Anti-virus tool:
Using anti-virus techniques for malware detection

Author: Radoslav Ralinov  
Supervisor: Ning Zhang

The School of Computer Science
Compute Science BSc (Hons) with Industrial Experience
Third Year Project Report
May 2016
Abstract

Anti-virus tool: Using anti-virus techniques for malware detection

Malware detection in the 21st century has become one of the most if not the most important aspects of security software in the Computer Science area. With the growing numbers of personal computer users the need for good malware detection algorithms has been increasing with considerable rate.

The aim of this project is to build an application that demonstrates some of the main virus detection techniques used by anti-virus software to detect and remove malicious files. The techniques considered in the report could be split in two – signature-based and heuristic-based malware detection. The report also investigates the possibility of having an application that could learn based on the heuristic approaches.

The report begins with a review of one of the existing malware scanning programs and the approaches it used for malware detection. At the end of the report, performance of the techniques implemented is compared as well as the effectiveness of each one of them is evaluated.

Author: Radoslav Ralinov

Supervisor: Ning Zhang
Acknowledgements

Firstly I would like to thank my supervisor Ning Zhang for her support and guidance throughout the project. Her questions as well as answers really helped me drive this project forward in the right direction.

I would also like to thank my family for believing in me and continuously supporting me throughout the whole third year.
Contents

1 Introduction .................................................................................................................. 5
  1.1 Motivation ............................................................................................................. 5
  1.2 Project Aim and Objectives .................................................................................. 6
  1.3 Report Structure .................................................................................................. 6

2 Background .................................................................................................................. 7
  2.1 Types of malware ................................................................................................. 7
  2.2 ClamAntiVirus anti-virus scanner ......................................................................... 7
    2.2.1 ClamAV malware signatures ......................................................................... 7
    2.2.2 ClamAV heuristic .......................................................................................... 9
  2.3 Anti-virus techniques nowadays ........................................................................... 9

3 Design .......................................................................................................................... 10
  3.1 Requirements ....................................................................................................... 10
    3.1.1 Functional ..................................................................................................... 10
    3.1.2 Non-Functional ............................................................................................ 11
  3.2 Technologies .......................................................................................................... 12
    3.2.1 Programming languages .............................................................................. 12
    3.2.2 Orchestrate ................................................................................................... 12
    3.2.3 Version control .............................................................................................. 13
    3.2.4 Java Native Interface .................................................................................. 13
  3.3 The malware detection cycle .................................................................................. 14

4 Implementation .......................................................................................................... 16
  4.1 Key libraries ......................................................................................................... 16
  4.2 Signature-based detection ..................................................................................... 17
  4.3 Heuristic-based detection ..................................................................................... 18
  4.4 Concurrency .......................................................................................................... 18
  4.5 Malware management ........................................................................................... 19
  4.6 User interface implementation ............................................................................. 19

5 Testing and evaluation ............................................................................................... 21
  5.1 Automated testing ............................................................................................... 21
  5.2 Manual Testing ..................................................................................................... 21
  5.3 Evaluation ............................................................................................................. 22
Table of Figures

FIGURE 1 CLAMAV'S FULL HASH SIGNATURE FORMAT ......................................................... 8
FIGURE 2 CLAMAV'S PARTIAL HASH SIGNATURE FORMAT ............................................... 8
FIGURE 3 CLAMAV'S BYTE SIGNATURE FORMAT .............................................................. 9
FIGURE 4 A BYTE SIGNATURE IN JSON FORMAT IN ORCHESTRATE ................................. 13
FIGURE 5 JNI IN AN APPLICATION [15] ........................................................................... 13
FIGURE 6 THE MALWARE DETECTION CYCLE .................................................................. 14
FIGURE 7 AHO-CORASICK TRIE WITH KEYWORDS "AB" "BCA" AND "CAA" [17] ............... 16
FIGURE 8 COMPARISON BETWEEN MD5 AND SHA256 .................................................... 17
FIGURE 9 HASH SIGNATURE DETECTION PROCESS ....................................................... 18
FIGURE 10 MULTITHREADING EXAMPLE .......................................................................... 19
FIGURE 11 SCREENSHOT OF THE TOOL WHILE SCANNING ......................................... 20

Table of Tables

TABLE 1 FUNCTIONAL REQUIREMENTS .............................................................................. 10
TABLE 2 NON-FUNCTIONAL REQUIREMENTS .................................................................... 11
TABLE 3 COMPARISON BETWEEN AVAST ANTIVIRUS AND THE PROJECT TOOL .......... 22
Chapter 1

1 Introduction

Nowadays anti-virus software applications have become essential to the everyday personal computer user. These modern tools for computer malware scanning use a mixture of algorithms to detect and prevent malicious software from causing damage.

This project is about building an online anti-virus tool for scanning file systems and detecting malware. The project encompasses the two main areas of malware discovery techniques.

One approach which has been around since the inception of antivirus tools is the signature-based detection which examines the file’s key aspects for a known static fingerprint. The signature itself could be a sequence of bytes that represent the malicious code in the file or the cryptographic hash of the whole infected file. [1] Once the program has access to the malware signature it goes through the suspected file and looks for a match. The current number of virus signatures is over 100,000 and it is growing constantly. This is why it is really important what string-matching algorithm is chosen.

The other and more modern technique for virus detection is the heuristic-based. A heuristic technique means an approach to problem solving that is not guaranteed to be optimal or perfect but sufficient enough. The heuristics-based detection approach relies on inspecting files for suspicious characteristics without the help of signatures. For instance, the tool might check for malicious instructions or junk code in the file. Another example for heuristic detection is executing the file in emulation environment and tracking the behaviour. [1] This specific instance of heuristic is called file emulation meanwhile the previous is called file analysis. The main difference between the two is one focuses on the actions the file does when executed in a controlled virtual environment while the other one focuses on the file itself. [2] This is the reason that file analysis is also being called static analysis and file emulation - dynamic analysis. [3]

This chapter is about an introduction to the project, its aim and motivation.

1.1 Motivation

The increasing number of personal computers [4] and malware [5] poses an imminent threat to the casual person behind his desktop. Modern antivirus software vendors struggle to convince people that they need their product. This is partially due to the software updates that most of them have frequently so that they keep up to date with the most advanced malware. Another reason is that people do not realize they need it until an unfortunate event happens.

In 2014, Edward Snowden, a former Central Intelligence Agency employee, leaked documents proving that National Security Agency intended to spy on millions of its citizens using malware [6]. Since then antivirus software companies have realized they need to protect their customers not only from private malware producers but from the governments as well.

All this proves the necessity for better malware detection software around the world. This project is about demonstrating the main techniques modern antivirus software
companies use in the discovery of malware and exploring the possibility of creating a malware detection cycle that could improve existing techniques.

1.2 Project Aim and Objectives

The key objectives in this project could be broken down into three aspects – the signature-based detection techniques, the heuristic-based detection technique and ensuring the performance of the tool is reasonable. In addition the report discusses the opportunity for a malware detection cycle that could improve the effectiveness and performance of modern day anti-virus software. The main three objectives are as follows:

- Implement two signature-based detection techniques: hash signature detection and byte signature detection. There should be an online database with all the signatures ready to be accessed when scanning is started.
- Develop and implement a heuristic detection algorithm as a second layer of detection after the signature-based ones. The algorithm should minimize the false positive entries as much as possible. The application will need to use a library to disassemble the executable files and access the assembly code.
- When developing the tool performance should be taken into account at every step. Due to the tool scanning through gigabytes of files the performance is essential for the completion of this project.

1.3 Report Structure

The rest of the report consists of five chapters. A brief outline for each of them is as follows:

- **Chapter 2** consists of reviews one of the current open-source anti-virus software applications and explains its approach to malware detection.
- **Chapter 3** explains the design decisions made throughout the project and how some of the techniques used by the application in the background chapter were applied.
- **Chapter 4** is about the development of the application. It describes how the main features function and interact with the system.
- **Chapter 5** discusses the testing and evaluation approaches taken. It includes performance comparison between the project tool and other similar application.
- **Chapter 6** draws the conclusions made of the whole project. In addition, it describes a possible improvement of the application that could improve the tool’s both performance and effectiveness.
Chapter 2

2 Background

In this chapter one of the most popular free and open source malware detection toolkits is reviewed. It also describes the types of malware detection it supports and ends with information on modern day anti-virus software.

2.1 Types of malware

Malware is malicious software which is designed with the sole purpose of doing harm or other unwanted actions on a computer system. Examples of malware are viruses, worms, spyware and Trojans. The heuristic part of this project focuses on the first two types.

Computer viruses could represent a piece of code or a program that got onto a user’s file system without his knowledge. One of the purposes of a virus could be to replicate itself, thus filling the host system memory. The even more dangerous types of viruses are the ones which are able to propagate onto another system over a network. Computer viruses require a host to spread the infections. This host must be executed so that the malicious code inside it is executed together with the harmless one.

Worms are similar to viruses with the exception that they do not alter files but reside in active memory and duplicate. They are usually unnoticeable until their replication takes system resources and slows down the tasks.

2.2 ClamAntiVirus anti-virus scanner

ClamAntiVirus (ClamAV) is one of the most widely used open-source antivirus toolkits available. It was initially developed for UNIX systems but latter a Windows version was produced. The toolkit consists of various command line programs such as virus signature generator as well as a core scanner library.

2.2.1 ClamAV malware signatures

ClamAV supports the two types of virus signatures – byte and hash signatures. Each of them has its advantages as well as disadvantages. For instance, hash signatures are useful if the malware you are trying to detect is old and non-changing. The problem with this kind of detection is that modern viruses are polymorphic which means they could change their structure. Modifying the infected file in any way makes the hash algorithm which generates the signature produce different result and leaves the malicious file undetected. On the other hand, the byte signature could not be bypassed that easily. This kind of signature relies on a string-matching algorithm to find the specific bytes in the file that make it malicious. ClamAV’s approach to byte signature detection is to use the Aho-Corasick pattern matching algorithm. The particular advantage of this algorithm is that it is multi-pattern exact matching and the running time is independent of the number of patterns. [7] In comparison with algorithms like Bayer-Moore string search algorithm which uses pre-
processing to skip sections of the text, Aho-Corasick constructs finite automaton from the patterns searched for and then it uses it when the actual string matching begins. The algorithm’s usefulness in anti-virus programs comes from the possibility to have the automaton constructed from the large number of signatures beforehand.

### 2.2.1.1 Hash signature format

ClamAV uses a MD5 hashing function to create its signatures. It requires three attributes to be present before it could create the signature format. The first one is the file size in bytes, the second one is the MD5 hash of the file and the third one is the file location. These attributes are separated by a colon and saved in a file with an extension “.hdb”. [8]

![Figure 1 ClamAV’s full hash signature format](image)

As mentioned earlier using full hash signatures could be a liability that anti-virus software could not afford. That’s why ClamAV supports partial hash signatures in addition. These signatures are useful when the code of the executable does not change but the data it uses does. In this situation the partial hash signature is a good option to hash only the code section of the executable. ClamAV’s partial hash signature format is as follows: first the section name (".code"), the section hash and then the malware name.

![Figure 2 ClamAV’s partial hash signature format](image)

Once the signature is created it could be saved into a file with extension “.mdb”. [8]

### 2.2.1.2 Byte signature format

Byte signatures could represent any type of data such as code or data inside a data stream of an executable. The criterion for a good byte signature is that it should exist in multiple variants of malware from the same family. Hence it should minimize the virus free files that it detects as infected (false positives) and the infected files that it does not flag as malware (false negatives). Similarly to the hash signature, the byte signature format of ClamAV consists of three attributes separated by a colon. The first one is usually the malware family. The second one is a value for what types of files the engine to scan for. The value “0” means any file, meanwhile “1” is only for Portable Executable files (.exe, .dll, .acm etc.). The third attribute is a sequence of bytes the engine is going to look for in the file (the byte signature). [8]
2.2.2 ClamAV heuristic

Heuristic-based detection comes when the signature-based detection fails. New malware is created every day and anti-virus software should be able to detect it even when they do not have its exact signature. The heuristic engine scans for certain instructions or commands within executable that are not found in typical software applications. As a result anti-virus software vendors are able to detect potentially malicious functionality in previously uninspected files. For instance, it could identify the replication mechanism of a virus or the distribution of a worm. [9]

Generally there are two types of heuristic detection – a rule based and weight based system.

The weight –based system is one of the first heuristic that appeared. This type of system works by analysing the each of the functionality of the file, giving it a weight and adding it to a value. At the end that value is compared against a threshold and if it is higher the file is a malware. The really big drawback of that technology is that it causes a lot of false positives. For instance, if non-infected file has a lot of copy operations it is very likely to be flagged as malware. [9] For this project we focus on this kind of heuristic.

The rule based system is the more modern of the two and nearly all anti-virus applications use it nowadays. This type of heuristic uses a component of its engine to extract rules from the potentially malicious file and compare them against pre-defined set of rules. If a match occurs, the file could be considered malware. It is very important in rule-based systems the rules to be exact so that it minimizes the false positives. [9] ClamAV’s heuristic engine is based on a rule system. Developers continually add new rules to the existing set to detect new viruses. For instance, a malware that already has around 1000 signatures should be prioritized for a rule implementation. Keeping that much signatures and potentially having even more of the same malware instead of having a single rule really slows down the whole application and becomes an obstacle for the scalability. [10]

2.3 Anti-virus techniques nowadays

It is important to note that modern day malware detection techniques – signature-based detection and heuristic-based – are always used together. Using just one of the techniques will leave the user unprotected. For example if the user uses only signature detection he will be shielded from known malware but will be vulnerable to the new. It is the opposite with heuristics since it is impossible to have rules for all the malware that ever existed.

Commercial anti-virus software vendors make use of the emulation part of the heuristic detection together with the file analysis. Executing a file in a simulated environment and monitoring the actions it takes proves just as valuable as the analysis done on the file contents. It drastically reduces the false positives and increases the efficiency.
Chapter 3

3  Design

This chapter focuses on the design decisions taken throughout the development of the tool. It includes artefacts such as the requirements at the beginning of the project as well as how what approach did the author of this report take in term of software development.

3.1 Requirements

Due to the nature of the project giving the author a lot of possible paths of development the requirements had to be given priorities. The advantage of having prioritization at the beginning of the project is so that the high value/risk requirements are implemented. The aim is to minimize the cost but to maximize the value. The priorities are split into three main ones:

**High** - this gives the requirement a high priority. This means that all efforts should be focused on finishing this before starting any requirements that are labelled with the other categories. Finishing requirements with this priority is essential to the proper functioning of the system

**Medium** – medium priority is used when the requirement is necessary but not vital for the system. Not implementing a requirement with this priority will not affect the project in a major way.

**Low** – requirements labelled with this priority will only be implemented if the higher priority requirements are already implemented. These requirements could be considered out of scope.

3.1.1 Functional

Functional requirements represent the technical specification of the system. The following table shows what the system tries to accomplish:

<table>
<thead>
<tr>
<th>Priority</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Visualize the detected malware (Graphical User Interface)</td>
<td>The system must have a Graphical User Interface so that the user can run the malware scanning on his file system. Without the GUI the user will be unable to tell if a malware is detected.</td>
</tr>
</tbody>
</table>
High | Use Hash-signature based malware detection | Due to the project being about virus detection techniques this is necessary for the project to be considered a success.

High | Use Byte-signature based malware detection | This is the second type of signature-based detection that must be implemented for the system to function properly.

High | Manage malware | This type of requirement is about managing the found malware. This includes the delete/quarantine options. Without this the user will be left with no possible actions against the malicious files that were discovered.

Medium | Use weight-based heuristic malware detection | The heuristic detection in this project is important to achieve the author’s intended design. But it is not crucial for the system.

Low | Have network or web based protection | Web protection is out of the scope of this project and it was decided that it is unlikely to be implemented by the end of the project.

### 3.1.2 Non-Functional

In contrast with functional requirements, non-functional requirements are used to describe what the system should be, rather than what the system should do. They focus on the quality of the system. That being said, the following non-functional requirements were taken into consideration in this project:

<table>
<thead>
<tr>
<th>Priority</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Performance</td>
<td>The performance is fundamental for the application considering the size of the data being checked. Not focusing on performance could mean that the application will run for a really long period of time.</td>
</tr>
<tr>
<td>High</td>
<td>Scalability</td>
<td>This non-functional requirement is just as important as the performance one. Without scalability with the growing number of malware in the database the tool will be useless.</td>
</tr>
<tr>
<td>High</td>
<td>Reliability</td>
<td>The mean time between failures when using the system must be minimal.</td>
</tr>
<tr>
<td>Medium</td>
<td>Testability</td>
<td>The system structure should aim to be capable of being tested thoroughly with automated tests. This will improve Reliability.</td>
</tr>
</tbody>
</table>
3.2 Technologies

During the design phase of the project important decisions such as technologies needed for successful implementation had to be made. In this section the choices and the reasoning behind them are presented.

3.2.1 Programming languages

The programming languages used to develop the tool are Java 8 and C programming language. One of the main reasons for using Java for more than half of the project is because Java supports the software platform for creating and delivering desktop applications called JavaFX. Furthermore there was the possibility of providing the project with possibility of using concurrency by multi-threading.

The JavaFX software platform in the project is used to give a modern look and feel to the application. JavaFX is supposed to replace its current alternative library Swing as a new client User Interface library in the foreseeable future [11] which gives the project the opportunity to have a stylish look. The User Interface of the tool is implemented with the latest JavaFX version.

The decision behind choosing Java 8 instead of an older version is justified by the performance improvements of the Java Virtual Machine as well as the usage of lambda expressions. Lambda expressions are used to replace anonymous inner classes with more readable code. This improves the cleanliness of the code especially when you are implementing a Graphical User Interface. These are just some of the benefits of Java 8.

The C language part of the project focuses on the heuristic. This is due to the need to have access to the underlying operating system (Windows 7 in this project). Resources like registry values are monitored by modern day anti-virus software. Thus, using a low level language as C gives the opportunity to further develop this project into a well-functioning anti-virus application. In addition, there are no disassembling libraries available in Java, which is a drawback to the project since the heuristic part relies on such capabilities.

3.2.2 Orchestrate

Orchestrate is an API platform that provides a NoSQL databases-as-a-service for developing Web applications. The Orchestrate Representational State Transfer API stores and retrieves JSON objects in/from collections by giving them key/value format. [12] The collections would usually represent a table in a SQL database and they act as a list full of objects. The advantage of using Orchestrate as a database service is that it indexes objects and they could be searched in a record time. Due to the fact that the tool will perform a lot of searches from the database during runtime this fits perfectly with the project.

In addition Orchestrate provides the project with the option of scalability by having the capability to scale from prototype to tens of thousands of requests per second. [13] Thus, the adoption of Orchestrate as a database fulfils the requirements of scalability and performance.

The project database structure consists of two collections – “ByteSingature” and “HashSignature”.

Figure 4 shows how objects are represented in Orchestrate. The “collection” word shows you that this is the byte signature collection and if you look further down you could
see the “value” word which gives you the attributes of the object - the signature and the malware name.

```
"results": [ {
  "path": {
    "kind": "item",
    "collection": "BytesSignatures",
    "key": "0f9b8afa4c48f0fe",
    "ref": "bfbf489c5cc6c3a",
    "reftime": 1455568485964
  },
  "value": {
    "signature": "X501P%@AP[4\PZX54(P^)ZCC)7]$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*$",
    "malwareName": "Eicar-test-signature2"
  }
},
"reftime": 1455568485964
```

Figure 4 A byte signature in JSON format in Orchestrate

### 3.2.3 Version control

The version control of choice for the application development was Git. The reasoning behind using version control instead of storing the project’s artefacts on the file system is that it provides you with the opportunity to rollback to older versions of the code in case something goes wrong. Keeping track of changes in the code for projects like these is really important due to the possibility of breaking the system easily. In addition, it acts as a preventative measure in case a failure of the host computer occurs by storing the project on a remote server.

### 3.2.4 Java Native Interface

Java Native Interface is a programming interface used for writing Java Native methods and embedding the Java Virtual Machine into native applications. [14] JNI gives the project the opportunity to use Java and C in one application. On Figure 5 you can see how

![JNI in an application](image)

Figure 5 JNI in an application [15]
the interface acts as a bridge between the C side of the code and the Java side. The JNI enables Java applications to use native methods written in C to their advantage. For instance in this project Java calls the native methods who deal with the heuristic detection and then return a result based on what they have found. One of the big disadvantages of JNI is that once it is used with Java, the application loses its platform independency. This is the reason that the project only runs on Windows. The C code containing the java native methods is compiled into a library using that operating system and there is strong possibility of it not being compatible with other platforms.

3.3 The malware detection cycle

![Diagram of the malware detection cycle](image)

The malware detection cycle of the project consists of the three virus discovery techniques, file extraction from the file system and the Graphical User Interface. The process starts with the tool getting the desired file to be scanned. Then it passes it to the
hash signature detection, followed by byte signature and heuristic-based detection. At each step if the file is found to be malicious, the GUI is updated with information about the malware and the cycle is restarted. If the file goes through all three stages of detection it is not infected and the whole process can begin again. This is repeated until all the chosen directories/files are scanned. The reasoning behind choosing this order of the detection algorithms is to minimize the time taken for the system to find that the file is infected. Hash signature detection takes the least time since it only checks if the hash of the file matches any hash from the database. This is followed by the byte signature detection whose performance is slower. The usage of string matching algorithm to examine the bytes for an existing byte signature in the database makes this type of detection a good candidate for the second step of the cycle. The last stage is lacking the performance of the previous ones since it does processing of the assembly instructions of the suspected file. In addition if the heuristic detection finds a malware it updates the database with its hash signature and if the same malware is present again it will be caught earlier in the process. At Fig. 6, a diagram demonstrating the top-down structure of the cycle is shown.

The aim of this proposed approach is:
- To improve the performance of the tool by skipping detections whenever the file is already found as malicious.
- To enhance the effectiveness of the tool – a suspected file will go through all types of malware detections algorithms.
- To remove the need for malware signature updates by storing the signatures in a database rather than a file and automatically updating it.

This cycle has several drawbacks which had to be considered:
- It relies on the effectiveness and correctness of the heuristic analysis. There is a possibility of creating signatures for false positives which will lead to problems.
- All selected files from the user will go through the whole process even if they are not infected. The probability of having a regular file instead of a malware is much higher.

Possible improvements:
- Ideally the heuristic part of the system should create and update the database with byte signatures as well. Due to polymorphism (constantly changing structure) of modern day malware the hash signatures, which are completely different if the smallest change is made, are not as effective as the byte ones.
- Improving the heuristic algorithm to minimize false positives and false negatives would be a constant requirement for the tool to be relevant.
Chapter 4

4 Implementation

This chapter discusses the implementation specifics of the project. It gives high-level overview of the techniques used as well as some other interesting aspects of the project. It also gives the reader an outline of the key libraries that aided the creation of the tool.

4.1 Key libraries

Before reviewing the main and interesting parts of the implementation of the project it is important to go over the two major libraries that are essential for the tool to run.

Aho-Corasick java library

This java implementation of the Aho-Corasick string matching algorithm that was mentioned in the background section, was created by Robert Bor. The algorithm works by building up a construct called Trie from the all the patterns that the user wants to search for in the text. The Trie in the context of the previous explanation is the automaton. The algorithm consists of three main components: goto, fail and output. Once the Trie is ready the algorithm goes through the text. Each of the characters is checked if it matches a state in the Trie and if it matches it moves it to the next state with the goto function. If there is a character that is not matched with a state a fail function is invoked and it go back to states with less depth. Whenever a state that matches a whole keyword is reached it starts the output function which saves that word in an output set that could be accessed when the scan is completed. [16]

![Figure 7 Aho-Corasick Trie with keywords "ab", "bca" and "caa" [17]](image-url)
The significance of this library for the project is that it gives the string matching algorithm runs in O(n) complexity. This means that Aho-Corasick is the perfect choice for the byte signature detection.

**Libdasm 1.5**

Libdasm is a disassembler library which provides basic disassembly of Intel x86 instructions from a binary stream. In the scope of the project this Libdasm is really important part of the heuristic detection. The library is used to get the executable files’ instructions so that the heuristic algorithm can analyse them.

**4.2 Signature-based detection**

Two signature detection techniques were implemented throughout the project – hash and byte signature detection. Both of the techniques store the respective signatures in the project database. In this subsection the approaches as well as considerations taken when implementing the techniques are described.

The hash signature detection of the system is based on the Secure Hash Algorithm-256, which is a cryptographic hash function that has no found collisions. A hash function having collisions means that there is a possibility of the existence of two files with the same hash. This is compensated by the lack of known found collisions. A hash function having collisions means that it is vulnerable to false positives. Hence, if one file being infected has a hash signature, there exists the possibility of classifying another file with the same signature as malicious, despite it not being one. This is the primary reason for choosing SHA-256 instead of MD5 for this project.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Output size (bits)</th>
<th>Security (bits)</th>
<th>Example performance (MiB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5</td>
<td>128</td>
<td>&lt;64 (collisions found)</td>
<td>335</td>
</tr>
<tr>
<td>SHA-256</td>
<td>256</td>
<td>128</td>
<td>139</td>
</tr>
</tbody>
</table>

**Figure 8 Comparison between MD5 and SHA256**

The hash signature format in this project consists of three components – size of the infected file, the cryptographic hash and the malware name. These three attributes represent a hash signature object in the database. The detection in the tool works by querying the database for all hash signature objects that have the size of the file being scanned. If a list of objects is returned that means that the file might be infected. The next step is to compute cryptographic hash of the suspected file. Once this is finished, it can now be compared with the returned hash signatures from the database. If there is a match – the file is flagged as malware. The main advantage of this approach is that it takes into consideration the non-functional performance requirement by querying the database with the file size rather than the hash. The algorithm improves the performance of the tool since computing the SHA256 hash of each file would significantly slow down the process.
In contrast with hash signature detection, byte signature detection in the system works by using the Aho-Corasick pattern matching algorithm. The Trie (finite state automaton) is constructed from all byte signatures available in the database at the beginning of the scanning. Once the Trie is ready, the byte matching can start without querying the database.

### 4.3 Heuristic-based detection

The heuristic detection of the tool is implemented by using the Libdasm disassembly library. With its help, the system acquires the instructions of the executable. As mentioned in the Chapter 2, the developed heuristic is based on the weight system which gives weight to each functionality found and then checks if the sum is higher than the threshold. The particular difference of the current implementation is that it does not classify weight to each instruction. It only looks through them and if it encounters a delete instruction it gives it a weight of 1. Due to the high possibility of a malware containing such instructions the threshold is set to two.

### 4.4 Concurrency

Program concurrency was achieved by using java multithreading. By definition multithreading is the ability of a program to manage multiple user requests without the need of having multiple instances of the application. [19] In the context of the project it allows the program to scan multiple files at the same time. On Fig. 10 the tool’s concurrency
design is shown. Once a user presses the button to start the scanning a thread with a coordinating task in it is started. This on its turn kicks off multiple other threads (each containing a task) based on how many folders exist in the directory being scanned.

![Multithreading example](image)

4.5 Malware management

There are two ways of managing malware when it is found – deleting it or putting it under quarantine. Modern anti-virus software provides the used with both of these options. An infected file could be put under quarantine by making it harmless. Typical ways of doing that is using some kind of encryption. This tool uses Advanced Encryption Standard encryption to render the file useless. After encryption the malicious file cannot be executed. The key used to encrypt the suspected malware is stored on the user’s file system in the program’s directory. Whenever the user decides that he needs that file he can just unencrypt it by right clicking on the malware choosing the “Quarantine/Remove from Quarantine” option. The tool can put multiple files under quarantine at the same time.

4.6 User interface implementation

The user interface of the program was implemented using the JavaFX framework. The tool implements a tabbed pane look with three tabs – overview, scan and statistics. The overview and statistics tab fell out of scope throughout the project. The scanning tab contains three buttons each with different functionality:

- The quick scan button scans vulnerable directories on a Windows such as “Program Files” and “System 32”
The full scan button goes through the whole file system – including mounted devices. (USB drive, CD disk, etc.)

The custom scan button pops up a dialog asking the user to choose a file/directory.

Once the scan is ran a table appears giving information about the malware such as the name of the file and if the file is currently in quarantine. The table is updated in real time and the application is fully responsive. This is due to the multithreading functionality implemented. There is a scanning progress bar as well as stop and pause buttons. A “Back” button was added (Fig. 11 on the left side of the pause button) which stops the scanning process and returns the user back to the three scanning options.

![Screenshot of the tool while scanning](image)

**Figure 11** Screenshot of the tool while scanning
Chapter 5

5 Testing and evaluation

This chapter reviews the testing and evaluation approach taken while developing the tool.

5.1 Automated testing

Unit testing is a software development process used to test the smallest parts of an application called units. [20] These “units” are the smallest entities in the program and could represent a class or a method. It usually could be manual or automated. In this project automated unit testing framework for Java called JUnit was used. The purpose of this kind of testing is to have automated test for the most important parts of a project. It isolates the single parts of the program and assures that the individual parts are correct. [21] One of the benefits of unit testing is that it facilitates change which is particularly useful in this project since it gives the ability to tidy up code and make the implementation much cleaner. In the context of the tool development unit tests were created for features like the quarantine, stop/pause buttons which halt the threads, and file system traversal.

The limitations of unit testing are not being able to detect every error in the program. If the application being developed is too complex and has many units interacting with each other, unit testing appears to be unable to evaluate every execution path. That’s why in the next subsection an alternative approach to unit testing was used.

5.2 Manual Testing

Due to the nature of the project, manual testing was the primary of system correctness. A anti-malware test file called “EICAR”, developed by the European Institute for Computer Antivirus Research. It is an executable that when ran produces a message but also acts as a virus and is used to test commercial anti-virus software. [22] In the scope of the project the EICAR file was an immense help in the testing of the tool. Features such as uncompressing files and checking their contents for malware were tested by the provided “eicar.zip” files.

The hash signature-based detection techniques were manually tested by checking if the hash algorithm (SHA-256) computes the correct result. This was done by altering the contents of a file that already has a hash and rehashing the file. The two hashes were then compared and if they were different this meant that the algorithm works properly. Another test that was considered was spreading a file that has a hash signature in the database all over different directories and then running the tool on the parent folder/disk.

The byte signature detection had several different test cases. These included having a file that contains a byte signature as its content as well as mixing the byte signature with normal bytes.

The heuristic part of the project was the most challenging to test since finding a file that contains a “del” instruction in its machine code was difficult. There is still plenty of
work to be done in this area of the tool. Due to time limitations a full test of the technique was not possible. Such test would include writing example malware and researching into the techniques malware use to avoid anti-virus software.

5.3 Evaluation

The evaluation of the project was done by having different amount of signatures in the database. Due to the performance being important part of the project the tool’s performance was compared to some of the anti-virus scanning software. On Table 3 you can see a comparison with the free commercial anti-virus Avast. There are considerable amount improvements to be done in the efficiency of the tool as well as performance for it to achieve a commercial anti-virus software.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Size of directory/file</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avast</td>
<td>17.0 GB</td>
<td>30 seconds</td>
</tr>
<tr>
<td>Virado (the project tool)</td>
<td>17.0 GB</td>
<td>2 minutes 50 seconds</td>
</tr>
<tr>
<td>Avast</td>
<td>2.0 GB</td>
<td>5 seconds</td>
</tr>
<tr>
<td>Virado</td>
<td>2.0 GB</td>
<td>50 seconds</td>
</tr>
<tr>
<td>Avast</td>
<td>50 GB</td>
<td>1 minute 40 seconds</td>
</tr>
<tr>
<td>Virado</td>
<td>50 GB</td>
<td>7 minutes 24 seconds</td>
</tr>
</tbody>
</table>
Chapter 6

6 Conclusion

In this section conclusion is drawn about the project and what was achieved. In addition challenges encountered.

6.1 Challenges

The first challenge when developing the tool was estimating the amount of time it will take to implement particular features. This resulted in not fully implementing one of the main requirements – the heuristic-based detection. Although the technique was implemented it was not integrated with the rest of the system due to memory leak that appeared when transitioning from C language to Java. Another obstacle was designing the system. Due to the project being related to security software, there are not many sources of information how modern day anti-virus software companies optimize their applications.

Finally, implementing the multi-threading to interact with the User Interface appeared to be a considerable obstacle. Updating a table from a multi-threaded environment required me to think outside the box and

6.2 Gained knowledge

The author of this project had previous experience with Java and C. Although being more skilled in Java than C, the C part of the heuristic detection was implemented. Knowledge was gained throughout the development of the java multi-threading as well as the implementation of the User Interface using JavaFX. Additionally, the author learned about JNI, which he was inexperienced in. Another technical ability gained was the usage of NoSql databases.

Last but not least, the author acquired overall knowledge about how anti-virus software works and the main detection techniques that are used today.

6.3 Future work

The tool developed has potential of being optimized even further. In addition the heuristic algorithm has to be changed to a rule-based system which is the more up to date detection technique. Once that is implemented new rules for existing malware have to be created. Furthermore the database has to be filled with valid malware signatures. Features like the statistics tab and the overview tab have to be developed in order to give useful information to the potential customers. Partial hash could be added to boost the efficiency and a proper analyse component should be added so that it decides which parts of the file are most likely to be infected.
7 References


