial distance. The Pareto-front statistics for mutation and selection operators indicate 4 different operators having the best score for the 4 different calculated metrics.

4.2.3. Selecting best configuration

We have looked at scatterplots and Pareto-fronts for all the operators and gained an overview of the performance of AGONI. This has not shown a set of operators to be better overall, with every operator successfully generating adversarial examples. Every operator should therefore be able to bypass an NIDS, meaning every configuration of operators is acceptable. We need to select one set of operators to use, since performing the remaining experiments with all possible configurations is out of scope. The choice between all operators is a trade-off in 3 dimensions; the degree to which an operator finds high similarity scores, the degree to which it finds low adversarial distances and the diversity of the generated adversarial examples. To find the best operators, we sum the values of the statistics from Table 4.1

Selection	All points	Roulette	Stochastic	Tournament
Similarity: Mean	0.201 ± 0.001	**0.204 ± 0.003**	0.199 ± 0.003	0.197 ± 0.003
Distance: Mean	0.616 ± 0.012	0.698 ± 0.030	0.595 ± 0.028	**0.567 ± 0.028**
Similarity: STD	0.043	**0.044**	0.042	**0.044**
Distance: STD	0.450	0.468	0.438	0.442
	Linear Rank	**Exponential Rank**	**Truncation**	
Similarity: Mean	0.202 ± 0.003	0.199 ± 0.003	0.202 ± 0.003	
Distance: Mean	0.653 ± 0.031	0.581 ± 0.026	0.601 ± 0.027	
Similarity: STD	**0.044**	0.043	0.042	
Distance: STD	**0.492**	0.410	0.433	

Table 4.3: The mean, standard error of the mean and standard deviations ($n = 250$) for all selection operators for both fitness metrics. The operators with best-performing statistics are highlighted in green. No operator stands out, with very similar statistics. The maximum difference in standard deviation for the similarity score is only 0.002.

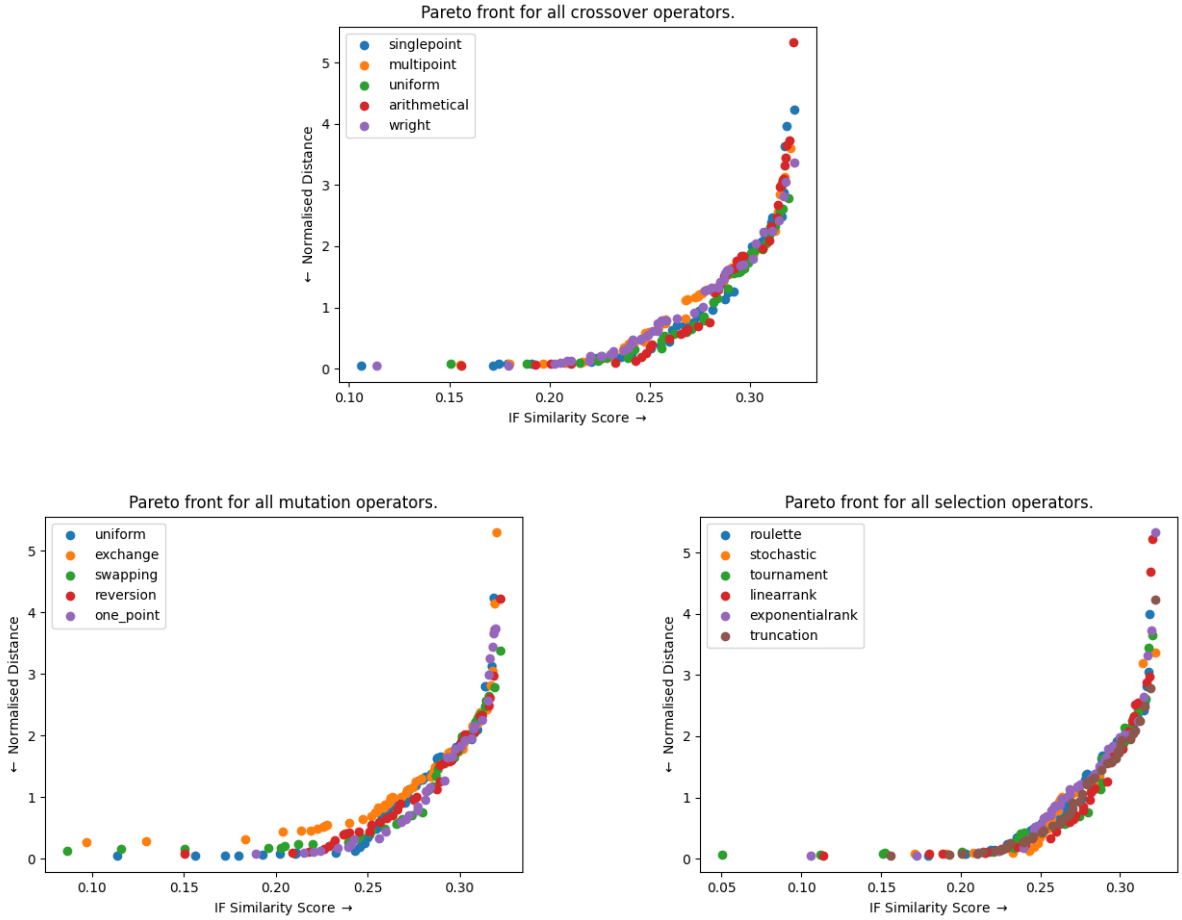


Figure 4.5: The fitness scores from generated individuals part of the Pareto-front per operator. All operators follow a very similar Pareto-front and are able to find solutions of roughly the same quality.

to 4.6. To do this fairly, we make 2 adjustments to the data. We normalise the data, since adversarial distance has a larger range of values, causing the distance mean and standard deviation to be more impactful if the values were summed regularly. The second adjustment is inverting the normalised distance data (using the formula $1 - Distance$), since the adversarial distance is a minimisation problem. Summing the data would mean that larger adversarial distances would cause a better overall score for an operator. By inverting the data, lower distances, which are better, cause the overall score for an

Crossover	All points	Single Point	Multi Point
Similarity: Mean	0.266 ± 0.003	0.259 ± 0.008	0.264 ± 0.005
Distance: Mean	1.198 ± 0.07	1.125 ± 0.178	1.168 ± 0.137
Similarity: STD	0.042	**0.049**	0.039
Distance: STD	1.044	1.142	0.967
	Uniform	**Arithmetical**	**Wright**
Similarity: Mean	0.273 ± 0.006	**0.279 ± 0.006**	0.258 ± 0.006
Distance: Mean	1.172 ± 0.134	1.604 ± 0.207	**0.985 ± 0.121**
Similarity: STD	0.037	0.039	0.040
Distance: STD	0.849	**1.296**	0.844

Table 4.4: The mean and standard deviations based on the Pareto-front for all crossover operators for both fitness metrics. The Arithmetical operator stands out the most with a high mean distance for both metrics and high distance standard deviation.

Mutation	All points	Uniform	Exchange
Similarity: Mean	0.268 ± 0.003	0.262 ± 0.006	0.267 ± 0.006
Distance: Mean	1.274 ± 0.066	**1.112 ± 0.130**	1.388 ± 0.129
Similarity: STD	0.044	0.044	0.044
Distance: STD	0.982	0.919	0.945
	Swapping	**Reversion**	**One Point**
Similarity: Mean	0.264 ± 0.009	0.272 ± 0.006	**0.278 ± 0.006**
Distance: Mean	1.220 ± 0.148	1.283 ± 0.161	1.387 ± 0.194
Similarity: STD	**0.054**	0.037	0.035
Distance: STD	0.933	1.006	**1.112**

Table 4.5: The mean and standard deviations based on the Pareto-front for all mutation operators for both fitness metrics. The One Point operator stands out the most with a high mean distance for both metrics and high distance standard deviation.

operator to also be better. The resulting data and summation can be found in Appendix D, but here we only show the final scores for every operator in Table 4.7. From the results of this table, we select Single Point crossover, Uniform mutation and Tournament selection as the set of operators to use for further experiments.

4.2.4. Errors per feature in generated packets

Looking at the defined fitness metrics, we get a good indication of how the configurations perform relative to each other. However, it is complicated to define a label that tells us when the GA is performing 'good' or 'bad'. A packet with a similarity score of x and an adversarial distance of y might look better in comparison with other generated solutions, but if the GA is performing bad overall, there is no way of telling. To get a better overview of the packets that AGONI generated and the patterns that emerge, we calculate per feature the mean difference from the original instance and the standard deviation. We do this for both the real differences and the differences between normalised values. The real differences give insight into what actual values for certain headers AGONI generates, while normalised differences give a better insight into how close generated values are to their original. To give an example, if the real difference in a flag is 1, then from the 2 options the header had, it now has the wrong one. However, if the Acknowledgement Number has a real difference of 1, the real value is only 1 off from the 2^{32} possible values that an Acknowledgement Number can take.

Selection	All points	Roulette	Stochastic	Tournament
Similarity: Mean	0.266 ± 0.003	0.266 ± 0.005	**0.270 ± 0.005**	0.255 ± 0.010
Distance: Mean	1.198 ± 0.070	1.142 ± 0.128	1.176 ± 0.134	1.086 ± 0.168
Similarity: STD	0.042	0.036	0.032	**0.060**
Distance: STD	1.044	0.924	0.837	0.982
	Linear Rank	**Exponential Rank**	**Truncation**	
Similarity: Mean	0.269 ± 0.008	0.269 ± 0.006	0.266 ± 0.006	
Distance: Mean	1.322 ± 0.204	1.307 ± 0.152	**1.081 ± 0.135**	
Similarity: STD	0.046	0.039	0.038	
Distance: STD	**1.240**	1.008	0.898	

Table 4.6: The mean and standard deviations based on the Pareto-front for all selection operators for both fitness metrics. No operator particularly stands out, with all operators having their own strong and weak points.

Crossover	**Single Point**	Multi Point	Uniform	Arithmetical	Wright	
Score	**3.233**	3.174	3.155	3.162	3.173	
Mutation	**Uniform**	Exchange	Swapping	Reversion	One Point	
Score	**3.191**	3.030	3.139	3.141	3.160	
Selection	Roulette	Stochastic	**Tournament**	Linear Rank	Exp. Rank	Truncation
Score	3.169	3.155	**3.224**	3.222	3.163	3.191

Table 4.7: The scores based on the summed normalised statistics, indicating that Single Point crossover, Uniform mutation and Tournament selection are the statistically best choice.

Table 4.8 shows the results of these calculations. While several features have a mean real difference of several orders of magnitude, the mean normalised difference shows that, relatively, the generated features are still close to the original instance. For 4 features, the generated solution always has the same value, resulting in the difference always being 0. The feature with the largest differences (the time to live) has a normalised mean of 0.0918, meaning that on average the difference is 9.18% of the largest difference possible. Looking at all normalised average differences, the average of that value is 0.0212, meaning that the difference of every generated value for any feature with its original instance is on average 2% of the largest possible difference.

4.3. Conclusions

This chapter aimed to answer RQ2: *What effect do different GA versions have on the generation of network packets aimed for anomaly-detection evasion?* We executed the algorithm on all configurations and investigated the resulting data. We presented graphs and statistics for both the overall performance and the Pareto-front created by each operator. Finally, we looked into the generated features and their differences from the original instances.

From the experiment results, we can conclude the following: AGONI successfully generates valid adversarial examples 100% of the time. Different operators have little to no effect on the quality of generated adversarial examples. From the generation graphs, we find that AGONI finds improvements over the adversarial distance faster than improvements over the similarity score. Finally, generated adversarial examples closely resemble the original instances of malicious traffic, with generated features having a difference averaging 2% from the largest possible difference.

Feature	Mean	Std. Dev.	Mean (Normalised)	Std. Dev. (Normalised)
Total Length	$0.872 \cdot 10^4$	$0.395 \cdot 10^4$	0.133	$0.602 \cdot 10^{-1}$
Time To Live	$0.234 \cdot 10^2$	$0.617 \cdot 10^1$	$0.918 \cdot 10^{-1}$	$0.242 \cdot 10^{-1}$
ToS	$0.190 \cdot 10^{-1}$	0.511	$0.745 \cdot 10^{-6}$	$0.200 \cdot 10^{-2}$
Identification	$0.258 \cdot 10^4$	$0.252 \cdot 10^4$	$0.395 \cdot 10^{-1}$	$0.385 \cdot 10^{-1}$
Source Port	$0.341 \cdot 10^4$	$0.147 \cdot 10^4$	$0.521 \cdot 10^{-1}$	$0.225 \cdot 10^{-1}$
Destination Port	$0.465 \cdot 10^4$	$0.494 \cdot 10^4$	$0.709 \cdot 10^{-1}$	$0.754 \cdot 10^{-1}$
Sequence Number	$0.129 \cdot 10^8$	$0.265 \cdot 10^8$	$0.301 \cdot 10^{-2}$	$0.617 \cdot 10^{-2}$
Ack. Number	$0.736 \cdot 10^8$	$0.767 \cdot 10^8$	$0.172 \cdot 10^{-1}$	$0.179 \cdot 10^{-1}$
Urgent Offset	0	0	**0**	0
Window Size	$0.377 \cdot 10^4$	$0.260 \cdot 10^4$	$0.576 \cdot 10^{-1}$	$0.396 \cdot 10^{-1}$
CWR Flag	0	0	**0**	0
ECE Flag	0	0	**0**	0
URG Flag	$0.100 \cdot 10^{-2}$	$0.316 \cdot 10^{-1}$	$0.100 \cdot 10^{-2}$	$0.316 \cdot 10^{-1}$
ACK Flag	0	$0.$	0	0
PSH Flag	$0.200 \cdot 10^{-2}$	$0.447 \cdot 10^{-1}$	$0.200 \cdot 10^{-2}$	$0.447 \cdot 10^{-1}$
RST Flag	$0.100 \cdot 10^{-2}$	$0.316 \cdot 10^{-1}$	$0.100 \cdot 10^{-2}$	$0.316 \cdot 10^{-1}$
SYN Flag	0	0	**0**	0
FIN Flag	$0.400 \cdot 10^{-2}$	$0.631 \cdot 10^{-1}$	$0.400 \cdot 10^{-2}$	$0.631 \cdot 10^{-1}$

Table 4.8: The mean difference between the original instances and the adversarial examples, with the standard deviation on the left ($n = 5000$). The right column shows the same heuristics over normalised differences. For 4 features, the difference and standard deviations are zero. The feature with the largest differences is the time to live, with a mean normalised difference of 0.0918.

5

Evaluating GA in attack scenarios

The previous chapter showed the performance of AGONI with only constraints on the packet validity. However, when the objective is to create a network attack, more constraints are in place for header values to ensure attack functionality. With more constraints, the search space gets more limitations. A limited search space likely affects the quality of the generated solutions. This chapter presents experiments to show the performance of AGONI when applied in attack scenarios where a defined set of features must keep their original value. The attack scenarios represent varying levels of constraints and affect different IP/TCP headers. We perform these experiments to answer RQ3: *Do attack constraints affect the quality of adversarial examples generated by Genetic Algorithms?* Section 5.1 shows the different attack scenarios and what features they affect. Section 5.2 shows the effect that each scenario has on the performance of AGONI before we conclude in 5.4.

5.1. The attack scenarios

We selected a set of attack scenarios to apply constraints on multiple features, in order to evaluate AGONI. The scenarios are shown in Table 5.1, along with the features that are affected by those scenarios. The affected features of the original instance need to have the same value in every adversarial example that is generated.

Scenario 1 constrains the algorithm in changing the total length of the packet. Since the header length of both the IP and TCP protocol is known since we are not including the IP and TCP options header, this feature implicitly represents the payload size. So this scenario represents a setting where the attacker has to get a particular payload past an NIDS. Scenario 2 represents an attack that must access a specific port, for example a specific service that would not be available on any other port. Scenario 3 tests the ability of the algorithm to create packets that a network sends in response to other packets with correct sequence and acknowledgement numbers. Attacks that need to maintain a connection will have to respond with correct values for those features, else the network will cut off their connection. Scenarios 4 and 5 increase the level of constraint by combining the first three scenarios. Scenario 6 checks the influence of the TCP flags on generating adversarial examples, for example when an attacker needs to perform the TCP 3-way handshake that can only performed by setting the correct flags. Finally, scenario 7 puts constraints on all IP/TCP features that were unaffected by previous scenarios. Scenario 7 does not represent a particular attack scenario, but helps to bring to light the importance of said features when generating adversarial examples. Every scenario ran 100 times with the configuration of operators chosen in Chapter 4. We store the same data as during the experiments of said chapter to get an insight into the performance.

5.2. Results

Similar to Chapter 4, we store the best individuals from each run and the Pareto front accumulated during all the runs. Figure 5.1a shows the best individuals, with the Pareto fronts in Figure 5.1a. Along with the results from the scenarios, we plotted results that were unconstrained to compare and judge the effect certain scenarios had. From these graphs, we see that constraints do affect the quality of the

Scenario #	Description	Affected features
1	The attacker cannot change the payload.	• Total Length
2	The attacker cannot change the destination port.	• Destination Port
3	The attacker cannot change the sequence and acknowledgment numbers.	• Sequence Number • Acknowledgement Number
4	The attacker cannot change any feature from Scenario 1-2.	• Total Length • Destination Port
5	The attacker cannot change any feature from Scenario 1-4.	• Total Length • Destination Port • Sequence Number • Acknowledgement Number
6	The attacker cannot change any of the TCP flags.	• All TCP Flags
7	The attacker cannot change any features not affected by Scenario 1-6.	• Remainder of IP/TCP headers

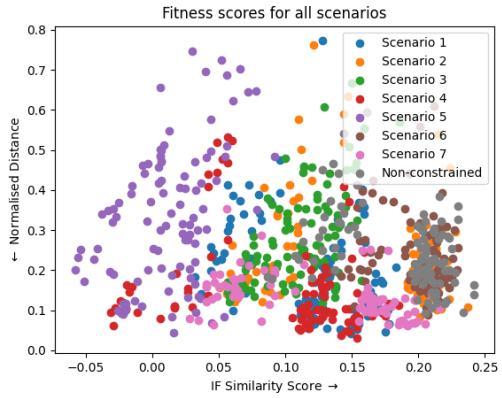
Table 5.1: The 7 scenarios used, with the features that are not allowed to be altered per scenario. The number of affected features grows with the number of scenarios, to investigate the level of constraint we can put on the algorithm. Note that scenario 7 does not necessarily represents an attack scenario, but is added to investigate the importance of the features unaffected by the previous scenarios.

generated solutions. Scenarios with more constraints affect the overall performance more than those with less constraints. This makes sense, since more constraints block more of the search space where the algorithm is looking. Scenario 4 and 5 show that their constraints cause the algorithm to occasionally fail in generating a successful adversarial example. This can be seen in Figure 5.1a, where the point of scenario 4 and 5 receive negative similarity scores, meaning the anomaly detector would not classify it as benign traffic.

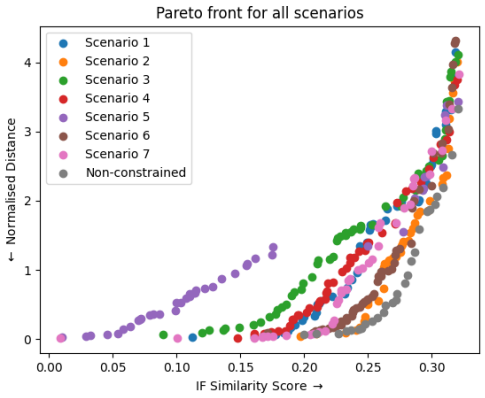
Limiting more features does not necessarily mean a worse performance. In scenario 7, there are 8 different features cannot be altered. However, the performance in Figure 5.1a is still very similar and the Pareto front in Figure 5.1b is very close to the Pareto front of the unconstrained results. From this we see that features have different levels of influence on the performance of AGONI. The total length of a packet and the destination port appear to be important for generating valid adversarial examples. The combination of sequence numbers and acknowledgment numbers also influence the quality of generated solutions. The TCP Flags have a slight influence whereas the set of features from Scenario 7 have no influence on the performance at all, generating similar solutions as to when no constraints are applied.

5.3. Special scenario: an adaptive defense

In the case that AGONI bypasses NIDSs on multiple occasions, the defending party will be able to adapt their ML model from their anomaly-based NIDS to new data. This results in an adaptive defense that is better prepared for the adversarial examples that AGONI generates. We generated a dataset, exclusively with packets generated by AGONI ($n = 500$), and trained a new IF to run a different version of AGONI. In Figure 5.2, the fitness scores are plotted of solutions generated by AGONI with the updated IF. We evaluated these solutions with the old IF from AGONI to inspect the effect that an updated IF had on the solutions. Since the solutions only receive a different similarity score, the adversarial distance remains the same. The 2 sets of points are only shifted along the horizontal axis. The figure shows that AGONI with an updated IF is still able to generate valid adversarial examples. These solutions are



(a) The fitness scores obtained for all scenarios. In scenario 4 and 5, AGONI occasionally fails to generate successful adversarial examples.



(b) The Pareto front for all scenarios. Scenarios with more constraints push the front back, limiting the quality of AGONI's adversarial examples.

Figure 5.1: The performance of AGONI is visualised when applied to different scenarios. With more constraints, the quality of the generated solutions decreases.

also successful adversarial examples when evaluated with the old IF, however the similarity score is lower overall. Updating the classifier of the model is uneffective in stopping the packets generated by AGONI. It is unclear whether or not updating the model multiple times is more effective.

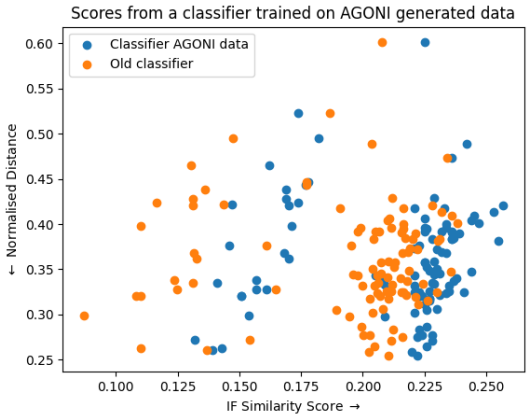


Figure 5.2: The performance of AGONI with an IF that was trained on data generated by AGONI. We see that AGONI is still able to generate valid adversarial examples. When evaluated on the old original IF from AGONI, the adversarial examples are still successful but with a lower similarity score.

5.4. Conclusions

This chapter aimed to answer RQ3: *Do attack constraints affect the quality of adversarial examples generated by Genetic Algorithms?*. We presented 7 scenarios for generating network packets and applied the corresponding constraints to AGONI to evaluate the effect those constraints had on the performance. For 5 scenarios, AGONI is still able to consistently generate valid adversarial examples but with lower similarity scores on average. For 2 scenarios, the success rate decreased where the algorithm generated valid network packets that were not adversarial examples. These 2 scenarios showed that certain features have more influence on the performance than others, indicating that the total length of a packet and the destination port cause the largest drop in performance.

6

Evaluating GA with Suricata

Chapter 4 and 5 showed experiments to determine the quality of the generated solutions measured by the defined fitness metrics. In this chapter, we aim to answer RQ4: *Do existing NIDSs detect adversarial internet packets?* Confirming whether or not the generated solutions of AGONI can bypass an existing NIDS gives extra insight into the quality of the solutions. We use Suricata to detect anomalies, an open-source, signature-based NIDS. Suricata has an existing set of detection rules, with which it can detect various types of malicious traffic, but also check for packet validity.

6.1. Inspecting CTU-13

We want to create adversarial examples for the CTU-13 dataset [58]. We first let Suricata inspect the dataset to see whether it can detect malicious behaviour. It would not be significant if our adversarial examples are able to bypass Suricata if the original dataset can already do that. After inspecting the full PCAP, Suricata raised an alert 128 times with 9 unique alerts. Since we are generating packets individually, we took a fragment of the full PCAP and split it into separate PCAPs consisting of 1 packet and let Suricata inspect those. That resulted in 44 alerts with 6 unique alerts divided over 20,000 PCAPs. These results show that Suricata does catch malicious traffic within the CTU-13 dataset.

6.2. Results

From the experiments in Chapter 5, AGONI generated 4677 unique individuals in the final populations. We put all packets into separate PCAPs and let Suricata inspect them. From the 4677 PCAPs, Suricata raised 3 alerts in total, meaning that 99.93% of the generated solutions went undetected. The alerts that Suricata raised were all of the type:

SURICATA STREAM 3way handshake SYNACK in wrong direction

This alert indicates that a packet tried to initiate a TCP three-way handshake with the wrong flags. In Figure 6.1, we visualise the three-way handshake that consists of 3 steps:

- The sender sends a packet with the SYN flag.
- The receiver sends a packet back with both the SYN and the ACK flag.
- The sender ends the handshake by sending a packet with the ACK flag.

The alerts detected that the sender is starting off the handshake with both the SYN and the ACK flag, therefore the alert of 'SYNACK in wrong direction' is raised. The packet validity does not cause this alert. The packet does not follow the rules of network communication. However, we cannot add a constraint that prevents having both the SYN and ACK flag, since this can occur in other scenarios (e.g., other devices trying to connect to you). Therefore, we can state that while it is possible to reliably generate valid internet packets that bypass existing NIDSs, it is extremely hard to generate an exhaustive set of constraints that guarantees the packet following the rules of network communication. When looking at generating packet sequences, we state the hypothesis that anticipating the rules of internet traffic is

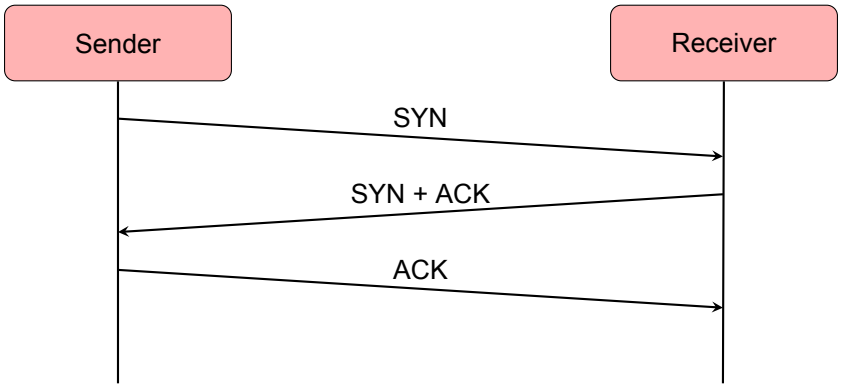


Figure 6.1: The structure of the TCP 3-way handshake. First, the sender sends a packet with the SYN flag. Then, the receiver sends a packet back with both the SYN and the ACK flag. Finally, the sender ends the handshake by sending a packet with the ACK flag.

more challenging than only using individual packets since more communication rules of networks apply. Individual packets can already occasionally break those rules and raise alerts in NIDSs, so for packet sequences we expect this to occur quicker and more often.

6.3. Conclusions

This chapter aimed to answer RQ4: *Do existing NIDSs detect adversarial internet packets?* We confirmed that Suricata was able to detect malicious traffic in the CTU-13 dataset and then let Suricata inspect the individual packets that AGONI generated. This showed that AGONI was able to bypass Suricata 99.93% of the time. The remaining 0.07% cannot be prevented with extra constraints, since such constraints cause violations in other scenarios, meaning that creating an exhaustive set of constraints to guarantee that packets follow the rules of network communication is challenging.

7

Comparing GA against other solutions

This chapter aims to answer RQ5; *How does AGONI perform compared to existing black-box methods for generating adversarial examples?* We compare AGONI with two methods: randomised fuzzing and the Boundary Attack from Brendel et al. [33]. We inspect the performance of every method and analyse the (dis-)advantages of every method to determine the contributions of AGONI.

We run the 3 methods several times with the following quantities:

- AGONI: 100 times.
- Boundary Attack: 100 times.
- Random: 5000 times.

We allow the Random Attack to have more attempts to let the randomised fuzzing take advantage of the short computation time to find better solutions.

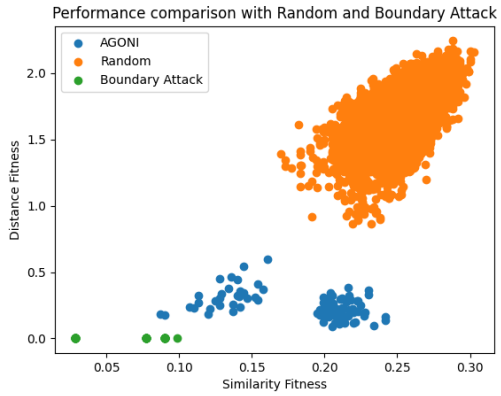
7.1. Results

The results of the 3 methods are shown in Figure 7.1. The Random Attack forms one big cluster with overall better similarity fitness, but also with adversarial distances that are around 3 to 4 times as large as the adversarial distances of AGONI. AGONI overall does a better job at maximising the similarity score and minimising the adversarial distance. The Boundary Attack consistently finds solutions with an adversarial distance of near-zero, but does so with similarity scores that are worse than both the Random Attack and AGONI.

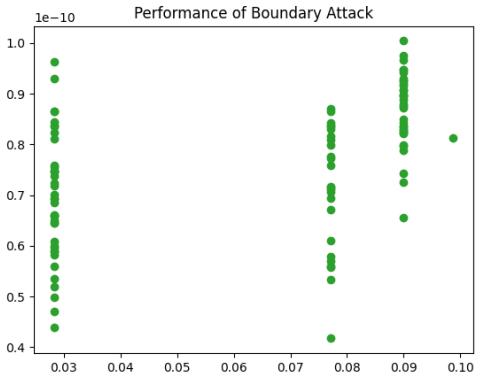
Another aspect that we compare is the validity rate of the methods; how many of the generated solutions are valid network packets? We apply the constraints on each packet, defined in Section 3.4, and sum the number of invalid packets for each method. The results, shown in Table 7.1, tell us that 18% of the Random Attack solutions are invalid packets and that the Boundary Attack is unable to generate a single valid packet. AGONI is the only method to consistently generate valid packets.

Method	Number of packets	Invalid packets
AGONI	100	0
Random Attack	5000	889
Boundary Attack	100	100

Table 7.1: The number of invalid packets each method generates. AGONI has an invalid packet rate of 0%, unlike the Random Attack (18%) and the Boundary Attack (100%).



(a) The results of all the methods. AGONI performs better in optimising for both fitness metrics. The Random Attack receives better similarity score but achieves that with high adversarial distances. The Boundary Attack produces the opposite, with solutions of a low adversarial distance but also a low similarity score.



(b) The results of just the Boundary Attack to better distinguish individual results. All results have a low adversarial distance, but also a very low similarity score.

Figure 7.1: The performance of AGONI compared to the Random Attack and the Boundary Attack.

Step #	BA Value	Step Size
1	63	1
2	62	1
3	61	1
4	60.5	0.5
5	60.25	0.25

Table 7.2: The values and stepsize of the Boundary Attack per step. The attack creates invalid non-integer values when approaching the classification border. The step size becomes smaller than 1, causing the next steps to result in non-integers.

7.1.1. Networking Boundary Attack

The Boundary Attack is not effective at finding valid adversarial examples. The decreased step sizes of the Boundary Attack cause for non-integer values that invalidate the result. Table 7.2 show the steps of a hypothetical scenario when executing the Boundary Attack, where the classification border exists at 60. The value 60 is no longer an adversarial example, so the algorithm tries smaller steps to get closer to the border of 60, resulting in non-integer values that are unsuited for network packets. To solve these issues, we introduce an adjusted version of the original algorithm, the *Networking Boundary Attack* (NBA). The NBA adds 3 changes to the original attack to allow it to generate valid packets.

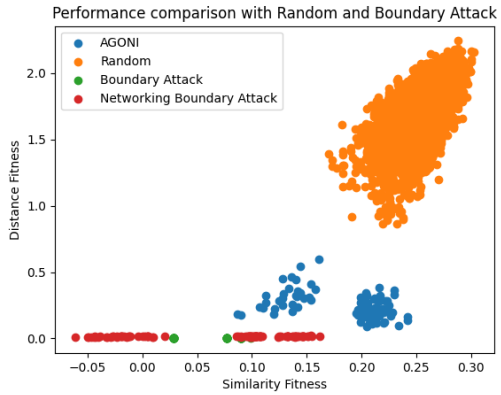
The first change is rounding the values for each feature to integer values, so that each feature has a valid value. The issue that arises from this change is that some of the adversarial examples generated are so close to their original instance, that rounding the adversarial example just results in the original instance. To solve this, the second change is to follow the previous steps taken by the Boundary Attack until an adversarial example is found that does not round to the original instance. Together, these changes allow the NBA to find an adversarial example with integer values that are different from the original instance. However, these 2 changes are still not sufficient to find valid adversarial examples since every packet still ends up violating the contextual constraints defined in Chapter 3.4. Therefore, the third and final change is to apply these contextual constraints making sure the packets no longer violate them.

The changes performed by the NBA influence both the adversarial distance and the similarity score. Figure 7.2 shows the results of all the methods including the NBA. The NBA now consistently generates valid packets with only a slight increase in the adversarial distance. However, the similarity score occasionally turns negative due to the changes of the NBA, meaning it is no longer a successful adversarial example. Table 7.3 shows the number of packets that are invalid together with the number of pack-

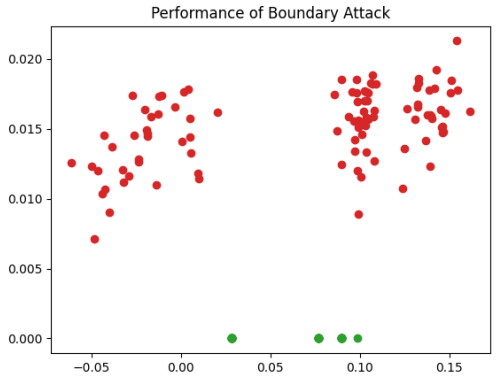
Method	Number of packets	Invalid packets	Adversarial Examples	Success
AGONI	100	0	100	100%
Random Attack	5,000	889	5,000	82%
Boundary Attack	100	100	100	0%
NBA	100	0	74	74%

Table 7.3: The number of invalid packets and successful adversarial examples, indicating the overall success rate of each method. AGONI is the only method to consistently generate valid adversarial examples.

ets that are not successful adversarial examples. This gives every method a success rate indicating how many packets generated by that method are both valid and successful adversarial examples. We see that only AGONI has a 100% success rate, with the Random Attack and the NBA performing less consistent.



(a) The results of all the methods including the Networking Boundary Attack. The NBA occasionally fails to generate adversarial examples.



(b) The results of the Boundary Attack and the Networking Boundary Attack to better distinguish the effects of the changes performed by the NBA. The NBA now consistently generates valid packets with slightly increased adversarial distance, but occasionally fails to generate an adversarial example.

Figure 7.2: The performance of AGONI compared to the other methods, including the Networking Boundary Attack.

7.1.2. Networking Boundary Attack with confidence levels

The NBA has shown to reliably generate valid network packets, but not all of them are adversarial examples. We evaluate the behaviour of the NBA when constrained with different confidence levels, which represents the lowest score the similarity score is allowed to receive. With these confidence levels, the success rate of the NBA should increase since no packets can be generated anymore with a negative similarity score. In Figure 7.3, we show the performance of the NBA over multiple confidence levels. For a confidence level of 0.10, the NBA generates solutions that are of similar quality to the solutions that AGONI generates. With higher confidence levels of 0.15 and 0.20, the NBA performs worse and generates solutions that require a higher adversarial distance. We observe that with a higher confidence level, the NBA finds the boundary much earlier and once found that it has less area to move to. A confidence level of 0.10 results in the best balance between enforcing high similarity score while also allowing for enough movements around the boundary to achieve low adversarial distances.

7.1.3. Differences in adjustments by every method

The methods mentioned in this chapter have different approaches to generate their answers. In Table 7.4, we show the resulting adversarial examples generated by different methods. The Genetic Algorithm of AGONI was able to evolve the features to values that are close to the original instance. The Random Attack overall generated no features that are close to the original, except for some flags where the Random Attack had a 50% chance to pick the correct flag. This also caused the Random Attack to be the only attack to fail to get every flag correct. All other methods with a more structured approach

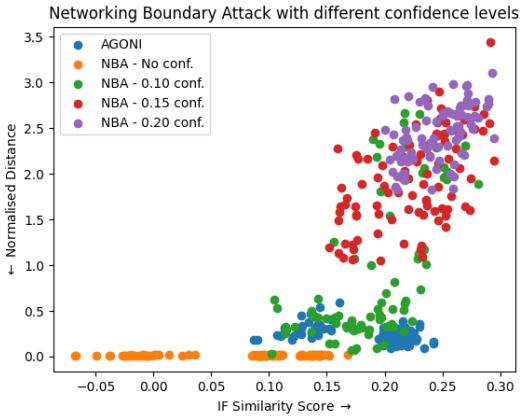


Figure 7.3: The performance of the Networking Boundary Attack with different confidence levels. When the NBA has to perform with a confidence level of 0.10, the NBA occasionally finds solutions that can compete with the results from AGONI.

were able to successfully generate the correct values for the TCP flags. The Boundary Attack generates values for every feature that are nearly identical to the original instance, but part of those values are non-integer values causing the packet to be invalid. Finally, the Networking Boundary Attack generates values slightly further off than the Boundary Attack but the values do represent a valid network packet.

7.2. Conclusions

This chapter aimed to answer RQ5: *How does AGONI perform compared to existing black-box methods for generating adversarial examples?* We compared AGONI to the Random Attack and the Boundary Attack, and introduced an improved version of the Boundary Attack called the Networking Boundary Attack. When comparing the results, the Boundary Attack showed to be unfit for generating network packets, failing to generate a single valid packet. The Random Attack fails to find valid packets 18% of the time. The Networking Boundary Attack performs better than the original, only finding valid packets, but fails to find adversarial examples 26% of the time. Combining the success rate of valid packets and adversarial examples, AGONI outperforms the other methods with a success rate of 100%. The Random Attack, Boundary Attack and Networking Boundary Attack only achieved a success rate of 82%, 0% and 74% respectively.

Feature	Original	AGONI	Random	Boundary	Networking Boundary
Total Length	62	8,192	4,977	62.01	579
Time to Live	128	95	218	128.50	129
ToS	0	0	55	0	2
Identification	61,591	64,747	31,168	61,591.94	61,274
Source Port	1,105	6,144	54,734	1,105.02	1,237
Destination Port	25	8,192	43,726	25.01	416
Seq. Number	0	0	731,012,111	0	35,848,093
Ack. Number	0	0	1,704,538,114	0	1,298,159
Urgent Pointer	0	0	27,778	0	0
Window	64,240	65,024	30,755	64,240.98	63,776
CRW Flag	0	0	0	0	0
ECN Flag	0	0	1	0	0
URG Flag	0	0	0	0	0
ACK Flag	0	0	1	0	0
PSH Flag	0	0	0	0	0
RST Flag	0	0	0	0	0
SYN Flag	1	1	0	1	1
FIN Flag	0	0	0	0	0
Similarity	-	0	0	0	0
Distance	0	$0.129 \cdot 10^{5}$	$0.185 \cdot 10^{10}$	$0.144 \cdot 10^{1}$	$0.35871590 \cdot 10^{8}$
Distance (norm.)	0	0.237	$0.226 \cdot 10^{1}$	$0.196 \cdot 10^{-2}$	$0.179 \cdot 10^{-1}$

Table 7.4: A single solution generated by each method. The Boundary Attack has a low adversarial distance, but non-integer values for features causing the packet to be invalid. The Random Attack has a good similarity score but the generated values are far off from the original.

8

Discussion

In this thesis, we discussed the problem of generating individual network packets as adversarial examples aimed to bypass NIDSs. We discuss the difficulties of generating network sequences in Section 8.1, cover ethical concerns for this line of research in Section 8.2, limitations of AGONI in its current state in 8.3 and final recommendations to improve or further verify the performance of AGONI in Section 8.4.

8.1. Guaranteeing validity of packet sequences

AGONI is designed for generating singular packets. When moving on from generating singular packets to packet sequences, we have to take more factors into account. The sequence of packets now needs to follow protocol expectations from the network, which requires significant in-depth knowledge about the rules, parameters and dependencies that exist in network traffic. This means the set of constraints for individual packets must be extended to incorporate validity for packet sequences. Additionally, computer networks can respond differently to traffic than others. A simple example is a network that resets their connection after a certain time-period as a security measure. This already requires the traffic for communicating with this network to look different than when communicating with a regular network that does not reset its connections.

Apart from the validity of packets, a sequence introduces other dimensions that can signal to an NIDS if a sequence is regular traffic or a handcrafted set of packets. First is the time in between packets. If two packets are sent after each other, it does not matter whether the interval is 0.1 or 0.2 seconds. The sequence is still valid. However, if the interval is an irregular value compared to benign traffic, an NIDS could label it as suspicious. The second aspect is the number of packets in a sequence. A sequence of x packets can be just as valid as a sequence of $x+1$ packets, but similar to the time intervals, NIDSs can use the number of packets as an indication for the suspiciousness of the packet sequence.

The most problematic factor is not that multiple packets must be generated, but that the packets must interact with the response of the network. A network response to packets is different per network [64], which significantly increases the complexity of generating packets. An example of a core network interaction is the 3-way handshake that is required to start a connection when using the TCP protocol. Here, the sender needs to generate a proper third packet that is dependent on the contents of the response of the receiver. Whilst not the most complicated interaction, numerous other interactions exist that all expect slightly different behaviour. Traffic patterns and requirements can vary across computer networks and can even fluctuate greatly within the same network. Depending on the application, scale of the network and security policy, the way that NIDSs verify and intercept traffic varies vastly. Networks are also dynamic, with traffic patterns and conditions changing constantly. Due to these problems, generating packet sequences was out of scope for this research and the focus was put on creating individual packets.

8.2. Ethical concerns

We created AGONI with the purpose of showing vulnerabilities in existing defense systems. This means that while we can use it to help improve an NIDS, malicious parties can use it to exploit the vulnerabilities that AGONI finds with the adversarial examples it generates. With the current state of AGONI, one cannot create network sequences that perform a network attack, so the possible damage that can be inflicted is minimal. However, one is able to craft a network attack and replace individual packets with packets generated by AGONI. By applying constraints on those packets, we have shown it still possible to generate valid adversarial examples, allowing one to maintain attack functionality. For this scenario, it is unknown if this helps the likelihood of success of the attack, but the possibility does exist.

We warn any party who proceeds with this line of research that any technological advancements in this field grants the same technological advancements to malicious parties. Therefore, one should research how to defend against any new attacks made possible by new research in this field. This gives parties that have to protect vital assets time to set up and strengthen defenses before malicious parties can deploy attacks.

8.3. Limitations

As stated in Section 3.3, we trained the IF on data of malicious traffic and invert the scores from the IF. This causes the algorithm to only be able to generate adversarial examples for attacks that are included in the training data. If an attack is not included in the training data, the IF needs to be retrained with a dataset that does include that attack in order to generate an adversarial example. An additional limitation is that we only trained the IF on the data of the CTU-13 dataset and did not investigate the performance of AGONI when the IF is trained on datasets with malicious network traffic other than the traffic in CTU-13 (e.g., the 1998 DARPA Intrusion Detection Evaluation dataset [1].

AGONI checks the validity of packets with a set of hard constraints. We created the set of constraints to the best of our knowledge, but we acknowledge the possibility that other constraints might exist. For example, constraints might exist related to the Options field for both IPv4 and TCP, which were excluded in this research due to their limited use.

One of the metrics we used for evaluating the quality of solutions is the adversarial distance. While we know that a smaller adversarial distance makes for a better adversarial example, we do not know when the adversarial distance is small enough to represent a good adversarial example. This means that, while we can clearly compare different methods, it is difficult to judge how well a method would work in practice.

8.4. Recommendations

We evaluated AGONI by testing it on a rule-based NIDS and comparing it with other black-box methods for generating adversarial examples. However, we did not test AGONI on an anomaly-based NIDS. It would be interesting to also test this, to further evaluate the performance of AGONI in different settings.

AGONI is configured to generate packets with the IPv4 and TCP protocol and is evaluated with a model trained on data consisting of IPv4 and TCP data. This choice was mainly motivated by the popularity of IPv4 and TCP. An interesting experiment would be to configurate AGONI to work with other often-used protocols (e.g., UDP) and evaluate if it yields a similar performance.

As stated in the limitations, only the CTU-13 dataset was used for training the IF that AGONI uses. If one were available, a dataset with benign network traffic would be very useful. We could train the IF on the benign data which would mean that AGONI would be able to construct adversarial examples for any type of malicious network traffic and not just the malicious traffic present in the used dataset. As stated, such datasets are hard to find, so one could use other datasets with different types of malicious traffic to investigate the effect of different datasets on the performance of AGONI.

[1] https://www.ll.mit.edu/r-d/datasets/1998-darpa-intrusion-detection-evaluation-dataset

9 Conclusion

Recent work has shown that adversarial examples can also be generated for network traffic to bypass Network Intrusion Detection Systems. However, said work has yet to show how to generate network packets that can be used to launch a cyber-attack. In this thesis, we proposed a Genetic Algorithm to generate adversarial examples representing individual valid network packets, able to bypass an NIDS. We presented a new algorithm called AGONI that is able to reliably find adversarial examples and guarantee validity with the use of a set of constraints.

We studied the effect of different crossover, mutation and selection operators on the performance of AGONI. We found that there was no significant difference in the performance of different operators. We evaluated the performance of AGONI on various attack scenarios and against the rule-based NIDS Suricata. We found that the success rate remained the same under the constraints from most of the scenarios. Only in particular scenarios with several constraints was the success rate of the algorithm not 100%, meaning that it was still able to find valid adversarial examples. Against Suricata, 99.93% of the generated packets successfully bypassed the defense. The few packets that failed, violated rules based on the expected behaviour of traffic. These violations could not be prevented with additional constraints. Finally, we compared AGONI with other black-box methods for generating adversarial examples. We compared with the Random Attack and the Boundary Attack, and introduced the Networking Boundary Attack that was more suitable than the Boundary Attack for generating valid network packets. We found that only AGONI had a 100% success rate in generating valid adversarial examples and outperformed the Random Attack (82%), the Boundary Attack (0%) and the Networking Boundary Attack (74%).

9.1. Contributions

We looked at the use of adversarial examples in the domain of internet traffic in our literature review. Modern work focuses on adversarial examples for network flows. Recreating network attacks from these adversarial examples is non-trivial, making it hard to show if NIDSs have any weaknesses that can be exploited. Our main contributions are:

- A new algorithm for generating valid network packets that bypass NIDSs.
- A study on the influence of different Genetic Algorithm operators on the quality of generated solutions.
- A quantification of AGONI's performance against the Suricata NIDS.
- A version of the Boundary Attack improved for the domain constraints of network traffic, called the Networking Boundary Attack.
- A study on the performance of AGONI compared to other black-box methods.

9.2. Future work

The objective for this line of research is to generate a sequence of packets that acts as an adversarial example for a network attack, while maintaining attack functionality. As shown in this thesis, guaran-

teeing traffic validity is challenging, both for individual packets and packet sequences. ... Future work can focus on a literature or field study to determine the expected behaviour of network packets during all stages of network communications. Such a study can help determine more constraints for individual packets and packet sequences, which helps with guaranteeing the traffic validity. Using these new constraints, future work can extend the structure of AGONI to create a new (MO-)GA in order to generate packet sequences that maintain attack functionality.

We have shown in Chapter 5 that retraining a classifier on data generated by AGONI can still be fooled by AGONI. We have also shown that the effectiveness of AGONI is limited when applying constraints that state what features AGONI can change. Future work can focus on creating a defense strategy that can reliably detect packets generated by AGONI.

References

[1] Sumit Kumar, Sumit Dalal, and Vivek Dixit. “The OSI model: Overview on the seven layers of computer networks”. In: *International Journal of Computer Science and Information Technology Research* 2.3 (2014), pp. 461–466.

[2] Physical Layer. “Ieee standard for ethernet”. In: (2018).

[3] Manal M Alhassoun and Sara R Alghunaim. “A Survey of IPv6 Deployment”. In: *International Journal of Advanced Computer Science and Applications* 7.9 (2016).

[4] Jon Postel. *Rfc0793: Transmission control protocol*. 1981.

[5] James P Anderson. “Computer security threat monitoring and surveillance”. In: *Technical Report, James P. Anderson Company* (1980).

[6] John R Vacca. *Computer and information security handbook*. Newnes, 2012.

[7] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. “Explaining and harnessing adversarial examples”. In: *arXiv preprint arXiv:1412.6572* (2014).

[8] Alex Kantchelian, J Doug Tygar, and Anthony Joseph. “Evasion and hardening of tree ensemble classifiers”. In: *International conference on machine learning*. PMLR. 2016, pp. 2387–2396.

[9] Gary J Saavedra et al. “A review of machine learning applications in fuzzing”. In: *arXiv preprint arXiv:1906.11133* (2019).

[10] Patrice Godefroid. “Fuzzing: Hack, art, and science”. In: *Communications of the ACM* 63.2 (2020), pp. 70–76.

[11] Martin Eberlein et al. “Evolutionary grammar-based fuzzing”. In: *International Symposium on Search Based Software Engineering*. Springer. 2020, pp. 105–120.

[12] Charles Darwin. “The descent of man”. In: *New York: D. Appleton* (1871).

[13] Tobias Blickle. “Tournament selection”. In: *Evolutionary computation* 1 (2000), pp. 181–186.

[14] David E Golberg. “Genetic algorithms in search, optimization, and machine learning”. In: *Addion wesley* 1989.102 (1989), p. 36.

[15] Peter JB Hancock. “An empirical comparison of selection methods in evolutionary algorithms”. In: *AISB workshop on evolutionary computing*. Springer. 1994, pp. 80–94.

[16] Riccardo Poli and William B Langdon. “Genetic programming with one-point crossover”. In: *Soft Computing in Engineering Design and Manufacturing*. Springer, 1998, pp. 180–189.

[17] Gilbert Syswerda et al. “Uniform crossover in genetic algorithms.” In: *ICGA*. Vol. 3. 1989, pp. 2–9.

[18] Tzung-Pei Hong et al. “Evolution of appropriate crossover and mutation operators in a genetic process”. In: *Applied intelligence* 16 (2002), pp. 7–17.

[19] Alden H Wright. “Genetic algorithms for real parameter optimization”. In: *Foundations of genetic algorithms*. Vol. 1. Elsevier, 1991, pp. 205–218.

[20] Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* University of Michigan, 1975.

[21] John J Grefenstette. “Optimization of control parameters for genetic algorithms”. In: *IEEE Transactions on systems, man, and cybernetics* 16.1 (1986), pp. 122–128.

[22] Tzung-Pei Hong, Hong-Shung Wang, and Wei-Chou Chen. “Simultaneously applying multiple mutation operators in genetic algorithms”. In: *Journal of heuristics* 6 (2000), pp. 439–455.

[23] J Hesser and R Männer. “Towards an optimal mutation probability”. In: *Proceedings of the International Workshop Parallel Problem Solving from Nature, Springer Verlag*. 1990.

[24] J David Schaffer et al. “A study of control parameters affecting online performance of genetic algorithms for function optimization”. In: *Proceedings of the 3rd international conference on genetic algorithms*. 1989, pp. 51–60.

[25] Dirk Schlierkamp-Voosen. “Optimal interaction of mutation and crossover in the breeder genetic algorithm”. In: *International Conference on Genetic Algorithms; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA*. 1993.

[26] Tzung-Pei Hong and Hong-Shung Wang. “A dynamic mutation genetic algorithm”. In: *1996 IEEE International Conference on Systems, Man and Cybernetics. Information Intelligence and Systems (Cat. No. 96CH35929)*. Vol. 3. IEEE. 1996, pp. 2000–2005.

[27] Khalid Jebari and Mohammed Madiafi. “Selection methods for genetic algorithms”. In: *International Journal of Emerging Sciences* 3.4 (2013), pp. 333–344.

[28] David E Goldberg et al. *Real-coded genetic algorithms, virtual alphabets and blocking*. Citeseer, 1990.

[29] Babatunde A Sawyerr, Aderemi O Adewumi, and M Montaz Ali. “Benchmarking rcgau on the noiseless bbob testbed”. In: *The Scientific World Journal* 2015 (2015).

[30] P Kaelo and MM Ali. “Integrated crossover rules in real coded genetic algorithms”. In: *European Journal of Operational Research* 176.1 (2007), pp. 60–76.

[31] Yuri Lavinas et al. “Experimental analysis of the tournament size on genetic algorithms”. In: *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE. 2018, pp. 3647–3653.

[32] Anupriya Shukla, Hari Mohan Pandey, and Deepti Mehrotra. “Comparative review of selection techniques in genetic algorithm”. In: *2015 international conference on futuristic trends on computational analysis and knowledge management (ABLAZE)*. IEEE. 2015, pp. 515–519.

[33] Wieland Brendel, Jonas Rauber, and Matthias Bethge. “Decision-based adversarial attacks: Reliable attacks against black-box machine learning models”. In: *arXiv preprint arXiv:1712.04248* (2017).

[34] Battista Biggio et al. “Evasion attacks against machine learning at test time”. In: *Joint European conference on machine learning and knowledge discovery in databases*. Springer. 2013, pp. 387–402.

[35] Christian Szegedy et al. “Intriguing properties of neural networks”. In: *arXiv preprint arXiv:1312.6199* (2013).

[36] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. “Transferability in machine learning: from phenomena to black-box attacks using adversarial samples”. In: *arXiv preprint arXiv:1605.07277* (2016).

[37] Houda Jmila and Mohamed Ibn Khedher. “Adversarial machine learning for network intrusion detection: A comparative study”. In: *Computer Networks* (2022), p. 109073.

[38] Nicholas Carlini and David Wagner. “Towards evaluating the robustness of neural networks”. In: *2017 ieee symposium on security and privacy (sp)*. Ieee. 2017, pp. 39–57.

[39] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. “Deepfool: a simple and accurate method to fool deep neural networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2574–2582.

[40] Nicolas Papernot et al. “The limitations of deep learning in adversarial settings”. In: *2016 IEEE European symposium on security and privacy (EuroS&P)*. IEEE. 2016, pp. 372–387.

[41] Kathrin Grosse et al. “Adversarial examples for malware detection”. In: *European symposium on research in computer security*. Springer. 2017, pp. 62–79.

[42] Bojan Kolosnjaji et al. “Adversarial malware binaries: Evading deep learning for malware detection in executables”. In: *2018 26th European signal processing conference (EUSIPCO)*. IEEE. 2018, pp. 533–537.

[43] Javid Ebrahimi et al. “Hotflip: White-box adversarial examples for text classification”. In: *arXiv preprint arXiv:1712.06751* (2017).

[44] Robin Jia and Percy Liang. “Adversarial examples for evaluating reading comprehension systems”. In: *arXiv preprint arXiv:1707.07328* (2017).

[45] Nicholas Carlini and David Wagner. “Audio adversarial examples: Targeted attacks on speech-to-text”. In: *2018 IEEE security and privacy workshops (SPW)*. IEEE. 2018, pp. 1–7.

[46] Arkadiusz Warzyński and Grzegorz Kołaczek. “Intrusion detection systems vulnerability on adversarial examples”. In: *2018 Innovations in Intelligent Systems and Applications (INISTA)*. IEEE. 2018, pp. 1–4.

[47] Maria Rigaki. *Adversarial deep learning against intrusion detection classifiers*. 2017.

[48] Kaichen Yang et al. “Adversarial examples against the deep learning based network intrusion detection systems”. In: *MILCOM 2018-2018 ieee military communications conference (MILCOM)*. IEEE. 2018, pp. 559–564.

[49] Pin-Yu Chen et al. “Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models”. In: *Proceedings of the 10th ACM workshop on artificial intelligence and security*. 2017, pp. 15–26.

[50] Ian Goodfellow et al. “Generative adversarial networks”. In: *Communications of the ACM* 63.11 (2020), pp. 139–144.

[51] Mohammad J Hashemi, Greg Cusack, and Eric Keller. “Towards evaluation of nidss in adversarial setting”. In: *Proceedings of the 3rd ACM CoNEXT Workshop on Big Data, Machine Learning and Artificial Intelligence for Data Communication Networks*. 2019, pp. 14–21.

[52] Zilong Lin, Yong Shi, and Zhi Xue. “Idsgan: Generative adversarial networks for attack generation against intrusion detection”. In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2022, pp. 79–91.

[53] Ryan Sheatsley et al. “Adversarial examples for network intrusion detection systems”. In: *Journal of Computer Security* Preprint (2022), pp. 1–26.

[54] Fernando Gont, R Atkinson, and Carlos Pignataro. *Recommendations on filtering of IPv4 packets containing IPv4 options*. Tech. rep. 2014.

[55] Scott Bradner and Vern Paxson. *IANA allocation guidelines for values in the internet protocol and related headers*. Tech. rep. 2000.

[56] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. “Isolation forest”. In: *2008 eighth ieee international conference on data mining*. IEEE. 2008, pp. 413–422.

[57] Songqiao Han et al. “Adbench: Anomaly detection benchmark”. In: *arXiv preprint arXiv:2206.09426* (2022).

[58] Sebastian Garcia et al. “An empirical comparison of botnet detection methods”. In: *computers & security* 45 (2014), pp. 100–123.

[59] Trevor Hastie et al. *The elements of statistical learning: data mining, inference, and prediction*. Vol. 2. Springer, 2009.

[60] Apoorv Shukla et al. “Runtime Verification for Programmable Switches”. In: *IEEE/ACM Transactions on Networking* (2023).

[61] *Manually create and send raw TCP/IP packets*. `https://inc0x0.com/tcp-ip-packets-introduction/tcp-ip-packets-3-manually-create-and-send-raw-tcp-ip-packets/`.

[62] Ahmad Hassanat et al. “Choosing mutation and crossover ratios for genetic algorithms—a review with a new dynamic approach”. In: *Information* 10.12 (2019), p. 390.

[63] Matthew S Gibbs et al. “Minimum number of generations required for convergence of genetic algorithms”. In: *2006 IEEE International Conference on Evolutionary Computation*. IEEE. 2006, pp. 565–572.

[64] Wolfgang John, Sven Tafvelin, and Tomas Olovsson. “Trends and differences in connection-behavior within classes of internet backbone traffic”. In: *Passive and Active Network Measurement: 9th International Conference, PAM 2008, Cleveland, OH, USA, April 29-30, 2008. Proceedings 9*. Springer. 2008, pp. 192–201.

A

Scatter plots Operators

This appendix shows all the scatterplots for every operator, showing the fitness metrics of all the best solutions generated during the execution of the GA. The first 3 images are the combined plots as shown in Chapter 4. Then 4 images follow that show the results from each of the crossover operators. After that, 5 images with results from every mutation operator. Finally, 6 images containing results from every selection operator.

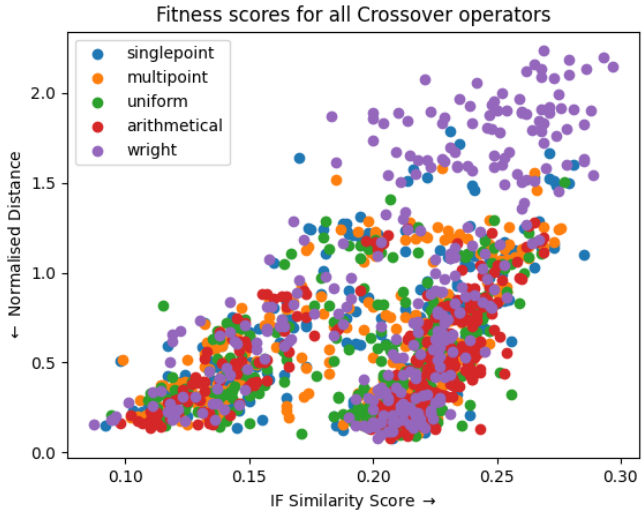


Figure A.1: Fitness values from individuals found by all crossover operators.

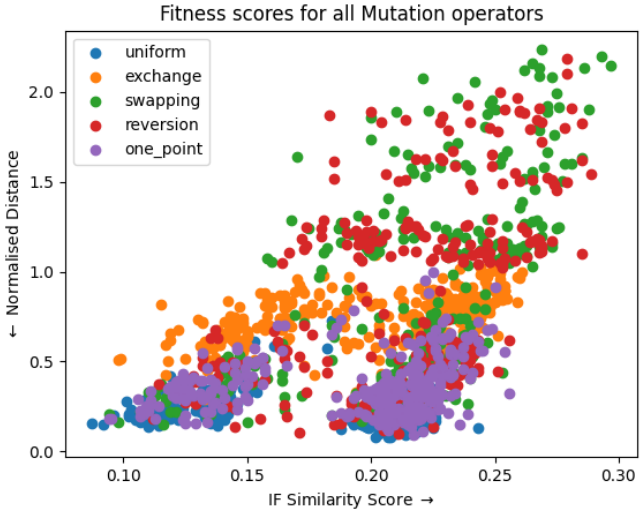


Figure A.2: Fitness values from individuals found by all mutation operators.

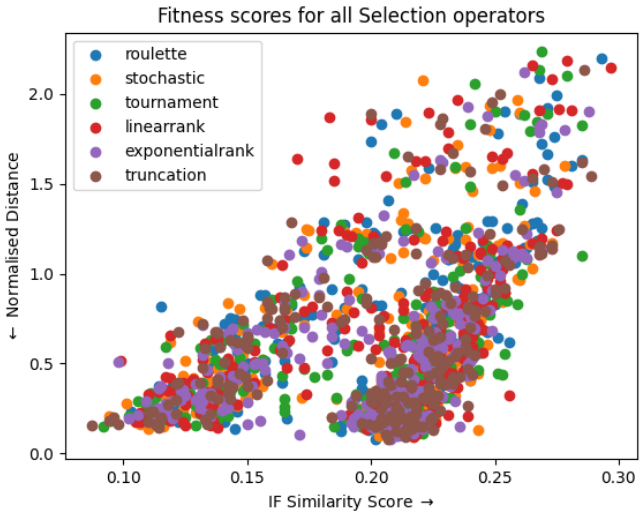
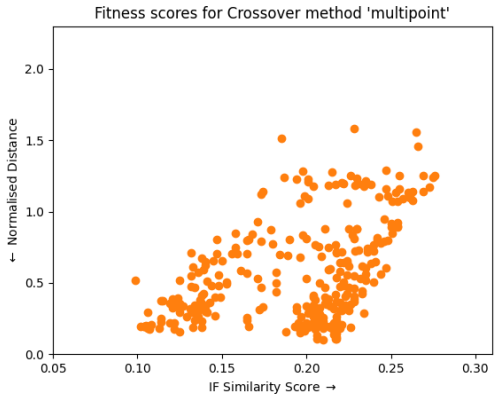
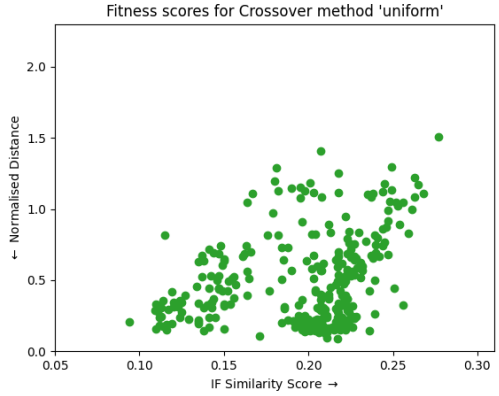


Figure A.3: Fitness values from individuals found by all selection operators.

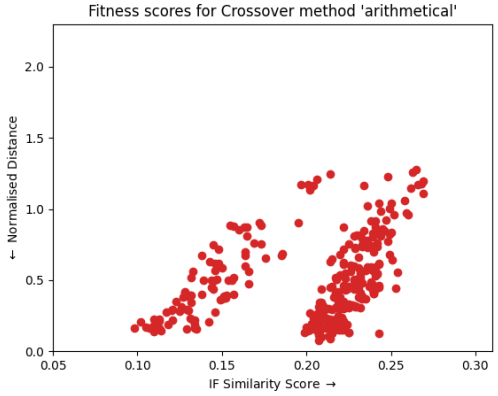
(a) Single Point Crossover



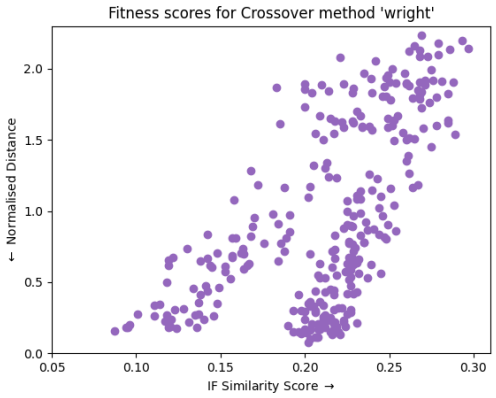
(b) Multi-Point Crossover



(c) Uniform Crossover



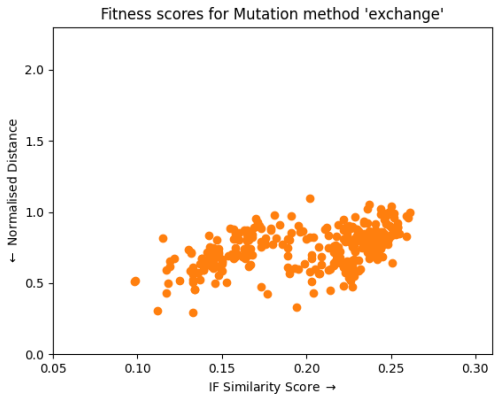
(d) Arithmetical Crossover



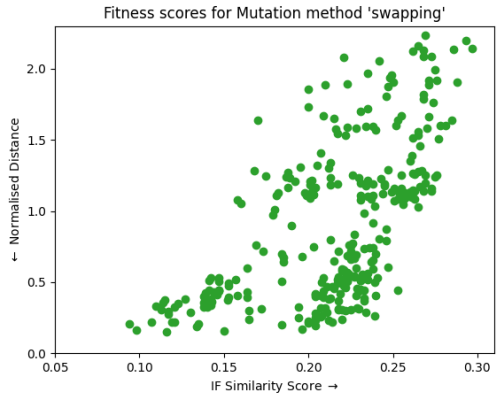
(e) Wright Crossover

Figure A.4: Fitness values from individuals, divided per crossover operator.

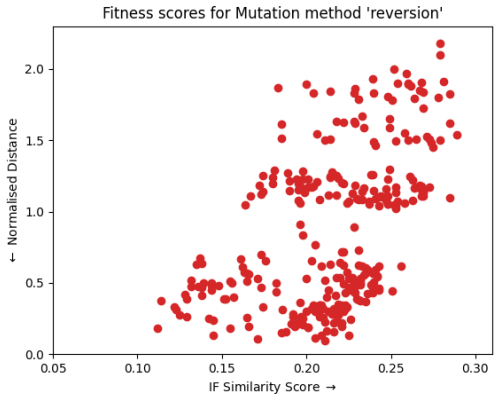
(a) Uniform Mutation



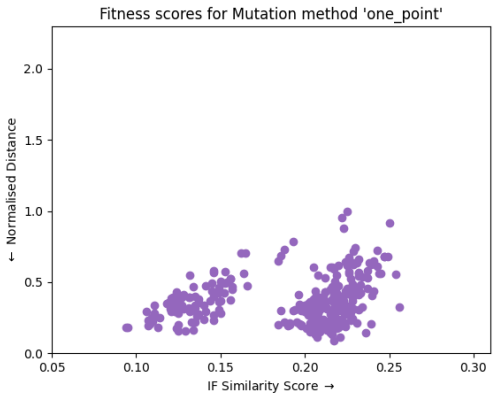
(b) Exchange Mutation



(c) Swapping Mutation

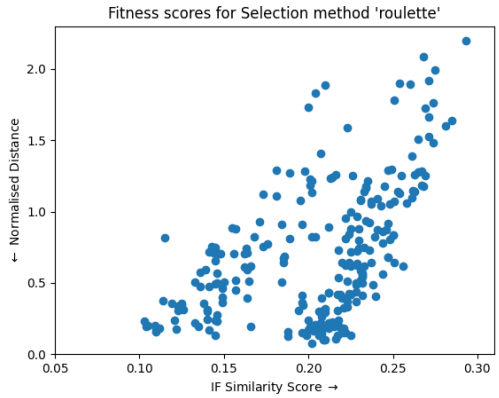


(d) Reversion Mutation

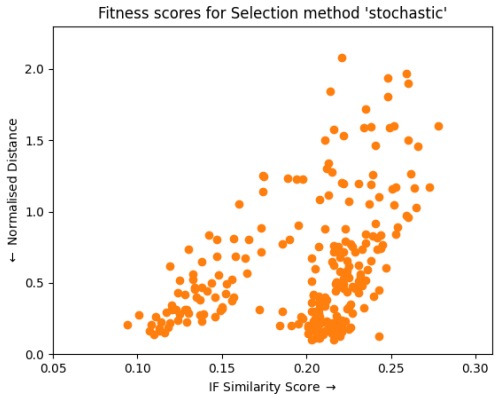


(e) One-Point Mutation

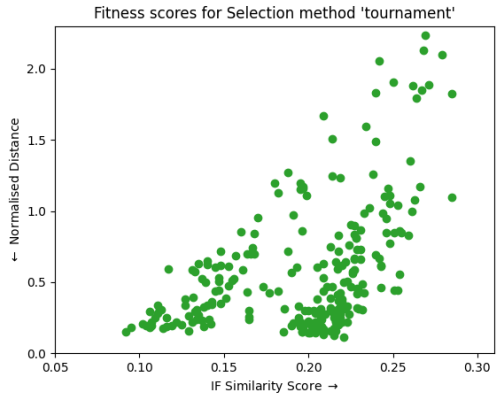
Figure A.5: Fitness values from individuals, divided per mutation operator.



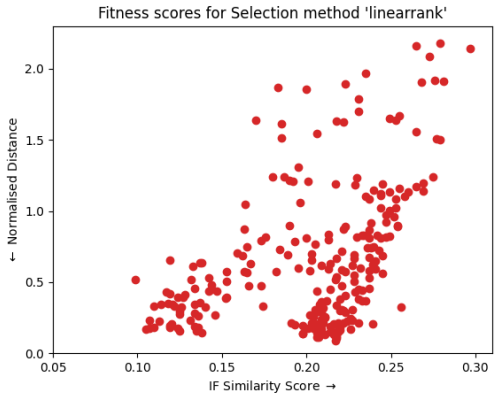
(a) Roulette Wheel Selection



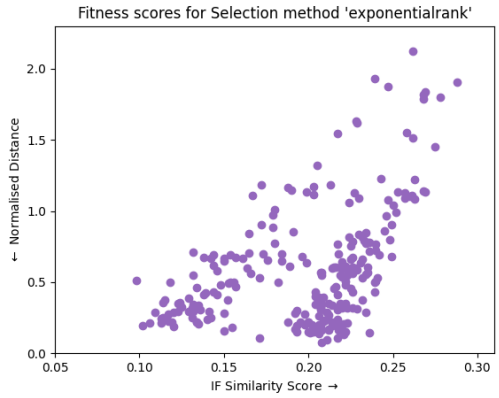
(b) Stochastic Universal Selection



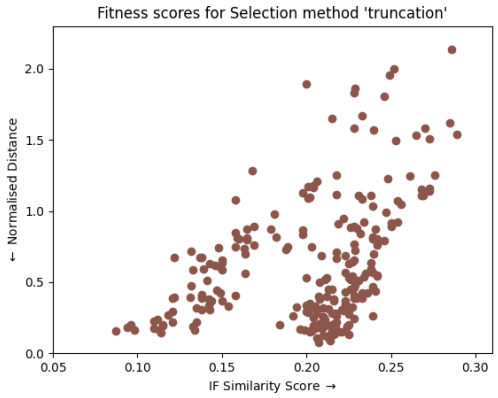
(c) Tournament Selection



(d) Linear Rank Selection



(e) Exponential Rank Selection



(f) Truncation Selection

Figure A.6: Fitness values from individuals, divided per selection operator.

B

Scatter plots Pareto-fronts

This appendix shows all the scatterplots for the Pareto-front found by every operator, showing the fitness metrics of all the solutions that are part of that front. The first 3 images are the combined plots as shown in Chapter 4. Then 4 images follow that show the results from each of the crossover operators. After that, 5 images with results from every mutation operator. Finally, 6 images containing results from every selection operator.

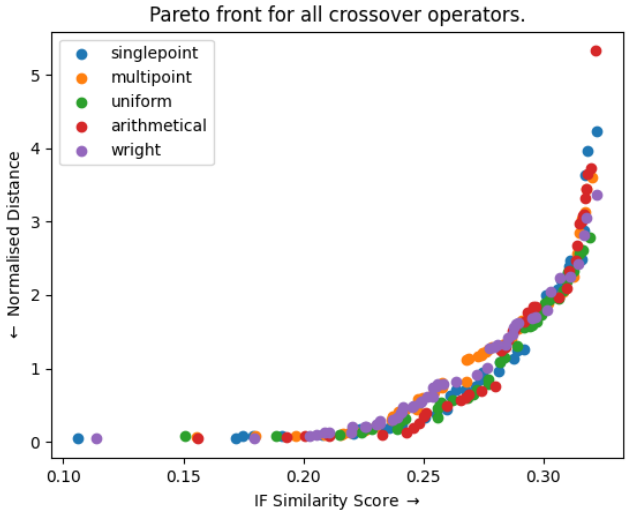


Figure B.1: Fitness values from the Pareto-front found by all crossover operators.

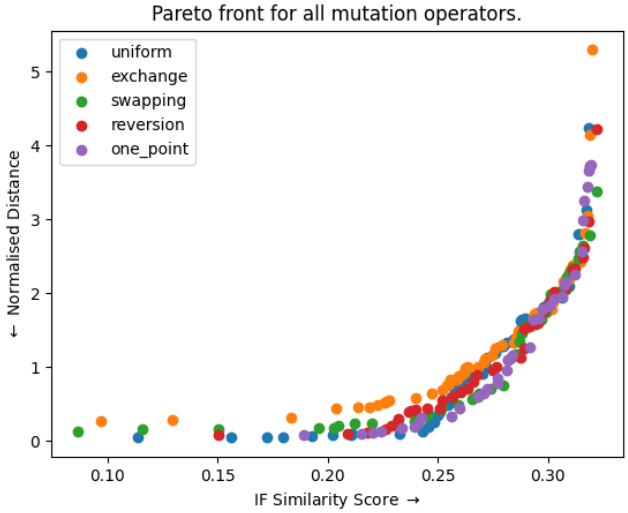


Figure B.2: Fitness values from the Pareto-front found by all mutation operators.

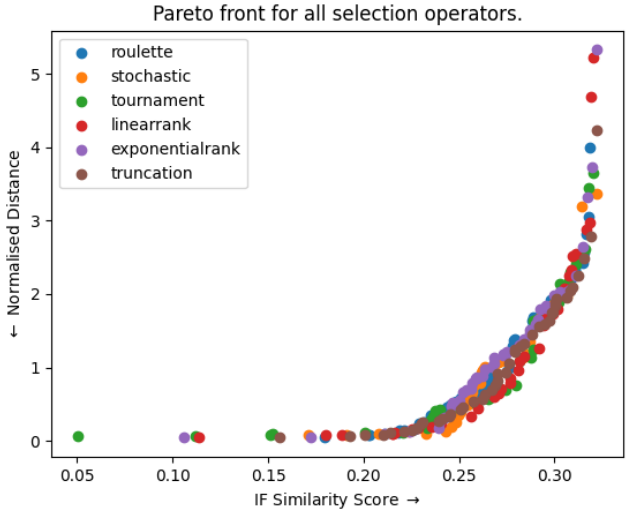
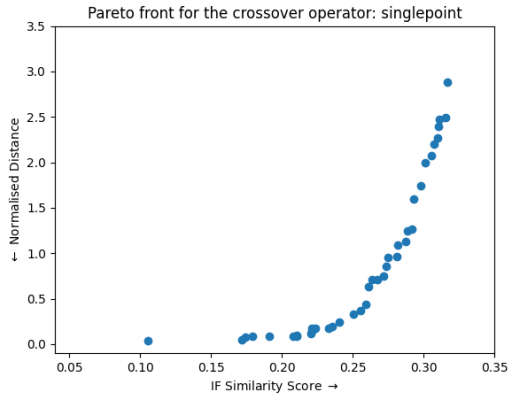
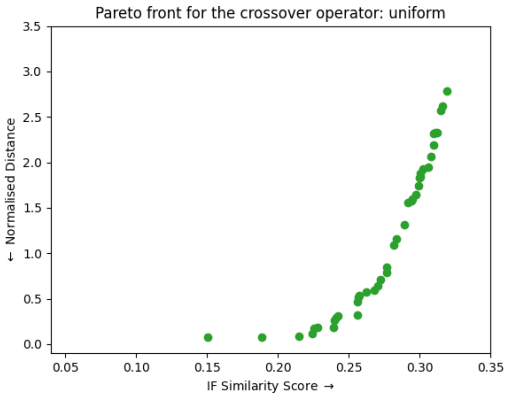


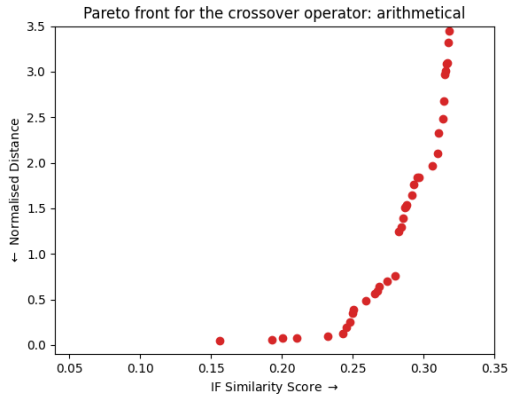
Figure B.3: Fitness values from the Pareto-front found by all selection operators.



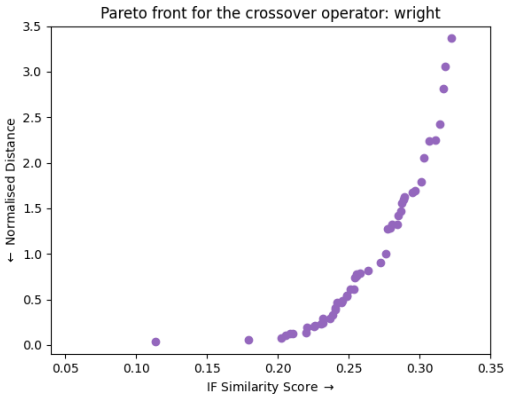
(a) Single Point Crossover



(b) Uniform Crossover

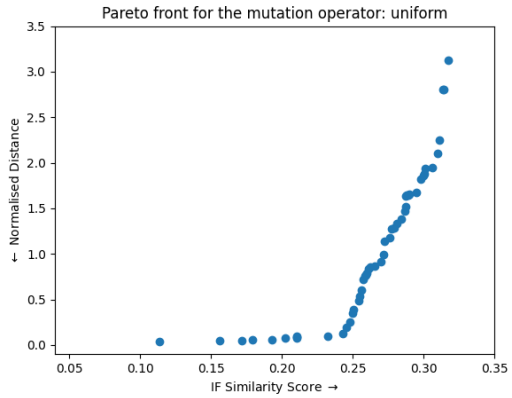


(c) Arithmetical Crossover

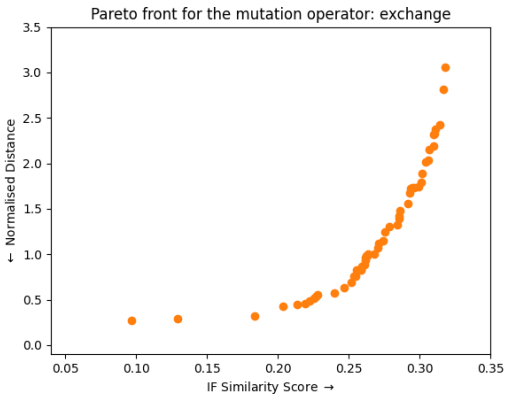


(d) Wright Crossover

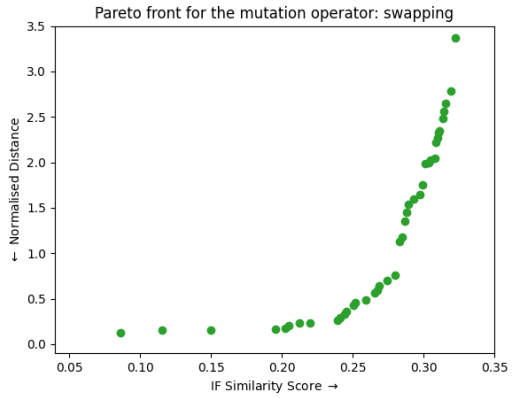
Figure B.4: Fitness values from individuals in the Pareto-fronts, divided per crossover operator.



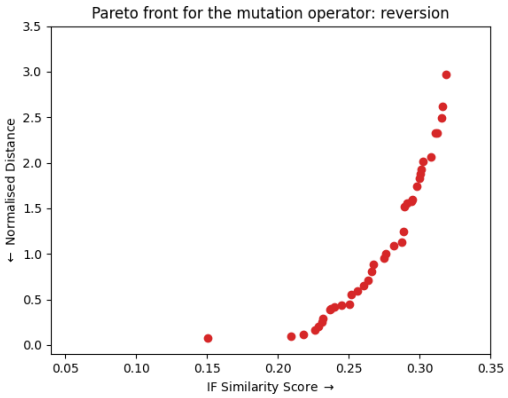
(a) Uniform Mutation



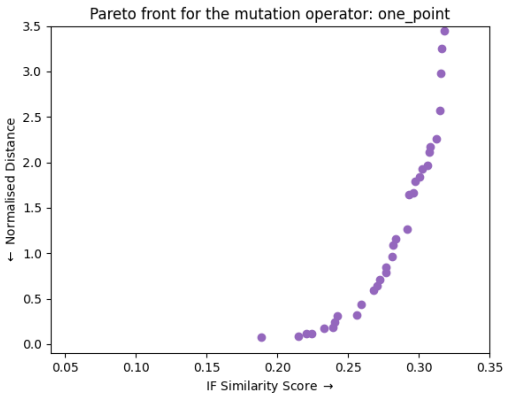
(b) Exchange Mutation



(c) Swapping Mutation

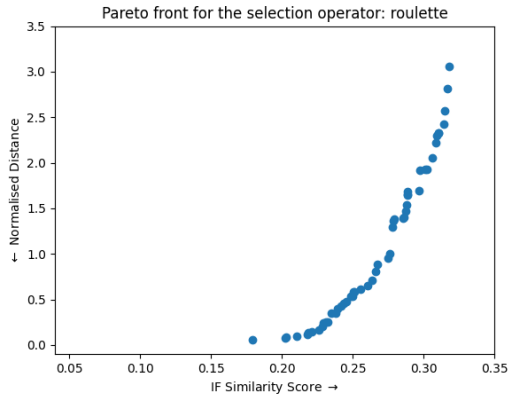


(d) Reversion Mutation

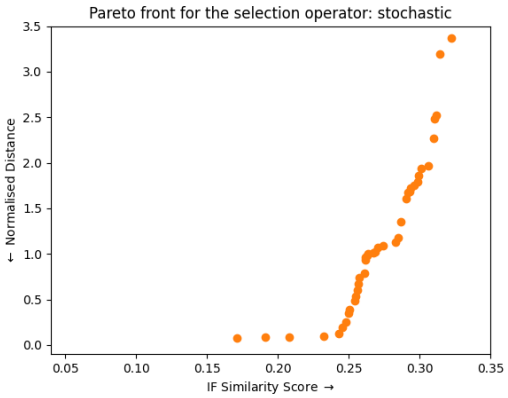


(e) One-Point Mutation

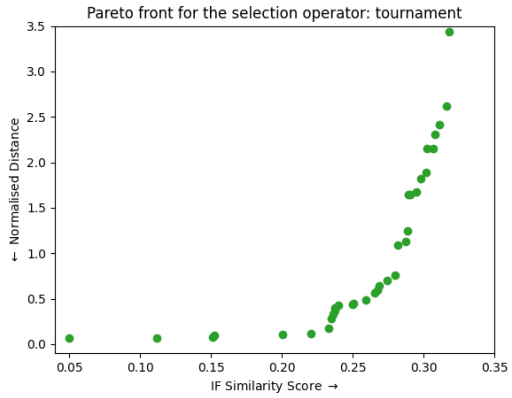
Figure B.5: Fitness values from individuals in the Pareto-fronts, divided per mutation operator.



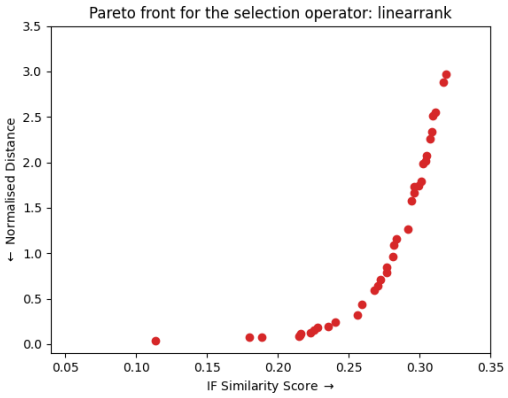
(a) Roulette Wheel Selection



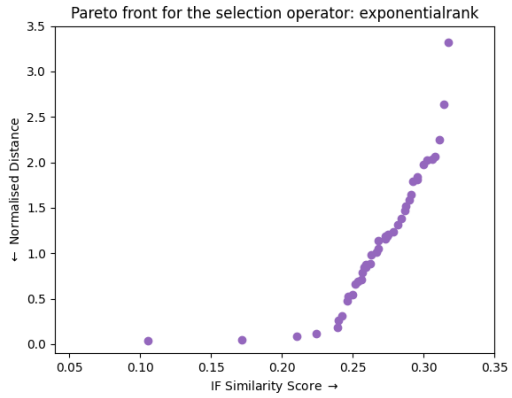
(b) Stochastic Universal Selection



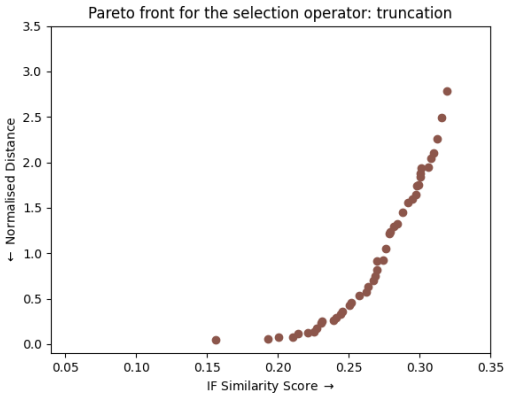
(c) Tournament Selection



(d) Linear Rank Selection



(e) Exponential Rank Selection



(f) Truncation Selection

Figure B.6: Fitness values from individuals in the Pareto-fronts, divided per selection operator.

C

Generation graphs

This appendix shows all the generation graphs for every individual operator. Every graph shows the highest fitness achieved, the lowest fitness achieved, but also the median, first and third quartile to indicate the spread of fitness scores from all runs with that operator.

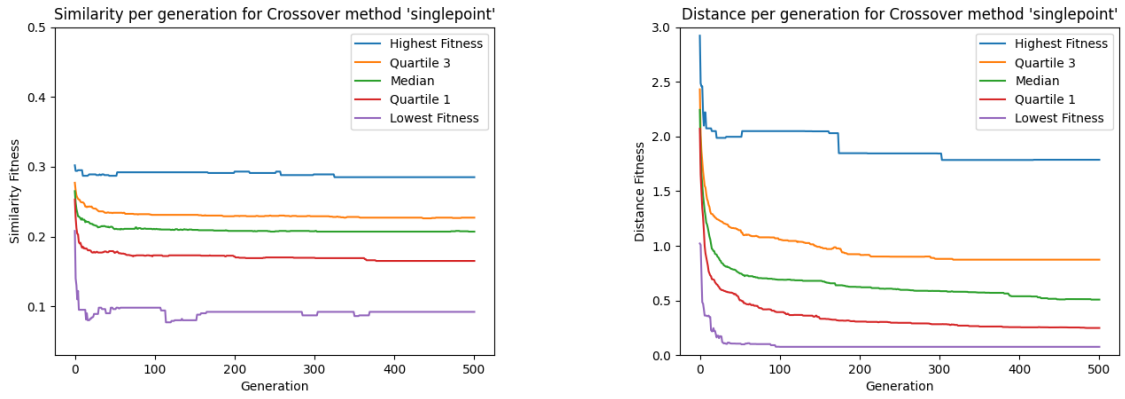


Figure C.1: Generation graph for both the similarity and distance of the Single Point crossover operator.

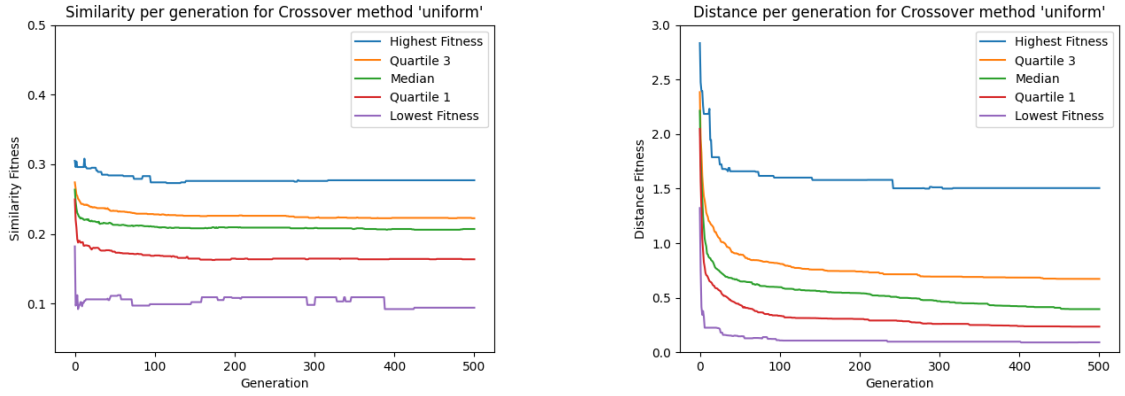


Figure C.2: Generation graph for both the similarity and distance of the Uniform crossover operator.

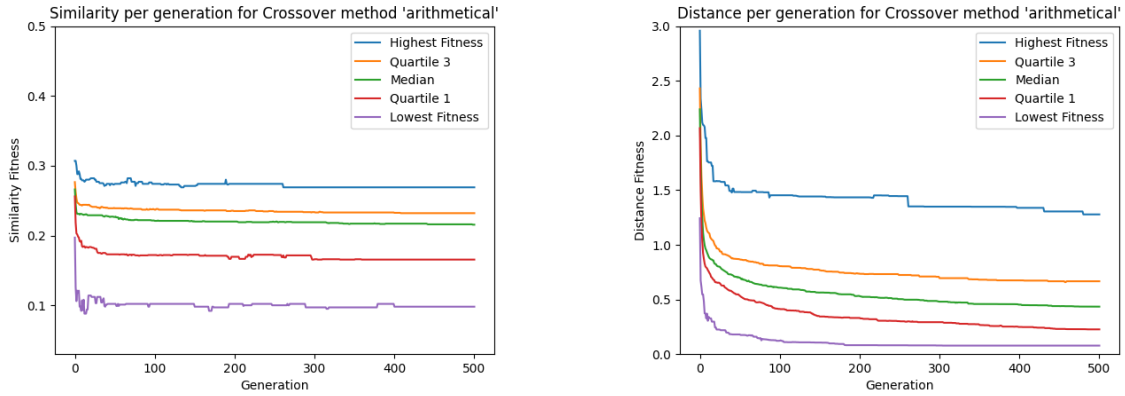


Figure C.3: Generation graph for both the similarity and distance of the Arithmetical crossover operator.

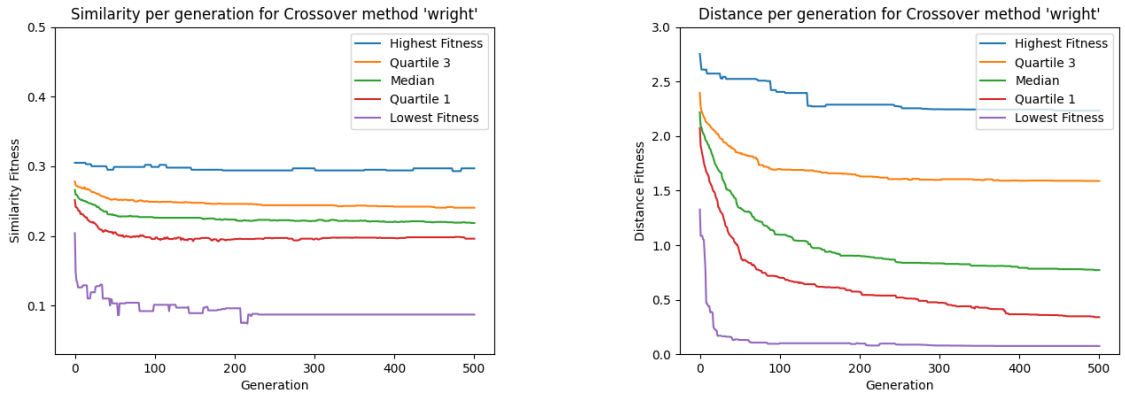


Figure C.4: Generation graph for both the similarity and distance of the Wright crossover operator.

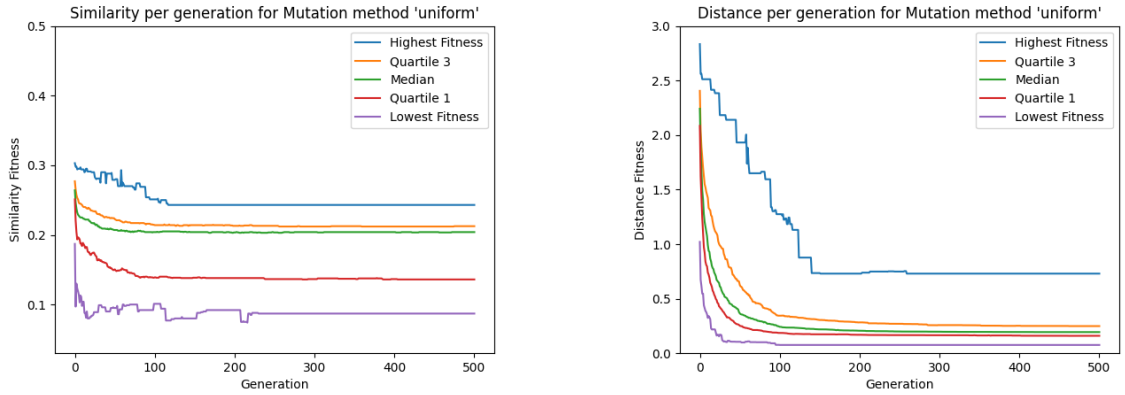


Figure C.5: Generation graph for both the similarity and distance of the Uniform mutation operator.

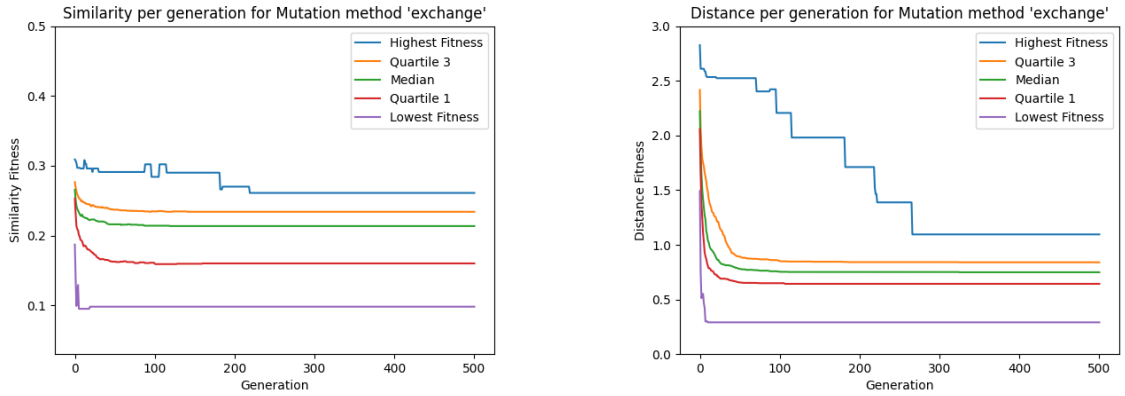


Figure C.6: Generation graph for both the similarity and distance of the Exchange mutation operator.

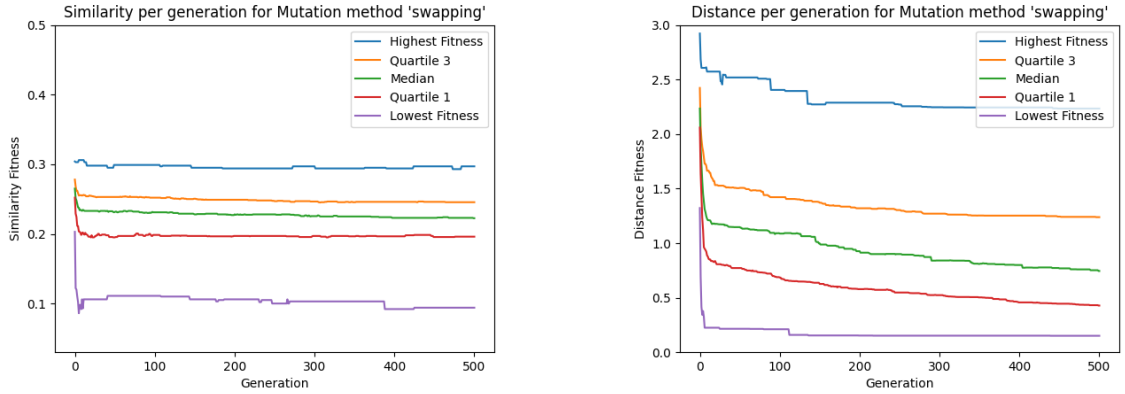


Figure C.7: Generation graph for both the similarity and distance of the Swapping mutation operator.

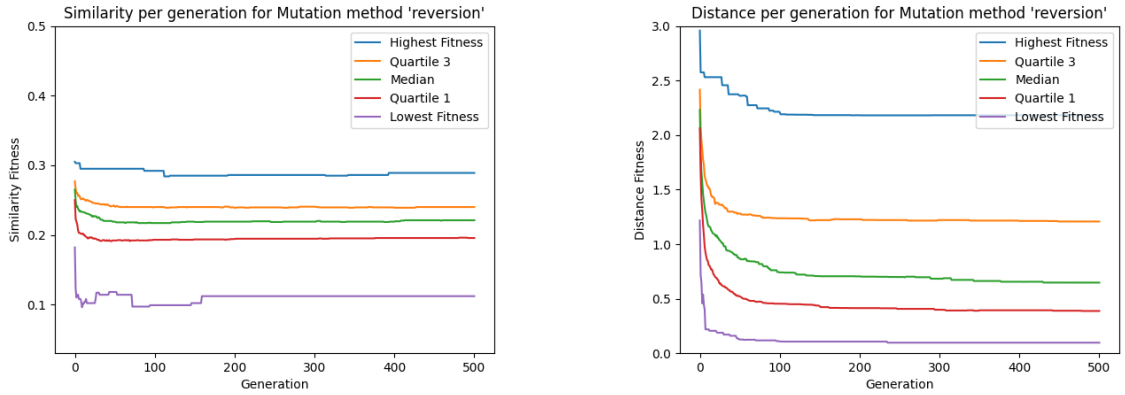


Figure C.8: Generation graph for both the similarity and distance of the Reversion mutation operator.

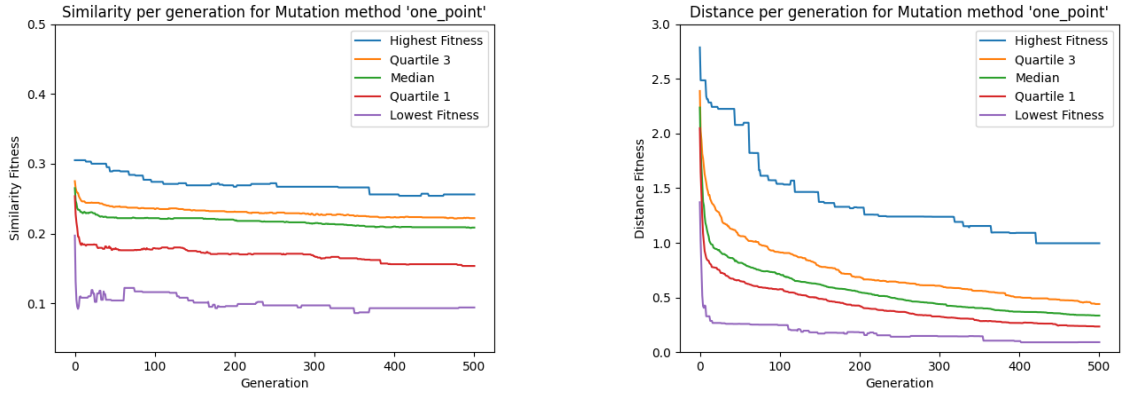


Figure C.9: Generation graph for both the similarity and distance of the One Point mutation operator.

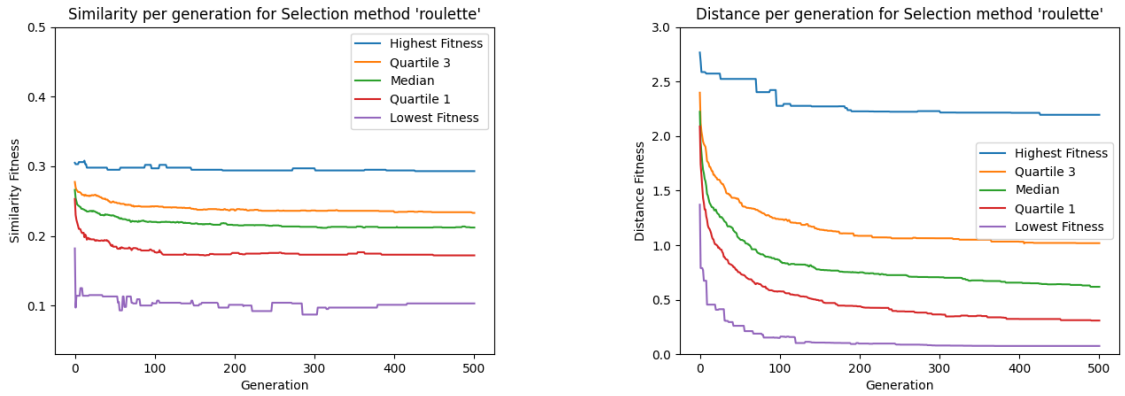


Figure C.10: Generation graph for both the similarity and distance of the Roulette Wheel selection operator.

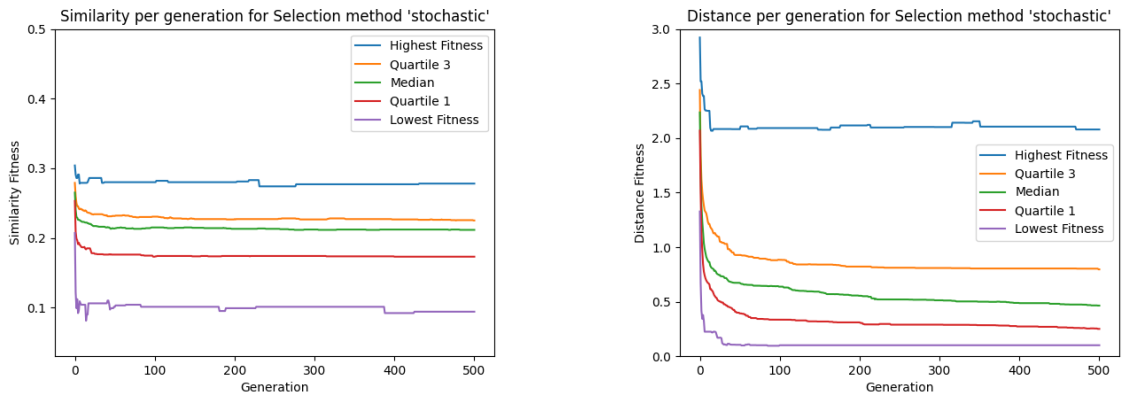


Figure C.11: Generation graph for both the similarity and distance of the Stochastic Universal selection operator.

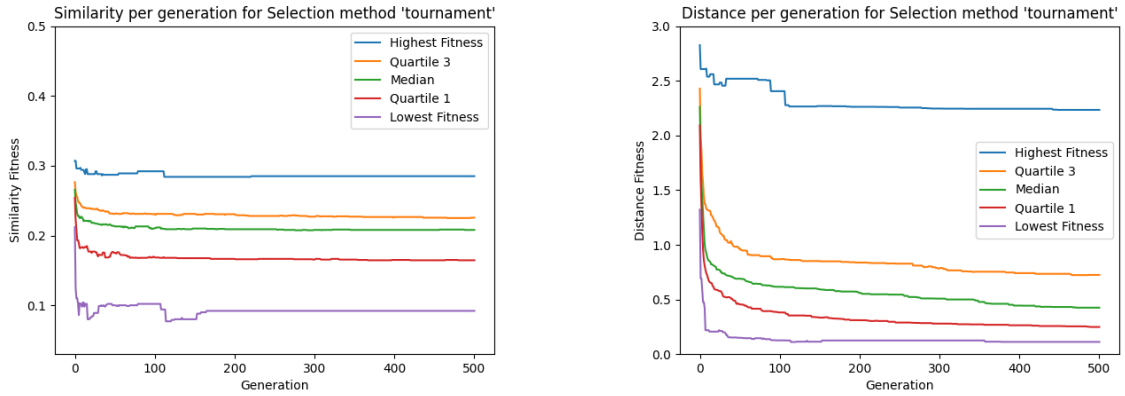


Figure C.12: Generation graph for both the similarity and distance of the Tournament selection operator.

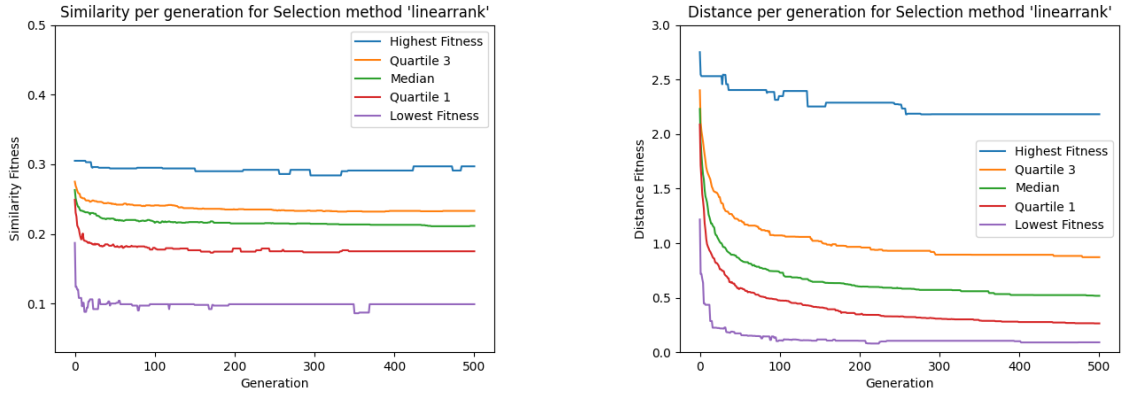


Figure C.13: Generation graph for both the similarity and distance of the Linear Rank selection operator.

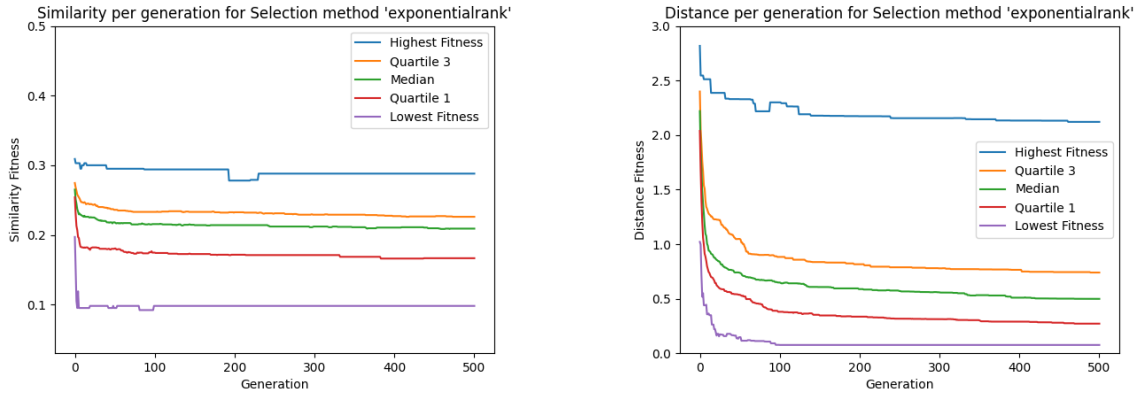


Figure C.14: Generation graph for both the similarity and distance of the Exponential Rank selection operator.

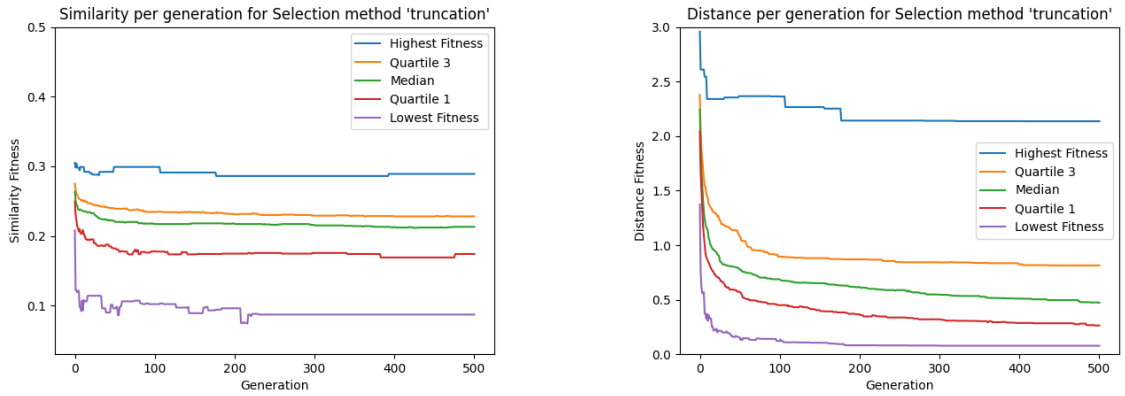


Figure C.15: Generation graph for both the similarity and distance of the Truncation selection operator.

D

Calculating score for optimal operators

This appendix shows Tables D.1, D.2 and D.3, containing the normalised data for the regular scatter-plots and Pareto-fronts for the crossover, mutation and selection operators. The adversarial distance has been inverted to represent a maximisation problem. All values per operator are summed and used to compute a final score that is used to determine the best overall operator. From these tables, we see that the best crossover operator is Single Point crossover, Uniform is the best mutation operator and Truncation performs the best for selection.

Crossover	**Single Point**	**Multi Point**	**Uniform**	**Arithmetical**	**Wright**
Similarity: Mean	0.599	0.598	0.598	0.601	0.606
Inverted Distance: Mean	0.857	0.867	0.884	0.886	0.779
Similarity: STD	0.022	0.022	0.020	0.021	0.022
Inverted Distance: STD	0.096	0.083	0.073	0.068	0.150
Crossover - Pareto					
Similarity: Mean	0.630	0.632	0.637	0.639	0.629
Inverted Distance: Mean	0.735	0.725	0.724	0.622	0.768
Similarity: STD	0.025	0.019	0.019	0.020	0.020
Inverted Distance: STD	0.269	0.228	0.200	0.305	0.199
Total	3.233	3.174	3.155	3.162	3.173

Table D.1: The summed statistics indicating that Single Point crossover performs best.

Mutation	Uniform	Exchange	Swapping	Reversion	One Point
Similarity: Mean	0.590	0.599	0.608	0.607	0.597
Inverted Distance: Mean	0.949	0.825	0.788	0.799	0.914
Similarity: STD	0.021	0.021	0.022	0.019	0.020
Inverted Distance: STD	0.023	0.033	0.129	0.126	0.038
Mutation - Pareto					
Similarity: Mean	0.631	0.634	0.632	0.636	0.639
Inverted Distance: Mean	0.738	0.673	0.713	0.698	0.673
Similarity: STD	0.022	0.022	0.027	0.019	0.017
Inverted Distance: STD	0.217	0.223	0.220	0.237	0.262
Total	3.191	3.030	3.139	3.141	3.160

Table D.2: The summed statistics indicating that Uniform mutation performs best.

Selection	Roulette	Stochastic	Tournament
Similarity: Mean	0.602	0.600	0.598
Inverted Distance: Mean	0.835	0.860	0.866
Similarity: STD	0.022	0.021	0.022
Inverted Distance: STD	0.110	0.103	0.104
Selection - Pareto			
Similarity: Mean	0.633	0.635	0.628
Inverted Distance: Mean	0.731	0.723	0.744
Similarity: STD	0.018	0.016	0.030
Inverted Distance: STD	0.218	0.197	0.232
Total	3.169	3.155	3.224

Selection	Linear Rank	Exp. Rank	Truncation
Similarity: Mean	0.601	0.599	0.601
Inverted Distance: Mean	0.846	0.863	0.858
Similarity: STD	0.022	0.021	0.021
Inverted Distance: STD	0.116	0.097	0.102
Selection - Pareto			
Similarity: Mean	0.634	0.634	0.633
Inverted Distance: Mean	0.688	0.692	0.745
Similarity: STD	0.023	0.019	0.019
Inverted Distance: STD	0.292	0.238	0.212
Total	3.222	3.163	3.191

Table D.3: The summed statistics indicating that Tournament selection performs best.

E

AGONI example run: feature values

We show all the generation graphs in Figure E.1 to E.6 that display the values of every feature during the generations for the example run. We show the value that AGONI's best individual has for that feature in blue, along with the value from that feature in the original network packet in blue. This generation graphs show the difference between the actual value and AGONI's value during the generations.

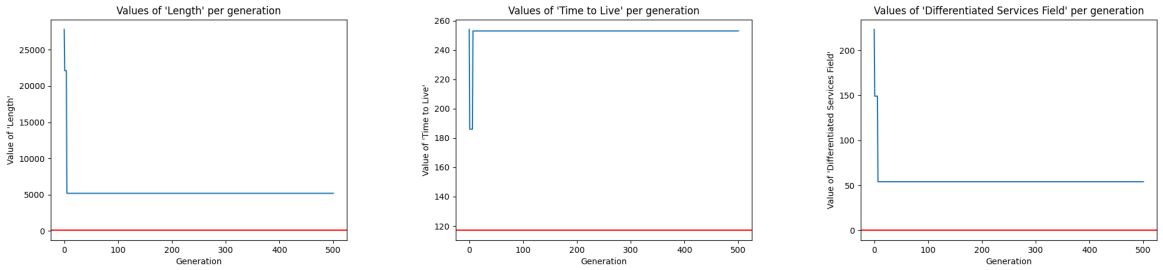


Figure E.1: Generation graph for the total length, time to live and differentiated services.

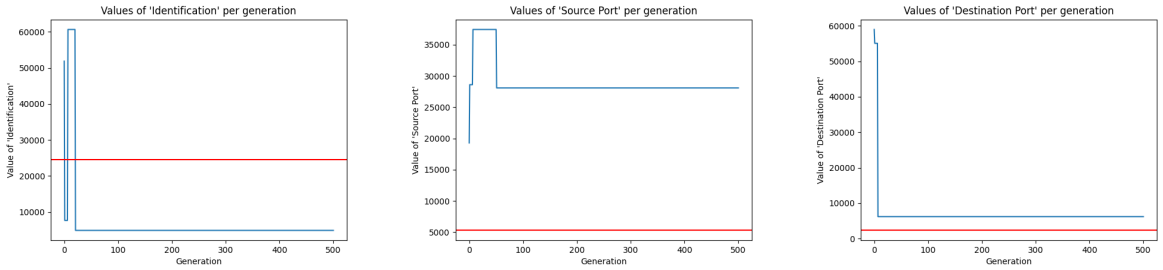


Figure E.2: Generation graph for the identification, source port and destination port.

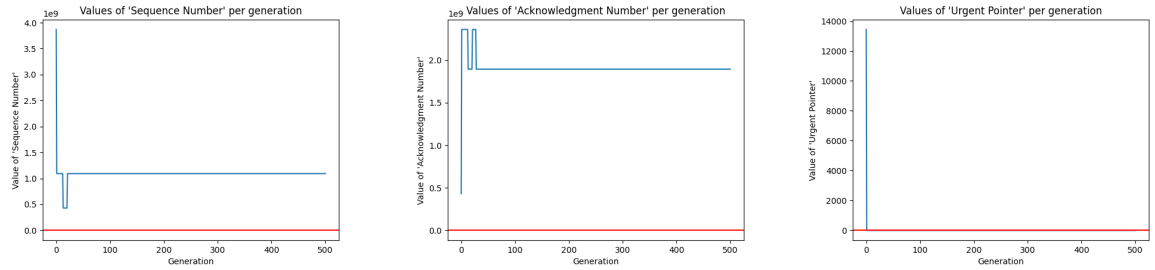


Figure E.3: Generation graph for the sequence number, acknowledgement number and urgent pointer.

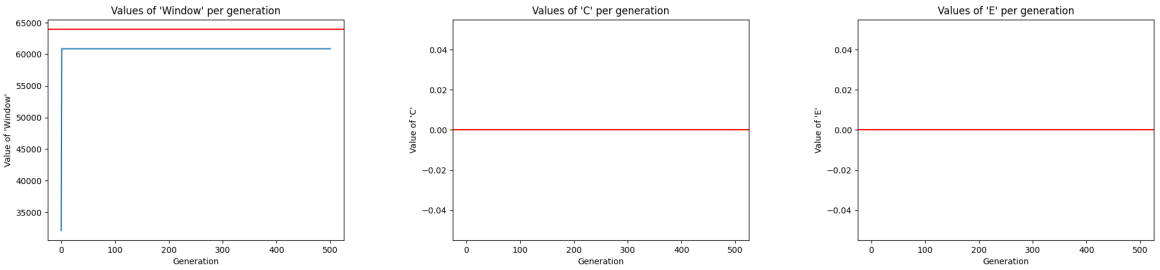


Figure E.4: Generation graph for the window, CWR flag and ECE flag.

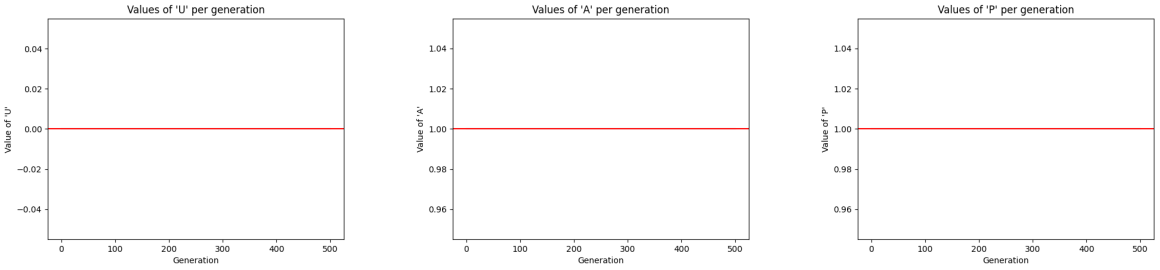


Figure E.5: Generation graph for the URG flag, ACK flag and PSH flag.

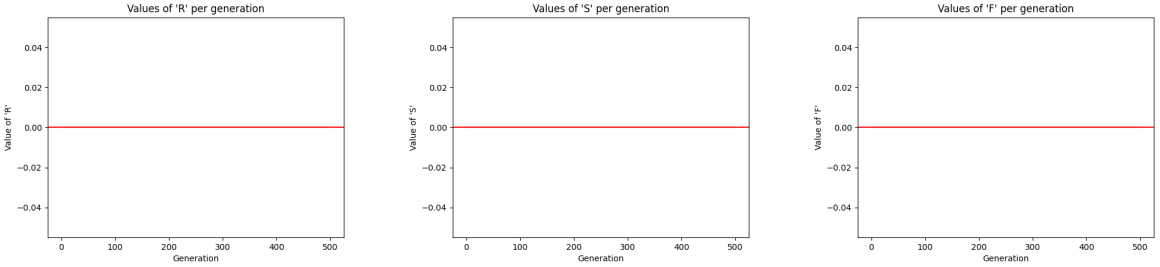


Figure E.6: Generation graph for the RST flag, SYN flag and FIN flag.