Application of Genetic Algorithms for Malware Obfuscation and Static Vulnerability Analysis in Android Environments

Aaron Henderson
Supervisor: Dr Felipe Maldonado

A dissertation submitted for the degree of:
Masters by Dissertation - Applied Mathematics

School of Mathematics, Statistics and Actuarial Science
University of Essex

Date of submission for examination:

January 15, 2026

Abstract

As digitisation grows, from social media to banking applications, malicious software (malware) threats have become increasingly problematic. A large proportion of malware is developed directly based on previous malware samples. This allows low-skilled developers to modify known malware and reap the benefits. This dissertation highlights how effective a standard programming paradigm, and a low-performance computer can be in evading modern antivirus systems, demonstrating that the security for mobile applications can be overcome without the need for high-performance computers.

This dissertation aims to provide a standardised way to make Android application code unrecognisable/hide its characteristics (obfuscate), highlighting security vulnerabilities without the need for professional knowledge on the subject. Obfuscation can be used for legitimate purposes, such as protecting intellectual property and hiding sensitive information or vulnerabilities in software code; conversely, it has the illegitimate purpose of making malware more evasive. In this dissertation, a program was developed using a genetic algorithm combined with several preexisting tools and then tested using malicious Android applications.

The contributions of this research are two-fold. Firstly, existing research was recreated using recent Android applications rather than pre-1995 DOS-era applications, making its insights relevant to current computer systems. Secondly, achieving this with standard computer resources (i.e., not a high-performance computer), as is typically used in related research, broadens the audience to which the results are applicable. Key findings from this research are that, with obfuscation methods built in the year 2020, a genetic algorithm can find sequences of obfuscation that can bypass antivirus systems in 2025. The program achieved this with only control and data flow manipulation-based obfuscation and produced good results with a population of five after twenty-one generations. This finding is significant because the methods used are not highly complex to implement, and in terms of computer technology, five years is a significant amount of time.

Contents

CHAPTER 1

Introduction

1.1 Background

With the ubiquitous presence of digitisation, from social interaction to daily administrative tasks, the problematic effects of malicious software (malware) have increased. There are consistent reports of the growing threat from malware in academic literature, policies, and the media. For instance, the UK government is continually updating policies in this regard, most recently [21]. An up-to-date report from a cybersecurity company [9] shows the exponentially growing threat from Android malware. Consequently, there is a need to protect against such ever-growing threats in the cat-and-mouse game of digital security ([65], [36]).

Currently, a large proportion of malware is developed directly based on previous malware samples [43]. This allows low-skilled developers to modify known malware and reap the benefits. This dissertation highlights how effective a relatively standard programming paradigm and a low-performance computer can be in evading modern commercial antivirus systems, demonstrating that the security for mobile applications can be overcome without the need of a high-performance computer.

The contributions of this research are two-fold. Firstly, existing research was recreated using recent Android applications rather than pre-1995 DOS-era applications, making its insights relevant to current computer systems. Secondly, achieving this without the need for high-performance computers, as is typically used in related research, broadens the audience to which the results are applicable. This dissertation lies at the intersection of three domains: malware analysis, the Android operating system, and genetic algorithms. The dissertation aims to provide a standardised way to make Android application code unrecognisable/hide its characteristics (obfuscate), highlighting security vulnerabilities, without the need for professional knowledge on the subject to get the best results. Obfuscation can be used for legitimate purposes, such as protecting intellectual property, hiding sensitive information, and hiding vulnerabilities in the code; conversely, it has the illegitimate purpose of making malware more evasive. Obfuscation and detection of malware are continually evolving, techniques used on a given application that successfully bypassed an antivirus system that worked months ago may no longer be as effective today. For this dissertation the developed program will rely on a genetic algorithm combined with several preexisting tools, and will be tested using known malicious Android application samples.

Android was selected as a suitable domain for this dissertation because it provides a wide range of freely available tools for malware analysis. For example, the Android emulator which makes simulating an Android device possible, on a standard laptop. Additionally, a lot of the malware for Android is on the application layer, which means that less is needed to be known about the underlying framework of the operating system to understand how malware samples work. The aforementioned conveniences of using Android could also be found when looking at malware on embedded systems, like Internet of Things (IoT) devices. However, a quick online search shows considerable literature on Android and many developer tools available, most likely due to Android being a global, open-source platform. This abundance of resources makes Android a good choice compared to embedded systems, aligning this domain with the constraints of the dissertation (see Section 1.4).

1.2 Research Questions

This dissertation aims to answer the following research questions:

- Genetic algorithms are known to solve a range of combinatorial optimisation problems. How effective are they, in different configurations, at finding combinations of malware modifications that can surpass modern antivirus systems?
- A wide range of modifications can evade antivirus systems, from encryption to randomisation. This dissertation addresses the question of how complex do the modifications need to be to achieve this goal?
- When looking for ideal combinations, what are the ideal parameter settings for the genetic algorithm, given the hardware it is running on?

1.3 Research Objectives

The research objectives below correspond to the chapters in this dissertation. The first objective is covered by Chapter 2. The second and third objectives are addressed in Chapter 3, with the final objective covered by Chapter 4.

1. Design and implement a genetic algorithm based program for obfuscation (Chapter 2).
2. Test different implementation choices and their effectiveness at bypassing modern commercial antivirus systems (Chapter 3).
3. Understand the computational performance of different algorithm configurations (Chapter 3).
4. Highlight key issues in implementing such a system and further developments that could be made (Chapter 4).

1.4 Scope and Constraint

The three core constraints of this dissertation are:

1. The project needed to be completed within the one-year time limit, imposed by the length of a master's by dissertation.
2. There is no additional funding to be spent on developing this project (e.g., AWS could not be used).
3. The computational power available was limited to that of a typical laptop, see Section 2.2.1 for details.

As the project developed, the scope was added to focus specifically on applications on the Android operating system and use modern commercial antivirus systems to investigate the effectiveness of obfuscation.

1.5 Literature Review

This literature review aims to establish an understanding of theoretical ideas and empirical results, which would then direct an applied research project within the domain of machine learning and malicious software (malware). The literature review will start by examining the general fields of machine learning and malware to then concentrate on genetic algorithms applying concealment (obfuscation) to Android applications. Android was selected as a platform as it is a widely used operating system, and the literature revealed several factors that made it a practical system to work with, given the three constraints from Section 1.4.

In the following section, the literature review begins with a textbook [43] and a survey paper [51] to form a starting point, and sets the dissertation within the broader context of machine learning and malware on Android.

1.5.1 Preliminary Sources

The textbook [43] framed this research project within the broad domain of malware in the Android environment. This textbook helped to understand the security features of the Android operating system and highlighted several interesting ideas.

- That Android defence is performed in layers, and each layer can detect some, but not all malware.
- Key security issues known on the Android platform, namely privilege escalation malware, data theft or manipulation, accessing executable memory, accessing kernel-level flags, and adding apps from outside Google Play, known as side-loading.
- The accessibility application programming interface (API) affects security on the Android platform because it has permission to cross the sandboxes setup on each app, such as a screen reader assistant, which is classified as a security weakness.
- That malware developers for Android can use different languages to make automated analysis more complex.
- Reducing the surface area of attack to help reduce attack possibilities is done by deleting unused code, making it inaccessible via permission, API structure, or sandboxing apps.

The authors covered the broader implications of security within the Android ecosystem. They explained how attacks are performed by large-scale, sophisticated attackers and how chains of exploits can bypass Android and lead to a compromised system. These complex chains of events can be executed over a period of months for additional sophistication. Google has systems to scan applications on Google Play [41], as well as system images produced by phone manufacturers [40], to detect security issues. Due to the large-scale deployment of the Android operating system, there is a wide range of attack options for malware developers. From a software perspective, an abstraction layer exists between the core operating system and phone manufacturers' add-ons in the Android system, which helps speed up security updates. [43] details how malware is defined on the Android system, for example, how rooting is performed to escalate malware privileges. The authors report that using public software libraries to spread malware is the most successful vector for infection, in terms of the number of affected devices. The general intention of malware developers on Android is to keep the malware operations within a grey area to avoid detection. For example, taking personal information and selling it to facilitate accessing private bank accounts is a crime, while taking personal information and selling the data for targeted advertising is not necessarily a crime. The fact that there is crossover between legitimate data brokers and malware developers makes the detection of illegitimate actors harder. [43] provided a step-by-step guide to understand issues concerning malware within the Android system. In terms of machine learning, the book's scope was limited, as it only focused on detecting malware and provided a few classification examples. Due to the wide range of academic literature on malware and machine learning a survey paper was employed to explore this topic.

The paper selected [51] focused on both machine and non-machine learning methods to distinguish between benign software and malware. Non-machine learning techniques used included statistical analysis, game theory, and entropy (e.g., using game theory in the prediction of advanced persistent threats). Machine learning techniques include clustering, decision trees, neural networks and reinforcement learning (e.g., neural networks being used to understand network traffic). [51] also introduces new topics like oligomorphic programs, programs that can change themselves during execution, and using novel ideas for malware classification, like heat emissions and converting source code to an image or sound. The paper also looks at well-known technical approaches like API calls.

Consistent with reports from outlets across the internet, this paper states that in recent years, malware has sharply risen for Android, IoT and Windows computers. The authors state that the most common attack on Android devices is infecting legitimate applications, intending to collect the users' personal information, which is confirmed in [43]. The most common infection for internet-connected, IoT devices is harnessing the device in a botnet, for use in a denial of service (DoS) attack. The authors state that ransomware is regarded as the most laborious malware to deal with once an infection occurs.

[51] gives a broad understanding of possible uses of different machine learning techniques in the context of malware. This topic was discussed from the perspective of generalised ideas and did not detail specific projects. This paper focused on detection methods, highlighting the need for a suitable source of malware-related information to perform research. There are several available databases online; some have been suggested in this paper (e.g,. DREBIN [10] and EMBER-2018 [48]). Each database will influence the type of research that can be done. This is because the type of information recorded in the database influences what type of machine learning can be used effectively. In the next section, we discuss a series of articles, selected to give a practical understanding of projects developed using machine learning in the malware domain.

1.5.2 General Machine Learning and Malware Sources

In recent years reinforcement learning has been investigated as a suitable method for malware detection (e.g., [34] and [71]). [34] looked at how Q-learning can be used for static feature detection of potentially malicious, portable executable (PE) files. PE files are the instructions used when running an EXE file. Q-learning was used to tune the parameters of several of the feature detectors. Namely, Support Vector Machine (SVM), Random Forest, K-Nearest Neighbours (KNN), Decision Tree and Naive Bayes. The system extracted characteristics from the PE files, in some cases using the most common n-grams as a comparison. The core idea in this prototype was for Q-learning to identify the most successful characteristics to be used by the feature selection algorithm.

This prototype used a Tesla V100 GPU in an Nvidia DGX server, a high-performance computer, with the code available on GitHub [35]. The results showed that having a limited number of well-selected characteristics provided highly accurate solutions.

Some classification detectors performed better than others, and using a limited number of features helps reduce resource requirements and training times.

The authors assume that if one feature performs well in identifying a sample as malware, then that feature is likely to be found in other malware samples. This is a valid statement, as malware developers often copy known strategies and build on top of pre-existing software [43]. On the other hand, this could mean that the developed prototype will be limited to detecting specific malware families [24]. Malware families are groups that identify how specific malware samples' behaviour evolves, members of a family also have similar features. The author used pyCUDA in preference to CUDA C, which emphasises the flexibility of prototyping over computation speed. The improvement obtained from applying Q-learning seems linked directly to the underlying classification model. SVM and Naive Bayes had notable improvement compared to the others, presumably because these methods are better at this type of problem, regardless of parameter tuning. The idea of taking random action based on a set of rules is interesting for an environment without well-defined actions. Which are a requirement when using reinforcement learning. While [34] did focus on the Windows operating system, which is outside the scope of this dissertation. Insight into the application of reinforcement learning was valuable, although their prototype relied on a large amount of computation during development.

Also using reinforcement learning, but this time, with an abstract representation of malware behaviour, [71] looks at making a Breach and Simulation (BAS) system to create an automated program to identify security issues. The idea is to enable computer administrators to make the most effective decisions to secure their systems. The authors compare multi-agent reinforcement learning (MARL) and a grammatical evolutionary algorithm (GEA) [2] to solve a game theory structure problem. The GEA's distribution and the MARL convergence rate were used to compare the two algorithms. They grouped the MARL episodes to correspond to the generations of the GEA. This made the comparison easier.

The prototype looks for a mixed strategy Nash equilibrium. A concept from game theory, whereby each player in the game does not benefit from changing their own strategy, resulting in the game stagnating. When looking for equilibrium, MARL and GEA were used in the attack and defence roles, and during testing, GEA was better at attacking. They believe this could be due to excessive exploration of MARL agents. MARL also had issues repeatedly trying the same actions. Both are well-known problems when using reinforcement learning [66], known as the exploration-exploitation dilemma, which makes MARL weak at defending. They believe low mutation in the GEA preserved reasonable solutions, while high crossover increased stability between different generations. The authors claim the unique element of this paper is the use of mixed strategies rather than a pure equilibrium, which they state previous papers have done. They used standardised scores from a database of known attack and defence patterns to give context to the system, namely the common vulnerability scoring system (CVSS) [64].

In [71], MARL was insufficient to solve the problem, but the authors clarified that there were many more possibilities to explore to make MARL more suitable, such as applying a rule to stop the agent from repeating itself. Malware was modelled using predefined attack and defence patterns from CVSS. The authors did not infer how this could be generalised to work when the answers to specific events are not already recorded. An interesting idea gained from this paper is to have the parameters of each algorithm inform each other so that they can train together, in a Generative Adversarial Network (GAN) type setup [39]. This paper also highlighted the demanding training requirements for MARL when dealing with complex environments, which can also be seen in [34]. Both [34] and [71] suggest that reinforcement learning requires extensive and computationally expensive training which goes against a constraint of this dissertation (see Section 1.4).

In [63], the authors looked at techniques to obfuscate Android malware and hide it from detection, using a non-machine learning algorithm. The authors developed a prototype called DroidChameleon, which can apply various obfuscation techniques to Android applications. For instance static obfuscation such as repacking, and dynamic ones, such as runtime encryption.

DroidChameleon uses APKtool [8]. They used *DroidDream* and *FakePlayer* malware samples to test if the application's original functionality was intact after DroidChameleon's transformation. The results of applying DroidChameleon were confirmed by testing to see if commercial antivirus programs could detect the samples. The authors used malware that predated 2011 to ensure that the malware signature was well-known to existing antivirus programs. The transformations were applied with increasing complexity, and the test would be stopped if the antivirus failed. The authors claim that only combining two transformations was typically needed to bypass the antivirus systems. They suggest that their findings show that very few antivirus systems for Android use static analysis. This is probably due to the privileges granted to third-party apps on Android, restricting access to the source code of other applications.

The authors state that the bytecode of a transformed sample retains some of its core semantic properties after compiling, and as such, they infer that compiling automatically removes some of the source code transformations that were added. The ideas suggested in this paper align with this dissertation's research questions, however, the authors relied on a considerable manual manipulation of code in testing and transformation, whereas this dissertation aims to create a more systemic approach. Nevertheless, this paper introduced two useful tools. Firstly, the web service av-test.org [69], that provides benchmarking results for common commercial antivirus software on different operating systems. Secondly, APKtool [8], a tool that allows source code level access to precompiled applications, which is useful when using source code and malware samples.

Following the previous paper, a more in-depth look at non-machine learning was explored. In [83] the authors look at generating malware samples by implementing dynamic binding and reflection, which can then be used for testing antivirus software. They called this system MYSTIQUE-S. MYSTIQUE-S uses a Dynamic Software Production Line (DSPL), whereby the program is structured as a set of modular components which can then be combined, on the fly, using linear programming. MYSTIQUE-S first gets system information, then selects a modular feature based on three characteristics:

aggressiveness, latency and detectability. The Behavioural Description Language (BDL) provides a high-level description of behaviour and functionality. BDL was used as an intermediary representation of the programs. The authors divide Android attack types into four categories: financial, privacy leakage, phishing, and extortion. The authors state that often malware shares similar code, and it is common for malware to be clones of each other. This is also stated in the book [43]. They used sixteen real devices from different brands during testing and ran the experiments on an Intel Xeon CPU with 64GB memory, which is a high-performance computer.

The authors claim that the application of additional obfuscation was unnecessary due to the low detectability of the generated samples. This paper and its previous counterpart [54] proposed using two methods, Software Product Line Engineering (SPLE) and DSPL, similar to an evolutionary algorithm, in generating source code samples. The authors claim that software production lines are more computationally efficient than evolutionary algorithms, as they allow targeted control of outcomes, compared to evolutionary algorithms that use much computation exploring suboptimal options. The deployment of software production lines in this paper seems to require substantial computer resources, considering how much the samples were transformed.

Performance is a repeating theme in the literature ([83], [34] and [71]). Along this theme, [36] looks at deploying a performance sensitive prototype on Android, using deep-learning methods for on-device malware protection; they call this system MobiTive. On Android devices, some security features are done server-side [43]. If people download apps from third-party locations, server-side checks can be bypassed. Another potential issue with server-side security is that the system can be manipulated if the network is compromised. The authors state that the volume of data being collected on mobile devices is increasing due to widespread use, which in turn produces a growing need for better mobile device protection. The paper is developed on top of the author's previous work and the work from [17]. To reduce computational complexity, they looked directly at the API calls and the manifest files from source code, without decompiling. The *Classes.dex* file contains an API table which matches the executable symbols to API strings. Only API calls suspected of being malware were checked, which was assessed by manual inspection. The experiment took place on six real-time physical devices. The prototype model was trained on a server, after which the trained model can be run on the mobile device. The prototype was built on a high-performance computer, 192GB RAM and a GeForce 2080Ti GPU with Linux OS.

[36] used an Inter-procedural Control Flow Graph (ICFG) and Call Graph (CG) to map the API calls, and evaluated the performance of using different features in the model. The authors trained seven different models to see which would be best. The Recurrent Neural Network (RNN) worked better than the rest. They compared their results to three other learning-based systems that run on Android devices and found that their results were faster and more accurate. This paper showed that combining different features improved detection rates and using abstract data types, ICFG and CG to represent the malware helped to make model training manageable, but there was a significant requirement for computer power. While some scaling down of this project could be done, previous experience has shown that any non-trivial training of a network-based model is time and resource-consuming. Besides using the TensorFlow Lite framework, [36] did not highlight many factors when considering performance. It also did not cover aspects of data poisoning, a well-known issue when training networks like the authors have proposed. Data poisoning is the process of feeding data to a model with the intention to affect its training, which has been discussed in [43].

For this dissertation finding an appropriate benchmark was challenging. There are two main reasons for this. Firstly, the computers being used were often much larger than what is available for this dissertation; this renders the insights yielded from the papers slightly irrelevant. Secondly, it was found that how a framework is deployed affects its performance; for example, if it is distributed or on a GPU/CPU, which can make results irrelevant as well. Some frameworks are optimised for specific deployments, and often papers would not fully cover this aspect. This is one area where the academic literature does not cover specific information on the topic. While this is a research gap, the practical application of such research, which benchmarks these specific configurations, would not be commonly used and is probably not of great academic value. Nonetheless, [12] highlights some of the previously mentioned limitations for benchmarking papers that can be applied to this dissertation. Interestingly [12] shows that, depending on how often the fitness function needs to be evaluated during a run, effects which framework is faster. For example, the authors compared several frameworks, such as GPlearn and karooGP. In the lower half of evaluations, GPlearn is better than karooGP, but in the top half karooGP is faster than GPlearn.

In the article [56] the authors used a computer system comparable to the one available for this dissertation. Using PE files, the authors used Genetic Programming (GP) to evolve malware samples. They called this system MAGE. MAGE aims to make as varied malware as possible; it does not focus on making new malware variants. The generated samples can then be used to train antivirus systems. MAGE modifies two malware samples, one a COM file, the other an EXE file; the samples were called *Timid* and *Intruder*.

Intra-population Jaccard similarity and Euclidean distance were used as a fitness function in [56]. Jaccard similarity was implemented using a vector, so that the intra-population fitness could be calculated, making the fitness value more robust. As expected, the fitness function had a major impact on run time. When the fitness value was created by scanning generated samples with an online antivirus system, a single run of 500 generations took 72 hours. With intra-population fitness, the time was reduced to thirty minutes. The authors claim that comparing individuals in the population to each other consistently drove new changes in the population and led to better evolution than an antivirus detector by itself. This is because an antivirus detector can be tricked with some changes, and it can take some time before the system gets updated with the potential to recognise these changes. The authors report that against sixty antivirus systems supplied by VirusTotal, evolved samples evaded detection with a 90% success rate.

To be operated on by the GP, samples were disassembled, if the resulting individual could not be reassembled, then it was regarded as invalid and excluded from the experiment. The operators were designed to keep the samples intact, and the authors assumed the generated code is valid. Mutations are the modifications made to the code. The mutations are control flow, data transformation and code layout. A single-point crossover, in a fixed position, was used, as care needed to be taken so that the crossover did not affect the execution of the program and invalidate the sample.

The total number of samples in the GP was very low compared to evolutionary algorithms in general, with a total population of twenty. Mutation is higher than typical at 80%. The total generations were 500, within the range expected when using GP. The experiments were performed on a standard machine using an Intel i5, 8GB of RAM and Linux. The samples were run on a virtual machine, and consisted of Assembly programs from the DOS era that would not run on a modern computer. The use of computer programs of this age makes this project much more computationally effective, as old programs from DOS are much smaller and simpler than programs today.

Shannon entropy and Mann-Whitney tests were also used in [56] to assess how MAGE changed the population after running, and could be used to judge the effectiveness of the fitness functions. Using a multi-objective fitness function that uses multiple values to generate an individual fitness value is a useful idea to have when training an evolutionary algorithm to discern something as complex as a malware sample. This paper helped evaluate several methods that can be used for a fitness function. Considering the quality of the results obtained with a working prototype that runs on a typical-sized computer, this paper was regarded as a reasonable proposition on which to base this dissertation.

There were also two heavily referenced programs in the literature, ADAM [70], AAMO [58]. These prototypes either did not work or had limited functionality, and as such, they were not reviewed further. The use of an evolutionary algorithm was mentioned in several papers as a possible solution method. Previous experience has shown that evolutionary algorithms can obtain good results without large amounts of computation but require a relatively large amount of memory storage. Evolutionary algorithms operate on source code samples, which removes the reliance on premade databases, and gives more development flexibility to the dissertation.

1.5.3 Adversarial Systems and Evolutionary Algorithms

Adversarial Systems have gained popularity in recent years after the development of the Generative Adversarial Network (GAN) in the seminal paper [39]. In [71] the authors inferred the use of a generator/discriminator architecture in the malware domain. The core idea to be explored in this section is developing two converse agents that help train each other in a unified goal, in the context of malware generation and detection.

The authors of [15] use Genetic Programming (GP) to find adversarial malware samples by injecting code into PE files. The prototype they developed was called AIMED. Generated samples are evaluated by antivirus systems, Sophos, ESET and Kaspersky, and a machine learning classifier, a gradient boosted decision tree. Kaspersky was believed to be the best at detecting the generated malware. The authors highlight the importance of static analysis as it can be performed before the sample executes and is able to act maliciously.

In each generation, the generated samples were tested for validity in terms of their ability to execute. This could be a resource consuming process, compared to having non-invalidating operators. One novel idea from [15] was to create part of the fitness value from how many generations a malware sample has been operated on. The two fittest population members are selected to create the offspring in the next generation. An individual's sandboxing and antivirus evaluation took about 15-30 seconds. The authors state that a 10% mutation rate was required to be better than a completely random program.

Benchmarking was performed by testing against random code injection and a reinforcement learning agent. The random system worked comparably well when applying small changes to the samples, but performed poorly when injecting lots of code. Similarly reinforcement learning produced a large amount of corrupted files, which waste available computation and inevitably slows development. The authors tested cross-validation by training AIMED on one specific antivirus system, then seeing if the generated sample would bypass another. The generated samples achieved up to 82% in this regard. From a database of 6880 files, samples were randomly selected to be transformed by AIMED. AIMED had an invalidation rate of about 24%, but this was after classifying 76% of the available files as unmodifiable, which means this system is not very good in a general application.

Following a similar idea [65] looked at transforming malware by applying obfuscation, which can be used to predict future malware modifications. Therefore, it could highlight new malware before it becomes an issue. The project uses GP for generation and detection, and is a fully automated system. Each individual is a mobile application; the initial population is all valid malware examples. The training data was taken from MalGenome [84], and the benign applications were downloaded directly from the Google Play Store. Testing was performed for each application on six Android emulators and each emulator was running a different antivirus program. The GP algorithm terminates once a set number of generations have eventuated. The applications were decompiled and converted into call graphs (CG), which the GP operates on. The CG can then be converted back into an application using the author's custom software, which is unavailable for download. The authors use several tools to confirm that each tree was valid. Using a CG limits what programs can be run, excluding those with recursive functions.

Crossover was known to produce broken trees as a node's parameters and return values always need to align with its neighbours; rather than use a check to preserve the tree's integrity, the crossover rate was set low at 10%. The mutation rate was 80%, and the population size was 15. Mutation was the application of six different obfuscation techniques: rename local identifier, junk code insertion, data encryption, two-fold code reordering, three-fold code reordering, and register realignment. The fitness score was between 0 and 1, 0 being the best; if the malware did not run, it would be set to 1. The malware samples run for one minute on each emulator, and an antivirus system would attempt to detect malware within this time frame; the result from all emulators was added together to create the fitness value of the individual. The one-minute limit was decided from experiential data, and by definition, it would exclude complex malware that takes more time to analyse. This, of course, was a trade-off to reduce the algorithm's run time to a manageable scale.

Additionally, [65] developed a GP based antivirus detector, as the second part of the adversarial system. This system classified samples using known malware features and was trained to select the most common features needed to identify a given sample. The fitness value was calculated using the true positive rate * false positive rate. To test the detection rate of co-evolved samples, a comparison was made between evolved, co-evolved and the application of the professional obfuscation tool, Kelix Klassmaster. The authors claim that the result of this comparison depended on which antivirus system was used in the test.

In some cases, identical results were obtained, as seen in Table IV of [65], implying that neither system could effectively bypass the features used by the antivirus system. Testing each evolved sample required a lot of computation and relied on static analysis. This paper showed rigorous testing and even tested the system during different antivirus updates, which has not been seen in prior research.

1.5.4 Additional Tooling

Additional software tools were sought to make the project manageable within the given time constraints (see Section 1.4). The following paper was found to be the most suitable for this purpose [7]. In this paper the authors developed a tool to protect against code analysis called Obfuscapk. It aimed to provide a free, off-the-shelf solution to apply a range of obfuscation techniques, to be used in further academic research. It has a range of techniques that can be applied via a terminal interface. Several trivial techniques can be applied, as well as renaming, encryption, code injection and reflection. Obfuscapk was tested on 1000 Android applications; the tests were performed using *Monkey*, a tool from the Android development platform. The tests were used to confirm if Obfuscapk leaves an application functionally intact. In 83% of the applications tested, functionality was unaffected by Obfuscapk. The authors state that applying reflection caused the most failures during testing. VirusTotal [74] was used to check how well Obfuscapk applied obfuscation. A single case study was performed manually to perform this test. They used a Dell XPS with 16 GB of RAM and an OnePlus Android phone with 8 GB of RAM for testing.

Obfuscapk was developed several years ago, and as such, the obfuscation techniques it implements are not as effective as they were at the time of publication (see Section 2.2.3). This is an expected outcome as antivirus systems are constantly updated to better combat malware development techniques, including obfuscation. While testing on 1000 applications is substantial. Due to the wide range of Obfuscapk's functionality and the evolution of Android application development over time. Using a single case study leaves many assumptions about how well Obfuscapk will perform in general regarding obfuscation effectiveness. However, the authors did make it clear that the testing of 1000 applications was for stability of the application, not effectiveness at obfuscation.

1.5.5 Summary

The sections in this chapter represent categories used to guide the review process. Additionally, the papers reviewed revealed several common themes that further influenced the dissertation. Abstract data types like CG or feature databases, as seen in [36] and [71], were commonly referred to when using machine learning, but not with evolutionary algorithms specifically. Fitness functions were often cited as being simpler, metric-based functions. The authors state that this approach was more effective due to the efficiency of running a metric-based function repeatedly. In papers directly modifying malware samples, there were two groups these papers could be categorised into: creating new malware or changing existing malware, for example [15] and [56] respectively. A key note about these two definitions is that they are not formally defined and the difference between the two is unclear. Without a formal definition, these descriptions can become arbitrary.

Several papers reviewed noted the importance of static analysis when detecting malware. Interestingly, *Windows Defender* seems to rely mainly on dynamic analysis.

This was determined during the development of this dissertation, as very few malicious files were deleted before they were interacted with. It was far more typical for malicious files to be removed by Windows Defender once interaction with the file had begun. This observation is backed up by the Windows Defender documentation [33]. According to *av-test.org* [69], Windows Defender has consistently been a top-performing antivirus system. As such, a critical and large scale commercial antivirus system relies heavily on dynamic analysis; the importance attributed to static analysis is of questionable validity. From a theoretical standpoint, detecting malware before it starts running its malicious code would be best. Static analysis is also typically more straightforward to implement because it does not require running the malware to perform analysis. Nevertheless, it would seem, from the behaviour of Windows Defender, that in a real-world setting, static analysis might not be the best practice, which contradicts the commonly held belief in the literature.

The testing performed in different papers varied significantly. As well as the interpretation of a successful test. There were roughly five testing styles in the reviewed papers. Either testing prototypes against each other, testing a prototype against a small case study, or testing a large sample size of batch testing. [71] and [15] where at either end of this spectrum. The final two styles were using locally installed antivirus systems or a pre-configured antivirus web service, for example [65] and [56] respectively. Each one of these testing styles has its own benefits and disadvantages. The most influential factor in selecting a testing method for this project was feasibility. Large sample testing requires additional programming, planning and computer power, so only a small case study will be possible. The use of a pre-configured antivirus web service is of the same benefit. The definition and intentions of the testing performed in each paper were always clearly stated, but when reading them from an outside perspective, the test may not cover all aspects of development required. For example, in [7], there is no indication of how well the tool generally works in obfuscation, just whether it produces uncorrupted applications.

By using the tool developed in [7], some of the extensive testing required to ensure the correct functionality of the program developed in this dissertation, was mitigated. The reliance on a preexisting tool, which had associated testing, reduced development time and computational resource requirements for this project. By adopting this prototyped tool, the developers' testing could also be adopted. In addition to the preexisting testing done by the paper's authors, testing was also performed during development of this dissertation, which can be seen in Section 2.2.1 and 2.3.4. This includes a recreation of the case study from the publication. To confirm the tool's current performance.

As already stated in Section 1.5.2, there is a gap in the literature on the topic of benchmarking for such a specific project as this one. While benchmarking will not be the primary focus of this project, attention will be given to the effectiveness of a chosen implementation in addressing the given problem and computational expense. Besides the general idea for a project, one of the most valuable pieces of information gained from this literature review was to understand approximate numbers for the parameter settings when using an evolutionary algorithm. Parameter tuning of machine learning is a time-consuming process. The parameter values, provided by several papers (e.g., [56] and [65]), provided an initial starting point for the parameter tuning process and shortened development time. The provided values also helped legitimise development decisions. The reviewed papers also detailed how to implement the operators of an evolutionary algorithm within the domain of malware, and this again reduced the trial-and-error required to get viable results.

This dissertation will lay a practical groundwork towards making more intelligent and robust antivirus systems. To put more pressure on malware developers. Consequently, more expert knowledge will be required to develop undetectable malware and reduce the volume of successfully evasive malware being created. Currently, a large proportion of malware is developed directly based on previous malware samples [43]. This allows low-skilled developers to modify known malware samples and reap the benefits. By training antivirus systems on a range of yet unseen modifications to known malware, the systems will be able to predict what types of modifications could be made to malware families in the future. The additional defence these systems provide will help protect everyday activities on mobile devices. Mobile devices form a dominant part of everyday life, from banking to socialising. This concept of social benefit has been directly adopted from [65], [36] and [43].

CHAPTER 2

Developing a Genetic Algorithm for Software Obfuscation

2.1 Introduction

This chapter details the prototype developed during this dissertation. It provides background information on the theoretical and technical aspects of the three core elements: the Android system, malware analysis and genetic algorithms. Section 2.3 details specific design choices that were made, of which a more in-depth investigation is found in Chapter 3.

2.1.1 Genetic Algorithms

Genetic algorithms (GA) are computer algorithms for manipulating lists, which are a subdomain of evolutionary algorithms (EA). One of the most famous results of an EA was using it to evolve a novel design for an antenna used on a NASA spacecraft that had not been thought of via current methods at the time [45]. GA's are heuristic search algorithms based on ideas from evolution in biology. They are designed to simulate "survival of the fittest" as proposed by Darwin. GA's use a guided random search to find a solution to a problem. GAs are typically used for combinatorial optimisation problems, for example, the Knapsack Problem, maximising a total value while being constrained by a total cost.

The key components of a GA are the individual's representation, the population of potential solutions, and its operators [50].

The individual representation, also known as a chromosome or genome in GAs, is an abstraction of the solution. It has two interpretations: the phenotype and the genotype. The genotype represents the problem within the GA and allows the operator to manipulate the individual effectively. In contrast, the phenotype is the actual real-world solution to the problem, which is directly converted from the genotype. In some cases, genotype and phenotype are the same, or in the case of this dissertation, they are entirely separate entities. In this dissertation, the genotype and phenotype are separate entities because of how the mutation function is implemented. An element in an individual is a gene, an individual is the chromosome, and a set of chromosomes is a population (see Figure 2.1).

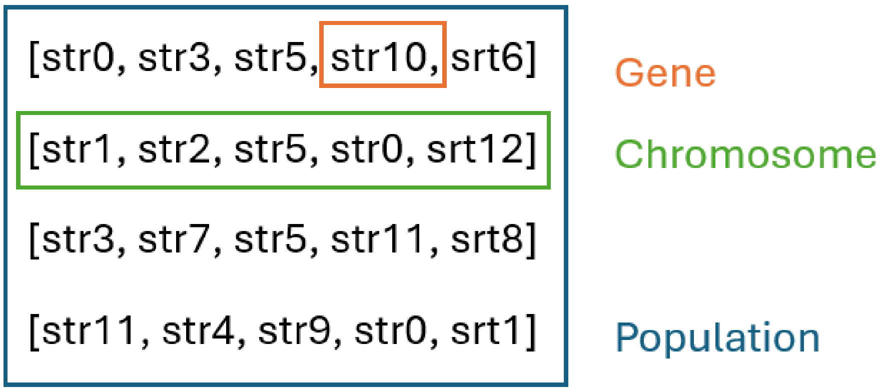

[str0, str3, str5, str10, srt6]
[str1, str2, str5, str0, srt12]
[str3, str7, str5, str11, srt8]
[str11, str4, str9, str0, srt1]
Gene
Chromosome
Population

Figure 2.1: An illustration of a gene, chromosome, and population. *strX* is a string.

The population is the search space for the solution. The global search space is all possible solutions that an individual can represent. The population is made from an initial starting set of individuals. The population could start with a specific set or a randomly generated set. In this dissertation, the population starts with an identical set of individuals. The process of applying a sequence of operators to individuals in the population is called a generation. Once a stopping condition is met, given the parameter set, the best individual from any generation is the best solution yielded. The stopping condition can be defined by the fitness function, a limit on the number of generations or computational time.

The operators are how the GA searches the problem space. These operators are crossover, mutation and selection. *Selection* allows chromosomes to pass to the next generation. Selection uses a fitness function to measure how well an individual fits the solution criteria. The criteria could be a specific result or a measurement taken from a specified result. The selection of individuals can be performed in several ways, for example, completely replacing the next generation or only replacing a selection of a few individuals. *Mutation* is about exploring the problem space. It makes slight changes to an individual; this could be random changes or specific alterations. *Crossover* exploits the current best solutions in the population; it combines elements from existing individuals to create new ones.

Developing an efficient GA is a nuanced and time-consuming process; therefore, using one of the well-known GA libraries is typical. There are many available implementations of GA. The main languages used are Java, Python, C++ and traditionally LISP. Python, by far, has the most implementations, but several are not customisable and can only be used for numerical problems. Typically, implementations based on TensorFlow or PyTorch are focused on numerical optimisation problems; their operators can not be easily customised and are designed to work on vectorised numbers. Conversely, this dissertation requires high customisability to enable the GA to operate with Android applications. Because of this, flexibility was the key characteristic in the selection process when narrowing down the wide range of Python-based implementations.

2.1.2 Malware Analysis

Malware analysis lies in the domain of cybersecurity. Its intention is to understand the underlying functionality of a program and determine whether its functionality is malicious or benign. Analysis can be performed manually by a human or by an automated computer system. In most cases, automated systems are relied upon due to their cost-effectiveness and scalability, which are needed to handle the vast amount of daily threats. Manual analysis is used in special cases and to help develop automated systems. Such an analysis will aim to determine if the program's actions could have malicious intent and then grade the program appropriately.

A typical way to grade programs after analysis would be to place the program within a specific category. For example, the dropper category, whereby a seemingly benign program is designed to install a malicious component in the future. Often, a specific definition of what malicious behaviour is exhibited is immaterial; however, from an analysis perspective, attention must be given to the trade-off between false positives and false negatives of an analytical interpretation. Automated techniques typically use records of learned behaviour to understand patterns that indicate malicious behaviour. A recent famous example of malware analysis was during the infection of WannaCry malware in 2017, which affected the NHS (some classify this malware as a ransomware worm [3]). In the case of WannaCry, researchers determined that the program accessed a remote server, known as a command and control server, after installation. Then, by redirecting traffic from the server, WannaCry's malicious actions were neutralised.

There are four basic steps to malware analysis:

1. **Acquisition**: Getting a copy of the program for analysis requires mitigating potential infections and successfully decompiling and running the program. This step prepares the program for the next two steps.

2. **Static Analysis**: Examining the program without executing it. The source code and package can be analysed to get any available metadata and understand its functionality. This information can help inform the next step of analysis, and understand possible threats and characteristics. Permissions and access granted by the app can indicate what the program is capable of, but techniques like permission escalation, colluding with another app, and reflection can hide this. Other notable issues include using encryption, naming conventions or code injection.

3. **Dynamic Analysis**: Executing the program in a sandboxed environment. Then observe its behaviour and determine its interactions with the system and network.

This step helps infer information that can not be determined from the source code alone. Command and control servers can play a large role in dynamic analysis. These servers can be used to get additional malware code, as well as customise the execution of the malware to suit different environments, which helps the program evade detection methods.

4. **Analysis Report**: Determine if any threats are present and how effective they are. In complex examples, static and dynamic analysis will need to be repeated, as they both help uncover the truth about what the program is doing.

Malware developers utilise obfuscation techniques to hide the intention of program code, which hinders the analysis of potential malware. By making malware more challenging to detect, weaknesses can be exposed in current detection methods, highlighting areas where these detectors could improve. There are several types of obfuscation: encryption, encoding, control/data flow, and renaming/randomisation. Their definitions can be found below. Different methods have different objectives; some are more effective against manual analysis than automated analysis, and vice versa. While others are more effective against different stages of analysis, one method could bypass a static analysis but would be detected by dynamic analysis.

- **Encryption:** Involves encrypting any part of the program. That could be an external asset, data or an entire part of the program. Once bytes are encrypted, it is hard to determine what has been encrypted until it is decrypted. For example, in polymorphic code [13], a program continually re-encrypts itself as it executes, constantly changing the program's signature. Besides making analysis in general difficult, this method would be highly effective against an automated analysis tool.

- **Encoding:** When parts of the program are written in different programming languages. This obfuscates code because known behaviour can be rewritten in different target-dependent languages, which automated analysis would be specifically susceptible to. For example, a program can have malicious content written into machine code for a specific target computer's CPU [43].

- **Control and data flow:** Additional functionality added to the program, which changes data access or how control passes through the program, for example, adding additional *if* statements that do not change the program's core functionality but still need to be evaluated at runtime. The *if* statement could be made more complex by adding a mathematical equation that needs to be evaluated to determine the Boolean value required in the conditional statement. This can affect a wide range of analysis methods.

- **Renaming and randomisation:** Changing the names of libraries and variables makes manual analysis harder and bypasses checks for sensitive functionality like access to encryption libraries. The order of lines in files can also be manipulated. For example, randomising a configuration file can affect methods like n-grams.

2.1.3 Android Operating System

The Android operating system (OS) is the most widely used OS in the world [18]. It is predominantly for touch screen, mobile systems and is based on the Linux kernel; it functions as a multi-user system whereby each application is a different user. Android OS is a free and open-source project developed by the Open Handset Alliance, a technology consortium whose most predominant member is Google [5]. Applications of the Android OS system have the *.apk* extension, which is a *.zip* file that contains all that is needed to install the application on a device. Applications can be downloaded from various online stores as well as manually installed by the user. Large device manufacturers have their own third-party application stores like Amazon Appstore, Samsung Galaxy Store, and Huawei AppGallery [44]. Different stores provide different services. For example, the *Google Play* store has servers that can host the apps for you, do basic stress testing on different physical devices, and allow the use of the Android Software Development Kit (SDK), which enables developers to access a wide range of pre-made software features like mapping. Android OS has a range of built-in security features. For example, each Android application has its own security sandbox, which includes the following features:

- A unique user ID, which the application is unaware of.
- Application permissions can only be used by the designated ID.
- A virtual machine is used to run he application's code, isolating it from other applications.
- The OS controls the running of application processes.

Android applications are commonly written in Java and other supported languages such as C++ and Kotlin. There are software libraries that allow the use of JavaScript and native machine code as well. There are four main components to the APK package: assets, metadata, manifest file and *DEX* files. Assets contain any additional data the application has stored. Metadata contains information about compilation and the development of the application. The manifest file configures the application for the system; it contains information about entry points to the application, what version of Android is required, and requested permissions. The *DEX* file is the main code for the application. There are four core components of an application's code: activities, services, broadcast receivers and content providers [27].

- **Activities:** are the main entry points, for example, the windows you see on screen.
- **Services:** are entry points for background functionality, such as footsteps counting.
- **Broadcast receivers:** are accessed resources generated from outside the application, for example, receiving a text message.
- **Content providers:** are shared memory access, for example, a SQL database.

Execution of a program is performed by the Android runtime (ART). Historically, this was performed by Dalvik, hence the Dalvik executable file (DEX). ART was fully released around 2014, but many of Dalvik's features are still supported. One of the reasons for the development of ART is improved garbage collection. ART uses ahead-of-time compilation (AOT) and the on-device tool dex2oat to compile applications at install time [25].

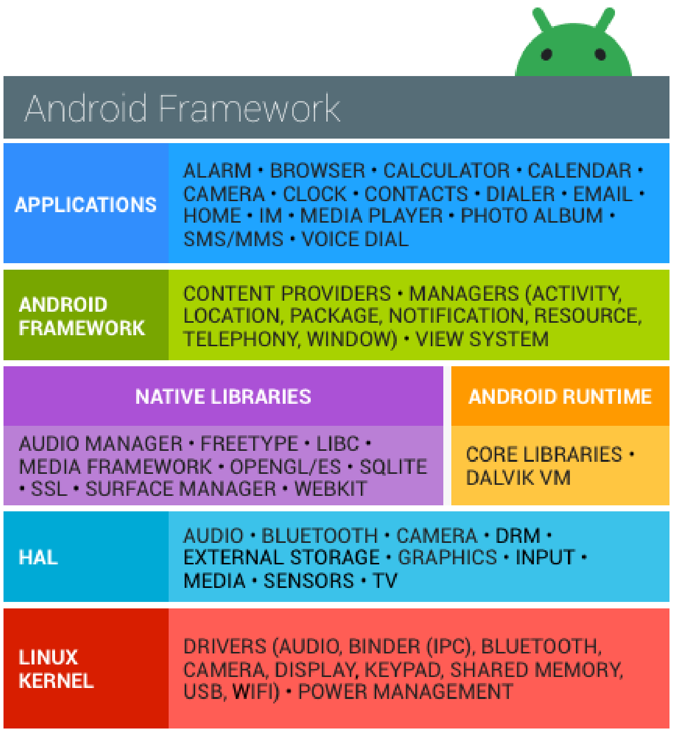

Android Framework
APPLICATIONS
ALARM • BROWSER • CALCULATOR • CALENDAR • CAMERA • CLOCK • CONTACTS • DIALER • EMAIL • HOME • IM • MEDIA PLAYER • PHOTO ALBUM • SMS/MMS • VOICE DIAL
ANDROID FRAMEWORK
CONTENT PROVIDERS • MANAGERS (ACTIVITY, LOCATION, PACKAGE, NOTIFICATION, RESOURCE, TELEPHONY, WINDOW) • VIEW SYSTEM
NATIVE LIBRARIES
AUDIO MANAGER • FREETYPE • LIBC • MEDIA FRAMEWORK • OPENGL/ES • SQLITE • SSL • SURFACE MANAGER • WEBKIT
ANDROID RUNTIME
CORE LIBRARIES • DALVIK VM
HAL
AUDIO • BLUETOOTH • CAMERA • DRM • EXTERNAL STORAGE • GRAPHICS • INPUT • MEDIA • SENSORS • TV
LINUX KERNEL
DRIVERS (AUDIO, BINDER (IPC), BLUETOOTH, CAMERA, DISPLAY, KEYPAD, SHARED MEMORY, USB, WIFI) • POWER MANAGEMENT

Figure 2.2: Android framework overview [29].

2.2 Methodology

2.2.1 Setup

- **Hardware specifications and runtime environment:** Python, Windows 11 home laptop, 16GB RAM, Ryzen 7 4000 series CPU.

- **Software and tools used:** Standard Python libraries, PowerShell, CURL, DEAP, Obfuscapk, Android Studio Suite.

2.2.2 Genetic Algorithm

A genetic algorithm (GA) was selected to search for the best combination of obfuscation that can be applied to a given sample. A GA can manage large problem spaces, is adaptable to different domains and has reasonable computational complexity. The GA was combined with an obfuscation library called Obfuscapk [7]. There are twenty obfuscation techniques available to use in Obfuscapk, but in this dissertation, up to fourteen techniques will be used in total. Different levels of obfuscation can be produced depending on which techniques are applied and in which order. Taking three of the fifteen techniques would give 2,730 permutations. Selecting all fifteen techniques gives 1,307,674,368,000 permutations. Obfuscation is also relative to individual programs, hence trying to exhaustively explore this large problem space by brute force would be unfeasible. A GA can heuristically explore a space of this magnitude and is also adaptable to complex problems requiring domain-specific implementation.

Typically, a GA framework will provide a template for managing the population, creating operators, applying operators in each generation, and miscellaneous features like storing statistics to assess the algorithm's performance. For non-numerical problems, the developer supplies an individual's representation, mutation and fitness function.

When choosing an efficient programming framework for this dissertation, support for a GA and genetic programming (GP), a related evolutionary algorithm, was an important factor. When searching for a solution, a GP seemed more powerful than a GA at performing task-specific operations. In the initial stages of development, a GP was used to apply obfuscation directly to the *APK*. A core issue that arose when implementing an GP, was that once the *APK* had been modified, the modifications were not necessarily still in the source code after compilation and disassembly, either creating an invalid APK or creating no change in the course code. The development time taken while working with GP provided a good understanding of the underlying principles for applying obfuscation. Consequently, the dissertation transitioned to using a GA with a prebuilt obfuscation library for *mutation*. When considering the computational complexity of a potential framework, a trade-off between the programs requirements and speed of development was considered. The slowest parts of this program are mainly external tools used by the GA. Additionally, in this dissertation, the GA performs a limited number of generations rather than requiring a specific end goal, further reducing the algorithm's potential load on the system. In some GA based projects, there could be thousands of generations with thousands of individuals in each population. In this dissertation, each generation has fewer than twenty individuals and runs for fewer than one hundred generations. These points meant practical considerations for the GA framework performance were more about reducing development time than computational time.

The key choice when selecting a *representation* was whether to use an abstract data type (e.g., a tree structure or a feature list) or source code samples. A benefit of using an abstract representation would be the application of preexisting methods, of which better algorithms would be available to solve the problem. On the contrary, using source code via disassembly gives a one-to-one mapping for the application's functionality and is computationally efficient, taking seconds to perform the task. Using an abstract representation requires more computation time and is an onto mapping. An onto mapping makes converting back to source code difficult, which makes the testing of obfuscation results less reliable. Abstract data types also make the system vulnerable to the algorithm trying to exploit the abstract representation to get the best results. Additionally, using source code as opposed to an abstract data type means that any choices made do not place a limit on later development choices. In summary, the phenotype is the *APK* file as it allows more robust testing of the results of the GA. The genotype was dictated by how the operators apply the obfuscation; in this case, it is a list of techniques for Obfuscapk to apply.

Ideally, each phenotype would be checked against an array of antivirus programs. However, this would be a slow process if performed in each generation and could create a bias regarding the program's size being operated on, because at this point, it is unclear if obfuscation techniques are size-independent. For instance, when adding junk code, if the original code has 100-line and 25 junk lines are added, the overall effect on the project is 25% change in the source code; conversely, inserting 25 lines into a 10,000-line program, that is only 0.25% change. In this dissertation, the program was structured as shown in Figure 2.3. The GA is run based on the program's size, using a lightweight fitness function to compare the file data of each *APK*, and at regular intervals, a more robust antivirus system test is applied. Comparison of APK file data shows how obfuscation has changed the byte structure of the application, but it does not show how well the obfuscation works. An antivirus system determines the effectiveness of obfuscation after a number of generations have passed. There are several popular online tools for providing antivirus system tests, the most common being VirusTotal [74]. This type of online service allows access to a range of antivirus tools and can be utilised as the primary testing metric for this project.

Fitness Metrics

To choose an appropriate fitness function there are two primary considerations, good computational performance and accurate representation of the relative differences of individuals in the population. Six metrics were investigated for a suitable fitness function: Jaccard, Levenshtein, n-gram, cdifflib, entropy and hashing. Consideration was given to metrics, computational efficiency and accuracy at mapping the individuals in the population, as the fitness function can be a bottleneck of a GA (see Section 3.2.2 for in-depth experiments)

Jaccard similarity [14] is used to compare the similarity between two data sets. This was the first attempt at a fitness function. This function works by dividing strings, using white space, into subsets and then comparing the sets of subsets. A trivial implementation suffers significantly from the order in which changes are inserted into an individual; a line changed at the beginning of the file will have the most impact, as it will shift the order of all subsequent lines. Cross-referencing checks in-between lines will be very resource-intensive. Jaccard similarity compares code using the following calculation:

$$setX = split(strX)$$

$$inter = set1 \cap set2$$

$$union = set1 \cup set2$$

$$similarity = count(inter)/count(union)$$

n-gram [53] can measure underlying features in the program. It works by cutting the sequences into blocks of length n+1 and dividing the common blocks by the total unique blocks. *n* is typically set between one and three. An n-gram compares code using the following calculation:

$$i = [0 \ldots len(strX) - n + 1]$$

$$gramX = step([i : i + n])$$

$$common = gram1 \wedge gram2$$

$$total = gram1 \vee gram2$$

$$similarity = count(common)/count(total)$$

For example:

1. Similarity: 101 / 101101 = 0.667
2. Split both strings in to length two words, n=2
   (a) 101 = (10): 1, (01): 1
   (b) 101101 = (10): 2, (01): 2, (11): 1
3. a = total identical characters in both strings (10): 1, (01): 1
4. b = total unique characters from both strings (10): 2, (01): 2, (11): 1
5. Similarity = length(a) / length(b) = 0.667 rounded to three decimal places.

cdifflib is a modern version of Gestalt pattern matching algorithm. It measures how different sequences are in a given set. It is written in C and packaged specifically for Python [16]. cdifflib works by counting matching words in each sequence. This function runs in the worst case in quadratic time and in linear time in the best case. Its behaviour is dependent on how many elements the sequences have in common. cdifflib compares code using the following calculation:

$$l = len(min(str1, str2))$$

$$i, j \in \{0 \ldots l-1\} \text{ where } i \leq j$$

$$n \in \{0 \ldots l\}$$

$$matches = step(str1[i : i+n] == str2[j : j+n])$$

$$M = count(matches)$$

$$similarity = 2.0 * M / len(str1 + str2)$$

For example:

1. Similarity: 101 / 101101 = 0.667
2. Match(i=0,j=0,n=1)=10, match(i=0,j=0,n=2)=101, match(i=1,j=1,n=1)=01
3. Similarity = 2*3/(3+6) = 0.667 rounded to three decimal places.

Entropy, a well-known algorithm for measuring the noisiness of a signal, is also used in malware detection. Entropy can measure a program's noise and show it changes over time. The noisiness of a sample can be used to indicate the application of obfuscation, repacking and encryption, which is used to speculate on the presence of malware [60]. Entropy is calculated using the following formula:

$$u = unique(strX)$$

$$p = [0 \ldots u/len(strX)]$$

$$entropy(strX) = -\Sigma(p * log_2(p))$$

$$similarity = |(entropy(str1) - entropy(str2))| / max(entropy(str1), entropy(str2))$$

Levenshtein distance [59] can be used to compare how many changes are needed to make the two words the same. It runs in $O(m * n)$, where 'm' and 'n' are the lengths of the strings being compared.

Hashing and *fuzzy hashing* are algorithms whereby data is converted to a hash and then compared. This has the following issues: there is a very slim chance that an identical hash can be produced for different data, but most importantly, only slight changes in the data can yield significant changes in the hash produced, making it unreliable for a fitness function.

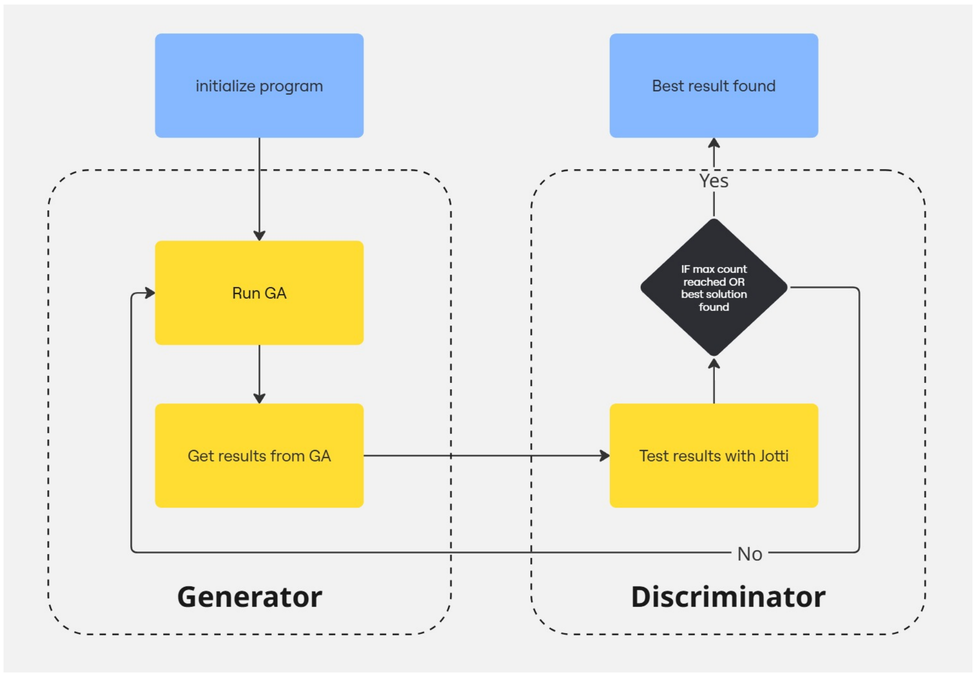


Figure 2.3: Program architecture overview.

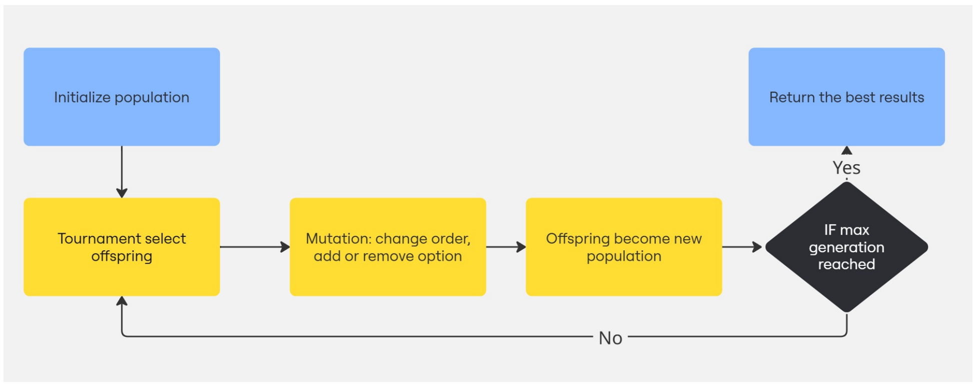

Initialize population
Return the best results
Yes
Tournament select offspring
Mutation: change order, add or remove option
Offspring become new population
IF max generation reached
No

Figure 2.4: Genetic algorithm flowchart.

2.2.3 Obfuscation

Initially, a custom obfuscation program would be built, however, it became clear this would be unfeasible given the time constraints of this dissertation. Therefore, a pre-existing obfuscation tool would be required. Android Studio has an inbuilt tool to do this [30]. *R8* is a compiler allowing developers to shrink, obfuscate, and optimise an application code. Customisability of *R8's* functionality is limited, and from manual inspection of malware source code, it appears to be a commonly used tool by malware developers, so further application of this tool will not give additional obfuscation to the program. *R8* use can be observed by comparing source code traits to samples provided on *R8* webpage. For example, import statements that look like "a.a.a.b".

Another common tool used is Proguard, available on GitHub [42]. This tool has a broader range of features; it can shrink, optimise, obfuscate, and pre-verify Java source code. It has more features for obfuscation than R8. Code obfuscation is available on resource files, the manifest file, stack traces and package names [61]. There is a broader range of available obfuscation categories, such as renaming and data/control flow manipulation, but the tool still lacked separate control of methods in each category.

To increase the GA's success rate, a wide range of unique obfuscation methods must be applied so that an individual in the population can have a wide range of variation. Several other obfuscation tools were mentioned in the literature, which are not free to use or have a limited free trial version available. The dissertation did not consider these products due to the constraints of using a timed trial version. Among them are:

- Zelix klassmaster [49].
- Dexprotector [22].
- DashO [20].
- Allatori [4].

One project with a lot of potential in the literature review was Obfuscapk. Obfuscapk provides control over a wide range of obfuscation techniques, it is stable and can run in

APK Name	Original File	Applying 18 Techniques	Time Taken
com.spike.old	40/76 [75]	18/76 [76]	10 minutes
comet-bot	44/76 [77]	20/76 [78]	2 minutes

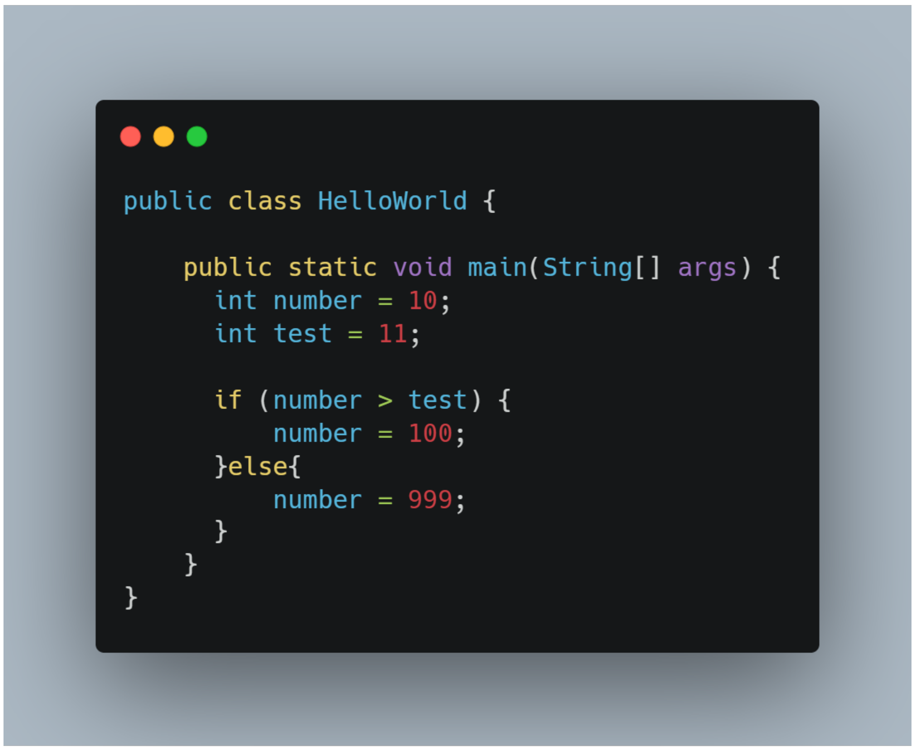
Table 2.1: Test results from VirusTotal, after applying all eighteen of Obfuscapk's obfuscation techniques.

a reliable amount of time on the available hardware. Therefore, Obfuscapk was deemed to be a suitable tool to combine with a GA. Before the GA could be developed, manual tests were performed to get a baseline understanding of how the obfuscation applied by Obfuscapk works. Two malware samples were selected for testing *comet-bot.apk* and *com.spike.old*. *comet-bot.apk* was selected as it was used to benchmark in the Obfuscapk paper. This allows a direct comparison to understand the current effectiveness of Obfuscapk. *com.spike.old* was selected as this sample has been used since the early research stage of this dissertation to understand different malware related concepts.

The following test, (see comet-bot in Table 2.1) was performed in March-April 2025 to recreate an experiment result from the Obfuscapk paper [7]. In Table 2.1, the VirusTotal ratio is made out of 76 because this includes the vendors that could not process the file, making the ratio more consistent between different tests, as the number of available vendors can change between experiments. It can be seen that the techniques are not as successful as they were when the paper was released. The authors in [7] claim a ratio from VirusTotal of 0/58 (zero detections out of 58 vendors). Checking the same link today, thereby accessing the historical records at VirusTotal, shows 26/63. The results shown in Table 2.1 confirm that the radio has increased. [7] was published in 2020, the results are expected to differ as antivirus systems improve.

Obfuscapk relies on three external dependencies. Firstly, Android Studio's *zipalign* is a zip archive alignment tool and optimiser that is required to make an installable application. Secondly, Android Studio's *apksigner* signs an APK, allowing supported Android OS versions to verify the signature. Finally, *APKtool* is a popular assembler/disassembler for APK packages, which uses smali/baksmali to convert an APK into assembly-like code, which can be repacked into an APK afterwards. Baksmali is generated from the DEX file, which contains machine code. In some cases, when using smali/baksmali, the APK changes in size, while the program remains functionally the same. Experimentation showed this is because smali/baksmali performed some optimisation (see Figures 2.5 and 2.6).

As Obfuscapk would be run repeatedly on an APK, understanding the repeated application's effects would be valuable. Three experiments were performed to achieve this. One, manually adding a single line to the source code and repacking using APKtool (see Figures 2.7 and 2.8). Two, the repeated repacking, assembling/disassembling, using APKtool (see Table 2.2). Three, using VirusTotal to check experiment two's results and the results of applying all Obfuscapk techniques at once (see Table 2.3).

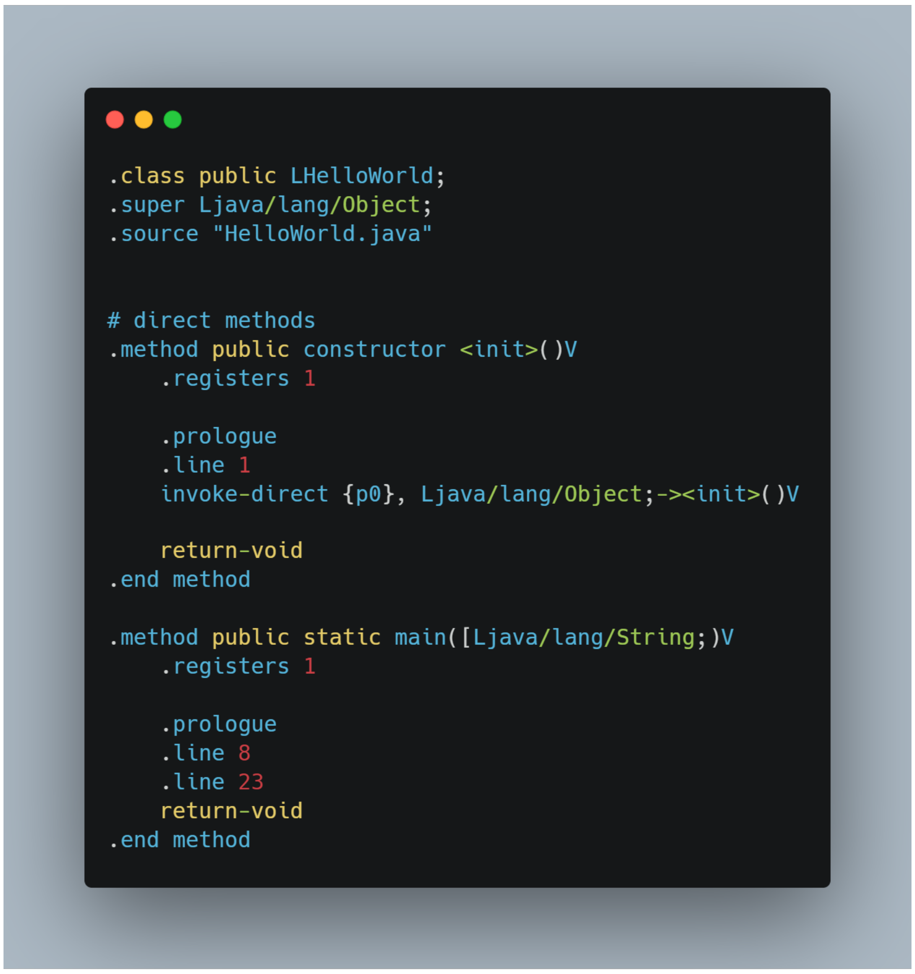


```java
public class HelloWorld {

    public static void main(String[] args) {
      int number = 10;
      int test = 11;

      if (number > test) {
          number = 100;
      }else{
          number = 999;
      }
    }
}
```

Figure 2.5: Java hello world example, *if* statement is always true, so the code will always assign the value 999, then does nothing with it.

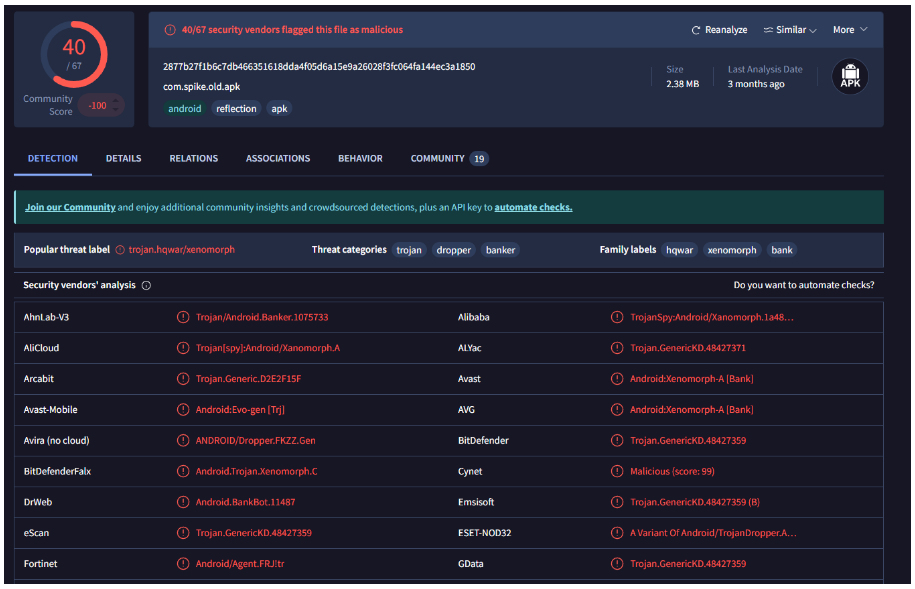
Experiment One: Added a single NOP instruction (No-Operation, spend a defined number of CPU cycles) to com.spike.old.apk, then repacking with APKtool. A vast drop in detection was found when uploading to VirusTotal. The sample's behaviour and genealogy were still detectable (see Figures 2.7 and 2.8).

Experiment Two: Repeatedly repacking an APK with APKtool. cdifflib ratio was used to measure this process's effect. Therefore, understanding how much effect, if



```
.class public LHelloWorld;
.super Ljava/lang/Object;
.source "HelloWorld.java"


# direct methods
.method public constructor <init>()V
    .registers 1

    .prologue
    .line 1
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    return-void
.end method

.method public static main([Ljava/lang/String;)V
    .registers 1

    .prologue
    .line 8
    .line 23
    return-void
.end method
```

Figure 2.6: Smali hello world example. The code has an entry point structured and one register assigned, but otherwise the code literally does nothing.



40/67 security vendors flagged this file as malicious

2877b27f1b6c7db466351618dda4f05d6a15e9a26028f3fc064fa144ec3a1850
com.spike.old.apk
android reflection apk

Size 2.38 MB
Last Analysis Date 3 months ago

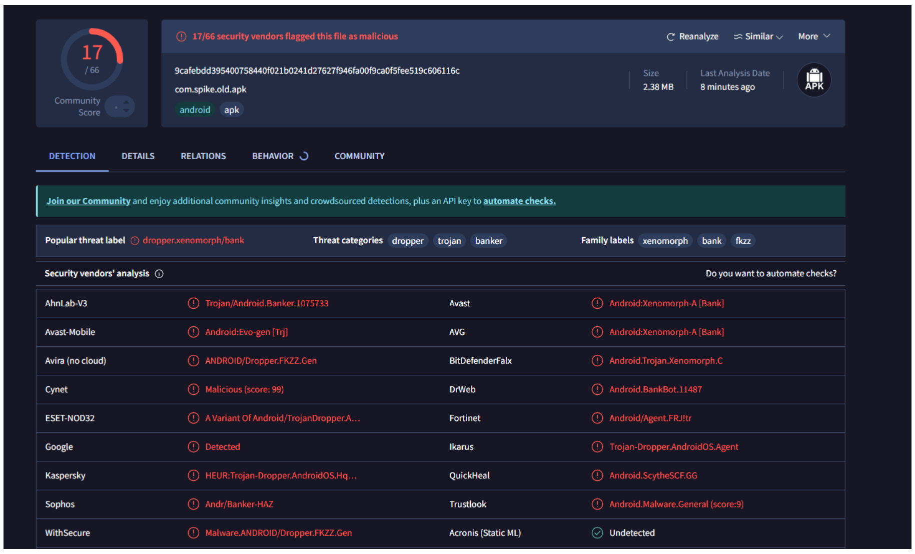
Join our Community and enjoy additional community insights and crowdsourced detections, plus an API key to automate checks.

Popular threat label trojan.hqwar/xenomorph
Threat categories trojan dropper banker
Family labels hqwar xenomorph bank

Security vendors' analysis
Do you want to automate checks?

AhnLab-V3	Trojan/Android.Banker.1075733	Alibaba	TrojanSpy:Android/Xanomorph.1a48...
AliCloud	Trojan[spy]:Android/Xanomorph.A	ALYac	Trojan.GenericKD.48427371
Arcabit	Trojan.Generic.D2E2F15F	Avast	Android:Xenomorph-A [Bank]
Avast-Mobile	Android:Evo-gen [Trj]	AVG	Android:Xenomorph-A [Bank]
Avira (no cloud)	ANDROID/Dropper.FKZZ.Gen	BitDefender	Trojan.GenericKD.48427359
BitDefenderFalx	Android.Trojan.Xenomorph.C	Cynet	Malicious (score: 99)
DrWeb	Android.BankBot.11487	Emsisoft	Trojan.GenericKD.48427359 (B)
eScan	Trojan.GenericKD.48427359	ESET-NOD32	A Variant Of Android/TrojanDropper.A...
Fortinet	Android/Agent.FRJ!tr	GData	Trojan.GenericKD.48427359

Figure 2.7: VirusTotal report for com.spike.old.apk with no changes made.



17/66 security vendors flagged this file as malicious

9cafebdd395400758440f021b0241d27627f946fa00f9ca0f5fee519c606116c
com.spike.old.apk
android apk

Size 2.38 MB
Last Analysis Date 8 minutes ago

Join our Community and enjoy additional community insights and crowdsourced detections, plus an API key to automate checks.

Popular threat label dropper.xenomorph/bank   Threat categories dropper trojan banker   Family labels xenomorph bank fkzz

Security vendors' analysis   Do you want to automate checks?

AhnLab-V3	Trojan/Android.Banker.1075733	Avast	Android:Xenomorph-A [Bank]
Avast-Mobile	Android:Evo-gen [Trj]	AVG	Android:Xenomorph-A [Bank]
Avira (no cloud)	ANDROID/Dropper.FKZZ.Gen	BitDefenderFalx	Android.Trojan.Xenomorph.C
Cynet	Malicious (score: 99)	DrWeb	Android.BankBot.11487
ESET-NOD32	A Variant Of Android/TrojanDropper.A...	Fortinet	Android/Agent.FRJ!tr
Google	Detected	Ikarus	Trojan-Dropper.AndroidOS.Agent
Kaspersky	HEUR:Trojan-Dropper.AndroidOS.Hq...	QuickHeal	Android.ScytheSCF.GG
Sophos	Andr/Banker-HAZ	Trustlook	Android.Malware.General (score:9)
WithSecure	Malware.ANDROID/Dropper.FKZZ.Gen	Acronis (Static ML)	✓ Undetected

Figure 2.8: VirusTotal report for com.spike.old.apk with added NOP instruction then repacked.

any, the continued use of APKtool would change the fitness value of an individual. When using cdifflib, a value of 1.0 would show no change, and 0.0 would be completely different. In summary, repacking has the effect of changing the original source code by a ratio of 0.426 in the first test, but a subsequent repack has a result of 0.978. Comparing the original with the repacked-repacked results in a ratio of 0.426 again. Showing that subsequent use of APKtool has little effect on the APK similarity (see Table 2.2).

APK Name	Original/Repack Once	Original/Repack Twice	Repack Once/Repack Twice
com.spike.old	0.426	0.426	0.978
comet-bot	0.358	0.358	0.904

Table 2.2: Test results comparing the repeated repacking of an APK file using cdifflib.

Experiment Three: Most interestingly, just using *APKtool* had almost the same effect as running all the 18 obfuscapk techniques on *com.spike.old.apk*, only bypassing one additional vendor. The difference ratio from cdifflib after applying all 18 techniques was 0.424 for *com.spike.old.apk/ com.spike.old-obfuscated-all.apk*. For *comet-bot.apk*, two to five additional vendors were bypassed using all 18 techniques compared to just applying APKtool. The difference ratio from cdifflib after applying all 18 techniques was 0.172 for *comet-bot.apk/comet-bot-obfuscated-all.apk* (see Table 2.3).

The experiments showed that the sample selected affected the outcome. *comet-bot.apk*, is particularly susceptible to changes from obfuscapk compared to *com.spike.old.apk*.

APK Name	Original File	Repacking Once	Repacking Twice	Applying 18 Techniques
com.spike.old	40/76 [75]	19/76 [79]	19/76 [80]	18/76 [76]
comet-bot	44/76 [77]	22/76 [81]	25/76 [82]	20/76 [78]

Table 2.3: Test results comparing the repeated repacking, and applying obfuscation of an APK file using VirusTotal.

This could be because the changes are, in part, relative to the volume of file data being operated on. *comet-bot.apk* is approximately 55KB while *com.spike.old.apk* is approximately 2,400KB. The smaller program is easier to manipulate with fewer changes. These experiments also show that while repacking had some effect on changing the static characteristics of the APK, consecutive runs of APKtool have only a minor effect on the file data. This means that changes made over many generations can be attributed to applying obfuscation techniques rather than simply reordering the bytecode by assembling/disassembling it. The GA can then search for complementary and effective combinations of obfuscation techniques that can be used to bypass modern antivirus vendors.

There are four options to determine if obfuscation is working: doing a manual inspection, feature-based methods, installing antivirus systems on a local machine, or getting results from an online antivirus system. In either case, static analysis will be used as it is more convenient to implement, and dynamic analysis typically requires static analysis as a preliminary step:

- **Manual inspection:** Would be time-consuming and not reflect what happens in the real world, as generally automated tools are used to detect malware, which makes this an impractical solution.
- **Local machine antivirus systems:** Would require additional computational resources, which makes this an unfeasible solution.
- **Feature-based methods:** Like YARA rule set, or a database of known malware features, limit analysis to specific samples and obfuscation techniques.
- **Online antivirus systems:** Will allow the developed algorithm to be tested in a real-world context and reduce the computation used by the local machine. The

ratio of detected/total antivirus vendors is used as a metric to compare results from an online testing system.

In the literature, the most popular web service for testing with antivirus systems is VirusTotal. Several other alternatives provide free access, as shown in the list below:

- **VirusTotal** [74] - There are a range of well-known antivirus vendors available about 76 AV in total including AVG , Avast-Mobile, SymantecMobileInsight, McAfeeD. The report provides results directly from, threat labels and categories family labels as well as YARA rule set and MITRE ATT&CK techniques. VirusTotal is also able to run the application in a sandboxed environment. There is a free version with limited functionality and number of request that can be made within a given time period.

- **Jotti** [46] - works well, a limited but specific information provided in the report. There are 13 vendors, has good API support and the free version is reliable and robust.

- **Metadefender** [55] - The provided report covers a wide range of details, but limited capability to detect malware. Using the free version, all vendors were bypassed once a few obfuscation techniques were applied.

- **Kaspersky** [1] - need to pay for a workable report but can get very basic report for free via API.

- **Cookkoo sandbox** [19] - produces a 10/10 score and provides some in-depth details, very low limit of use, you do not know when you will get your results, it can take over a day in the worst case.

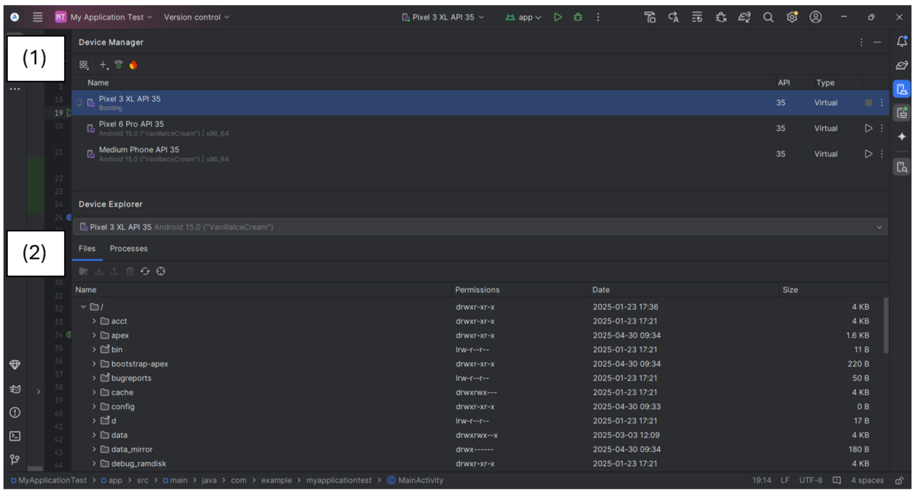
These antivirus web services provided a report detailing information about an uploaded APK. The reported information can range anywhere from a simple, malicious/benign label, to details about suspicious functions in the APK (see Figure 2.7). The graphical user interface (GUI) of VirusTotal is very reliable, but when accessing the service programmatically, it was found that the free access to VirusTotal had issues. The response from the server was often not fulfilled, and the time needed to wait varied from day to day, probably due to a queuing system used to control free requests to the server. This time frame was expanded, but still, on occasion, there would be no response. This might be due to the number of requests being posted to the service at that time. Jotti was the clear choice for this dissertation, due to its reliability and detailed reporting.

2.2.4 Android Operating System

There are some practical considerations that make the Android operating system (OS) a good choice for this dissertation. Android applications are relatively small, and there is a wide range of free resources for application development. Most importantly, working with Android applications on a Windows computer helps protect against some accidental infection. This is because few malware samples are developed as multiplatform programs, meaning they do not have the functionality to perform malicious behaviour on an Android and Windows computer simultaneously. That said, Windows Defender does recognise some Android malware, and therefore, security exceptions need to be created in order to run a genetic algorithm that interacts with malware; otherwise, newly modified files get automatically deleted.

Android Studio is a free resource that comes with a built-in emulator that allows developers to run applications on various simulated hardware configurations, such as watches, tablets and phones. This emulator can be used to test the effects of malware. In this dissertation, testing was performed on simulated mobile phones, as they are a widely used Android consumer product, and there will also be lots of available malware samples to choose from. Android Debug Bridge (ABD) interacts with the emulator via a command-line interface. ABD allows tools and shell scripts to be run on the simulated device to aid analysis and testing.

2.2.5 Data Collection


My Application Test
Version control
Pixel 3 XL API 35
app
(1)
(2)
Device Manager
Device Explorer
Pixel 3 XL API 35 Android 15.0 ("VanillaIceCream")
Files
Processes
MyApplicationTest > app > src > main > java > com > example > myapplicationtest > MainActivity
19:14 LF UTF-8 4 spaces

Name	API	Type
Pixel 3 XL API 35 Booting	35	Virtual
Pixel 6 Pro API 35 Android 15.0 ("VanillaIceCream") | x86_64	35	Virtual
Medium Phone API 35 Android 15.0 ("VanillaIceCream") | x86_64	35	Virtual

Name	Permissions	Date	Size
/	drwxr-xr-x	2025-01-23 17:36	4 KB
acct	drwxr-xr-x	2025-01-23 17:21	4 KB
apex	drwxr-xr-x	2025-04-30 09:34	1.6 KB
bin	lrw-r--r--	2025-01-23 17:21	11 B
bootstrap-apex	drwxr-xr-x	2025-04-30 09:34	220 B
bugreports	lrw-r--r--	2025-01-23 17:21	50 B
cache	drwxrwx---	2025-01-23 17:21	4 KB
config	drwxr-xr-x	2025-04-30 09:33	0 B
d	lrw-r--r--	2025-01-23 17:21	17 B
data	drwxrwx--x	2025-03-03 12:09	4 KB
data_mirror	drwx------	2025-04-30 09:34	180 B
debug_ramdisk	drwxr-xr-x	2025-01-23 17:21	4 KB

Figure 2.9: Screenshot of Android Studio. (1) Different emulator for hardware phone can be created. (2) The file system of the emulated phone can be accessed via GUI.

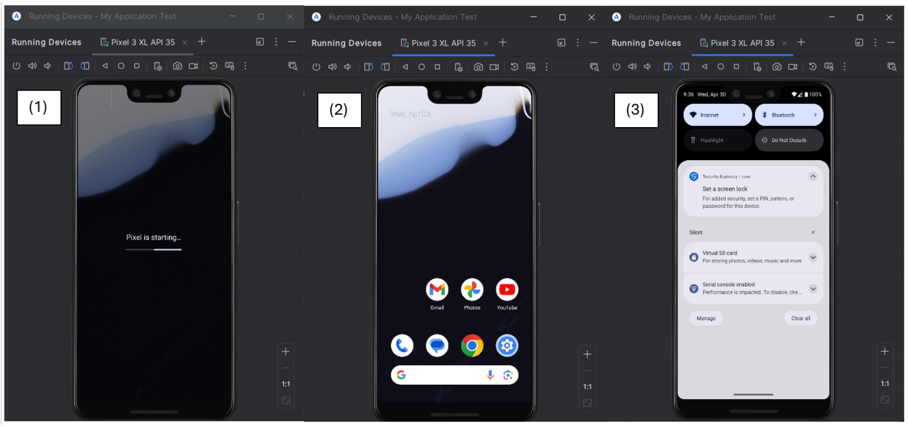


Figure 2.10: (1) Emulator from start up. (2) Emulator loaded. (3) Interaction with GUI can be done with mouse.

There are three main ways to obtain malware samples. Firstly, getting the source code, secondly getting an identifying hash, and finally downloading a database of features:

- **Source code:** Malware samples can be downloaded from GitHub [11] or other code respirators like Vx-underground [72].
- **Identifying hashes:** A unique number that can be used to identify a program, known as a hash ID. Online antivirus systems use hash ID's [74] to identify

malware samples. Either allowing the malware samples to be downloaded, or providing access to information stored about the related samples.

- **Databases of features:** Contain details like manifest file permissions and function calls. Databases can be found via GitHub [52] and also Kaggle [47], a common approach used in machine learning.

For this dissertation, source code was chosen because it allows direct manipulation of the APK, after which the functionality of the APK can be tested. The APK samples were collected from GitHub and identified from a reference in the literature [7] [43]. The samples were found by following web links directly, or referenced hashes were used to locate specific samples. The samples were tested on the Android emulator to see if they could run their basic functionality, be installed, and interact with the Graphical User Interface (GUI) and OS. When selecting samples, three key points were considered:

- **The disk size of the malware:** This affects the overall runtime, huge files make the program run slower due to the nature of moving and compiling and affect Obfuscapk runtime.

- **Anti-emulator design:** Some malware are designed not to work on an emulator to avoid being analysed. This can be determined with a trial-and-error method or using prior knowledge of the sample.

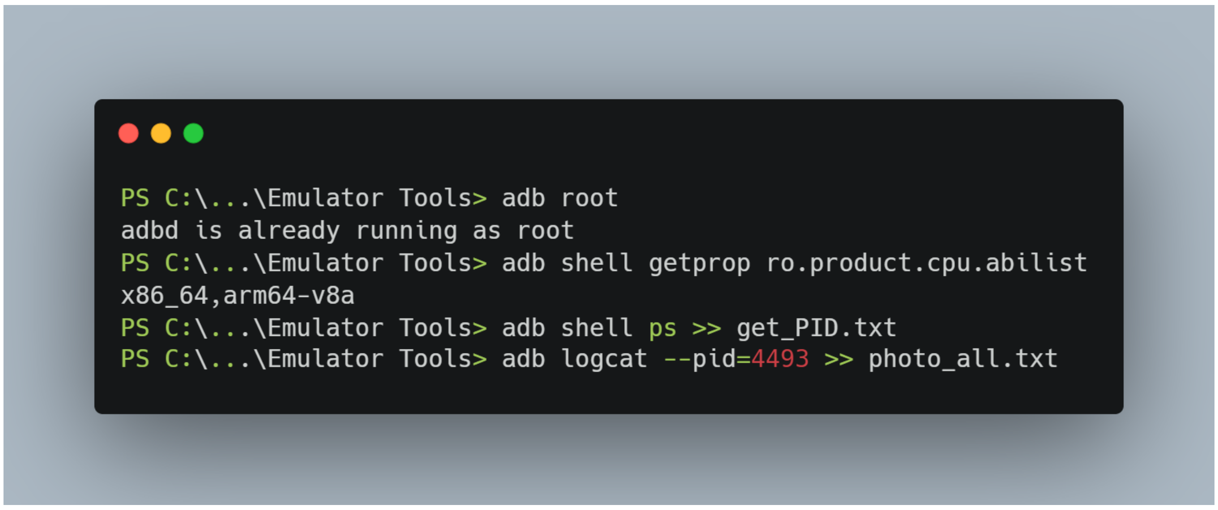
- **Malware structure:** In various cases, malware samples are not complete programs. Not having a full GUI component makes testing more complex. Therefore, a basic idea of the sample structure is required to understand whether it is a complete program. Source code samples stored online can come in various formats other than a complete malware sample. They can be a package to be loaded by another program, a specific GUI activity which needs a host activity to run, or a background activity without any explicit GUI activity. Structuring samples like this is typical in software development where developers often recycle and reuse well-known code bases to make software development easier.

- **Malware functionality:** For example, if the malware launches from the hardware power off button being pressed, which is not available on a software emulator.
- **SDK number:** Some applications have a maximum and minimum SDK number requirement, which can be an issue as the emulator does not simulate all SDK versions.

A simple testing procedure was used. An example of this is as follows:

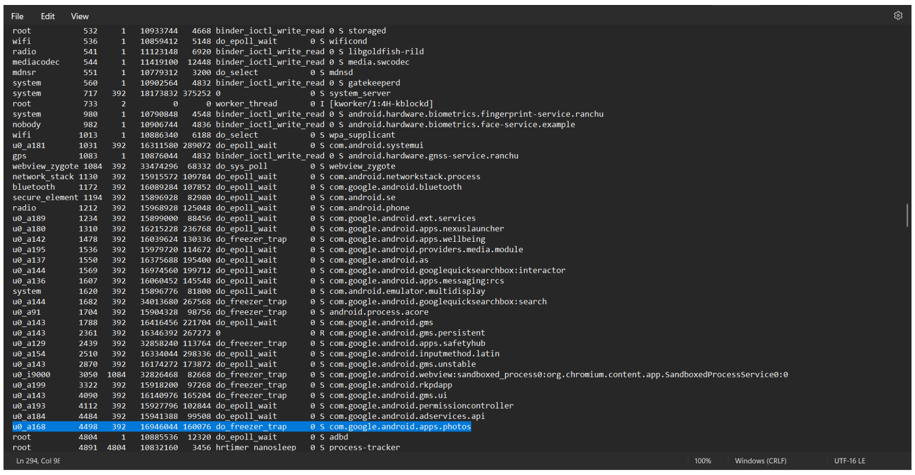
1. Is it installable?
2. Check if the file structure of the installed sample is present on the OS system. When looking at the emulator's file structure, some files may have the incorrect names (e.g., file.json != type(zip)). This is due to the prior application of obfuscation.
3. Does the application open from the home menu? Using the GUI interface, the installed application asks for the correct permissions. Permissions are requested by the application when launched for the first time.
4. Look for consistent system messages specific to the application (see Figure 2.11). *Logcat* is a command-line tool that logs all system messages, which includes messages written by the developer using the Log class (see Figures 2.12 and 2.13). *Logcat* is memory-limited, so the time the logs are taken dictates which logs can be seen. *Logcat* was used to record the first six seconds of the program running without selecting permissions. This record can then be used to compare whether the program makes the same function calls after being processed by the GA. The log records are checked using a text difference checker [23], combined with technical knowledge of how Android and the application work, for example, looking for suspect file names like sniffer or encryption.

There are two other methods that can be used for testing that were not used at this stage. Monitoring a program's access to shared memory locations/network traffic and *Frida*, a code injection tool. Frida allows interaction with the underlying code of the



```
PS C:\...\Emulator Tools> adb root
adbd is already running as root
PS C:\...\Emulator Tools> adb shell getprop ro.product.cpu.abilist
x86_64,arm64-v8a
PS C:\...\Emulator Tools> adb shell ps >> get_PID.txt
PS C:\...\Emulator Tools> adb logcat --pid=4493 >> photo_all.txt
```

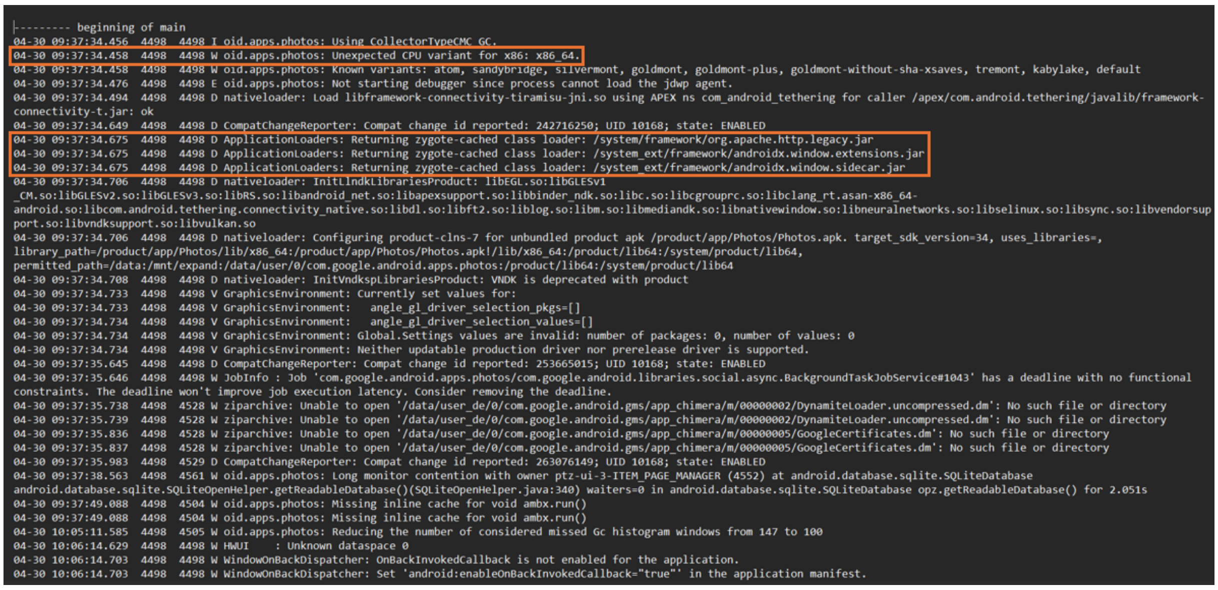
Figure 2.11: Example commands to interface with emulator and collect actions of an application.



```
root           532     1   10933744   4668 binder_ioctl_write_read 0 S storaged
wifi           536     1   10859412   5148 do_epoll_wait       0 S wificond
radio          541     1   11123148   6920 binder_ioctl_write_read 0 S libgoldfish-rild
mediacodec     544     1   11419100  12448 binder_ioctl_write_read 0 S media.swcodec
mdnsr          551     1   10779312   3200 do_select           0 S mdnsd
system         560     1   10902564   4832 binder_ioctl_write_read 0 S gatekeeperd
system         717   392   18173832 375252 0                   0 S system_server
root           733     2          0      0 worker_thread       0 I [kworker/1:4H-kblockd]
system         980     1   10790848   4548 binder_ioctl_write_read 0 S android.hardware.biometrics.fingerprint-service.ranchu
nobody         982     1   10906744   4836 binder_ioctl_write_read 0 S android.hardware.biometrics.face-service.example
wifi          1013     1   10886340   6188 do_select           0 S wpa_supplicant
u0_a181       1031   392   16311580 289072 do_epoll_wait       0 S com.android.systemui
gps           1083     1   10876044   4832 binder_ioctl_write_read 0 S android.hardware.gnss-service.ranchu
webview_zygote 1084  392   33474296  68332 do_sys_poll         0 S webview_zygote
network_stack 1130   392   15915572 109784 do_epoll_wait       0 S com.android.networkstack.process
bluetooth     1172   392   16089284 107852 do_epoll_wait       0 S com.google.android.bluetooth
secure_element 1194  392   15896928  82980 do_epoll_wait       0 S com.android.se
radio         1212   392   15968928 125048 do_epoll_wait       0 S com.android.phone
u0_a189       1234   392   15899000  88456 do_epoll_wait       0 S com.google.android.ext.services
u0_a180       1310   392   16215228 236768 do_epoll_wait       0 S com.google.android.apps.nexuslauncher
u0_a142       1478   392   16039624 130336 do_freezer_trap     0 S com.google.android.apps.wellbeing
u0_a195       1536   392   15979720 114672 do_epoll_wait       0 S com.google.android.providers.media.module
u0_a137       1550   392   16375688 195400 do_epoll_wait       0 S com.google.android.as
u0_a144       1569   392   16974560 199712 do_epoll_wait       0 S com.google.android.googlequicksearchbox:interactor
u0_a136       1607   392   16060452 145548 do_epoll_wait       0 S com.google.android.apps.messaging:rcs
system        1620   392   15896776  81800 do_epoll_wait       0 S com.android.emulator.multidisplay
u0_a144       1682   392   34013680 267568 do_freezer_trap     0 S com.google.android.googlequicksearchbox:search
u0_a91        1704   392   15904328  98756 do_freezer_trap     0 S android.process.acore
u0_a143       1788   392   16416456 221704 do_epoll_wait       0 S com.google.android.gms
u0_a143       2361   392   16346392 267272 0                   0 R com.google.android.gms.persistent
u0_a129       2439   392   32858240 113764 do_freezer_trap     0 S com.google.android.apps.safetyhub
u0_a154       2510   392   16334044 298336 do_epoll_wait       0 S com.google.android.inputmethod.latin
u0_a143       2870   392   16174272 173872 do_epoll_wait       0 S com.google.android.gms.unstable
u0_i9000      3050  1084   32826468  82668 do_freezer_trap     0 S com.google.android.webview:sandboxed_process0:org.chromium.content.app.SandboxedProcessService0:0
u0_a199       3322   392   15918200  97268 do_freezer_trap     0 S com.google.android.rkpdapp
u0_a143       4090   392   16140976 165204 do_freezer_trap     0 S com.google.android.gms.ui
u0_a193       4112   392   15927796 102844 do_epoll_wait       0 S com.google.android.permissioncontroller
u0_a184       4484   392   15941388  99508 do_epoll_wait       0 S com.google.android.adservices.api
u0_a168       4498   392   16946044 160076 do_freezer_trap     0 S com.google.android.apps.photos
root          4804     1   10885536  12320 do_epoll_wait       0 S adbd
root          4891  4804   10832160   3456 hrtimer_nanosleep   0 S process-tracker
```

Figure 2.12: List of all Process Ids (PID) on the Android emulator, photo application highlighted.

application at runtime, this can be used in conjunction with the other methods already mentioned above to dive deeper into what the program is doing. Frida required the installation of a server on the Android emulator. Android emulators need a substantial amount of resources to run. For example, if the emulated OS needs four gigabytes of RAM, the hardware system supporting the emulator must provide more than four gigabytes of RAM due to the additional overhead emulators require for additional computation. The running of the Frida server increases the already strenuous overhead of running an application on the emulator, which causes errors when accessing files remotely. Therefore, Frida will not be used for testing. Looking at shared memory locations/network traffic for installed applications can help further understand the program's functionality. However, in this dissertation, the emulator is not connected



```
|--------- beginning of main
04-30 09:37:34.456  4498  4498 I oid.apps.photos: Using CollectorTypeCMC GC.
04-30 09:37:34.458  4498  4498 W oid.apps.photos: Unexpected CPU variant for x86: x86_64.
04-30 09:37:34.458  4498  4498 W oid.apps.photos: Known variants: atom, sandybridge, silvermont, goldmont, goldmont-plus, goldmont-without-sha-xsaves, tremont, kabylake, default
04-30 09:37:34.476  4498  4498 E oid.apps.photos: Not starting debugger since process cannot load the jdwp agent.
04-30 09:37:34.494  4498  4498 D nativeloader: Load libframework-connectivity-tiramisu-jni.so using APEX ns com_android_tethering for caller /apex/com.android.tethering/javalib/framework-
connectivity-t.jar: ok
04-30 09:37:34.649  4498  4498 D CompatChangeReporter: Compat change id reported: 242716250; UID 10168; state: ENABLED
04-30 09:37:34.675  4498  4498 D ApplicationLoaders: Returning zygote-cached class loader: /system/framework/org.apache.http.legacy.jar
04-30 09:37:34.675  4498  4498 D ApplicationLoaders: Returning zygote-cached class loader: /system_ext/framework/androidx.window.extensions.jar
04-30 09:37:34.675  4498  4498 D ApplicationLoaders: Returning zygote-cached class loader: /system_ext/framework/androidx.window.sidecar.jar
04-30 09:37:34.706  4498  4498 D nativeloader: InitLlndkLibrariesProduct: libEGL.so:libGLESv1
_CM.so:libGLESv2.so:libGLESv3.so:libRS.so:libandroid_net.so:libapexsupport.so:libbinder_ndk.so:libc.so:libcgrouprc.so:libclang_rt.asan-x86_64-
android.so:libcom.android.tethering.connectivity_native.so:libdl.so:libft2.so:liblog.so:libm.so:libmediandk.so:libnativewindow.so:libneuralnetworks.so:libselinux.so:libsync.so:libvendorsup
port.so:libvndksupport.so:libvulkan.so
04-30 09:37:34.706  4498  4498 D nativeloader: Configuring product-clns-7 for unbundled product apk /product/app/Photos/Photos.apk. target_sdk_version=34, uses_libraries=,
library_path=/product/app/Photos/lib/x86_64:/product/app/Photos/Photos.apk!/lib/x86_64:/product/lib64:/system/product/lib64,
permitted_path=/data:/mnt/expand:/data/user/0/com.google.android.apps.photos:/product/lib64:/system/product/lib64
04-30 09:37:34.708  4498  4498 D nativeloader: InitVndkspLibrariesProduct: VNDK is deprecated with product
04-30 09:37:34.733  4498  4498 V GraphicsEnvironment: Currently set values for:
04-30 09:37:34.733  4498  4498 V GraphicsEnvironment:   angle_gl_driver_selection_pkgs=[]
04-30 09:37:34.734  4498  4498 V GraphicsEnvironment:   angle_gl_driver_selection_values=[]
04-30 09:37:34.734  4498  4498 V GraphicsEnvironment: Global.Settings values are invalid: number of packages: 0, number of values: 0
04-30 09:37:34.734  4498  4498 V GraphicsEnvironment: Neither updatable production driver nor prerelease driver is supported.
04-30 09:37:35.645  4498  4498 D CompatChangeReporter: Compat change id reported: 253665015; UID 10168; state: ENABLED
04-30 09:37:35.646  4498  4498 W JobInfo : Job 'com.google.android.apps.photos/com.google.android.libraries.social.async.BackgroundTaskJobService#1043' has a deadline with no functional
constraints. The deadline won't improve job execution latency. Consider removing the deadline.
04-30 09:37:35.738  4498  4528 W ziparchive: Unable to open '/data/user_de/0/com.google.android.gms/app_chimera/m/00000002/DynamiteLoader.uncompressed.dm': No such file or directory
04-30 09:37:35.739  4498  4528 W ziparchive: Unable to open '/data/user_de/0/com.google.android.gms/app_chimera/m/00000002/DynamiteLoader.uncompressed.dm': No such file or directory
04-30 09:37:35.836  4498  4528 W ziparchive: Unable to open '/data/user_de/0/com.google.android.gms/app_chimera/m/00000005/GoogleCertificates.dm': No such file or directory
04-30 09:37:35.837  4498  4528 W ziparchive: Unable to open '/data/user_de/0/com.google.android.gms/app_chimera/m/00000005/GoogleCertificates.dm': No such file or directory
04-30 09:37:35.983  4498  4529 D CompatChangeReporter: Compat change id reported: 263076149; UID 10168; state: ENABLED
04-30 09:37:38.563  4498  4561 W oid.apps.photos: Long monitor contention with owner ptz-ui-3-ITEM_PAGE_MANAGER (4552) at android.database.sqlite.SQLiteDatabase
android.database.sqlite.SQLiteOpenHelper.getReadableDatabase()(SQLiteOpenHelper.java:340) waiters=0 in android.database.sqlite.SQLiteDatabase opz.getReadableDatabase() for 2.051s
04-30 09:37:49.088  4498  4504 W oid.apps.photos: Missing inline cache for void ambx.run()
04-30 09:37:49.088  4498  4504 W oid.apps.photos: Missing inline cache for void ambx.run()
04-30 10:05:11.585  4498  4505 W oid.apps.photos: Reducing the number of considered missed Gc histogram windows from 147 to 100
04-30 10:06:14.629  4498  4498 W HWUI    : Unknown dataspace 0
04-30 10:06:14.703  4498  4498 W WindowOnBackDispatcher: OnBackInvokedCallback is not enabled for the application.
04-30 10:06:14.703  4498  4498 W WindowOnBackDispatcher: Set 'android:enableOnBackInvokedCallback="true"' in the application manifest.
```

Figure 2.13: Log dump from a photo application. A message on the second line states that the CPU model is unusual, the application is running on an emulated CPU. Three messages refer to a class loader, which shows the application loading additional system classes.

to an external network, so monitoring network traffic is redundant. Looking at shared memory location access is application dependent and, in some cases, can be visible from the log dumps; consequently, these methods were not explored further at this stage.

For automating an application's interaction, a tool called *Monkey* is referred in the literature. *Monkey* sends a stream of random user interaction to the application; this can be used to confirm that the application is functioning as expected. It was found that Monkey is error-prone, causing the application to crash, making it too unreliable for testing. There is the deprecated tool *Monkeyrunner* which can be used to send direct commands to the application and be used similarly to Monkey. App Crawler and UI Automator have superseded Monkeyrunner [28].

2.3 Implementation

This program can be formalized as below, see Algorithm 1 for further description of the developed prototype:

Search space: set S : $\{\widehat{x} \mid$ where $\widehat{x}$ is a valid APK$\}$

Goal: $x^* \in S$

Fitness function: $f : S \to \mathbb{R}$

Objective Function (GA): $x^* = \arg\max \mathit{or} \arg\min_{x \in S} f(x)$

Population: set $P(t) = \{x_1, x_2, \ldots, x_n\}$ where $x_i \in S$

Fitness: for $x_i \in P(t)$, compute $f(x_i)$

Selection: map $P(t) \to P'(t)$

Mutation : $P'(t) \to P(t + p_m)$ where $p_m \in [M_1 : x_t, M_2 : x_t, M_3 : x_t]$

Algorithm 1 Pseudocode of the developed prototype

```
Initialise program
Get Jotti score of sample
while $runs \leq maxRuns$ do
    Initialise SGA
    $\to$ randomly generate population
    while $generation \leq maxGeneration$ do                ▷ Run SGA
        Tournament selection from population
        Mutation on randomly selected individuals
        Fitness evaluation of whole population              ▷ Using n-grams
        $\to$ add fittest individual to hall of fame
    end while
    Select best individuals from hall of fame for this run  ▷ From any generation
    $\to$ Get Jotti score of selected individuals
end while
Select best individual from all runs          ▷ Individual with lowest Jotti score
```

The package Distributed Evolutionary Algorithms in Python (DEAP) laid the foundation of the GA in this dissertation. The modifications made to DEAP included a custom fitness and mutation function. This required the development of a class object to manage how the GA can make modifications to APK files. The development of the class object allowed easy scalability to the prototype, allowing future functionality to be added. Several of the operations performed by the GA are handled with command-line instructions. For example, renaming the APK after modifications to track changes. An individual from the GA creates a custom command-line instruction used to control

Obfuscapk. Obfuscapk is the package that provides the core functionality of applying mutation to an APK in the form of obfuscation. The GA is run from a main loop, after the GA runs the best APK is selected with Jotti (see Figure 2.3). Then, a selection process is used to choose the APK as the input for the next GA run. The prototype is set up in a generator/discriminator configuration, with the GA being the generator and Jotti being a discriminator. In future work Jotti could be replaced or enhanced with a more robust discriminator with a better feedback signal. This configuration also allows for easily resetting the GA, from a previous run, because any APK generated by the GA can be used as a starting point for another run.

When selecting a GA, there are two key architectures: steady state GA (SSGA) or Simple GA (SGA). *SGA* is known for faster convergence rates and exploiting the best solutions, which leads to the disadvantage of premature convergence and loss of population diversity. Conversely, in an *SSGA*, the population is replaced more slowly and a wider selection of individuals remain between generations, lowering premature convergence. Using a generator and discriminator configuration increases population diversity by resetting the population after each run, reducing premature convergence and increasing population diversity overall. Therefore, combining an *SGA* and the generator/discriminator configuration will lessen the negative effects of a *SGA's* low population diversity and benefit the *SGA* in exploiting the best available solutions. This makes an *SGA* a clear choice compared to the slower and explorative *SSGA*.

When the prototype is executed, the main loop runs for a set number of runs. In each run, the SGA performs the same number of generations, so the total generations performed during one execution is *maximum number of generation * number of runs*. Statistical metrics of minimum, maximum, average and standard deviation were used to judge the prototype's performance. The prototype's success is measured by tracking the trend of whether more or fewer antivirus vendors can detect the APK files between consecutive runs. The antivirus vendors were provided by the web service Jotti. During development, a range of different runs and generations were tested to get statistical significance, as this is a stochastic system. In addition to the stochastic nature of an SGA, antivirus vendors are continually being updated, which can change their response to a specific malware sample, and there are no guarantees which antivirus vendors will be available when connected to Jotti. The issues related to Jotti are covered in the related Section 2.3.5. Testing was done over a few months, giving a more realistic view of how the prototype is performing as antivirus systems change over time.

A completely random version of the prototype was also built, which applied mutations with the probability of 1.0 and randomly selected an individual at the end of each run. This benchmarked the SGA against an entirely random, uncontrolled search. Testing was performed as an iterative process. The prototype was sensitive to the following parameters:

- **Population size:** With larger populations, the probability of generating good initial individuals increases. The larger the population, the slower the runtime of the prototype. This was set around 3-10 individuals in the population.
- **Mutation:** This is the main operation to search the solution space, and the secondary operation was crossover. Mutation was set high (0.5-0.8), and crossover was eventually disregarded as it led to a higher level of invalid individuals (i.e., created invalid commands for Obfuscapk). The high level of mutation allows each run to explore many possibilities from the starting population.
- **Number of Generations:** This parameter affected how often the mutation was applied during each run. The higher the number, the slower the runtime. There was a trade-off between having more runs with fewer generations or vice versa. Maximum generations was set between 3 and 8.

An overview of the SGA can be seen in the list below; the following sections expand on these points in more detail.

- **Representation:** List of strings [str, ..., str] = one individual.
- **GA architecture:** Simple GA, tournament selection is used to select the offspring, the operators are applied, then the offspring replace the current generation.

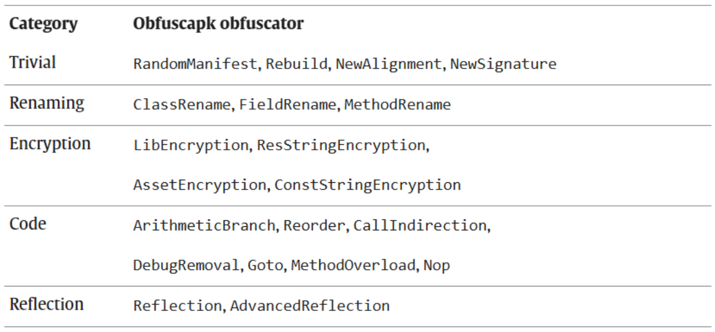
- **Crossover:** None.
- **Mutation:** Code injection, randomisation, renaming, encryption via Obfuscapk.
- **Selection:** One tournament with k individuals.
- **Fitness function:** File data matching via similarity metric and antivirus detection via Jotti.

2.3.1 Generation

In this dissertation, the SGA will terminate based on the maximum number of generations. The SGA runs for a set number of generations, and after each run, another fitness value is calculated using Jotti. The amount of modifications applied by each mutation is fixed. Therefore, the relative number of modifications applied to an APK is greater if the file is smaller, for modifications like code injection. Adjusting the number of generations relative to the file size means that larger samples will get more obfuscation applied to them.

An example of how the SGA explores possible solutions is as follows. The starting size is two, with five generations per run and three runs. Each run would have an upper bound of $1.6 * 10^6$ possible permutations of modifications to explore. There are two starting options for modifications taken when the population is initialised, and up to five additional modifications taken via mutations in each run. At the end of each run, the best individual is entered into the next run as the new phenotype, and the genotype is reset, so that the same number of permutations can be explored again.

There is no cloning of parents in the new generations; offspring make up the new generation entirely. Tournament selection helps reduce loss of diversity, as even low-fitness individuals may still be selected as parents and have a chance to be modified in the next generation to explore their unseen potential. There is a trade-off when considering how many runs and generations to use. Small populations have a quicker runtime, but provide lower exploration. While large populations provide higher exploration but much slower runtime, there is a trade-off between having lots of short runs or fewer



Category	Obfuscapk obfuscator
Trivial	`RandomManifest`, `Rebuild`, `NewAlignment`, `NewSignature`
Renaming	`ClassRename`, `FieldRename`, `MethodRename`
Encryption	`LibEncryption`, `ResStringEncryption`, `AssetEncryption`, `ConstStringEncryption`
Code	`ArithmeticBranch`, `Reorder`, `CallIndirection`, `DebugRemoval`, `Goto`, `MethodOverload`, `Nop`
Reflection	`Reflection`, `AdvancedReflection`

Figure 2.14: Obfuscators implemented in Obfuscapk [7].

long runs. After each run, the best individual is saved independently of the GA. Section 3.2.3 shows experimentation with different selection processes after each SGA run.

2.3.2 Representation

Each genotype is a list of strings. The strings are different commands for the Obfuscapk package (see Figure 2.14), and the commands are selected without replacement. An example individual could be *[MethodRename, Nop, RandomManifest]*. *MethodRename* renames methods in the APK. *Nop*, injects a NOP instruction. *RandomManifest*, will randomise the manifest file. Each genotype also has the following Obfuscapk commands automatically added to the end *[Rebuild, NewAlignment, NewSignature]*. *Rebuild* reapplies APKtool to ensure the signature of the DEX file has changed. *NewAlignment* ensures all uncompressed files in the APK are aligned relative to the start of the file. *NewSignature* re-signs the APK, a correct file signature is required for installing on Android OS. These three commands are automatically added to an individual because they are all standard functions that would be applied after modifications to an APK are made. Their application ensures higher stability and enables the APK to be installed and run. The phenotype of the individual is the APK.

When applying Obfuscapk to an APK, there is no guarantee that the APK will remain a functionally valid application. The Obfuscapk paper [7] showed that after applying Obfuscapk, 47 out of 1000 APKs were broken. Therefore, if Obfuscapk broke an APK, the fitness value of that individual was set to the worst possible outcome, effectively removing it from the selection process. Consequently, the final individuals from the

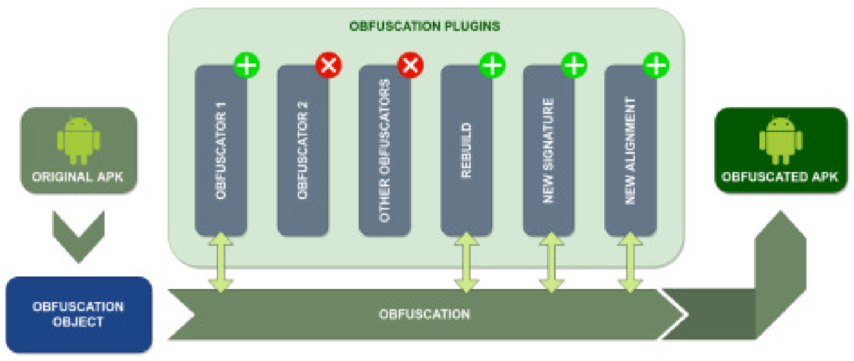


Figure 2.15: Obfuscapk architecture.[7]

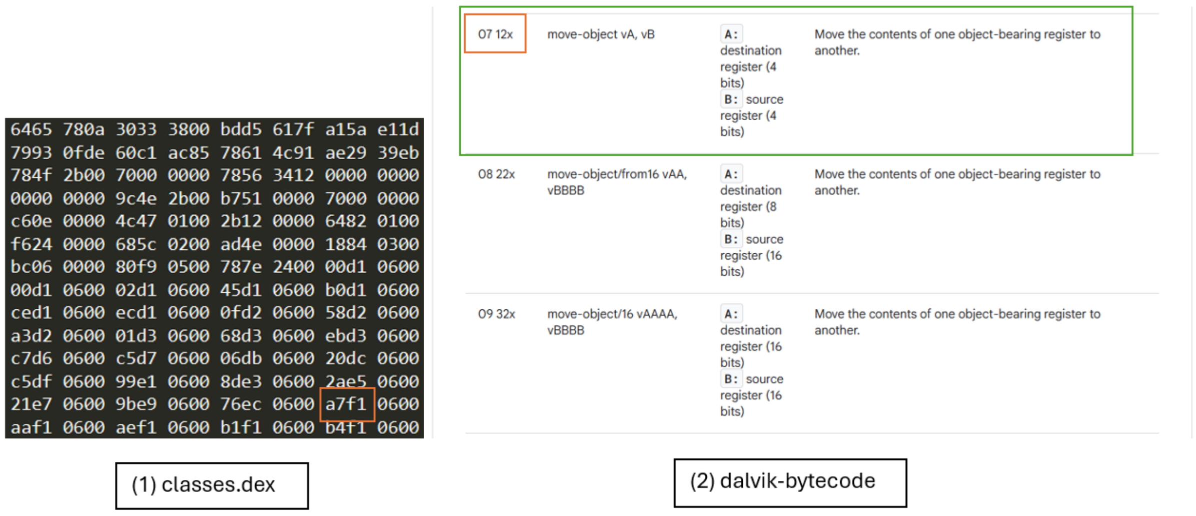
SGA will always result in a correctly functional input for Obfuscapk, so no need for corrective measures. All individuals are initialised to the same size; their length can change by applying mutation, resulting in variable-length final solutions. Once the phenotype individual had been created, its name was changed to record precisely what obfuscation techniques were used in its creation. The individual genotype represents commands sent to Obfuscapk; the resulting APK phenotype is then tested for fitness.

2.3.3 Crossover

Crossover is known to exploit good individuals by preserving parts of their genotype. However, using crossover can create invalid input for Obfuscapk, by creating duplicates of some commands, consequently, corrective measures would be required to use crossover, which would prove to be time consuming and inefficient. In some cases, a high level of mutation can, by chance, correct this issue by removing the offending command. To substitute the effect of crossover, the best possible individual is stored in an independent list after each generation, which is not included in the population selection process. This ensures that any good individual will not be lost due to mutation, preserving good genotypes.

2.3.4 Mutation

Mutation is performed on the genotype by selecting one of three functions: adding a new unique command to the individual, removing a command, or reordering the current commands the individual represents. The mutations are applied to the phe-



```
6465 780a 3033 3800 bdd5 617f a15a e11d
7993 0fde 60c1 ac85 7861 4c91 ae29 39eb
784f 2b00 7000 0000 7856 3412 0000 0000
0000 0000 9c4e 2b00 b751 0000 7000 0000
c60e 0000 4c47 0100 2b12 0000 6482 0100
f624 0000 685c 0200 ad4e 0000 1884 0300
bc06 0000 80f9 0500 787e 2400 00d1 0600
00d1 0600 02d1 0600 45d1 0600 b0d1 0600
ced1 0600 ecd1 0600 0fd2 0600 58d2 0600
a3d2 0600 01d3 0600 68d3 0600 ebd3 0600
c7d6 0600 c5d7 0600 06db 0600 20dc 0600
c5df 0600 99e1 0600 8de3 0600 2ae5 0600
21e7 0600 9be9 0600 76ec 0600 a7f1 0600
aaf1 0600 aef1 0600 b1f1 0600 b4f1 0600
```

07 12x	move-object vA, vB	A: destination register (4 bits) B: source register (4 bits)	Move the contents of one object-bearing register to another.
08 22x	move-object/from16 vAA, vBBBB	A: destination register (8 bits) B: source register (16 bits)	Move the contents of one object-bearing register to another.
09 32x	move-object/16 vAAAA, vBBBB	A: destination register (16 bits) B: source register (16 bits)	Move the contents of one object-bearing register to another.

Figure 2.16: Illustration of byte code inspection. (1) Shows encoded source file. (2) Shows encoded values represent elements in a lookup table, each element in the table translates into a machine instruction.

notype by Obfuscapk (see Figure 2.15). Obfuscapk creates an object that represents program features, like how many registers a function can access. Obfuscapk unpacks the DEX files to be converted to/from Dalvik bytecode via smali/baksmali, which allows the application of obfuscation to the source code. Unpacking is performed by APKtool, Figure 2.16 shows how this is done conceptually. The obfuscation techniques of Obfuscapk are applied with regular expressions, inserting and manipulating new lines of text and AES encryption.

A genotype minimum length of two made Obfuscapk more reliable. Obfuscapk internally tracks if a technique has already been used, and each time the fitness function runs, the internal state of Obfuscapk is reset. After each run, the command list to which the mutation operator has access is reset, this way, a thorough amount of obfuscation is applied to each APK during consecutive runs. In each generation, all APKs are saved for further research, regardless of their fitness.

There are five categories of obfuscation in Obfuscapk that can be applied with mutation: trivial, reflection, encryption, code manipulation and renaming (see Figure 2.14). The mutation in this dissertation excludes the use of the reflection and encryption categories. Reflection is when a Java function is assigned at runtime and can be used as an effective obfuscator. In the Obfuscapk paper, the reflection category was cited as the most common failure when testing, so it was excluded from experimentation. Encryption, the process of converting code into an inaccessible format, was also a source of common failures when applied repeatedly during testing. While it is theoretically possible to re-encrypt encrypted data, Obfuscapk does not have strong support for this. Encryption is also relatively easy to detect, as it produces noisy data, and its excessive use can be an indicator of malware, trying to hide its intentions, ([43], [60]) accordingly, it was excluded from this project. The remaining three categories have fourteen techniques available for application via mutation. From the trivial category, *Rebuild*, *NewAlignment* and *NewSignature*, were always applied, therefore not included in the mutation function. These techniques increase the stability of new individuals. The three obfuscation categories, applied by mutation, and their eleven corresponding techniques can be seen the list below, Algorithm 2 helps explain the functionality of Obfuscapk further:

1. **Trivial:** *[RandomManifest]*, Effective if someone is using the order of the manifest file as a signature.
2. **Renaming:** *[ClassRename, FieldRename, MethodRename]*, Which changes functions and variable names to make code inspection harder.
3. **Code manipulation:** *[ArithmeticBranch, Reorder, CallIndirection, DebugRemoval, Goto, MethodOverload, Nop]*, This wastes compute cycles and changes the running characteristics of the program. It includes manipulating data and control flow, removing debug information, using method overloading and injection of redundant code.

2.3.5 Selection

There are three principal ways to do selection within an SGA:

- **Proportional Selection:** The probability of selecting an individual positively correlates with the individual's fitness. So the fittest individual is most likely to be selected as a parent.

Algorithm 2 Pseudocode of how obfuscation is performed by Obfuscapk

```
Input sample and obfuscation methods to be applied
Initialise sample for processing
→ Decompile using APKtool and scan for locations to modify code
→ Example: finding the end of a function or creating a list of function names
Modify decompiled code to apply obfuscation, examples:
→ Insert new instruction                         ▷ with text insertion
→ Rename or redirection                          ▷ with text insertion
→ Reorder                                        ▷ with text insertion
→ Apply encryption                               ▷ with javax.crypto
```

- **Tournament Selection:** The population is randomly sampled. Out of the sample group, the fittests are selected as parents.
- **Ranked and Random selection:** All individuals have the same probability of being selected; there is no selection pressure. This is mainly used when picking from individuals with very similar fitness values.

Tournament selection was used in this dissertation; it is a popular selection method in the literature and aims to balance exploitation with exploration with a simple implementation. Elitism, whereby the best individual was copied directly to the next generation without modification, was not used, as SGA is known to exploit the best individuals. At the end of each generation, the best individual was stored in a list independent of the population to have a record of the best individuals at the end of each run. This was valuable as the termination criteria of the SGA is a set number of generations, not a specific fitness outcome. When using a fitness outcome as a termination criterion, keeping the best outcomes in the population with elitism could help steer the population towards a specific solution.

A mono-fitness score is used for each individual. Jotti produces an additional fitness value after the SGA has run and is used to select individual for the next run starting sample. For potential further development, a multi-variable fitness score could be used to combine both of these values. In this case, if proportional selection were experimented with, some additional customisation would be needed, as Jotti produces a minimising fitness score (i.e., better obfuscation means fewer antivirus detections); this would need to be inverted. Otherwise, the fittest, the closest to zero, would actually have the lowest probability of being selected. When using a multi-variable fitness score, the weighting of fitness values can also be explored, whereby the more valuable fitness (i.e., Jotti score) could have a multiplier applied, so that it has a larger impact on the selection process.

Fitness function

The metric used to calculate the fitness value needs to balance the measurement of obfuscation strength with the efficiency of implementation. The more variation between the original APK and the current individual APK's code, will result in a better fitness value. The idea being that modifications made via mutation, will show as a change in the file data of the APK. During consecutive runs of the SGA, the phenotype is used to judge an individual's fitness.

During early development, cdifflib was used as a fitness function, as it is easy to use. But after some experimentation, it became clear that cdifflib has issues mapping how the mutation affects the Jotti score (Figure 2.17). Further testing was required to deduce which algorithm would best suit the fitness function (see Section 3.2.2). The encoding of the APK was also considered during testing, as fundamentally, it is the encoded data being compared by the fitness function, and the encoding changes how the APK data is structured.

Jotti

The SGA looks at the file data of the APK to detect variation between its individuals. After the SGA is finished, the best individuals are uploaded to Jotti to judge the most well-obfuscated APK. Figure 2.18 shows the Jotti GUI. Based on the Jotti score, the best individual is selected for the next run of the SGA. Jotti is a web service that provides reports from various antivirus vendors. Once an APK is uploaded, all available antivirus vendors analyse the file, generating a report, that provides information about the current version of antivirus vendors as well as any information that vendors have inferred from the uploaded APK (see Figure 2.19). Jotti typically has access to up to thirteen antivirus

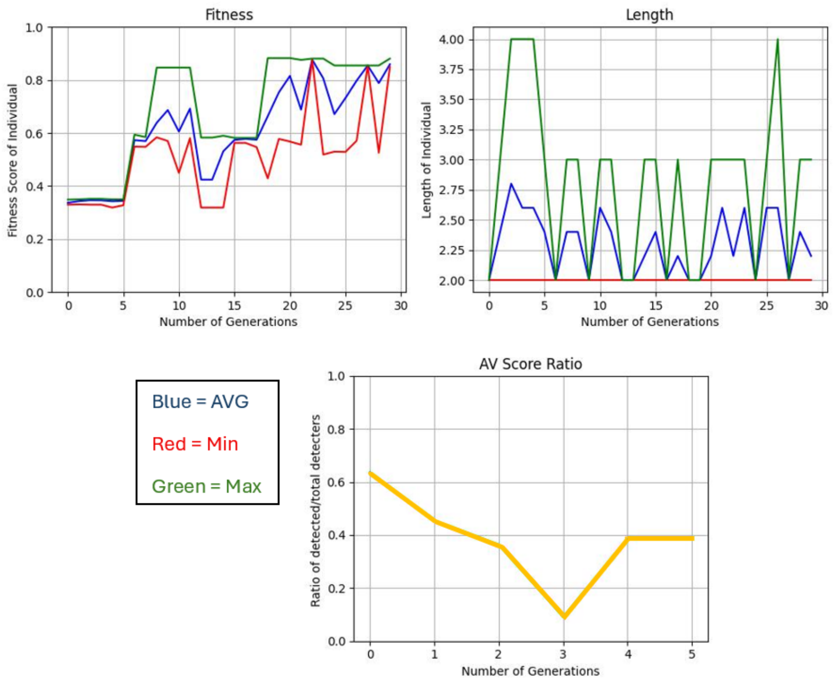
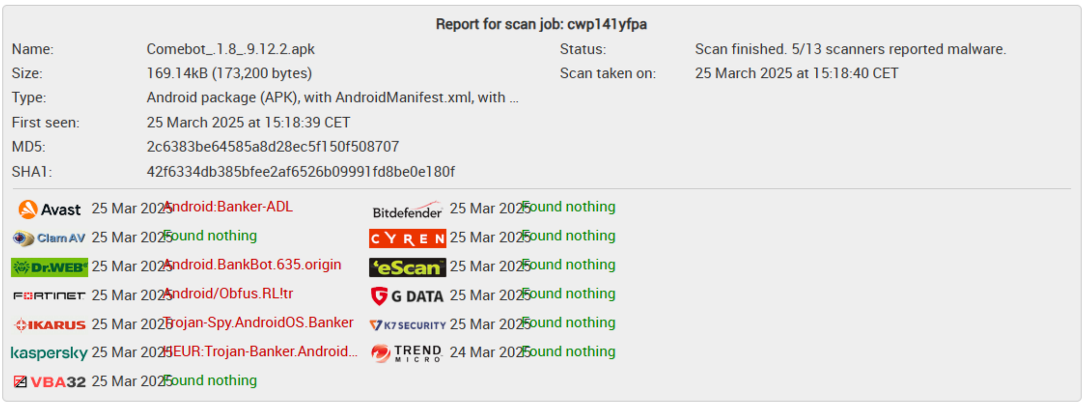


Figure 2.17: Test showing how cdifflib does not map well to the Jotti Score. *Fitness* measured by cdifflib (top left), does not match *AV score ratio* measured by Jotti (bottom). *AV score ratio* starts to go up, while *fitness* stays consistent.

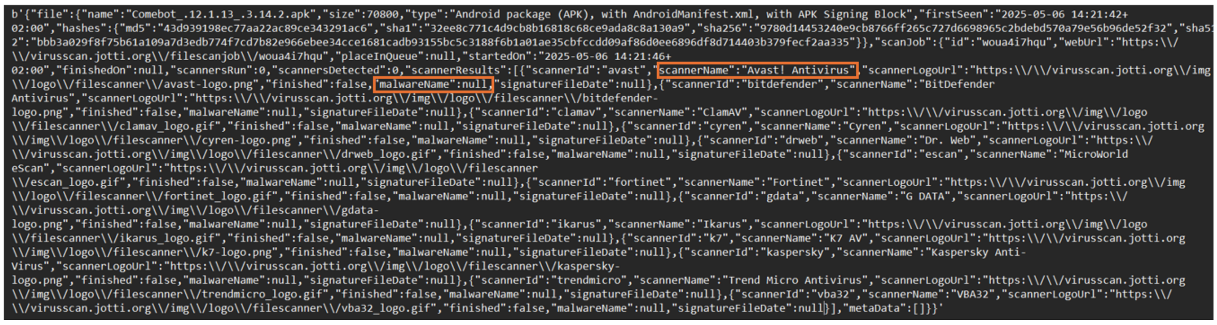


Report for scan job: cwp141yfpa

Name:	Comebot_1.8_9.12.2.apk	Status:	Scan finished. 5/13 scanners reported malware.
Size:	169.14kB (173,200 bytes)	Scan taken on:	25 March 2025 at 15:18:40 CET
Type:	Android package (APK), with AndroidManifest.xml, with ...		
First seen:	25 March 2025 at 15:18:39 CET		
MD5:	2c6383be64585a8d28ec5f150f508707		
SHA1:	42f6334db385bfee2af6526b09991fd8be0e180f		

Avast	25 Mar 2025	Android:Banker-ADL	Bitdefender	25 Mar 2025	Found nothing
ClamAV	25 Mar 2025	Found nothing	CYREN	25 Mar 2025	Found nothing
Dr.WEB	25 Mar 2025	Android.BankBot.635.origin	eScan	25 Mar 2025	Found nothing
FORTINET	25 Mar 2025	Android/Obfus.RL!tr	G DATA	25 Mar 2025	Found nothing
IKARUS	25 Mar 2025	Trojan-Spy.AndroidOS.Banker	K7 SECURITY	25 Mar 2025	Found nothing
kaspersky	25 Mar 2025	HEUR:Trojan-Banker.Android...	TREND MICRO	24 Mar 2025	Found nothing
VBA32	25 Mar 2025	Found nothing			

Figure 2.18: Jotti's Graphical User Interface.



```
b'{"file":{"name":"Comebot_.12.1.13_.3.14.2.apk","size":70800,"type":"Android package (APK), with AndroidManifest.xml, with APK Signing Block","firstSeen":"2025-05-06 14:21:42+
02:00","hashes":{"md5":"43d939198ec77aa22ac89ce343291ac6","sha1":"32ee8c771c4d9cb8b16818c68ce9ada8c8a130a9","sha256":"9780d14453240e9cb8766ff265c727d6698965c2bdebd570a79e56b96de52f32","sha51
2":"bbb3a029f8f75b61a109a7d3edb774f7cd7b82e966ebee34cce1681cadb93155bc5c3188f6b1a01ae35cbfccdd09af86d0ee6896df8d714403b379fecf2aa335"}},"scanJob":{"id":"woua4i7hqu","webUrl":"https:\\/
\\/virusscan.jotti.org\\/filescanjob\\/woua4i7hqu","placeInQueue":null,"startedOn":"2025-05-06 14:21:46+
02:00","finishedOn":null,"scannersRun":0,"scannersDetected":0,"scannerResults":[{"scannerId":"avast","scannerName":"Avast! Antivirus","scannerLogoUrl":"https:\\/\\/virusscan.jotti.org\\/img
\\/logo\\/filescanner\\/avast-logo.png","finished":false,"malwareName":null,"signatureFileDate":null},{"scannerId":"bitdefender","scannerName":"BitDefender
Antivirus","scannerLogoUrl":"https:\\/\\/virusscan.jotti.org\\/img\\/logo\\/filescanner\\/bitdefender-
logo.png","finished":false,"malwareName":null,"signatureFileDate":null},{"scannerId":"clamav","scannerName":"ClamAV","scannerLogoUrl":"https:\\/\\/virusscan.jotti.org\\/img\\/logo
\\/filescanner\\/clamav_logo.gif","finished":false,"malwareName":null,"signatureFileDate":null},{"scannerId":"cyren","scannerName":"Cyren","scannerLogoUrl":"https:\\/\\/virusscan.jotti.org
\\/img\\/logo\\/filescanner\\/cyren-logo.png","finished":false,"malwareName":null,"signatureFileDate":null},{"scannerId":"drweb","scannerName":"Dr. Web","scannerLogoUrl":"https:\\/
\\/virusscan.jotti.org\\/img\\/logo\\/filescanner\\/drweb_logo.gif","finished":false,"malwareName":null,"signatureFileDate":null},{"scannerId":"escan","scannerName":"MicroWorld
eScan","scannerLogoUrl":"https:\\/\\/virusscan.jotti.org\\/img\\/logo\\/filescanner
\\/escan_logo.gif","finished":false,"malwareName":null,"signatureFileDate":null},{"scannerId":"fortinet","scannerName":"Fortinet","scannerLogoUrl":"https:\\/\\/virusscan.jotti.org\\/img
\\/logo\\/filescanner\\/fortinet_logo.gif","finished":false,"malwareName":null,"signatureFileDate":null},{"scannerId":"gdata","scannerName":"G DATA","scannerLogoUrl":"https:\\/
\\/virusscan.jotti.org\\/img\\/logo\\/filescanner\\/gdata-
logo.png","finished":false,"malwareName":null,"signatureFileDate":null},{"scannerId":"ikarus","scannerName":"Ikarus","scannerLogoUrl":"https:\\/\\/virusscan.jotti.org\\/img\\/logo
\\/filescanner\\/ikarus_logo.gif","finished":false,"malwareName":null,"signatureFileDate":null},{"scannerId":"k7","scannerName":"K7 AV","scannerLogoUrl":"https:\\/\\/virusscan.jotti.org
\\/img\\/logo\\/filescanner\\/k7-logo.png","finished":false,"malwareName":null,"signatureFileDate":null},{"scannerId":"kaspersky","scannerName":"Kaspersky Anti-
Virus","scannerLogoUrl":"https:\\/\\/virusscan.jotti.org\\/img\\/logo\\/filescanner\\/kaspersky-
logo.png","finished":false,"malwareName":null,"signatureFileDate":null},{"scannerId":"trendmicro","scannerName":"Trend Micro Antivirus","scannerLogoUrl":"https:\\/\\/virusscan.jotti.org
\\/img\\/logo\\/filescanner\\/trendmicro_logo.gif","finished":false,"malwareName":null,"signatureFileDate":null},{"scannerId":"vba32","scannerName":"VBA32","scannerLogoUrl":"https:\\/
\\/virusscan.jotti.org\\/img\\/logo\\/filescanner\\/vba32_logo.gif","finished":false,"malwareName":null,"signatureFileDate":null}],"metaData":[]}}'
```

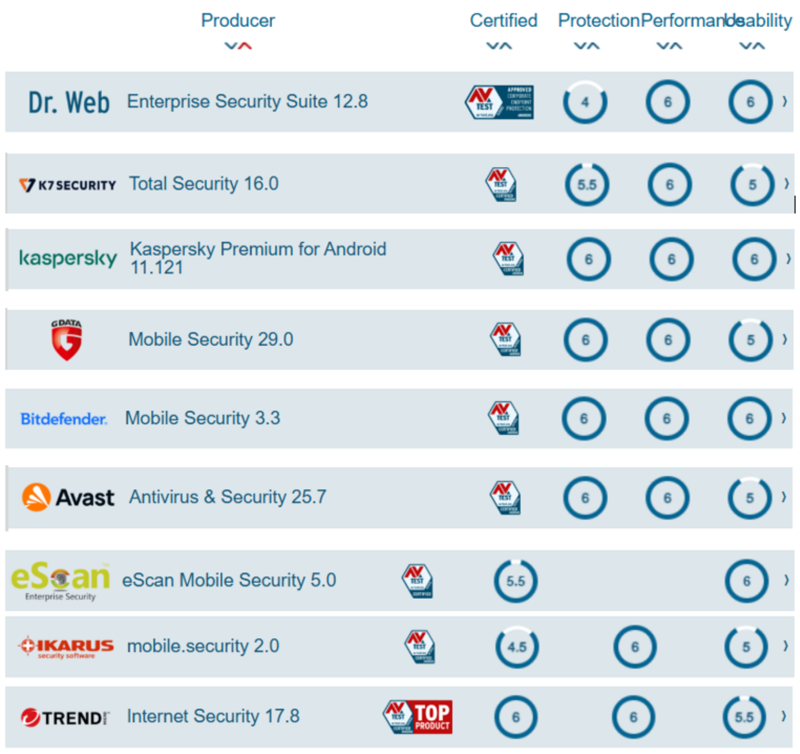
Figure 2.19: Raw Json report from Jotti.

programs. The number available to make a report is determined at the connection time. The vendors supplied by Jotti are Avast, BitDefender, ClamAV, Cyren, Dr. Web, MicroWorld, Fortinet, G DATA, Ikarus, K7 AV, Kaspersky, Trend Micro and VBA32.

Using real antivirus systems provided a real-world benchmark to determine if the SGA is able to search for more evasive malware. The antivirus vendors on Jotti are running on a Linux system, which is ideal as the Android OS is a Linux-based system. Regardless of how many vendors were accessible, when connecting to Jotti, all ratios were calculated using the maximum possible vendors available, thirteen. This makes the fitness value more reliable over time, as the available vendors on Jotti are not always the same. Otherwise, if fewer antivirus vendors are available in a subsequent run, the fitness score will increase, even if the same number of vendors detect the malicious APK.

The Jotti report contains details ranging from "found nothing" to more specific information about how the uploaded file functions, for example *Trojan-Spy.AndroidOS.Banker*. This definition can be broken down into three parts. *AndroidOS*, the operating system of the malware sample. *Trojan-Spy*, the sample behaviour, it pretends to be a different program and then spies on the user. *Banker*, the malware looks for banking details. The details provided are specific to each vendor and are not considered as part of the fitness value. Otherwise, the lack of uniformity will make the fitness value very noisy. To make the fitness score from the Jotti report, the total number of vendors that responded with malware detection is divided by the total number of vendors (i.e., detections/total vendors = fitness value). The fitness score ranges from the worst (1) to the best (0).

AV-Test.org can be used to understand how well the antivirus vendors on Jotti will perform. The exact version of the provided vendors could not always be found; however, AV-Test.org gives a general idea of how well the vendor's software could perform. Figure 2.20 shows the ratings from AV-Test.org. Four antivirus systems have not been graded by AV-Test.org: ClamAV, Cyren, Fortinet, and VBA32. ClamAV is an open-source antivirus system. Cyren and Fortinet are "threat intelligence" and presumably not included as an antivirus by AV-Test.org definition. VBA32 seems to be



Producer		Certified	Protection	Performance	Usability
Dr. Web	Enterprise Security Suite 12.8	AV TEST APPROVED CORPORATE ENDPOINT PROTECTION	4	6	6
K7 SECURITY	Total Security 16.0	AV TEST	5.5	6	5
kaspersky	Kaspersky Premium for Android 11.121	AV TEST	6	6	6
G DATA	Mobile Security 29.0	AV TEST	6	6	5
Bitdefender	Mobile Security 3.3	AV TEST	6	6	6
Avast	Antivirus & Security 25.7	AV TEST	6	6	5
eScan Enterprise Security	eScan Mobile Security 5.0	AV TEST	5.5		6
IKARUS security software	mobile.security 2.0	AV TEST	4.5	6	5
TREND	Internet Security 17.8	AV TEST TOP PRODUCT	6	6	5.5

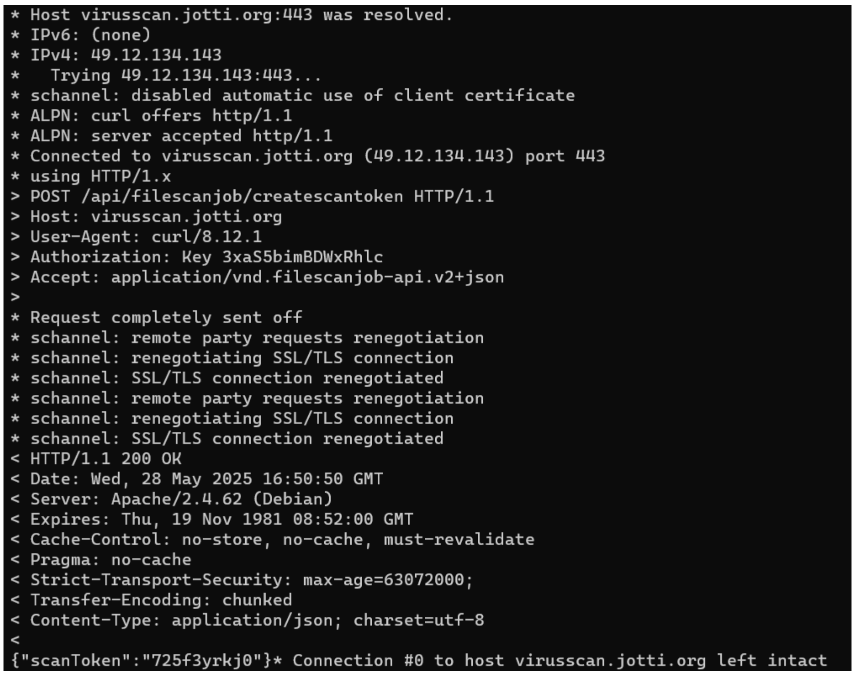
Figure 2.20: Associated score from AV-Test.org for related antivirus vendors available from Jotti. Six is the best score possible.

a typical antivirus system available on VirusTotal.

Jotti API is a premium service that allows free use for research purposes. Typically, the free version of these types of services has a limit on the number of requests within a given time frame. The terms and conditions received from Jotti did not explicitly state a limit of this kind, but as they were generous in providing access, and it plays a crucial role in this project, care has been taken to keep Jotti use to a minimum. By using Jotti in a generator/discriminator type system, access to Jotti has been dramatically reduced. If Jotti were used as the fitness function within the SGA, its use would dramatically increase, because every time a change is made to an individual, that individual needs to be uploaded to Jotti again.

Additionally, two practical considerations exist for not using Jotti inside the SGA. Firstly, the process of running Jotti is much slower than that of the previously mentioned fitness functions, so the overall time complexity of the SGA would increase. Secondly, as Jotti is a web service, on rare occasions, the connection may not be complete due to network outages or server-side issues. In this case, the SGA will have to wait for a response, or a substitute fitness metric will be required, slowing the program further. The generator/discriminator architecture (see Figure 2.3) helped speed up the development stage of this dissertation, as it naturally isolates Jotti and SGA programming code, which makes troubleshooting of both systems much quicker.

Originally, VirusTotal was used instead of Jotti. VirusTotal typically has 65-72 vendors available, while Jotti has typically 11-13, which makes using VirusTotal a better solution for more robust testing of APK samples. Unfortunately, when using the free API of VirusTotal, its reliability was much lower than Jotti's. This reliability is critical when doing multiple runs of the SGA, as it can significantly slow the running of the program, and in the case of VirusTotal, can completely halt the program when no response is found. It seems that on some days when using VirusTotal, the program can be left waiting an indeterminable amount of time for a response. Consequently Jotti was used instead, and to avoid issues with Windows Defender checks, client URL (CURL) was used to access the web services (see Figure 2.21). Using CURL makes access to web services much easier to debug as it provides a complete description of what is happening with the connection, rather than relying solely on debugging features implemented by the service provider.



```
* Host virusscan.jotti.org:443 was resolved.
* IPv6: (none)
* IPv4: 49.12.134.143
*   Trying 49.12.134.143:443...
* schannel: disabled automatic use of client certificate
* ALPN: curl offers http/1.1
* ALPN: server accepted http/1.1
* Connected to virusscan.jotti.org (49.12.134.143) port 443
* using HTTP/1.x
> POST /api/filescanjob/createscantoken HTTP/1.1
> Host: virusscan.jotti.org
> User-Agent: curl/8.12.1
> Authorization: Key 3xaS5bimBDWxRhlc
> Accept: application/vnd.filescanjob-api.v2+json
>
* Request completely sent off
* schannel: remote party requests renegotiation
* schannel: renegotiating SSL/TLS connection
* schannel: SSL/TLS connection renegotiated
* schannel: remote party requests renegotiation
* schannel: renegotiating SSL/TLS connection
* schannel: SSL/TLS connection renegotiated
< HTTP/1.1 200 OK
< Date: Wed, 28 May 2025 16:50:50 GMT
< Server: Apache/2.4.62 (Debian)
< Expires: Thu, 19 Nov 1981 08:52:00 GMT
< Cache-Control: no-store, no-cache, must-revalidate
< Pragma: no-cache
< Strict-Transport-Security: max-age=63072000;
< Transfer-Encoding: chunked
< Content-Type: application/json; charset=utf-8
<
{"scanToken":"725f3yrkj0"}* Connection #0 to host virusscan.jotti.org left intact
```

Figure 2.21: In depth details of URL response provided by CURL.

CHAPTER 3

Experimentation with Performance and Obfuscation

3.1 Introduction

This chapter details four experiments to better understand the effectiveness of the simple genetic algorithm (SGA) prototype in terms of performance and obfuscation. Each section details how the experiments were performed, displays associated graphs, and analyses their experimental results.

For all experiments, the samples were based on *comebot.apk*. It was found from experimentation that the algorithm starts to reach its transformation limit at around six runs as the best fitness values generated were around 0.011. This value cannot get much lower for a minimising fitness function as it is close to zero. In this case, zero represents a totally different APK. It was seen from experimentation that the range of transformation generated by the SGA is, approximately a Jotti score of 0.1 to 0.7. The lowest a Jotti score can be is 0 (no vendor detections). Sections 3.2.3 and 3.3.1 contain probabilistic experiments, therefore, they require a sample size to produce statistically significant results. The remaining experiments are deterministic consequently, repeating the experiment will not produce different results.

3.2 Performance of Genetic Algorithm

3.2.1 General Optimisation

The initiation parameters dramatically affected the run-time of the executed prototype. Consequently, increasing the prototype's runtime. Profiling was used for after-development optimisation, determining what optimisation could be performed. Profiling was performed with *cProfile* and visualised with *snakeViz*. Additionally, *timeit* was used for general speed analysis.

Figures 3.1 and 3.2 show the *snakeViz* visualisation of the profile of two different prototype configurations. The package Obfuscapk is clearly the slowest part of the prototype. By accessing the call stack, you can see that the slowest subprocesses are all linked to Obfuscapk. The total runtime was eighteen minutes for three runs with three generations and a population of three. Eighteen minutes is a reasonable time for execution, given the initial parameters at this stage in development. However, profiling adds additional overhead that slows the prototype's execution time. Additionally, the execution time will be affected by other active processing being managed by the operating system simultaneously.

Over 50% of the prototype's execution time was spent on Obfuscapk, accordingly, it makes sense to optimise Obfuscapk before any micro-optimisation is performed. As Obfuscapk performs a substantial amount of computation in the prototype, the first logical attempt at speeding up the prototype would be to run the prototype with a different interpreter. Python is a language specification, meaning different interpreters can be written to process Python code, each with unique design choices. The typical Python interpreter is built using the C language, *cPython* (see Figure 3.4) [38], but several other implementations exist.

Two popular specialised interpreters, *PyPy* and *Numba*, were inspected to see if they positively affect the prototype's runtime. *PyPy* is written in its own language called the *RPython* language [68]. *PyPy* does not fully support the complete ecosystem of Python packages and is intended to benefit large, complex Python prototypes. Figure

3.3 show that *PyPy* is slower than the standard Python interpreter; this seems to be due to the use of *yapsy*, a Python plugin management toolkit. This was determined by following the prototype's execution in the command console. *Numba* was unsuitable for this dissertation as it only speeds up specific types of code, like vectorised functions. Simply gaining runtime speed by switching to different interpreters was not an option.

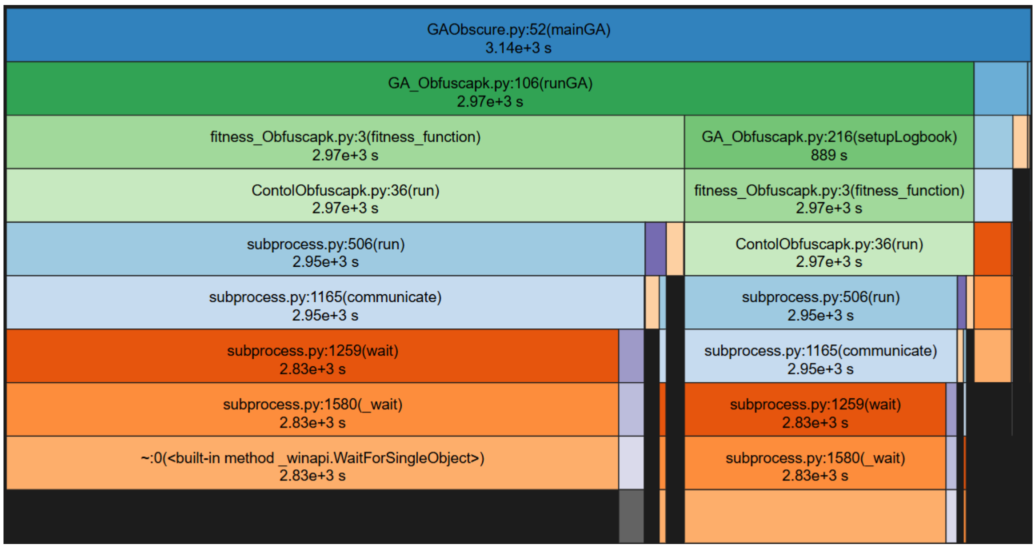


Figure 3.1: Profile of the *SGA* based prototype execution. Each rectangle represents a function in the prototype, the volume of each rectangle correspond to the runtime of the function compared to the overall runtime of the prototype.

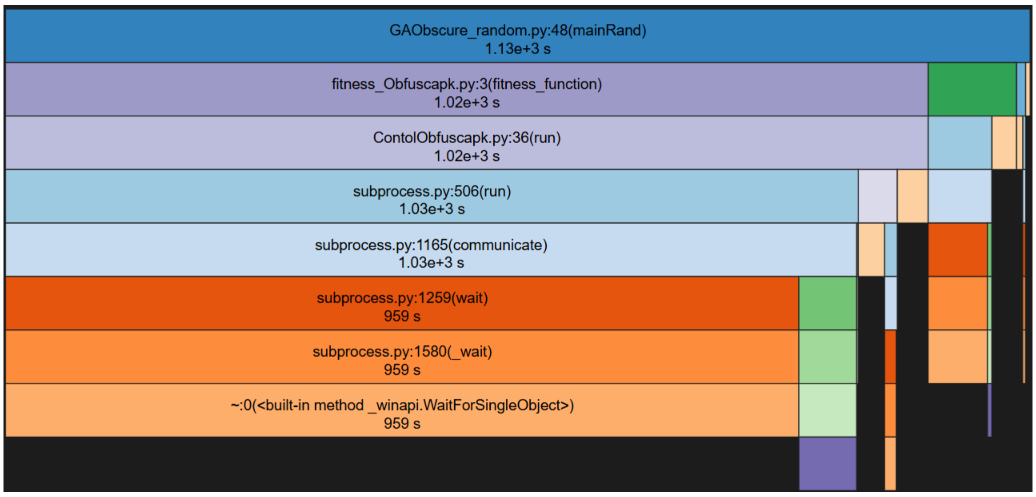
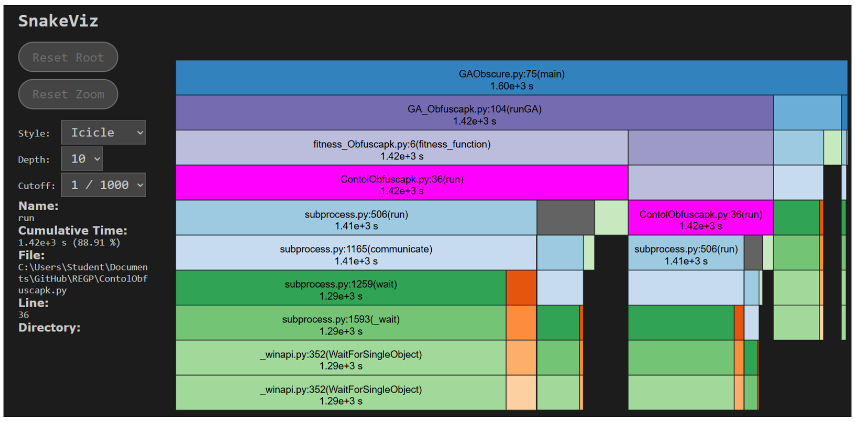


Figure 3.2: Profile of the *Random* based prototype execution. Each rectangle represents a function in the prototype, the volume of each rectangle correspond to the runtime of the function compared to the overall runtime of the prototype.

There are three remaining options to speed up the overall runtime of a prototype. Pre-compiling individual files, parallelisation and multithreading. To make the prototype suitable for such a process, pre-compiling individual files would require rewriting code, which, in the case of Obfuscapk would not be practical, as its code base is considerably large. There are two options for using a drop-in form of multithreading without the need to rewrite the logic of this prototype: using the built-in features of the DEAP package and the built-in multiprocessing package of Python. Both methods failed to work. It would seem that the APKtool, which provides core functionality to Obfuscapk, is incompatible with external threading. APKtool is a Java-based program that already has multithreading, and this does not seem to work correctly once Python implements its own multithreading functions. Multithreading or parallelising this prototype correctly would require substantially rebuilding the prototype, which was deemed not feasible. Consequently, only micro-optimisation could be performed. This was achieved by generating strings as efficiently as possible and making the code as concise as possible. Focus was specifically given to string generation because the SGA generates many different strings for each individual in the population. Figure 3.5 shows the runtime comparisons of using different strings generation methods in Python. To make the code more concise, rethinking the programming logic was required, as well as using list comprehension, where possible.


SnakeViz
Reset Root
Reset Zoom

Style: Icicle
Depth: 10
Cutoff: 1 / 1000
Name: run
Cumulative Time: 1.42e+3 s (88.91 %)
File: C:\Users\Student\Documents\GitHub\REGP\ContolObfuscapk.py
Line: 36
Directory:

GAObscure.py:75(main) 1.60e+3 s
GA_Obfuscapk.py:104(runGA) 1.42e+3 s
fitness_Obfuscapk.py:6(fitness_function) 1.42e+3 s
ContolObfuscapk.py:36(run) 1.42e+3 s
subprocess.py:506(run) 1.41e+3 s
ContolObfuscapk.py:36(run) 1.42e+3 s
subprocess.py:1165(communicate) 1.41e+3 s
subprocess.py:506(run) 1.41e+3 s
subprocess.py:1259(wait) 1.29e+3 s
subprocess.py:1593(_wait) 1.29e+3 s
_winapi.py:352(WaitForSingleObject) 1.29e+3 s
_winapi.py:352(WaitForSingleObject) 1.29e+3 s

Figure 3.3: Three runs of the prototype executed using the *PyPy* interpreter.

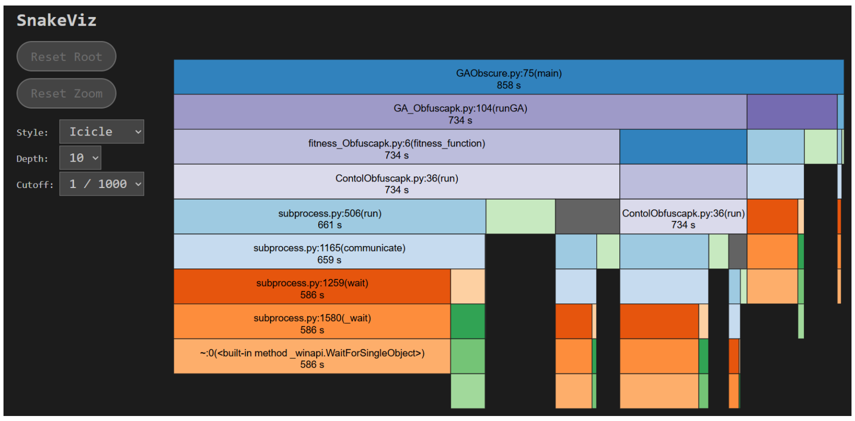
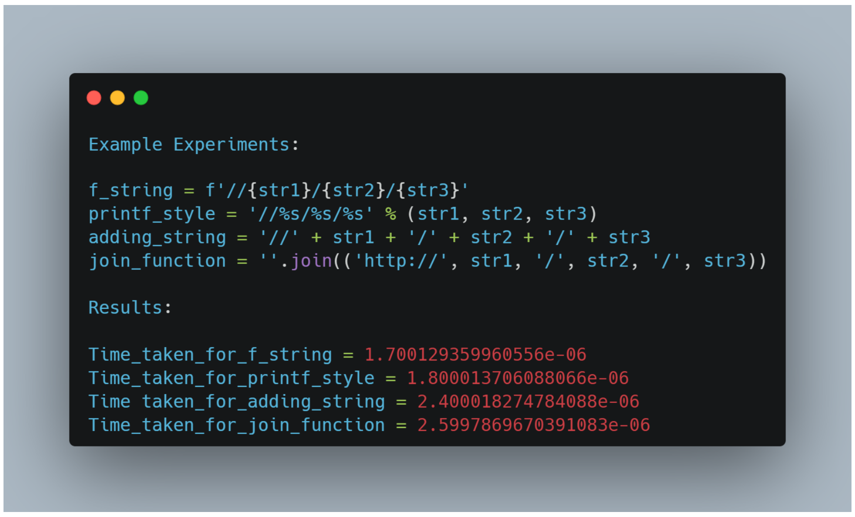


Figure 3.4: Three runs of the prototype executed using the *cPython* interpreter.

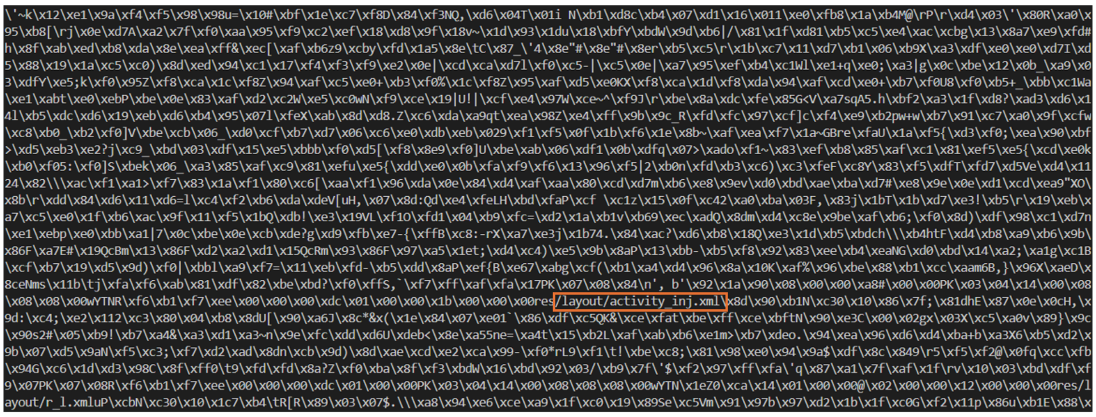


```
Example Experiments:

f_string = f'//{str1}/{str2}/{str3}'
printf_style = '//%s/%s/%s' % (str1, str2, str3)
adding_string = '//' + str1 + '/' + str2 + '/' + str3
join_function = ''.join(('http://', str1, '/', str2, '/', str3))

Results:

Time_taken_for_f_string = 1.700129359960556e-06
Time_taken_for_printf_style = 1.800013706088066e-06
Time taken_for_adding_string = 2.400018274784088e-06
Time_taken_for_join_function = 2.5997869670391083e-06
```

Figure 3.5: Time comparison of four different methods to make a string in Python.

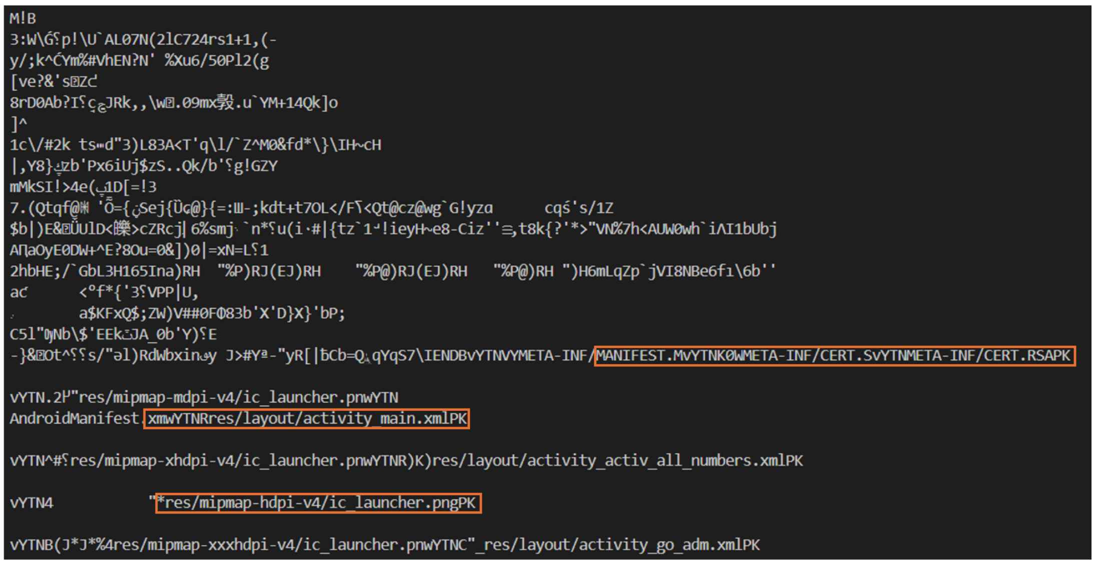
3.2.2 Compare the Effects of Different Fitness Functions and Encodings

This experiment aims to understand the effect, if any, of different algorithms used as the fitness function and how data encoding affects this calculation. The experiment was performed by comparing an APK's fitness value using different encoding and fitness function algorithms. The different encoding and algorithms were compared based on their runtime and the effectiveness of mapping modifications to the APK. The accuracy of the fitness function in mapping the desired solution is important if the SGA is going to be effective in finding the ideal solution. Every time an individual is modified, the fitness function is used, so even a small performance increase will benefit the overall performance of the prototype.



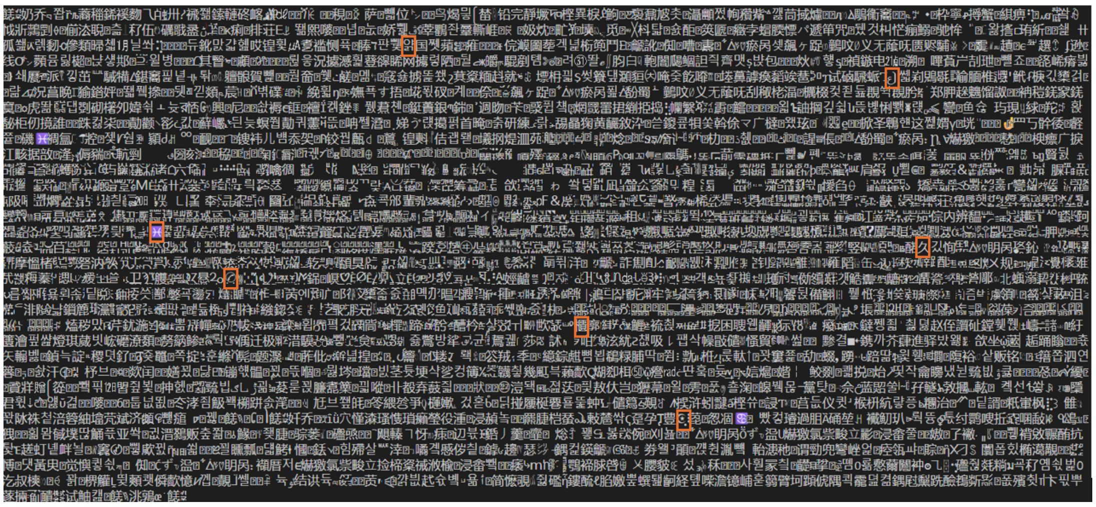
```
\'~k\x12\xe1\x9a\xf4\xf5\x98\x98u=\x10#\xbf\x1e\xc7\xf8D\x84\xf3NQ,\xd6\x04T\x01i N\xb1\xd8c\xb4\x07\xd1\x16\x011\xe0\xfb8\x1a\xb4M@\rP\r\xd4\x03\'\x80R\xa0\x
95\xb8[\rj\x0e\xd7A\xa2\x7f\xf0\xaa\x95\xf9\xc2\xef\x18\xd8\x9f\x18v~\x1d\x93\x1du\x18\xbfY\xbdW\x9d\xb6|/\x81\x1f\xd81\xb5\xc5\xe4\xac\xcbg\x13\x8a7\xe9\xfd#
h\x8f\xab\xed\xb8\xda\x8e\xea\xff&\xec[\xaf\xb6z9\xcby\xfd\x1a5\x8e\tC\x87_\'4\x8e"#\x8e"#\x8er\xb5\xc5\r\x1b\xc7\x11\xd7\xb1\x06\xb9X\xa3\xdf\xe0\xe0\xd7I\xd
5\x88\x19\x1a\xc5\xc0)\x8d\xed\x94\xc1\x17\xf4\xf3\xf9\xe2\x0e|\xcd\xca\xd7l\xf0\xc5-|\xc5\x0e|\xa7\x95\xef\xb4\xc1Wl\xe1+q\xe0;\xa3|g\x0c\xbe\x12\x0b_\xa9\x0
3\xdfY\xe5;k\xf0\x95Z\xf8\xca\x1c\xf8Z\x94\xaf\xc5\xe0+\xb3\xf0%\x1c\xf8Z\x95\xaf\xd5\xe0KX\xf8\xca\x1d\xf8\xda\x94\xaf\xcd\xe0+\xb7\xf0U8\xf0\xb5+_\xbb\xc1Wa
\xe1\xabt\xe0\xebP\xbe\x0e\x83\xaf\xd2\xc2W\xe5\xc0wN\xf9\xce\x19|U!|\xcf\xe4\x97W\xce~^\xf9J\r\xbe\x8a\xdc\xfe\x85G<V\xa7sqA5.h\xbf2\xa3\x1f\xd8?\xad3\xd6\x1
4l\xb5\xdc\xd6\x19\xeb\xd6\xb4\x95\x07l\xfeX\xab\x8d\xd8.Z\xc6\xda\xa9qt\xea\x98Z\xe4\xff\x9b\x9c_R\xfd\xfc\x97\xcf]c\xf4\xe9\xb2pw+w\xb7\x91\xc7\xa0\x9f\xcfw
\xc8\xb0_\xb2\xf0]V\xbe\xcb\x06_\xd0\xcf\xb7\xd7\x06\xc6\xe0\xdb\xeb\x029\xf1\xf5\x0f\x1b\xf6\x1e\x8b~\xaf\xea\xf7\x1a~GBre\xfaU\x1a\xf5{\xd3\xf0;\xea\x90\xbf
>\xd5\xeb3\xe2?j\xc9_\xbd\x03\xdf\x15\xe5\xbbb\xf0\xd5[\xf8\x8e9\xf0]U\xbe\xab\x06\xdf1\x0b\xdfq\x07>\xado\xf1~\x83\xef\xb8\x85\xaf\xc1\x81\xef5\xe5{\xcd\xe0k
\xb0\xf05:\xf0]S\xbek\x06_\xa3\x85\xaf\xc9\x81\xefu\xe5{\xdd\xe0\x0b\xfa\xf9\xf6\x13\x96\xf5|2\xb0n\xfd\xb3\xc6)\xc3\xfeF\xc8Y\x83\xf5\xdfT\xfd7\xd5Ve\xd4\x11
24\x82\\\xac\xf1\xa1>\xf7\x83\x1a\xf1\x80\xc6[\xaa\xf1\x96\xda\x0e\x84\xd4\xaf\xaa\x80\xcd\xd7m\xb6\xe8\x9ev\xd0\xbd\xae\xba\xd7#\xe8\x9e\x0e\xd1\xcd\xea9"XO\
x8b\r\xdd\x84\xd6\x11\xd6=l\xc4\xf2\xb6\xda\xdeV[uH,\x07\x8d:Qd\xe4\xfeLH\xbd\xfaP\xcf \xc1z\x15\x0f\xc42\xa0\xba\x03F,\x83j\x1bT\x1b\xd7\xe3!\xb5\r\x19\xeb\x
a7\xc5\xe0\x1f\xb6\xac\x9f\x11\xf5\x1bQ\xdb!\xe3\x19VL\xf1O\xfd1\x04\xb9\xfc=\xd2\x1a\xb1v\xb69\xec\xadQ\x8dm\xd4\xc8e\x9be\xaf\xb6;\xf0\x8d)\xdf\x98\xc1\xd7n
\xe1\xebp\xe0\xbb\xa1|7\x0c\xbe\x0e\xcb\xde?g\xd9\xfb\xe7-{\xffB\xc8:-rX\xa7\xe3j\x1b74.\x84\xac?\xd6\xb8\x18Q\xe3\x1d\xb5\xbdch\\\xb4htF\xd4\xb8\xa9\xb6\x9b\
x86F\xa7E#\x19QcBm\x13\x86F\xd2\xa2\xd1\x15QcRm\x93\x86F\x97\xa5\x1et;\xd4\xc4)\xe5\x9b\x8aP\x13\xbb-\xb5\xf8\x92\x83\xee\xb4\xeaNG\xd0\xbd\x14\xa2;\xa1g\xc1B
\xcf\xb7\x19\xd5\x9d)\xf0|\xbbl\xa9\xf7=\x11\xeb\xfd-\xb5\xdd\x8aP\xef{B\xe67\xabg\xcf(\xb1\xa4\xd4\x96\x8a\x10K\xaf%\x96\xbe\x88\xb1\xcc\xaam6B,}\x96X\xaeD\x
8ceNms\x11b\tj\xfa\xf6\xab\x81\xdf\x82\xbe\xbd?\xf0\xffS,`\xf7\xff\xaf\xfa\x17PK\x07\x08\x84\n', b'\x92\x1a\x90\x08\x00\x00\xa8#\x00\x00PK\x03\x04\x14\x00\x08
\x08\x08\x00wYTNR\xf6\xb1\xf7\xee\x00\x00\x00\xdc\x01\x00\x00\x1b\x00\x00\x00res/layout/activity_inj.xml\x8d\x90\xb1N\xc30\x10\x86\x7f;\x81dhE\x87\x0e\x0cH,\x
9d:\xc4;\xe2\x112\xc3\x80\x04\xb8\x8dU[\x90\xa6J\x8c*&x(\x1e\x84\x07\xe01`\x86\xdf\xc5QK&\xce\xfat\xbe\xff\xce\xbftN\x90\xe3C\x00\x02gx\x03X\xc5\xa0v\x89}\x9c
\x90s2#\x05\xb9!\xb7\xa4&\xa3\xd1\xa3~n\x9e\xfc\xdd\xd6U\xdeb<\x8e\xa55ne=\xa4t\x15\xb2L\xaf\xab\xb6\xe1m>\xb7\xdeo.\x94\xea\x96\xd6\xd4\xba+b\xa3X6\xb5\xd2\x
9b\x07\xd5\x9aN\xf5\xc3;\xf7\xd2\xad\x8dn\xcb\x9d)\x8d\xae\xcd\xe2\xca\x99-\xf0*rL9\xf1\t!\xbe\xc8;\x81\x98\xe0\x94\x9a$\xdf\x8c\x849\r5\xf5\xf2@\x0fq\xcc\xfb
\x94G\xc6\x1d\xd3\x98C\x8f\xff0\t9\xfd\xfd\x8a?Z\xf0\xba\x8f\xf3\xbdW\x16\xbd\x92\x03/\xb9\x7f\'$\xf2\x97\xff\xfa\'q\x87\xa1\x7f\xaf\x1f\rv\x10\x03\xbd\xdf\xf
9\x07PK\x07\x08R\xf6\xb1\xf7\xee\x00\x00\x00\xdc\x01\x00\x00PK\x03\x04\x14\x00\x08\x00\x08\x08\x08\x00wYTN\x1eZ0\xca\x14\x01\x00\x00@\x02\x00\x00\x12\x00\x00\x00res/l
ayout/r_l.xmluP\xcbN\xc30\x10\x1c7\xb4\tR[R\x89\x03\x07$.\\\xa8\x94\xe6\xce\xa9\x1f\xc0\x19\x89Se\xc5Vm\x91\x97b\x97\xd2\x1b\x1f\xc0G\xf2\x11p\x86u\xb1E\x88\x
```

Figure 3.6: APK file read with the standard encoding.



```
M!B
3:W\Ǵʕp!\U`AL07N(2lC724rs1+1,(-
y/;k^ĆYm%#VhEN?N' %Xu6/50Pl2(g
[ve?&'s□Zc
8rD0Ab?Iʕçج JRk,,\w□.09mx彀.u`YM+14Qk]o
]^
1c\/#2k ts╌d"3)L83A<T'q\l/`Z^M0&fd*\}\IH~cH
|,Y8}ئzb'Px6iUj$zS..Qk/b'ʕg!GZY
mMkSI!>4e(پ1D[=!3
7.(Qtqf@⋇ 'Ō={ؼSej{Ũɕ@}{=:Ш-;kdt+t7OL</F٦<Qt@cz@wg`G!yzɑ      cqś's/1Z
$b|)E&□ŬUlD<皪>cZRcj| 6%smj· `n*ʕu(i·#|{tz`1ᆜ!ieyH~e8-Ciz''≡,t8k{?'*>"VN%7h<AUW0wh`iɅI1bUbj
AՈaOyE0DW+^E?8Ou=0&])0|=xN=Lʕ1
2hbHE;/`GbL3H165Ina)RH  "%P)RJ(EJ)RH    "%P@)RJ(EJ)RH   "%P@)RH ")H6mLqZp`jVI8NBe6fı\6b''
ac      <°f*{'3ʕVPP|U,
ׅ       a$KFxQ$;ZW)V##0FΦ83b'X'D}X}'bP;
C5l"ŋNb\$'EEkـJA_0b'Y)ʕE
-}&□Ot^ʕʕs/"əl)RdWbxinوy J>#Yª-"yR[|ƀCb=Q qYqS7\IENDBvYTNVYMETA-INF/MANIFEST.MvYTNK0WMETA-INF/CERT.SvYTNMETA-INF/CERT.RSAPK

vYTN.2Ⴑ"res/mipmap-mdpi-v4/ic_launcher.pnwYTN
AndroidManifest.xmwYTNRres/layout/activity_main.xmlPK

vYTN^#ʕres/mipmap-xhdpi-v4/ic_launcher.pnwYTNR)K)res/layout/activity_activ_all_numbers.xmlPK

vYTN4          "*res/mipmap-hdpi-v4/ic_launcher.pngPK

vYTNB(J*J*%4res/mipmap-xxxhdpi-v4/ic_launcher.pnwYTNC"_res/layout/activity_go_adm.xmlPK
```

Figure 3.7: APK file read with the UTF-8 encoding.



[illegible]

Figure 3.8: APK file read with the UTF-16 encoding.

In an ideal situation, experimenting with encoding an APK would require unpacking the APK and then encoding each composite file to determine its suitability. As there are other valuable experiments to be performed, in this experiment, the whole APK file encoding will be altered as a whole. Figures 3.6, 3.7 and 3.8 show how the structure of a specific APK changes with different encodings. The fitness function uses this structure to compare two APK's. The Android documentation [32] details how the DEX file is encoded by combining Mnemonic and bit encoding. In the bottom half of Figures 3.6 and 3.7, raw data from the manifest file can be read (see highlighted text in figures), without opening the APK. Information about activities, the pixel density of the layout and a partial reference to a certification used can be seen. In Figure 3.8, the encoding translates the APK to a wide range of different languages and icons, predominantly Kanji/Hanzi characters, originating from China. In all cases the output of the encoding is primarily random to the observer. Once the correct encoding strategy is implemented the stored contents can be observed, as discussed in Section 2.1.3.

The first stage of the experiment was to determine which encoding should be used in the later stage of the experiment. Unigrams were used to compare various encoding types to see how they affect the similarity score of two APK files. When using different encodings, the larger encoding of 32-bit, results in fewer comparisons by the fitness function. At this stage in the experiment, only sensitivity to comparable differences is considered, not the speed of making the comparisons. Ideally, this experiment would include the encoding method the Android system does itself; unfortunately, an off-the-shelf solution to perform this could not be found, for this reason, encoding methods were selected from the standard Python codec [37]. For selecting encoding and results, see Table 3.1. The reasoning for selecting these encodings is as follows. Firstly, ASCII and 8-bit Unicode Transformation Format (UTF) are two of the most common encodings used. 32-bit UTF was selected because the encoding used by Android on DEX files is used to encode 32-bit values [32]. 16-bit UTF was selected to give an intermediary value between the 8-bit and 32-bit encoding. By experimenting with a range of UTF encoding, the size of the encoding blocks can also be compared, as initial experimentation showed

Encoding Name	Languages Represented	Similarity
ASCII	English [37]	1
UTF-8 Unsigned	All languages that have unicode [73]	0.554
UTF-8 Signed	All languages that have unicode [73]	0.554
gb2312	Simplified Chinese [37]	0.511
big5	Traditional Chinese [37]	0.499
UTF-16	All languages that have unicode [73]	0.432
UTF-16 Little-endian	All languages that have unicode [73]	0.432
UTF-16 Big-endian	All languages that have unicode [73]	0.432
UTF-32	All languages that have unicode [73]	0.347

Table 3.1: Results from comparing Comebot.apk and Comebot.1.8.apk using uni-grams and the respective encoding. The lower the similarity, the larger the measured dissimilarity between the two APKs is.

that Chinese style characters were generated, and an encoding for both simplified and traditional Chinese was selected for experimentation.

This experiment (see Table 3.1) showed that: unsigned, signed, big-endian or little-endian encoding had no notable effect on the similarity scores. UTF-8, UTF-16, UTF-32 and standard encoding (which the APK already is in) were selected for further experimentation. This selection was made because their similarity scores are spread between the two limits of possible outcomes (0.347 - 0.554). ASCII was excluded because the result of one shows that this encoding forces the fitness function to fail. A result of one indicates that both APK are the same, while both APKs are different in this test.

The next stage of the experiment involved testing the selected encoding with a range of different fitness functions. Namely, n-grams, cdifflib and Levenshtein distance. A custom and package implementations of n-grams were used, to test which was faster. In each trial, the fitness function was run ten times, therefore the fitness function was run for a total of 100 times. This simulated the maximum runtime of the SGA. The fitness function will be run a maximum of once for each individual in the population. Realistically, each generation of the SGA was smaller than this. The minimum runtime was selected to compare the runtime of each fitness function. If using average instead, the resulting time will include an amalgamation of other processes running on the CPU simultaneously. Using the minimum indicates the ideal speed at which the algorithm runs by itself, with the minimum interference from other processes. This observation is stated in the source code of the *timeit* package used to perform the experiment.

The main experiment was performed by comparing two APKs and getting a similarity score. Each comparison is labelled to indicate which APKs were used. For example, mod_0/mod_2 means *Comebot.apk* compared to *Comebot.0.10.apk*. Table: 3.2 provides more details on the APKs used in the experiment. For results see Figures 3.9, 3.10, 3.11 and 3.12. Levenshtein distance is a maximising fitness functions, while others are minimising. Therefore, when the two APKs compared are identical, Levenshtein distance will produce a value of zero, a perfect match, while minimising functions have a full bar, a value of one. While this can be modified by inverting the results of the function, the methods have been left in their natural form.

APK Name	Representative Name	Obfuscators Applied	Jotti Score
Comebot.apk	mod_0	none	0.69
Comebot.0.10.apk	mod_2	two	0.46
Comebot.8.0.10.13.3.6.apk	mod_6	six	0.46

Table 3.2: APK names and description of applied obfuscation used in this sections experiments. The higher the Jotti score the higher the number of detections.

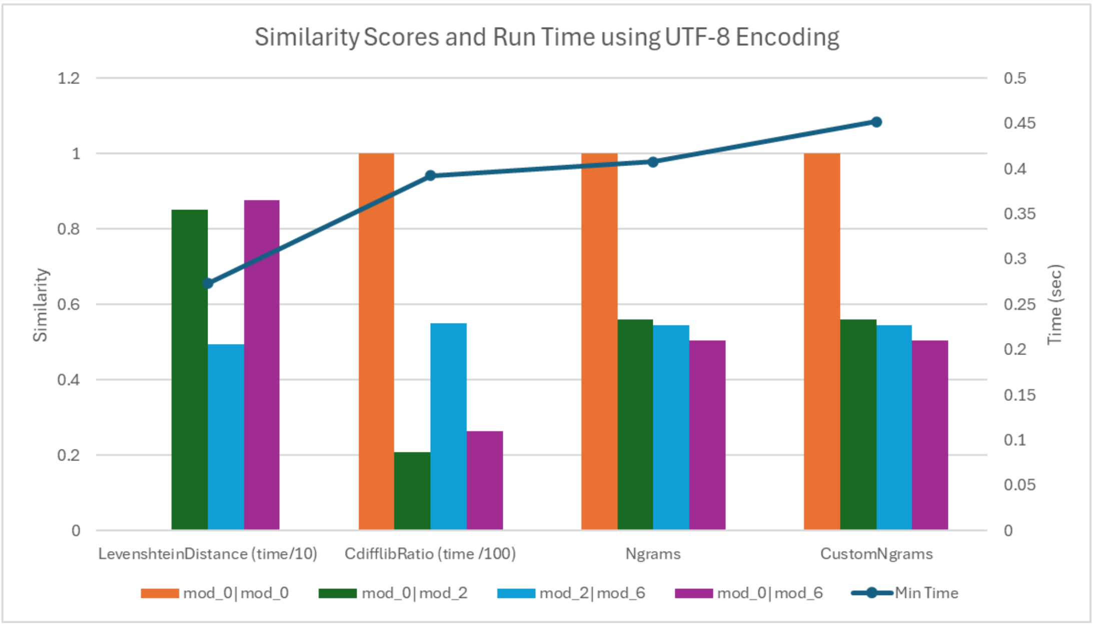


Figure 3.9: Pair-wise tests for all fitness functions using UTF-8 encoding. The relative difference between bar heights shows the relative similarity between APKs. For the maximising function of Levenshtein distance, as the bars get higher the dissimilarity is higher. The remaining functions are minimising, therefore vice versa. Note that time is scaled down for Levenshtein and cdifflib.

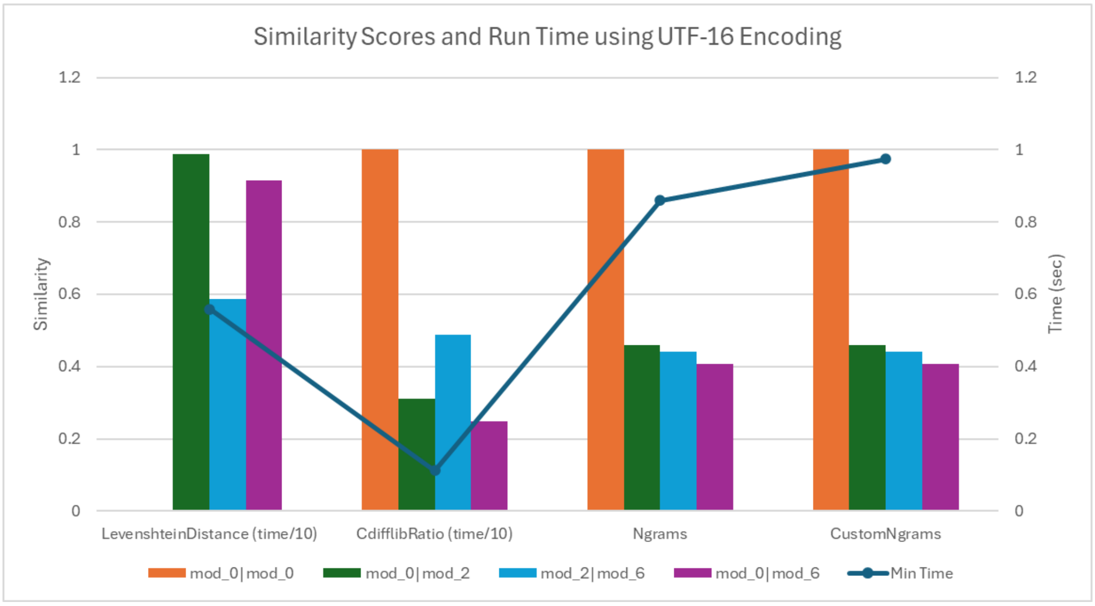


Figure 3.10: Pair-wise tests for all fitness functions using UTF-16 encoding. The relative difference between bar heights shows the relative similarity between APKs. For the maximising function of Levenshtein distance, as the bars get higher the dissimilarity is higher. The remaining functions are minimising, therefore vice versa. Note that time is scaled down for Levenshtein and cdifflib.

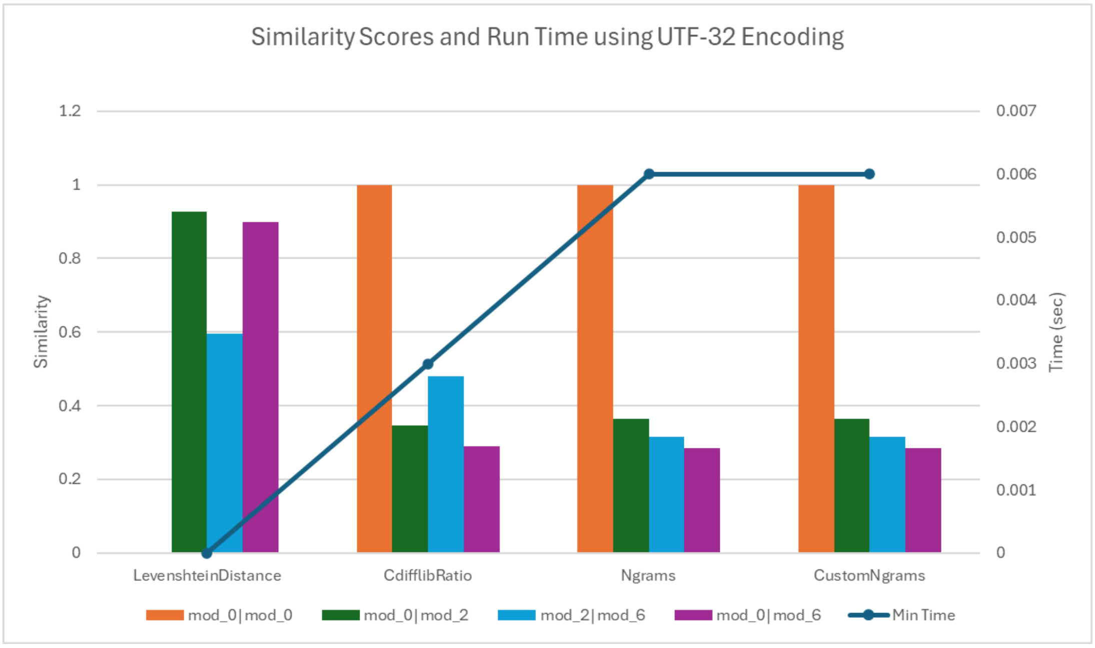


Figure 3.11: Pair-wise tests for all fitness functions using UTF-32 encoding. The relative difference between bar heights shows the relative similarity between APKs. For the maximising function of Levenshtein distance, as the bars get higher the dissimilarity is higher. The remaining functions are minimising, therefore vice versa.

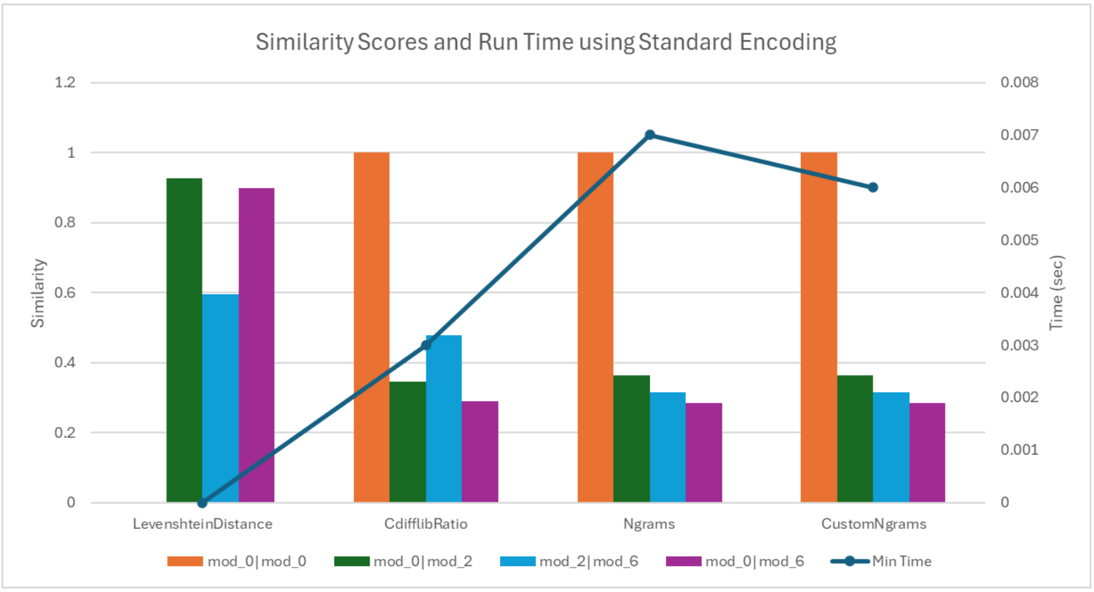


Figure 3.12: Pair-wise tests for all fitness functions using standard encoding. The relative difference between bar heights shows the relative similarity between APKs. For the maximising function of Levenshtein distance, as the bars get higher the dissimilarity is higher. The remaining functions are minimising, therefore vice versa.

Analysis of Results

Algorithm 3 depicts the structure for the experiments run in this section. The selected fitness function should be efficient, able to map to the Jotti score range and detect a difference between similar APKs. Comebot.0.10.apk (i.e., mod_2) and Comebot.8.0.10.13.3.6.apk (i.e., mod_6) have been selected due to their close similarity in terms of obfuscation techniques applied, requiring the fitness function to be sensitive to this difference. An insignificant difference between similar samples means that the metric will produce poor inter-generational fitness, whereby further sample modifications may not be recognised. The ideal outcome is that the selected fitness function shows each pair-wise test getting incrementally more dissimilar.

In both *Levenshtein* distance and *cdifflib*, the similarity score had unstable behaviour with respect to changes in the encoding. In Figures 3.9 and 3.10, the green and purple bars change relative height depending on which encoding was used. Both cdifflib and Levenshtein distance did not map the similarity as expected. In terms of performance, both cdifflib and Levenshtein were the worst affected, as the size of the encoding gets smaller (see Figures 3.9 and 3.10), both of these functions' runtime needed to be scaled by either a factor of 10 or 100 to fit within the same scale as the other results.

n-gram is the best choice as it is performance efficient and also sensitive as expected. To further investigate this function, the standard deviation (SD) of all n-gram results was compared (see Figure 3.13). It can be seen that both standard and UTF-32 encoding have the best performance and sensitivity. In standard and UTF-32, both have the same SD, which is higher than the other encoding methods, meaning the spread of values is the widest. The spread between the different tests is needed to define individuals in the population. As the Jotti score approaches zero, standard and UTF-32 encoding have proportionately less room for representing individuals, compared to UTF-8 or UTF-16. Overall, the performance benefit of using standard encoding makes it the best choice. Standard encoding requires less programming than UTF-32 as it is the format Python reads binary files by default. Other than using the same encoding methods as the Android system, an additional improvement to this experiment could be using a wider range of similarity metrics and encodings in the early stages of the experiment. Therefore, reducing any bias created by looking at a limited range of encoding using n-grams only.

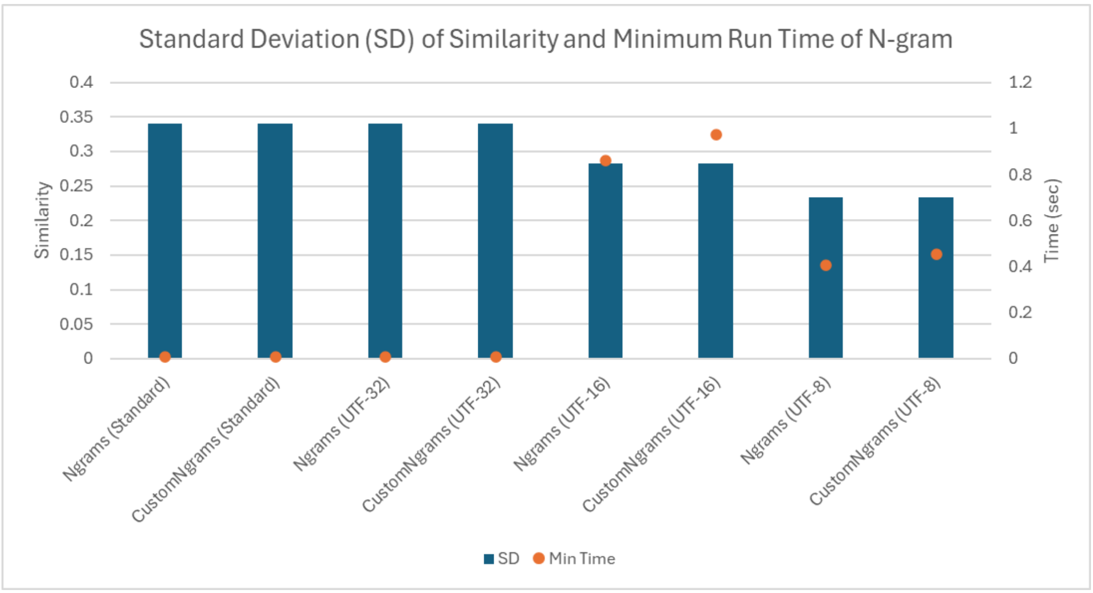


Figure 3.13: Standard deviation of similarity and minimum time performance, for all n-gram experiments. For UTF-32 and standard encoding the minimum time is between 0.06-0.07.

Algorithm 3 Pseudocode of Experiment from Section 3.2.2

```
For each encoding:                                          ▷ Select Encoding
→ compare Comebot.apk to Comebot.1.8.apk
For each Fitness function:                                  ▷ Select Fitness function
→ compare mod_0 to mod_0 and plot line and bar graph
→ compare mod_0 to mod_2 and plot line and bar graph
→ compare mod_2 to mod_6 and plot line and bar graph
→ compare mod_0 to mod_6 and plot line and bar graph
For each implementation of n-grams:                         ▷ Select Fitness function
→ compare and plot
```

3.2.3 Compare Elitism, Passing and Random Sampling in the Main Loop

Parameter Name	Elitism	Passing	Random
Number of trials	10	10	10
Number of runs per trial	4	4	4
Number of generations per run	4	4	4
Number of tndividuals per generation	5	5	5
Mutation rate	80%	80%	80%
Tournament size	3	3	3
Individual size	2	2	2

Table 3.3: Experiment parameters for Section 3.2.3. In this experiment the average of the final Jotti score, histogram, the average fitness value and size of all individuals in each generation were used to analyse results. The average runtime for one trial was 29 minutes.

During a prototype execution, the main loop collects the best individual from the last SGA run and tests each individual with Jotti to see how effective obfuscation has been. There are three ways the main loop can select which individual is passed to start the next SGA run. Either always passing the fittest individual from all runs, passing the best individual from the last run or random selection from the list of the best individuals so far. For this experiment, these three methods will be called elitism, passing and random, respectively.

This experiment looks at how each of these design choices affects the ability of the SGA to find the most effectively obfuscated APK. Each experiment had five trials. There were three experiments in total, firstly looking at the average fitness value of the population (see Figure 3.14), secondly looking at the average size of the individuals (see Figure 3.15) and finally looking at the Jotti score at the end of each run (see Figure 3.16). In each experiment, for each configuration, there were four individuals in the population, with 4 generations * 4 runs * 10 trials, a total 160 generations per test case. Each experiment produces results for fitness and size (maximum, minimum and average) and the Jotti score for each run. Each configuration can be judged on its overall trend by averaging these results.

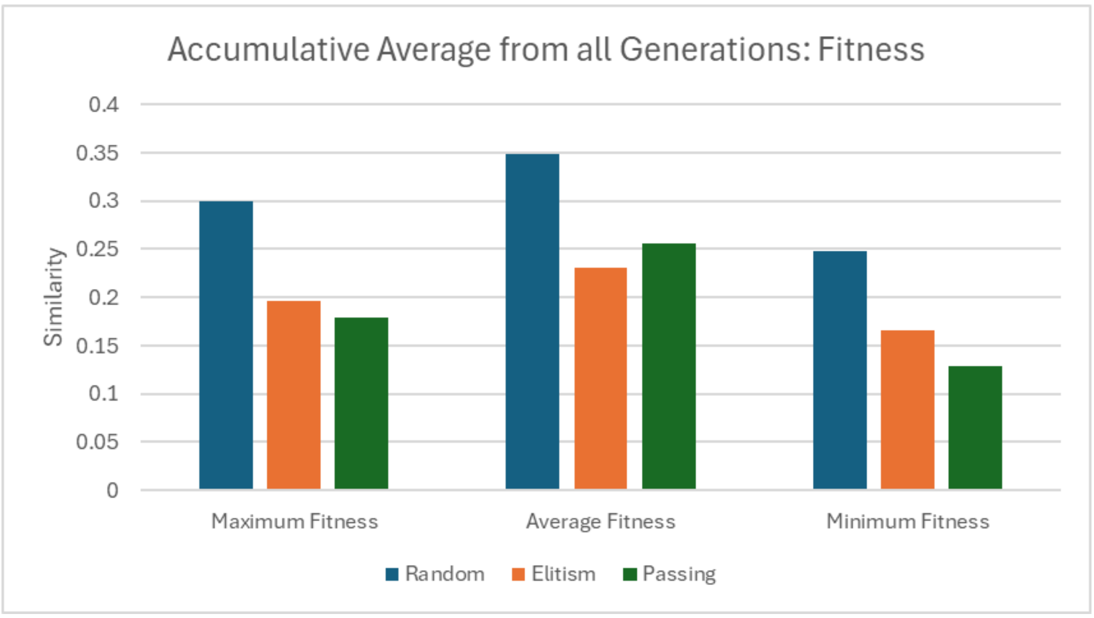


Figure 3.14: Average fitness of all individuals across five trials (i.e., 80 generations) using different selection methods. Each generation has the Max fitness (i.e., the worst) and Min fitness (i.e., the best).

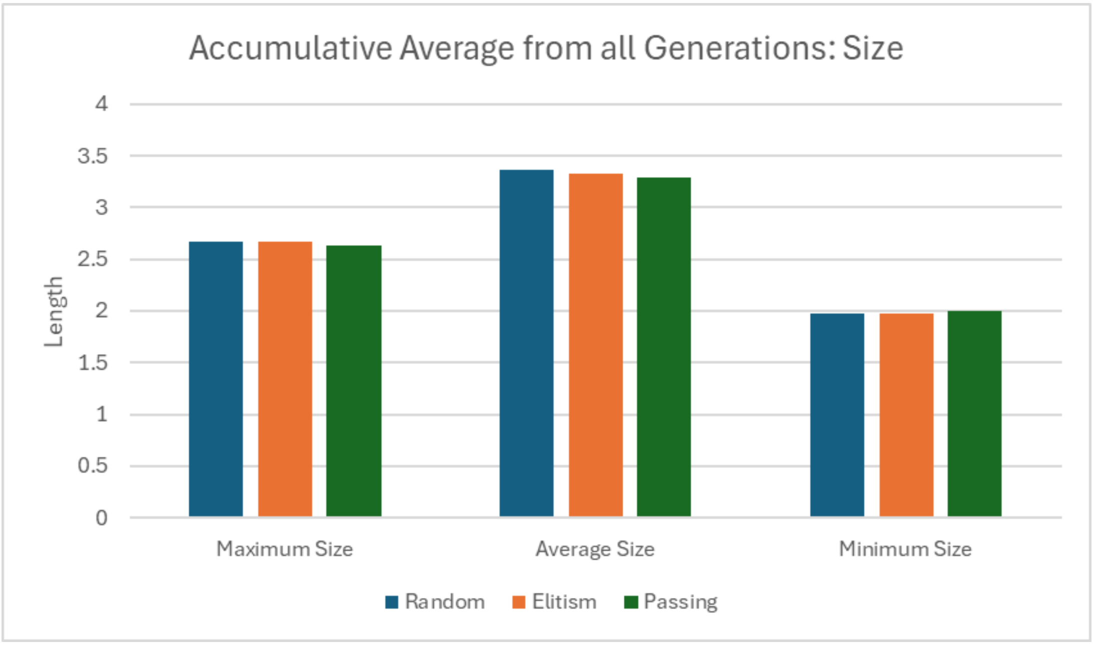


Figure 3.15: Average size of all individuals across five trials (i.e., 80 generations) using different selection methods. Each generation has the Max fitness (i.e., the longest) and Min fitness (i.e., the shortest).

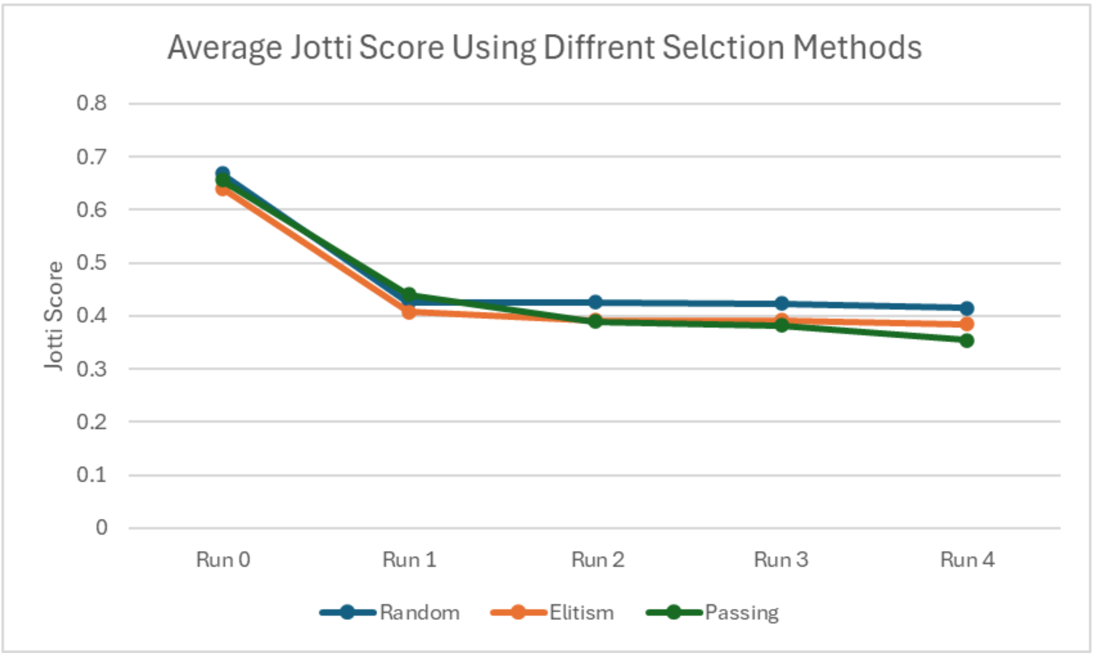


Figure 3.16: Average Jotti score at the end of each run across all five trials (20 runs in total). The lower the Jotti score the less antivirus detections.

Analysis of Results

Algorithm 4 depicts the structure for the experiments run in this section. A minimising fitness function was used therefore, in terms of best results *passing* is the most effective selection method, with random selection being the worst. The average fitness of *passsing* was also the lowest, even though its maximum was in the middle, which implies that the fitness distribution was biased towards lower values, which is ideally what should be explored by the SGA.

To confirm this observation, a count of the fitness being above or below the average was performed for each configuration. The data frequency was counted into two bins, either *more than* $>$ average or *less than* $<$ average. In Figure 3.17, the lighter coloured bars represent the values below the average, and the darker colours above the average. The best selection method would encourage the SGA to produce proportionately smaller fitness values overall. Green represents *passing*, showing that the *less than* bin has more values than the *more than* bin, ergo the population is trending towards lower values. *Passing* is clearly the best selection method. An improvement to this experiment would be to have performed the fitness function experiment first, as this experiment was performed using cdifflib, an ineffective fitness metric, hence the Jotti score flattens out quickly (see Figure 3.16).

3.3 Effectiveness of Obfuscation

3.3.1 Compare the Effects of Different Obfuscation Techniques

There are five categories of obfuscator provided by Obfuscapk namely: trivial, renaming, code, encryption and reflection. In this dissertation, both reflection and encryption have been excluded for performance reasons. In the trivial category, only the *RandomManifest* technique was applied via mutation, with the remaining technique automatically applied. Consequently, this experiment aims to understand the effectiveness of the renaming and code categories obfuscators, with the trivial category applied as usual.

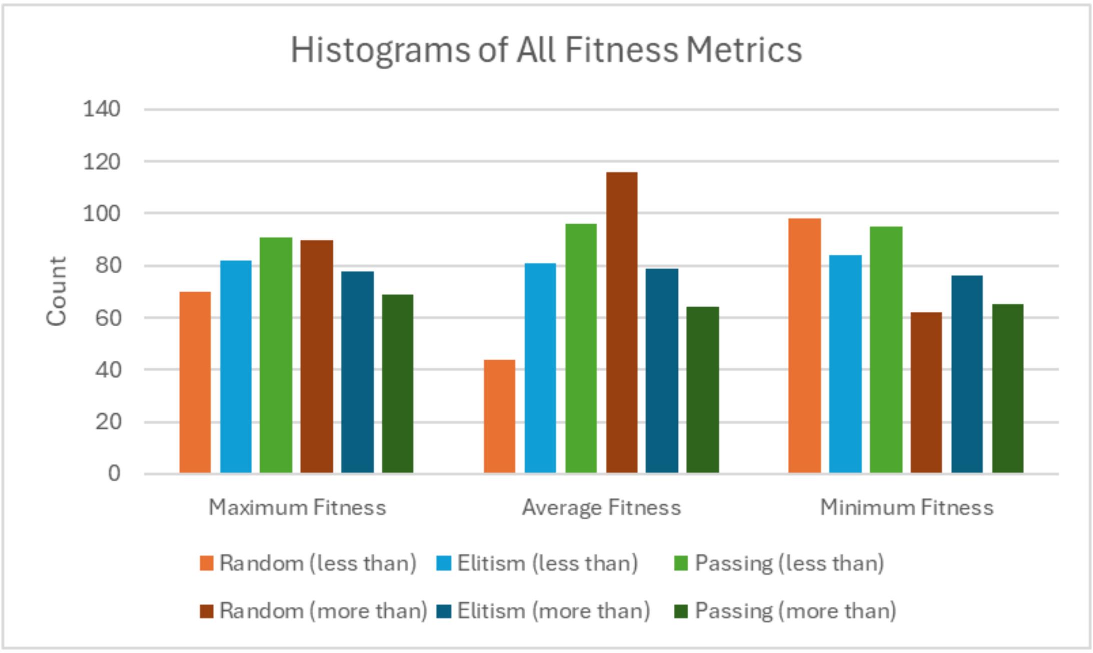


Figure 3.17: Comparisons of histograms for each selection method, from all three fitness metrics. The histograms are created around the respective data's average. *Less than* (the average) on the left-hand side of the respective histograms and *more than* on the right.

The Jotti score was used to determine which obfuscation categories were more impactful. The prototype was executed for each test case five times, with the initiation parameters of seven runs, five individuals in the population and three generations. Hence, with 5 generations * 7 runs * 10 trials, with a total of 210 generations per test case. There are five tests in total; their configuration can be seen in Table 3.4.

The first step was to look at the average Jotti score at the end of each run, in each test configuration, to see which one performed best. Taking the average across all five trials helps reduce the effect of outliers and gives an overall view of results (see Figure 3.18). In this experiment, the lower the score, the better and *tests two, three and five* show the most consistency. To further investigate these three tests, the minimum value of each run was selected to see which tests performed best in the ideal case (see Figure 3.19). *Test three* seems to fluctuate more than the other two, which could be since the obfuscators being selected are more random than the other two tests, or poor service coverage provided by Jotti at that time. To better understand these fluctuations, linear regression could be used to compare slopes calculated from each test. As so few observations per variable are available, a method suitable for small sample sizes was used. Ideally, the

Algorithm 4 Pseudocode of Experiment from Section 3.2.3

```
For each test case:                                          ▷ Create test data
while $trials \leq numberoftrials$ do
    while $runs \leq maxRuns$ do
        Initialise SGA with test case
        while $generation \leq maxGeneration$ do             ▷ Run SGA
            Selection
            Mutation
            Fitness evaluation
        end while
    end while
end while
Get min, max and average individual fitness and size:        ▷ Process test data
$\rightarrow$ Average values and plot bar chart
Get Jotti score from each run:
$\rightarrow$ Average values and plot line graph
Get min, max and average individual fitness:
$\rightarrow$ Do frequency analysis using the average then plot histogram
```

Jotti score would always be monotonically decreasing. To measure the "monotonicity" of each test, consecutive values were subtracted from each other $x[n+1] - x[n]$, producing a negative number if the values are decreasing. These results were counted if $X =< 0$, and the standard deviation of all values was taken. A lower standard deviation and higher total count would show that the test results are more consistent and closer to being negatively monotonic (see Figure 3.20). Using the minimum or average Jotti score for the count of $X =< 0$, made no difference in the distinction of each test's relative performance.

Test Name	Categories Included
Test One	Renaming
Test Two	Code
Test Three	Code and Renaming randomly mixed
Test Four	Code applied first then Renaming
Test Five	Renaming applied first then Code

Table 3.4: Obfuscator categories used in each test group.

Analysis of Results

Algorithm 5 depicts the structure for the experiments run in this section. It is clear from this experiment that *code* category by itself is the most effective obfuscator. It

Parameter Name	Test One	Test Two	Test Three	Test Four	Test Five
Number of trials	10	10	10	10	10
Number of runs per trial	7	7	7	7	7
Number of generations per run	3	3	3	3	3
Number of individuals per generation	5	5	5	5	5
Mutation rate	80%	80%	80%	80%	80%
Tournament size	3	3	3	3	3
Individual size	2	2	2	2	2

Table 3.5: Experiment parameters for Section 3.3.1. In this experiment the average of the final Jotti score after each run, histogram, standard deviation and the average and minimum fitness value of all individuals in each generation were used to analyse results. The average run time for one trial was 84 minutes.

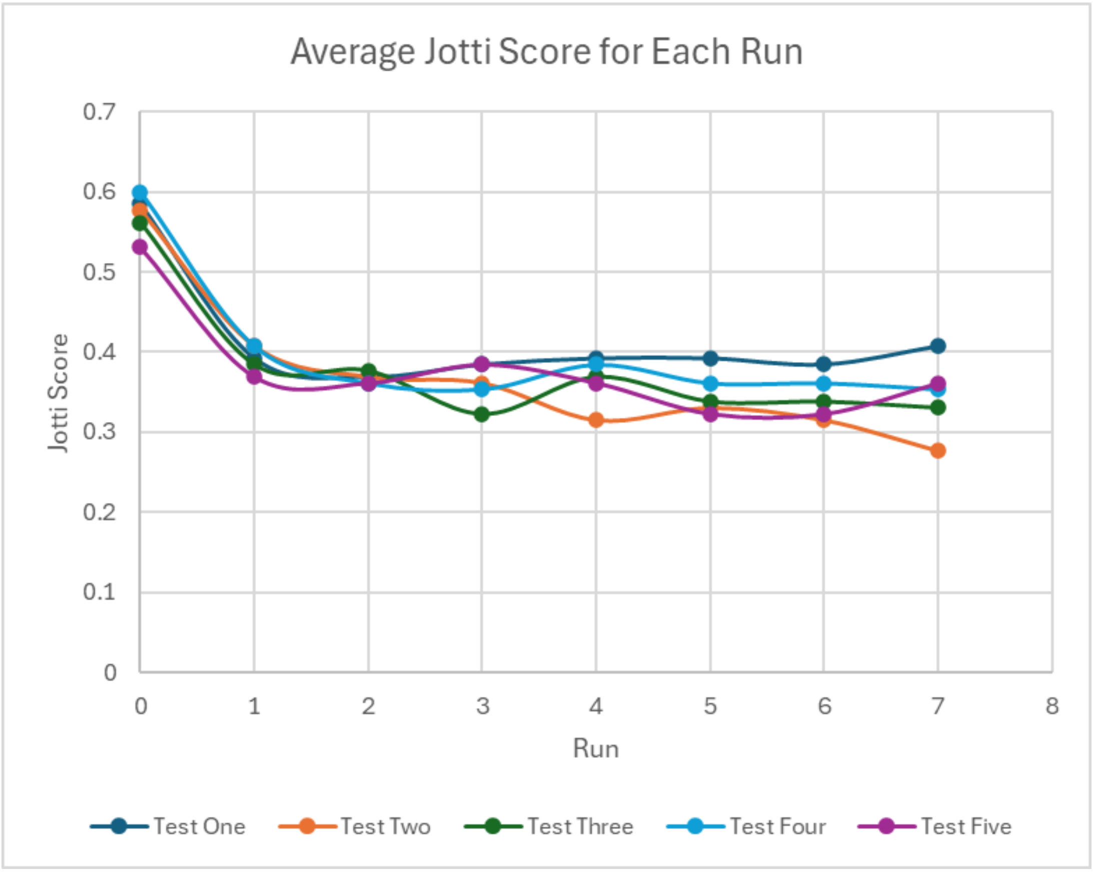


Figure 3.18: The Average Jotti scores of each run, for all tests.

produces the best final results and has the most consistent level of "monotonicity". Figure 3.21 shows a visualisation of the average Jotti score in the first and final run, showing each test configuration's progress towards the ideal outcome of zero, the centre of the hexagon. In this figure, it can clearly be seen that *test five* started in a better position than the other tests, indicating that not all vendors were available at the start of the test. Even with this observation, it can be clearly seen that *test two* had the most

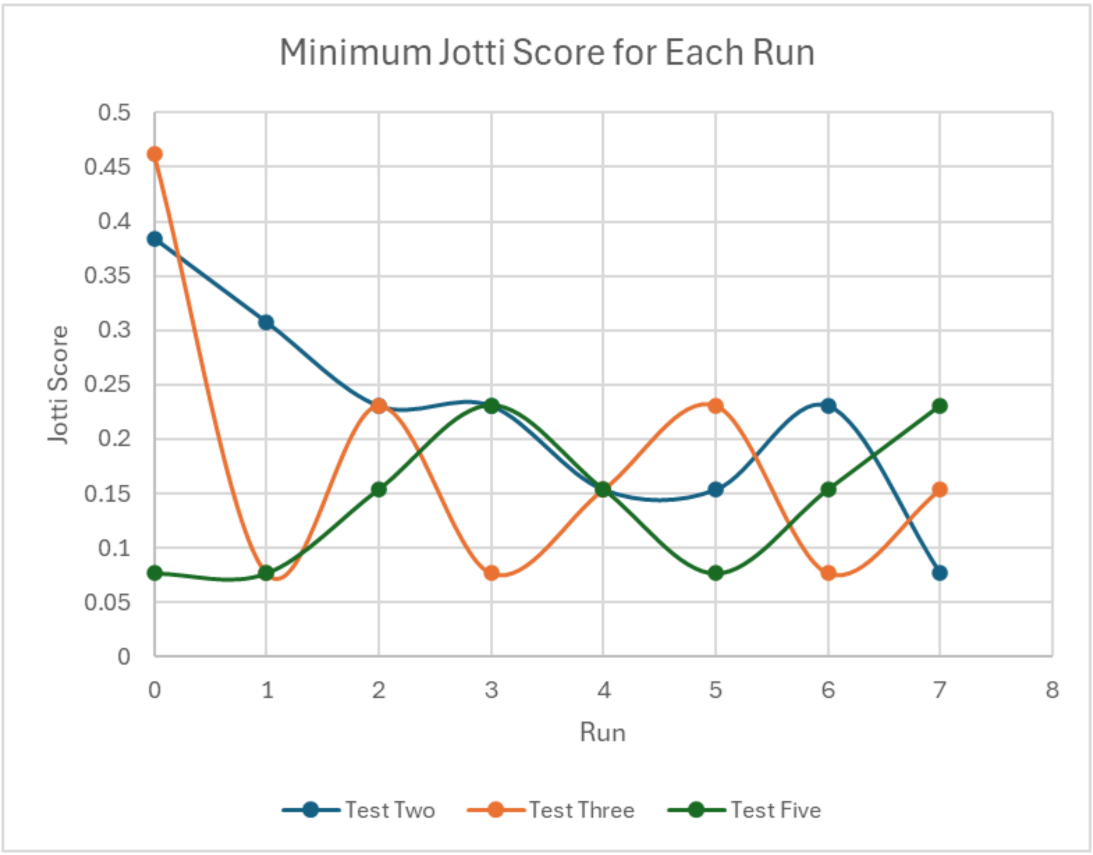


Figure 3.19: The minimum Jotti score of each run from *tests two, three and five*.

significant improvement. The radar graph was selected as it makes a clear comparison between starting and end Jotti values with the symmetry of the hexagon; the use of lines or bars does not get this point across as clearly.

It is important to note that because each test contains different categories, it also contains a different number of obfuscation techniques. *Renaming* has four techniques in total, and *code* has eight. This limits the largest possible size of an individual in the population and the number of variations an individual could have, as techniques cannot be applied twice in the same session due to the design of Obfuscapk. Consequently, this experiment alone does not prove unequivocally whether code modifications are more effective than renaming.

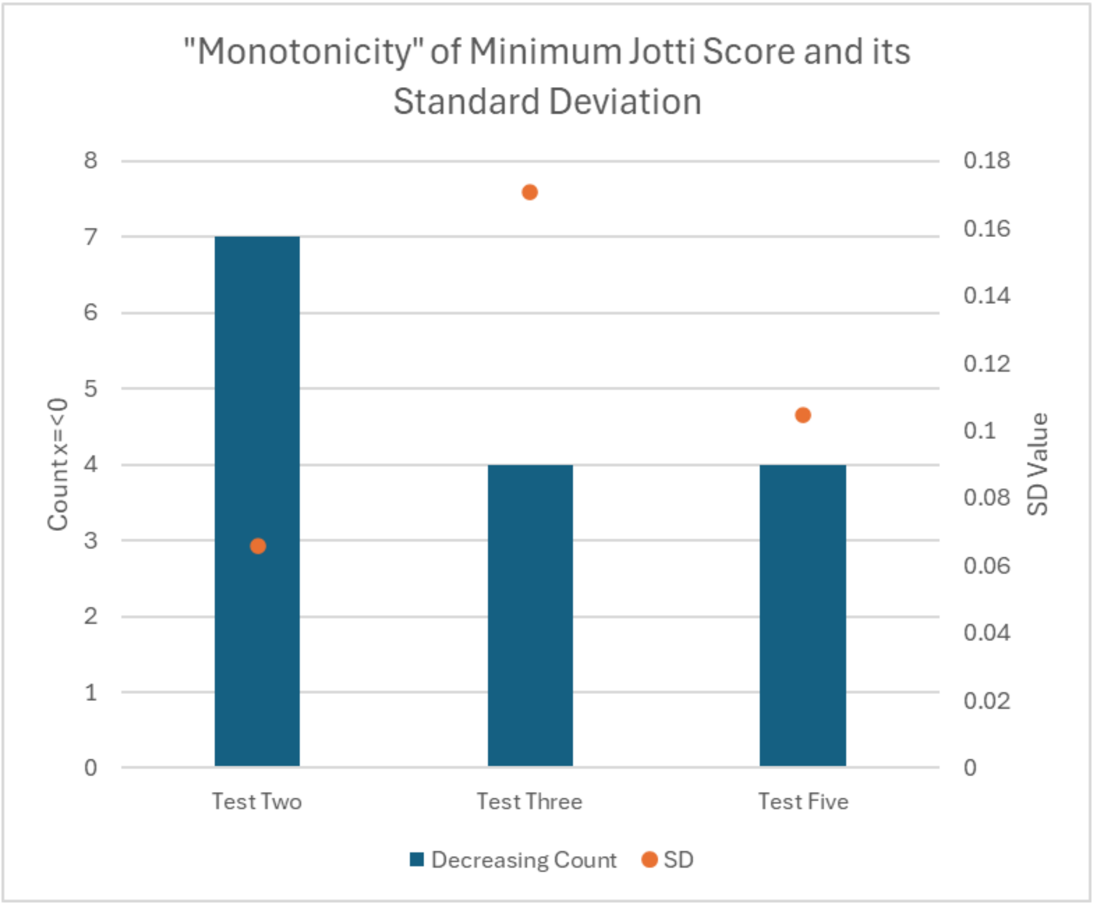


Figure 3.20: Histogram of Jotti score given (x[n+1] - x[n]) =< 0 and also it's standard deviation (SD).

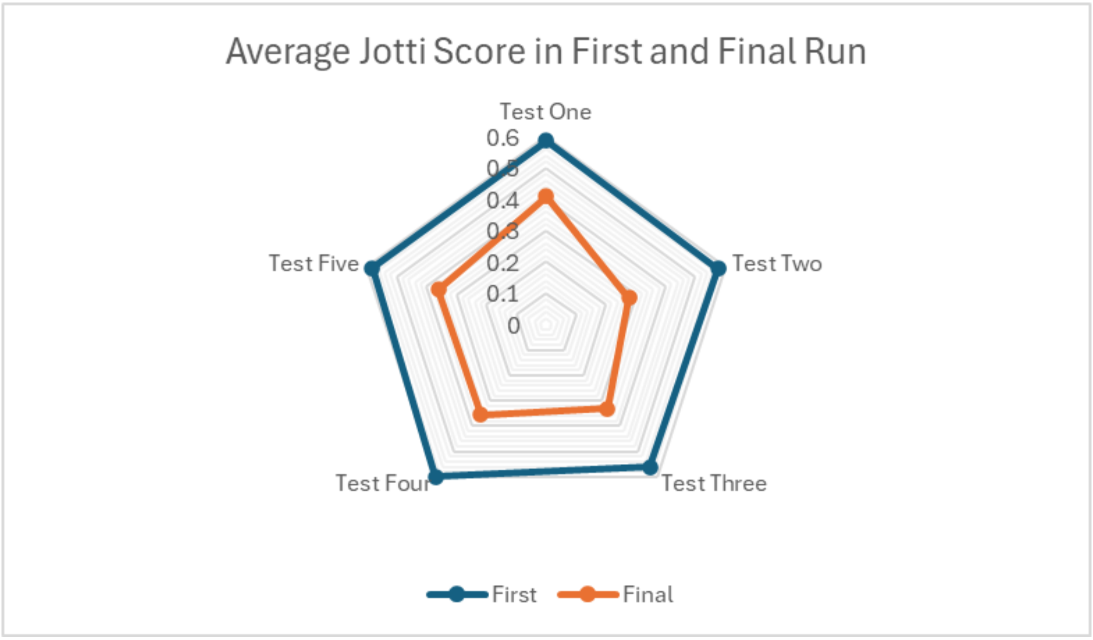


Figure 3.21: Comparisons between the average first and last run of all test groups.

Algorithm 5 Pseudocode of Experiment from Section 3.3.1

```
For each test case:                                          ▷ Create test data
while $trials \leq numberoftrials$ do
    while $runs \leq maxRuns$ do
        Initialise SGA with test case
        while $generation \leq maxGeneration$ do             ▷ Run SGA
            Selection
            Mutation
            Fitness evaluation
        end while
    end while
end while
Get Jotti score from each run:                               ▷ Process test data
$\rightarrow$ Average values from each trial and plot line graph
Get Jotti score from each run:
$\rightarrow$ Select minimum value from each trial and plot line graph
Get Jotti score from each run:
$\rightarrow$ Do frequency analysis using values that sequentially increase then plot histogram with SD
Get Jotti score from first and last run:
$\rightarrow$ Create radar plot
```

3.3.2 Comparison of Previous and Post Population

This experiment aims to understand the difference, if any, between the original population, the population generated by the SGA and a population generated by a random mutation, without selection pressure. For brevity, these test groups are known as Original, SGA and Random respectively. The populations were generated by running each experiment using the following initial parameters: three generations, five individuals in the population and seven runs. To make the comparison between populations, file size, number of unique API calls, percentage of unique external API calls and entropy were used. These metrics give a general sense of how the APK has been transformed. Both entropy and file size indicate the extent of obfuscation from a broad overview. API calls reveal obfuscation specific to function redirection and renaming, as well as a rough estimate of how much of the sample's source code appears to be outside the file system. To create the two test groups, fifty APKs were randomly selected from approximately 250 APKs generated from the SGA, and fifty were randomly selected from approximately 250 APKs generated from the random process. The sampled APKs represented the best APKs at the end of each generation. These APKs had a range of obfuscation applied from heavy to light, depending on whether they were created in an early or later stage of execution. API calls were traced from the main entry point of the APK using *Androguard* [6] as an analysis tool and were collected as if to make a call graphs (CG) and processed as a list of names.

The first step was to see if the dataset was normally distributed as this would indicate which statistical methods could be used. To test for normality the dataset's value range was divided in to bins and a count of values that fell into each bin was made. Sturges's rule [57] was used to select the number of bins for the histogram, $k = 1 + \log_2(n)$, with n = number of values. The interval was calculated by $(max(x) - min(x))/k$. Figures 3.22 and 3.23 confirm the dataset to be non-normally distributed, the data is multi-modal distribution as there is both a major and minor mode. Therefore an appropriate nonparametric test needs to be selected.

Out of the common nonparametric tests, Mann Whitney U and Wilcoxon signed rank

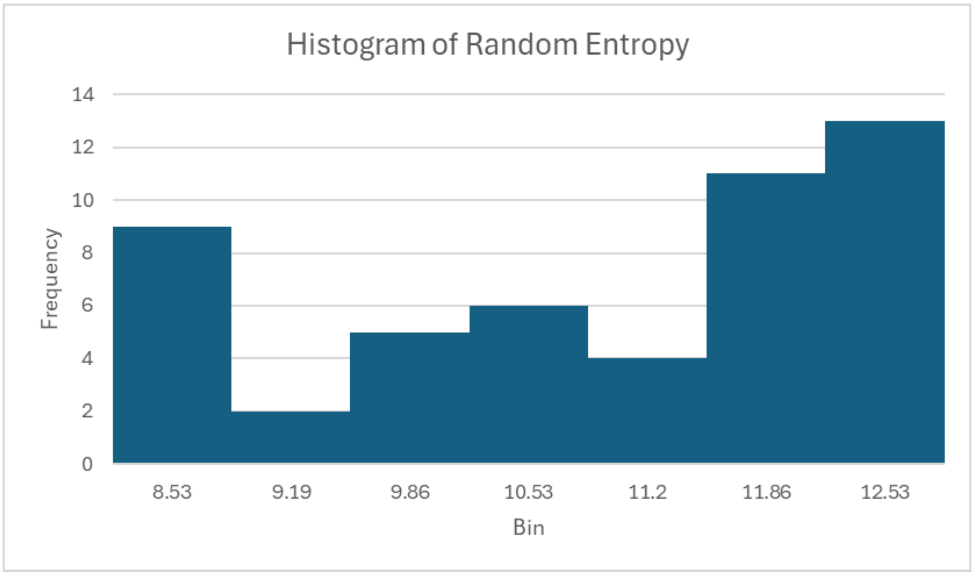


Figure 3.22: Histogram of Random APK's entropy, major mode right.

test could be suitable for this task as they are both used to compare two groups of data and produce a significance score. If the produced score is above or below a threshold value, then the given hypothesis can be accepted. In this early stage research there is no hypothesis to prove, the aim of this experiment is to identify what, if anything, has changed, hence common nonparametric tests are inappropriate at this stage.

Parameter Name	SGA	Random	Original
Number of trials	12	12	Not applicable
Number of runs per trial	7	7	Not applicable
Number of generations per run	3	3	Not applicable
Number of individuals per generation	5	5	1
Mutation rate	80%	80%	Not applicable
Tournament size	3	3	Not applicable
Individual size	2	2	Not applicable

Table 3.6: Experiment parameters for Section 3.3.2. File size, number of unique API calls, percentage of unique external API calls, entropy, histograms, percentage difference and Pearson correlation were used in the analyse of results. The *Original* test group, was created by duplicating the original sample used in SGA and Random 50 times. The tool Androguard was used to create call graphs that were used to determine API calls, this process takes approximately 8 hours for 50 samples as obfuscation makes analysis slower.

The next step of the experiment was to visually compare the results from the different test groups, namely Random to Original and SGA to Original. In both test cases the number of unique calls increase dramatically and the proportion of external calls deceased, with SGA having the clearest decrease (see Figures 3.24 and 3.25).

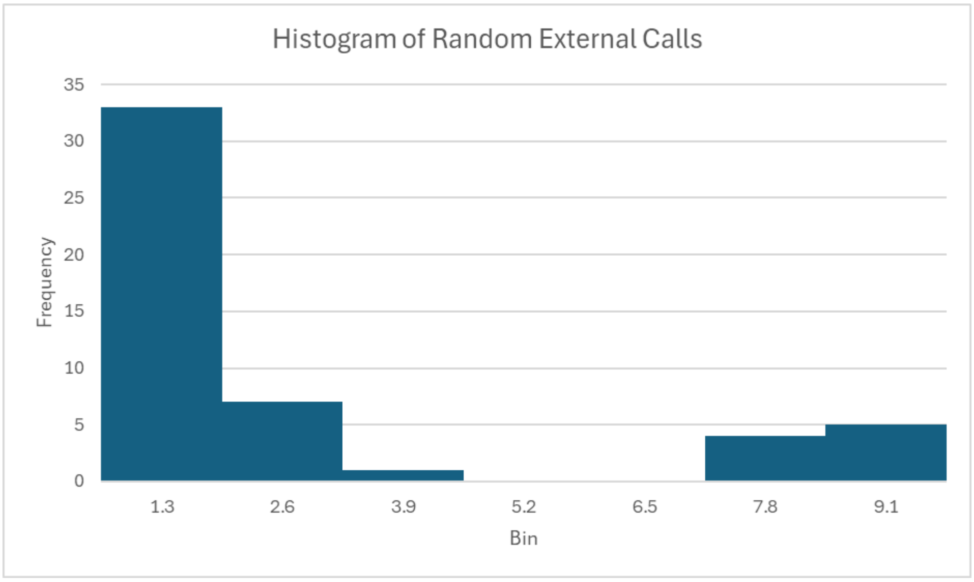


Figure 3.23: Histogram of Random APK's external API calls, major mode left.

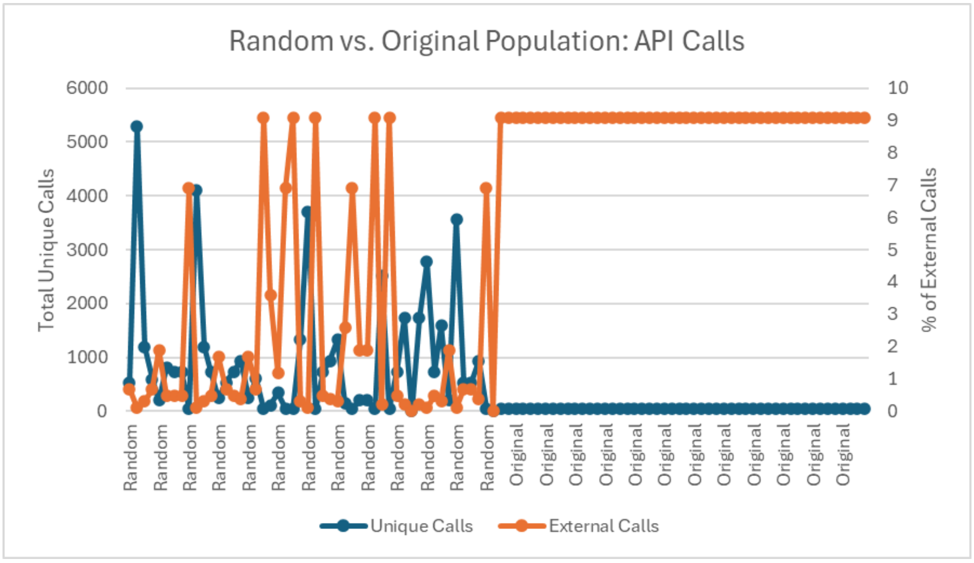


Figure 3.24: Unique API calls of Random vs. Original APKs. In the Original about 9% of the code is accessed for external libraries, as this is a unique count it is only a rough indication of what percentage of the programs code is stored in the APK.

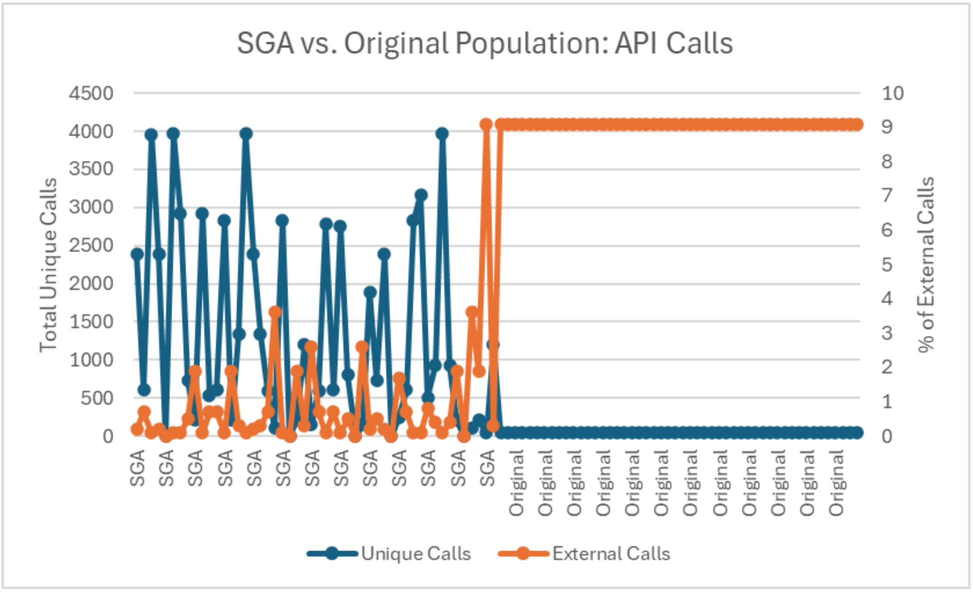


Figure 3.25: Unique API calls of SGA vs. Original APKs. In the Original about 9% of the code is accessed for external libraries, as this is a unique count it is only a rough indication of what percentage of the programs code is stored in the APK.

When comparing the largest amount of unique calls, there is little difference once the outlier in Random is taken in to account. In both cases, the external calls do not increase pass the original proportion as they become hidden and replaced with new unique calls. The original APK had 44 unique API calls in total, so in both cases the amount of obfuscation applied is significant.

It can be seen that in both cases the file size and entropy has been affected by obfuscation (see Figures 3.26 and 3.27). Low entropy means the data being measured is more predictable, an increase in entropy is used as a possible indication of malware [62]. Ideally the applied obfuscation would not increase entropy. Compared to the volume of entropy in the original file, the applied techniques entropy is relatively low and in the case of SGA the proportion did not increase significantly in relation to file size. This could be due to Obfuscapk using code templates in its eight *code* based obfuscators, consequently repeated application will create similar patterns throughout the code. To gain a clearer understanding of which generation method effected each metric the most, the Pearson correlation coefficient was determined for each comparison (see Table 3.7). There is a significant positive correlation in both cases for file size and entropy, as the file size gets bigger so does its entropy, there is negligible difference between the

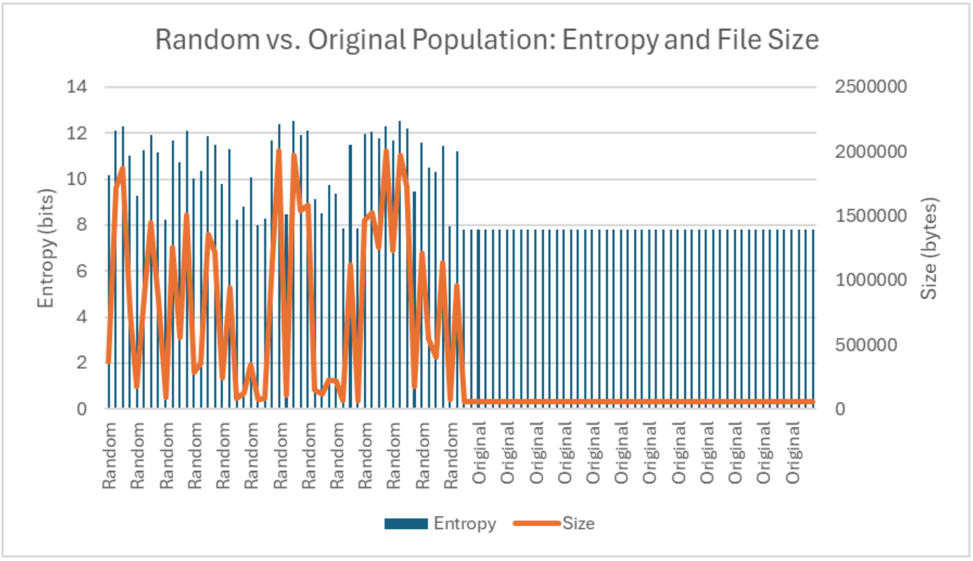


Figure 3.26: Entropy and file size of Random vs. Original APKs.

two test groups. In respect to API calls, there is a marginal negative correlation, again with negligible difference between the two test groups. Pearson correlation coefficient formula is:

$$n = \text{Number of pairs of observations}$$

$$\text{Pearson correlation} = \frac{n * \Sigma(\text{SGA} * \text{Random}) - \Sigma(\text{SGA}) * \Sigma(\text{Random})}{\sqrt{[n * \Sigma(\text{SGA}^2) - \Sigma(\text{SGA})^2][n * \Sigma(\text{Random}^2) - \Sigma(\text{Random})^2]}}$$

APK Origin	File Entropy - File Size	Unique Calls - External Calls
SGA	0.949051619	-0.432872374
Random	0.934926878	-0.455393819

Table 3.7: Pearson correlation coefficient for SGA and Random datasets.

Analysis of Results

Algorithm 6 depicts the structure for the experiments run in this section. To better understand if SGA or Random is producing more effectivity obfuscated APKs a comparison of ten final Jotti scores was made. Figure 3.28 shows that SGA is producing the more evasive results consistency, as the resulting Jotti scores are lower. To make a concise comparison the percentage difference formula was used:

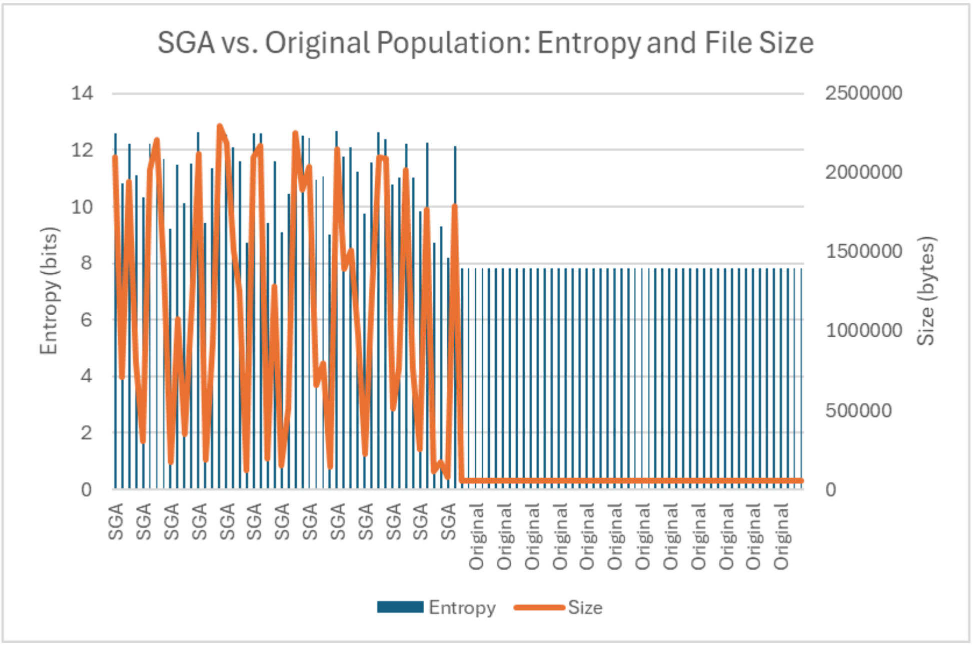


Figure 3.27: Entropy and file size of SGA vs. Original APKs.

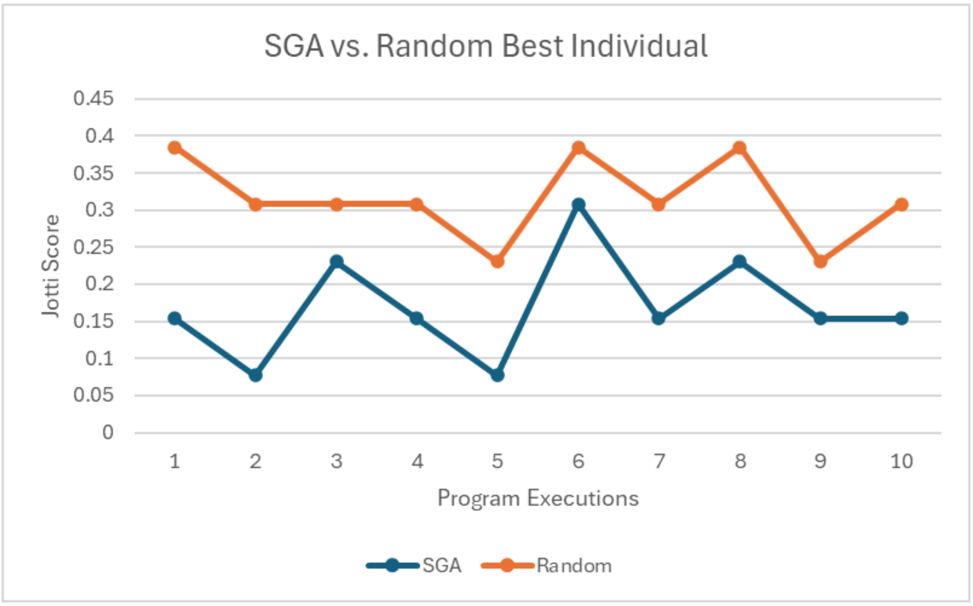


Figure 3.28: Results from ten consecutive executions of the SGA and Random based prototypes. A lower Jotti score indicates fewer antivirus detections.

APK Origin	Accumulative Jotti Score	Accumulative runtime
SGA	1.692307692	7 Hours : 15 Minutes
Random	3.153846154	5 Hours : 11 Minutes
Percentage Difference	60.32%	33.39%

Table 3.8: Overall comparison of ten SGA and Random executions, using percentage difference formula.

$$\text{Percentage difference} = \frac{|\Sigma(\text{SGA}) - \Sigma(\text{Random})|}{(\Sigma(\text{SGA}) + \Sigma(\text{Random}))/2} * 100$$

It can be seen in Table 3.8 that the Random's performance time is 33.39% less than SGA, which accounts for the additional selection process in SGA. The accumulative Jotti score of SGA is about 60.32%, lower than Random meaning there was a significant improvement with the selection method added.

Algorithm 6 Pseudocode of Experiment from Section 3.3.2

```
For each test case:                                          ▷ Create test data
while $trials \leq numberoftrials$ do
    while $runs \leq maxRuns$ do
        Initialise SGA with test case
        while $generation \leq maxGeneration$ do             ▷ Run algorithm
            Selection
            Mutation
            Fitness evaluation
        end while
    end while
end while
Get Random entropy and external API calls:                   ▷ Process test data
→ Do frequency analysis and plot histogram
Get All API calls:
→ Compare Random Vs Original and SGA Vs Original plot line graph
Get All entropy and file size:
→ Compare Random Vs Original and SGA Vs Original, plot line and bar graph
Get the final Jotti score from 10 trials:
→ Compare Random Vs SGA, plot line graph
```

CHAPTER 4

Discussions, Conclusions and Future Work

In terms of the three domains of this dissertation the key findings are, that with obfuscation methods built in the year 2020, a genetic algorithm can find sequences of obfuscation, for Android applications, that can bypass antivirus systems in 2025. The program achieved this with only control and data flow manipulation-based obfuscation and produced good results with a population of five after twenty-one generations. This finding is significant because the methods used are not highly complex to implement, as shown during the dissertation, and in terms of computer technology, five years is a significant amount of time. For the dissertation's design and application, using a generator/discriminator type implementation provides natural encapsulation of code structure which helps both troubleshoot during development and provides accessibility when applying new features.

The contributions of this research are two-fold. Firstly, existing research was recreated using recent Android applications rather than pre-1995 DOS-era applications, making its insights relevant to current computer systems. Secondly, achieving this with standard computer resources (i.e., not a high-performance computer), as is typically used in related research, broadens the audience to which the results are applicable. Comparing the research outcomes to expectations from the literature review, in particular [56], the outcomes were as expected. Repeated obfuscation can bypass antivirus systems.

4.1 Research Limitations

The first limitation was the use of Obfuscapk, while concurrently its use made the dissertation possible within the given constraints. The Obfuscapk package was a limiting factor for several reasons:

- It used boilerplate implementations of code obfuscation, which limits the range of obfuscation capabilities.
- It was built with various tools, making simple optimisation strategies fail. Its runtime contributed to the majority of the prototype's total runtime.
- Due to stability issues, reflection and encryption obfuscation methods could not be applied; ideally, these would have been experimented with to varying degrees.

The second set of limiting factors concerned the calculation of the Jotti feedback signal, used to guide the SGA search:

- As there was no definition between different vendors, the Jotti score is the same regardless of which vendors made the detection (i.e., if Avast or BitDefender make a detection, the same value of one detection is used to calculate the Jotti score). This reduces the uniqueness of the feedback signal. A solution could be for the antivirus vendors to be ranked based on their *AV-test.org* rank, then use the ranking to weigh the Jotti score calculation.
- Considering how many commercial antivirus companies are actively making products worldwide, Jotti has limited vendors available. Consequently, the Jotti score does not indicate how generated samples would be detected globally. Using VirusTotal, with a broader range of vendors, would make the detection ratio metric more realistic. This diversity could help drive the generation of more evasive samples.

The final limitations concern how testing was performed:

- Performing experiments with a wider range of APKs would yield more reliable test results, and allow the use of statistical methods that require a large observation set (see Section 3.3.1). This could be achieved by collecting samples representing a malware's historical developments (i.e., samples of the same application from different years, 2005, 2010, etcetera).
- Specific experiment parameters could have been improved. For example:
  - Using encoding methods that closely mimicked those used by the Android OS and a wider range of comparison metrics (see Section 3.2.2).
  - Simultaneity experimenting with the selection method from inside the SGA and the main loop (see Section 3.2.3).
- The functionality testing of an APK does not generalise well (see Section 2.2.5). Current testing depends on understanding each APK's operations and relies on the applied obfuscation not dramatically changing what is printed to the console log. More reliable testing of generated samples could be performed using the automation features of the Android App Crawler [26] and UI Automator [31] tools.
- There is no testing to investigate how often the SGA invalidates an APK, which is critical to understand the stability of this prototype. Additionally, some transformations can be lost after compiling (see [63] and Section 2.2.3). So precisely how effective this prototype is at applying obfuscation is unknown until the measure of post and previous compilation is taken as well as a large-scale test of the functional completeness of a range of generated samples.

scikit posthocs and *scipy stats* were used to investigate the suitability of statistical analysis functions that could better facilitate understanding of the generated data. Statistical functions suitable for non-normally distributed datasets with two or more sample groups showed few significant results, or results that showed significant for one function were not significant in another. For example in Table 4.1, Test 5 has significant results, but manual inspection of the dataset shows that Test 2 obtained the best Jotti score overall. Therefore, a range of more primitive statistical analyses was carried out, as seen in the dissertation. In this dissertation, the most significant results are the Jotti scores, which indicate the effectiveness of the applied obfuscation. The Jotti score tends to fluctuate, which is not ideal for statistical analysis. This could be due to three factors: There are only 13 vendors available, of varying degrees of obfuscation detectability. Genetic algorithms are non-deterministic, and applying more obfuscation does not necessarily produce better obfuscation. For better statistical analysis, multiple original APK samples (for example, taken from different time periods), as suggested in Chapter 4, could provide a more robust dataset for analysis; however, this is not realistic given the time frame and efficacy of the prototype. For example, it can be seen in Chapter 2 Table 2.1 that "spike.old.apk" takes ten minutes to apply all obfuscation methods while "comet.bot.apk" takes two minutes. The shortest probabilistic experiment was 3.2.3, when using "comet.bot.apk" it takes about fifteen hours in total to perform. Using "old.spike.apk" there would be up to a 5 times increase in runtime.

Test Name	Test One	Test Two	Test Three	Test Four	Test Five
Test One	1.0	0.894136	1.0	1.0	0.978233
Test Two	0.894136	1.0	1.0	1.0	0.007965
Test Three	1.0	1.0	1.0	1.0	0.2142
Test Four	1.0	1.0	1.0	1.0	0.0519
Test Five	0.978233	0.007965	0.2142	0.0519	1.0

Table 4.1: Dunn's tests on the Jotti scores from Section 3.3.1.

4.2 Future Research

The two primary considerations for future development of this dissertation would be in redeveloping the generator and discriminator components. This would aim to improve the application of obfuscation and enable the prototype to be used for larger applications. For example, using Windows programs ([15], [65]).

1. Developing a generator using a genetic program (GP) to enable custom-built

obfuscation functions, overcoming the limited flexibility of genetic algorithms and enabling the utilisation of crossover.

2. Consideration would need to be given to the performance of the mutation functions (see Section 3.2.1). Using CUDA C to find locations in the sample for code injection and encryption would make the sample transformation more time efficient [67]. Performing the disassembly of the sample on the CPU would give the benefit of using existing tools.

3. Upgrading the discriminator with a machine learning agent, like reinforcement learning. In this configuration, the generator can record metadata about generated samples to inform the discriminator during training, for example, which types of obfuscation have been used on a sample, and the discriminator can be trained in detecting malware and produce a more innate signal in return to maximise co-training between the two components, linking training parameters [71].

The development of the dissertation produced two unexpected by-products, which are both noteworthy for future research:

- Section 3.2.2 showed the application of different encodings revealed internal information from the APK. This could be explored as a heuristic when training a machine learning model to understand malware samples.

- As stated in Jotti's terms and conditions, the web service processes uploaded files for antivirus companies as training data. Investigating how dataset diversity benefits antivirus developers is a beneficial avenue for further research.

CHAPTER 5

Glossary

Antivirus program/antivirus system: A program for detecting malicious software.

API: Application Programming Interface - rules for software to communicate with each other.

APK: Android Package Kit - a file extension for Android Applications.

Application: A piece of software built for an end users.

DEAP: Distributed Evolutionary Algorithms in Python - a GA and GP framework.

Evolutionary Algorithm: A category of optimisation algorithms that includes the ideas of crossover, mutation, selection and generations. Both GA and GP are types of evolutionary algorithm.

Framework: A software suite that enables developers to build alongside an existing paradigm.

GA: Genetic Algorithm - a programming paradigm to manipulate lists.

Generator/Discriminator: A software architecture where one component classifies data (discriminator) and the other transforms data (generator).

GP: Genetic Program - a programming paradigm to manipulate programs.

GUI: Graphical User Interface.

Malware: A malicious piece of software.

Obfuscation: Making a program unintelligible, hiding its characteristics.

OS: A computer operating system.

Package: A piece of software intended for software development.

Program: A piece of software.

Prototype: A piece of experimental software.

Sample: A stored application.

Sandbox: A software tool that isolates running applications.

SDK: Software Development Kit.

SGA: Simple Genetic Algorithm - a genetic algorithm where an intensive level of modification is performed in each generation.

Tool/Tools: Software programs used for a specific purpose.

Vendor: An installation of an antivirus program.

CHAPTER 6

Bibliography

[1] AO Kaspersky Lab 2024. *Kaspersky Threat Intelligence Portal — opentip.kaspersky.com*. https://opentip.kaspersky.com/. [Accessed 11-08-2025].

[2] Algorithm Afternoon. *Grammatical Evolution | Algorithm Afternoon — algorithmafternoon.com*. https://algorithmafternoon.com/programming/grammatical_evolution/. [Accessed 09-08-2025].

[3] Maxat Akbanov and Vassilios Vassilakis. “WannaCry Ransomware: Analysis of Infection, Persistence, Recovery Prevention and Propagation Mechanisms”. In: *Journal of Telecommunications and Information Technology* 1 (Apr. 2019), pp. 113–124. DOI: 10.26636/jtit.2019.130218.

[4] ALLATORI. *Allatori Java Obfuscator - Professional Java Obfuscation since 2006 — allatori.com*. https://allatori.com/. [Accessed 10-08-2025].

[5] The Openhandset Alliance. *Alliance Members | Open Handset Alliance — openhandsetalliance.com*. https://www.openhandsetalliance.com/oha_members.html. [Accessed 23-04-2025].

[6] Androguard. *Welcome to Androguard 2019 documentation!; Androguard 3.4.0 documentation — androguard.readthedocs.io*. https://androguard.readthedocs.io/en/latest/. [Accessed 13-09-2025].

[7] Simone Aonzo et al. “Obfuscapk: An open-source black-box obfuscation tool for Android apps”. In: *SoftwareX* 11 (2020), p. 100403. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2020.100403. URL: https://www.sciencedirect.com/science/article/pii/S2352711019302791.

[8] Apktool. *Apktool — apktool.org*. https://apktool.org/. [Accessed 18-07-2025].

[9] Pieter Arntz. *Android threats rise sharply, with mobile malware jumping by 151 percent since start of year — malwarebytes.com*. https://www.malwarebytes.com/blog/news/2025/06/android-threats-rise-sharply-with-mobile-malware-jumping-by-151-since-start-of-year. [Accessed 08-09-2025].

[10] Daniel Arp. *The Drebin Dataset — drebin.mlsec.org*. https://drebin.mlsec.org/. [Accessed 25-09-2025].

[11] Ashishb. *GitHub - ashishb/android-malware: Collection of android malware samples — github.com*. https://github.com/ashishb/android-malware. [Accessed 13-06-2025].

[12] Francisco Baeta et al. “Speed benchmarking of genetic programming frameworks”. In: GECCO ’21. Lille, France: Proceedings of the Genetic and Evolutionary Computation Conference, 2021, 768â775. ISBN: 9781450383509. DOI: 10.1145/3449639.3459335. URL: https://doi.org/10.1145/3449639.3459335.

[13] Kurt Baker. *What is a Polymorphic Virus? Examples & More | CrowdStrike — crowdstrike.com*. https://www.crowdstrike.com/en-us/cybersecurity-101/malware/polymorphic-virus/. [Accessed 10-06-2025].

[14] Zach Bobbitt. *A Simple Explanation of the Jaccard Similarity Index — statology.org*. https://www.statology.org/jaccard-similarity/. [Accessed 11-06-2025].

[15] Raphael Labaca Castro, Corinna Schmitt, and Gabi Dreo. “AIMED: Evolving Malware with Genetic Programming to Evade Detection”. In: *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. 2019, pp. 240–247. DOI: 10.1109/TrustCom/BigDataSE.2019.00040.

[16] cdifflib. *cdifflib 1.2.9 — pypi.org*. https://pypi.org/project/cdifflib/. [Accessed 11-06-2025].

[17] Sen Chen. *Welcome to KuafuDet Homepage — sen-chen.github.io*. https://sen-chen.github.io/kuafuDet/kuafuDet.html. [Accessed 05-08-2025].

[18] Stat Counter. *Operating System Market Share Worldwide | Statcounter Global Stats — gs.statcounter.com*. https://gs.statcounter.com/os-market-share/. [Accessed 23-04-2025].

[19] Cuckoosandbox. *landing page*. https://sandbox.pikker.ee. [Accessed 11-08-2025].

[20] DashO. *Android Obfuscation and Java Security with DashO — preemptive.com*. https://www.preemptive.com/products/dasho/. [Accessed 10-08-2025].

[21] Innovation Department for Science and Technology. *Cyber Security and Resilience Bill — gov.uk*. https://www.gov.uk/government/collections/cyber-security-and-resilience-bill. [Accessed 13-09-2025].

[22] Dexprotector. *DexProtector — dexprotector.com*. https://dexprotector.com/. [Accessed 10-08-2025].

[23] Diffchecker. *Diffchecker - Compare text online to find the difference between two text files — diffchecker.com*. https://www.diffchecker.com/text-compare/. [Accessed 14-06-2025].

[24] Science Direct. *Malware Family - an overview | ScienceDirect Topics — sciencedirect.com*. https://www.sciencedirect.com/topics/computer-science/malware-family. [Accessed 19-09-2025].

[25] Android Documentation. *Android runtime and Dalvik Android Open Source Project — source.android.com*. https://source.android.com/docs/core/runtime. [Accessed 23-04-2025].

[26] Android Documentation. *App Crawler Android Studio Android Developers — developer.android.com*. https://developer.android.com/studio/test/other-testing-tools/app-crawler. [Accessed 21-09-2025].

[27] Android Documentation. *Application fundamentals App architecture Android Developers — developer.android.com*. https://developer.android.com/guide/components/fundamentals. [Accessed 23-04-2025].

[28] Android Documentation. *monkeyrunner Android Studio Android Developers — developer.android.com*. https://developer.android.com/studio/test/monkeyrunner#SampleProgram. [Accessed 14-06-2025].

[29] Android Documentation. *Secure an Android device Android Open Source Project — source.android.com*. https://source.android.com/docs/security/overview. [Accessed 23-04-2025].

[30] Android Documentation. *Shrink, obfuscate, and optimize your app Android Studio Android Developers — developer.android.com*. https://developer.android.com/build/shrink-code. [Accessed 01-05-2025].

[31] Android Documentation. *Write automated tests with UI Automator Test your app on Android Android Developers — developer.android.com*. https://developer.android.com/training/testing/other-components/ui-automator. [Accessed 21-09-2025].

[32] Android documentation. *Dalvik executable format Android Open Source Project — source.android.com*. https://source.android.com/docs/core/runtime/dex-format. [Accessed 01-09-2025].

[33] Microsoft Documentation. *Behavior monitoring in Microsoft Defender Antivirus - Microsoft Defender for Endpoint — learn.microsoft.com*. https://learn.microsoft.

com/en-us/defender-endpoint/behavior-monitor. [Accessed 16-07-2025].

[34] Zhiyang Fang et al. “Feature Selection for Malware Detection Based on Reinforcement Learning”. In: *IEEE Access* 7 (2019), pp. 176177–176187. DOI: 10.1109/ACCESS.2019.2957429.

[35] Fanmcgrady. *GitHub - fanmcgrady/select-features — github.com*. https://github.com/fanmcgrady/select-features. [Accessed 19-09-2025].

[36] Ruitao Feng et al. “A Performance-Sensitive Malware Detection System Using Deep Learning on Mobile Devices”. In: *IEEE Transactions on Information Forensics and Security* 16 (2021), pp. 1563–1578. DOI: 10.1109/TIFS.2020.3025436.

[37] Python foundation. *codecs - Codec registry and base classes*. https://docs.python.org/3.13/library/codecs.html#standard-encodings. [Accessed 01-09-2025].

[38] Python foundation. *python.org*. https://www.python.org/download/alternatives/. [Accessed 01-09-2025].

[39] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML]. URL: https://arxiv.org/abs/1406.2661.

[40] Google. *Pixel Binary Transparency Android Binary Transparency Google for Developers — developers.google.com*. https://developers.google.com/android/binary_transparency/pixel_overview. [Accessed 25-09-2025].

[41] Google. *Use Google Play Protect to help keep your apps safe and your data private - Google Play Help — support.google.com*. https://support.google.com/googleplay/answer/2812853?hl=en. [Accessed 25-09-2025].

[42] Guardsquare. *GitHub - Guardsquare/proguard: ProGuard, Java optimizer and obfuscator — github.com*. https://github.com/Guardsquare/proguard. [Accessed 01-05-2025].

[43] Qian Han et al. *The Android Malware Handbook: Detection and Analysis by Human and Machine*. No Starch Press, 2023.

[44] Joe Hindy. *10 best third-party app stores for Android — androidauthority.com*. https://www.androidauthority.com/best-app-stores-936652/. [Accessed 10-06-2025].

[45] Greg Hornby et al. "Automated Antenna Design with Evolutionary Algorithms". In: *Collection of Technical Papers - Space 2006 Conference* 1 (Sept. 2006). DOI: 10.2514/6.2006-7242.

[46] Jotti. *Jotti's malware scan — virusscan.jotti.org*. https://virusscan.jotti.org/en-GB/scan-file. [Accessed 11-08-2025].

[47] Kaggle. *Android Malware Detection — kaggle.com*. https://www.kaggle.com/datasets/subhajournal/android-malware-detection/data. [Accessed 13-06-2025].

[48] Kaggle. *Ember-2018-V2-features — kaggle.com*. https://www.kaggle.com/datasets/dhoogla/ember-2018-v2-features. [Accessed 25-09-2025].

[49] Zelix Pty Ltd. *Java Obfuscator - Zelix KlassMaster; — zelix.com*. https://zelix.com/klassmaster/index.html. [Accessed 10-08-2025].

[50] Sean Luke. *Essentials of Metaheuristics*. second. Available for free. Lulu, 2013. URL: https://cs.gmu.edu/~sean/book/metaheuristics/Essentials.pdf.

[51] Gopinath M. and Sibi Chakkaravarthy Sethuraman. "A comprehensive survey on deep learning based malware detection techniques". In: *Computer Science Review* 47 (2023), p. 100529. ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2022.100529. URL: https://www.sciencedirect.com/science/article/pii/S1574013722000636.

[52] ManSoSec. *GitHub - ManSoSec/Microsoft-Malware-Challenge — github.com*. https://github.com/ManSoSec/Microsoft-Malware-Challenge. [Accessed 13-06-2025].

[53] Daniel Jurafsky James H. Martin. *N-gram Language Models*. https://web.stanford.edu/~jurafsky/slp3/3.pdf. [Accessed 11-06-2025].

[54] Guozhu Meng et al. *Mystique: Evolving Android Malware for Auditing Anti-Malware Tools*. Xi'an, China, 2016. DOI: 10.1145/2897845.2897856. URL: https://doi.org/10.1145/2897845.2897856.

[55] Metadefender. *landingpage*. https://metadefender.opswat.com. [Accessed 11-08-2025].

[56] Ritwik Murali, Palanisamy Thangavel, and C. Shunmuga Velayutham. "Evolving malware variants as antigens for antivirus systems". In: *Expert Systems with Applications* 226 (2023), p. 120092. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2023.120092. URL: https://www.sciencedirect.com/science/article/pii/S0957417423005948.

[57] Springer Nature. *Sturges' and Scott's Rules — link.springer.com*. https://link.springer.com/rwe/10.1007/978-3-642-04898-2_578. [Accessed 08-09-2025].

[58] Necstaamo. *GitHub - necst/aamo: AAMO: Another Android Malware Obfuscator — github.com*. https://github.com/necst/aamo. [Accessed 26-09-2025].

[59] NIST. *Levenshtein distance — xlinux.nist.gov*. https://xlinux.nist.gov/dads/HTML/Levenshtein.html. [Accessed 11-06-2025].

[60] Pracsec. *Threat Hunting with File Entropy — practicalsecurityanalytics.com*. https://practicalsecurityanalytics.com/file-entropy/. [Accessed 11-06-2025].

[61] ProGuard. *ProGuard Manual: Examples | Guardsquare — guardsquare.com*. https://www.guardsquare.com/manual/configuration/examples. [Accessed 13-06-2025].

[62] Shyam Sundar Ramaswami. *Using entropy to spot the malware hiding in plain sight — umbrella.cisco.com*. https://umbrella.cisco.com/blog/using-entropy-to-spot-the-malware-hiding-in-plain-sight. [Accessed 08-09-2025].

[63] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. "Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks". In: *IEEE Transactions on Information Forensics and Security* 9.1 (2014), pp. 99–108. DOI: 10.1109/TIFS.2013.2290431.

[64] Jonathan Risto. *What is CVSS? — sans.org*. https://www.sans.org/blog/what-is-cvss. [Accessed 09-08-2025].

[65] Sevil Sen, Emre Aydogan, and Ahmet I. Aysan. "Coevolution of Mobile Malware and Anti-Malware". In: *IEEE Transactions on Information Forensics and Security* 13.10 (2018), pp. 2563–2574. DOI: 10.1109/TIFS.2018.2824250.

[66] David Silver. *Lecture 9: Exploration and Exploitation*. https://web.stanford.edu/class/cme241/lecture_slides/david_silver_slides/XX.pdf. [Accessed 19-09-2025].

[67] Radostin Stoyanov et al. *CRIUgpu: Transparent Checkpointing of GPU-Accelerated Workloads — arxiv.org*. https://arxiv.org/abs/2502.16631. [Accessed 17-09-2025].

[68] The PyPy Team. *PyPy - Features — pypy.org*. https://pypy.org/features.html. [Accessed 01-09-2025].

[69] av-test.org. *Test antivirus software for Windows 11 - April 2025 — av-test.org*. https://www.av-test.org/en/antivirus/home-windows/windows-11/april-2025/. [Accessed 16-07-2025].

[70] Somanath Tripathy, Narendra Singh, and Divyanshu N Singh. *ADAM: Automatic Detection of Android Malware — link.springer.com*. https://link.springer.com/chapter/10.1007/978-3-031-17510-7_2. [Accessed 26-09-2025].

[71] Matthew J. Turner, Erik Hemberg, and Una-May O'Reilly. "Analyzing multi-agent reinforcement learning and coevolution in cybersecurity". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '22 (2022), 1290â1298. DOI: 10.1145/3512290.3528844. URL: https://doi.org/10.1145/3512290.3528844.

[72] vx-underground.org. *Vx Underground — Home Page*. https://vx-underground.org/. [Accessed 13-06-2025].

[73] Unicode. *Supported Scripts — unicode.org*. https://www.unicode.org/standard/supported.html. [Accessed 01-09-2025].

[74] VirusTotal. *VirusTotal — virustotal.com*. https://www.virustotal.com/gui/home/upload. [Accessed 11-06-2025].

[75] VirusTotal. *VirusTotal — virustotal.com*. https://www.virustotal.com/gui/file/2877b27f1b6c7db466351618dda4f05d6a15e9a26028f3fc064fa144ec3a1850
[Accessed 28-05-2025].

[76] VirusTotal. *VirusTotal — virustotal.com*. https://www.virustotal.com/gui/file/4700a4e1f24f2d74a1d66f82ad12bd87f190756cfcf815c3e21b6c6282e0d0c3
[Accessed 28-05-2025].

[77] VirusTotal. *VirusTotal — virustotal.com*. https://www.virustotal.com/gui/file/642da73bc4c78004304dfed2e6e704ebb352ff9f1db19a19cc2296c86164e723
[Accessed 01-05-2025].

[78] VirusTotal. *VirusTotal — virustotal.com*. https://www.virustotal.com/gui/file/24e35247a430bd8d23d0c385aaf782269afb2f645b4e0fa4bbc32e3f8840417f
[Accessed 01-05-2025].

[79] VirusTotal. *VirusTotal — virustotal.com*. https://www.virustotal.com/gui/file/bde0d8617a0e0dccf8bf7e731c419f2296bf0522ddbab44296c834fcede61890
[Accessed 01-05-2025].

[80] VirusTotal. *VirusTotal — virustotal.com*. https://www.virustotal.com/gui/file/3b520dcd120d2ad472ef242e35d58e3ae9f2fefc6ad7ce1d9aa09b17c2048a2d
[Accessed 01-05-2025].

[81] VirusTotal. *VirusTotal — virustotal.com*. https://www.virustotal.com/gui/file/1be74307604525b456f80037cdd2ff03cb7d0b89346d9d83eec2a94ccf522d0e
[Accessed 01-05-2025].

[82] VirusTotal. *VirusTotal — virustotal.com*. https://www.virustotal.com/gui/file/c28a7f52c6ce6a00acbbc8ef769cc2a31f975259c8f25caf2ea7054806517bea [Accessed 01-05-2025].

[83] Yinxing Xue et al. “Auditing Anti-Malware Tools by Evolving Android Malware and Dynamic Loading Technique”. In: *IEEE Transactions on Information Forensics and Security* 12.7 (2017), pp. 1529–1544. DOI: 10.1109/TIFS.2017.2661723.

[84] Yajin Zhou. *Yajin Zhou@Zhejiang University — malgenomeproject.org*. http://www.malgenomeproject.org/. [Accessed 26-09-2025].