Thesis no: BCS-2016-09

Dynamic Heuristic Analysis Tool for Detection of Unknown Malware

Maciej Sokol, Joakim Ernstsson

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Contact Information:
Authors:
Maciej Sokol
E-mail: maso13@student.bth.se

Joakim Ernstsson
E-mail: joea13@student.bth.se

University advisor:
Docent Henric Johnson
Department of Computer Science and Engineering

Faculty of Computing	Internet	:	www.bth.se
Blekinge Institute of Technology	Phone	:	+46 455 38 50 00
SE–371 79 Karlskrona, Sweden	Fax	:	+46 455 38 50 57

Abstract

Context: In today's society virus makers have a large set of obfuscation tools to avoid classic signature detection used by antivirus software. Therefore there is a need to identify new and obfuscated viruses in a better way. One option is to look at the behaviour of a program by executing the program in a virtual environment to determine if it is malicious or benign. This approach is called dynamic heuristic analysis.
Objectives: In this study a new heuristic dynamic analysis tool for detecting unknown malware is proposed. The proposed implementation is evaluated against state-of-the-art in terms of accuracy.
Methods: The proposed implementation uses Cuckoo sandbox to collect the behavior of a software and a decision tree to classify the software as either malicious or benign. In addition, the implementation contains several custom programs to handle the interaction between the components.
Results: The experiment evaluating the implementation shows that an accuracy of 90% has been reached which is higher than 2 out of 3 state-of-the-art software.
Conclusions: We conclude that an implementation using Cuckoo and decision tree works well for classifying malware and that the proposed implementation has a high accuracy that could be increased in the future by including more samples in the training set.

Keywords: Malware, dynamic analysis, decision tree, heuristic analysis

List of Figures

List of Tables

Contents

Chapter 1
Introduction

In today's society computers are used almost everywhere, we trust them to handle economic resources and personal information. There are some vicious individuals who take advantage of this situation without respect to ethics. These individuals are often called hackers who commit cyber crimes to gain money, discover company secrets, or steal personal information such as credit card numbers and bank accounts. Some of these cyber crimes are performed using malware.

Malware, also called malicious software, is any software which performs malicious actions such as gather sensitive information, prevent legitimate usage, or gain unauthorized access. Some common malware types are virus, worm, and trojan horse. The main difference between these types are the techniques used to spread to the victims [7]. New malicious programs are constantly being developed and traditional signature-based antiviruses can only detect previously known malware [6][44]. Therefore there is a need to identify and protect against new malware.

Heuristic malware analysis works by examining the behavior of a program which gives it the ability to find unidentified threats. There are two main methods for heuristic analysis in malware detection. The first method is dynamic analysis, it executes the program in a sandbox environment to identify suspicious behavior. The second method is static analysis where the examined program is disassembled and the code is analyzed to detect suspicious behavior. Since static analysers examine the code without executing it, malicious programs can use obfuscation to hide software features to remain undetected.

Signature-based antivirus works with a list of known virus signatures and finds malware by generating signature and comparing it against the known malware signatures. This means that only malware that have already been detected and had its signatures added to the list can be discovered by the antivirus. Malware can be polymorphic to get a new unknown signature and avoid detection from signature-based antivirus [39]. Polymorphy will be explained later in background section.

During this thesis an implementation of a heuristic dynamic analysis tool for detecting previously unknown malware is performed. The objective of the thesis is to determine if an implementation using Cuckoo and decision tree can manage to produce even better results than the state-of-the-art due to the high amount of data that Cuckoo collects about a program's execution.

In the beginning of the thesis a background study is performed that presents previously developed implementations of heuristic malware analysis as well as different techniques that can be used for similar systems. This part of the study helped us understand how dynamic analysis works and how it can be implemented. In the second part a heuristic dynamic analysis tool has been implemented based on the knowledge gained from the first part. In the third part the research question RQ "How does the proposed implementation using Cuckoo and decision tree compare with the state-of-the-art antivirus tools in terms of accuracy when classyfing software?" is given an answer. This is done through a quasi-experiment where a sample of malicious and benign software was analysed by the implementation. Quasi-experiment was chosen because of difficulties with selecting subjects randomly. The implementations performance was then compared against the state-of-the-art and evaluated.

Clarifications of terms used:
True positives(TP): amount of programs correctly classified as malware.
True negative(TN): amount of programs correctly classified as benign.
False positive(FP): amount of benign programs incorrectly classified as malware.
False negative(FN): amount of malware incorrectly classified as benign.
Accuracy: measurement of how well the implementation classifies unknown software using the following formula:(TP+TN)/(TP+TN+FP+FN) [40]. Accuracy of 1 means that all programs have been classified correctly.

1.1 Challenges of the Field

This part of the thesis explains some problems that can be encountered by others while developing a heuristic dynamic analysis tool. One of the problems is to find enough benign and malicious samples which are used for training and experiment. Samples which are used for training need to differentiate from the samples used for evaluation. Another problem which may occur is extracting behaviour data from the analysed software, some of the software produced only a small amount of behaviour data which could not be used for classification. The behaviour data from the analysed software was formatted in json, json files could be up to 1 gigabyte big so it was hard to find a suitable json library to parse the file. Most of the libraries that were examined either crashed or consumed too much time while reading the json file.

1.2 Structure of the Thesis

Chapter 1 Introduction briefly explains the current situation in the fight against malware as well as introduces the proposed implementation which is evaluated in the study.
Chapter 2 Background and related work introduces static, dynamic heuristic, and related concepts. The first part walks through and explains the overall procedure of heuristic analysis. The chapter also presents related work and gives an overview of how the implementations work and how they perform.
Chapter 3 Research Design describes motivation and aim of the research as well as research methodology which was used.
Chapter 4 Implementation of the System goes into detail of how the proposed implementation works and presents all of its components. The order of execution is explained and the reasoning behind the selection of components are presented.
Chapter 5 Experiment Design explains experiment design, which samples are included and which state-of-the-art software are compared with the proposed implementation.
Chapter 6 Results presents results of the experiment.
Chapter 7 Analysis and Discussion presents analysis of the results and answers the research question. The advantages and disadvantages of implementation are discussed as well as fields where the proposed implementation could be applicable.
Chapter 8 Conclusions and Future Work describes the conclusion of the study by summarizing the results and presents future improvements to the implementation.

Chapter 2
Background and Related Work

A lot of research have been conducted to develop accurate malware detection for use in modern antivirus software. There is a constant arms race between malware creators and security experts. Malware creators develop new malware and techniques to avoid detection and security experts refine the techniques used for identifying malware. One of those techniques is called heuristic analysis. Heuristic analysis is divided into two steps, the first step is to collect the behaviour of a program. This can be done through static or dynamic analysis using some kind of feature collection technique. The main problem when collecting features is that malware can be obfuscated to prevent static analysis. The second step is to analyse the collected behaviour and draw conclusions whenever the evaluated program is malicious or benign. This is often called data mining. The concepts are explained in detail in their corresponding sections.

2.1 Heuristic Analysis Methods

The heuristic analysis methods can be divided into two categories, static analysis, and dynamic analysis. In this section we will describe those methods and discuss their advantages and disadvantages.
The static analysis method is a process of analysing the program's code without executing it. It is usually done by disassembling the program to translate the binary code into corresponding assembly instructions. Several static binary analysis techniques have been developed to detect different types of malicious code [20][21]. The advantage of static analysis is that it can cover the complete code of the program and is usually faster than its counterpart dynamic analysis. Static analysis can be easily evaded by using obfuscation that makes it harder to analyse the code. One of the big disadvantages is that we can not be sure if the analysed code is the code that will actually run, that is easily achievable by polymorphic, and metamorphic techniques as well as packed executables that unpacks themselves during runtime.
Dynamic analysis method works by examining the program during runtime. Such analysis can be dangerous to the executing computer and as a result dynamic analysis is often performed in a protected environment such as virtual machine

or emulation [15]. The main advantage of dynamic analysis is immunity to polymorphic, metamorphic, and packed executables. The program that is analysed is actually the program that would run in a real environment. As a trade-off to the execution of the program is time, compared to static analysis. The biggest disadvantage of dynamic analysis is the detection of simulated environment. Malware can often detect that they are being run in a simulated environment and change their behavior accordingly [35][17]. As of now there does not exist any publicly available virtual machines which are undetectable by malware.

2.2 Feature Collection

To perform a heuristic analysis the behaviour of a program needs to be collected first. In dynamic heuristic analysis, the behaviour data from the executing program needs to be collected and recorded in some way. In this section we present some methods for collecting software behaviour which have been used in previous research.

2.2.1 API Hooking

API Hooking is a method for collecting software features at runtime with the goal to negate the damage when testing malware. Another advantage with collecting features using API hooking is that it makes it harder for malware to detect. API hooking has been implemented successfully in a prior study where in-line code overwriting was used. In-line code overwriting is a process where DLL's API calls are overwritten in main memory to redirect them to the hook function [41].

2.2.2 N-grams

N-gram is a collection method which is both simple and highly scalable to use. N-grams are splitting up the given input into smaller parts. There are several types of n-grams which decide on how big the separated data should be [25]. Given the input "this is a test", using words as unit and N-gram of size 1 will produce the following output:"this,is,a,test", using N-gram of size 2 produces: "this is, is a, a test" and N-gram of size 3 produces: "this is a, is a test". N-grams are often used in static heuristic analysis to extract the behaviour of the samples where the assembly code is split using N-grams.

2.3 Obfuscation

Obfuscation is used by malware creators to prevent reverse engineering and avoid signature detection. There are many automatic tools to obfuscate code called

packers and crypters. Reports from antivirus vendors show that 80% of discovered malware is packed, creating a need to counter obfuscation [34]. For a obfuscated malware to be executed it must at some point decrypt/unpacks itself meaning that a dynamic heuristic analysis will still work on obfuscated malware. Static heuristic analysis can be done on an obfuscated malware but since they vary greatly in how they look they can be hard to classify. Alternatively the malware is decrypted or unpacked by the analyser so that the original executable can be examined.
Packers work by compressing an executable and wrapping a small unpacker around it, the unpacker unpacks the executable at runtime and transfers execution to it. Similarly an encrypted executable is first decrypted at runtime and then executed normally [39].Packing executables can be used for non-malicious purposes, for example by reducing the size of an executable or to protect intellectual property. However since such a large part of malware are obfuscated, knowing if an executable is packeted can be useful when determining if software is malicious.

Another method for malware to avoid detection is to use the knowledge that virtual environments and debuggers are used for heuristic dynamic analysis to its advantage. So if the malware detects a virtual environment or a debugger it then changes its own behaviour to stop the analysis to see any malicious behaviour. One example of a behaviour change is to not decrypt the malicious code segment and instead terminate the process. Malware can detect that it is running in a virtual environment since the virtualization never is perfect and there are many signs to look for. For example looking for a communication channel in the form of hardware ports that the virtual machine use for communication between the guest and host OS[19]. Another way is to look for differences caused by the fact that the guest uses the same memory as the host but mapps global items differently. Particular examples include the Interrupt Descriptor Table (IDT), the Global Descriptor Table (GDT), and the Local Descriptor Table (LDT) [35].
Some more advanced methods of obfuscation are metamorphic and polymorphic malware. Polymorphic malware are encrypted and contain randomly generated decryptors that will result in new signature for each malware instance. Metamorphic malware modifies the body of the code without changing the functionality. One method for doing this is via host code mutation where the binary code of a malware is disassembled and generates new binary code with the same functionality [37].
Many studies have been done to determine if a program is packed or not. One method is to look at the entropy of an executable since packed software are often more random [33]. Another method is to look for signatures of known packer tools or to look for structural differences associated with packed software. One study successfully used steganalysis where executables were mapped to bitmap images and the images were used as an input to a machine learning algorithm

which decided if the sample was packed [18].

2.4 Classification of Software

Data mining is the concept of generating knowledge or collect patterns from large sets of data. One type of data mining algorithms is classifiers. The goal of classifier algorithms is to determine what class a sample belongs to. Classifiers have many implementations so the sample can be anything, the classifier can decide between two or more classes depending on the context. One of the applications for a classifier algorithm is for deciding if a software is malicious or benign [26][27].

Examples of data mining algorithms used in research for detecting malware include Naive bayes, decision tree, MaTR, and Support vector machine. We will present a general explanation of these four methods in the following section.

2.4.1 Naive Bayes

Naive Bayes is a probabilistic method that is used for information retrieval and text classification. Naive bayes estimates probability of each class and the conditional probability of each attribute value given the class. Probability is estimated by counting the frequency of occurrence of the classes and the attribute values for each class using the training data. Assuming conditional independence of the attributes it uses Bayes rule to compute the probability of each class given an unknown instance [31].

2.4.2 Decision Tree

To detect malware using data mining the decision tree method can be used and several experiments have developed and tested implementations using this method [25][36].The decision tree works by using a training set of attributes and their values collected from benign and malicious software. The algorithm generates a tree by choosing an attribute that best describes the class and splitting the tree based on the value of the attribute and match the tree nodes to the given training elements class. This process is repeated recursively for the rest of the attributes. For subsequent training elements the tree is split only when it would generate more precise decision rules. When deciding the class of an unknown element the tree is traversed by comparing the test elements value with the attribute value of the node. The class is then determined at the end node [31][38].

2.4.3 MaTR

Malware Type Recognition (MaTR) is an extension of regular statistical malware detection methods that focuses on deciding the type of a malware for example backdoors, trojans, and worms. MaTR works in two steps first it detects if a program is malicious or benign using a decision tree. Secondly it classifies the type of malware using another decision tree with a class for each malware type to identify. The two decision trees use different sets of features to suit their purpose [22]. T Dube, R Raines, M Grimaila, K Bauer, and S Rogers performed a study where they compared MaTR, n-grams, and 3 commercial antivirus programs and found MaTR and N-grams far superior. Authors conducted an experiment to determine the scan time of each of the methods(besides N-grams) and found commercial antivirus to be a lot slower than MaTR, the fastest antivirus had a scan time of 43 seconds while MaTR had 0.9 seconds. The experiment also showed that MaTR had far superior TP rate of 0.93 compared to the best performing antivirus TP rate of 0.46 [24]. A similar research has been conducted where KM-based N-grams(N-gram method proposed by Kolter and Maloof) and MaTR were evaluated against commercial antivirus in terms of TP rate [31]. Both KM and MaTR used static heuristic analysis to extract the sample features and decision tree as a classification algorithm.The experimental results show that KM N-grams and MaTR detection models were found more appealing where both of the methods had a TP rate of over 0.9 while the commercial antivirus programs barely reached 0.47 [23].

2.4.4 SVM

Support vector machine also called SVM is a learning algorithm used for classifying which out of two classes an example belongs to. SVM maps the training set of parameters from the two classes to points in a space and divides them with a gap. In software classification API calls can be used as the parameters for the mapping. To classify new examples the algorithm maps them onto the same space and classifies them by looking at which side of the gap they fall [45].

2.5 Heuristic Implementations in Previous Research

2.5.1 Vaccine

Research have been done where heuristic dynamic analysis was used together with VMware to collect malware behavior. In the research, the VMware was using a Linux machine as host OS and Windows as guest OS on top of the virtual PC. The guest OS served as execution environment for the malware. The authors have also setup a DNS to ensure that network-based infections were possible. The connection between the host OS and guest OS worked as follow:

send the program to be examined from the host to the guest. Guest executes the program and collects the behavior. The gathered behavior is sent to the host OS and analysed to determine whenever the analysed software was a malware. If the analysed program has been marked as malware a vaccine was automatically developed based on the behavior of the malware to reverse its effects and remove it from the guest system [30].

2.5.2 Feature vector

Y Hu, L Cheng, M Xu, N Zheng, Y Guo present Argus [43]. Argus is a dynamic analysis system to detect unknown malware which is using a 35-dimensional feature vector where each dimension stands for a behavior feature. The Argus consists of several parts: (1) an executable is added into sample-database in PE-format. (2) the executable is transferred to the VMware machine where an API-Tracer is running and captures API calls. (3) the data is mapped into the feature vector. Argus is using Naive Bayes to classify the samples and an accuracy ranging from 70% to 89,9% has been reached.

J Ding, J Jin, P Bouvry, Y Ho, H Guan are presenting a very similar solution to Argus [28]. A software system was implemented that was using 35-dimension feature vector to dynamically extract the features. A VMware was used to capture the behavior. Afterwards a statistical method and MoE(model and mixture of experts) model developed by authors were used to determine if the program was malicious. The results show a True Positive(TP) of 96% using the statistical method and 75% TP using the MoE method compared to 70-80% of commercial heuristic scanners.

2.5.3 Dynamic instruction sequences

J Dai, R Guha, J Lee are proposing a dynamic analysis system where a VMware machine is used for creating classification models. The proposed system is capturing instruction sequences during runtime in debugging mode and logs the binary code. Afterwards the system disassembles the binary code to get the assembly code. The assembly code is used to generate the logic assembly by removing duplicated code. The logic assembly code is then simplified to create an abstract assembly code. Abstract assembly code is used for feature selection. Thereafter the frequent instruction associations are selected so that it is clear if a specific instruction is associated with benign or malicious software. Lastly, the classification models are generated using SVM and decision trees, and the sample is evaluated [29].

2.5.4 TTAnalyze

U Bayer, A Moser, C Kruegel, Engin Kirda are presenting a dynamic analysis system that uses emulation [17]. The system uses an external tool called TTAnalyze for analysing the executables and an emulation software called Qemu [11].The authors chose to emulate an entire computer system to achieve greater emulation accuracy at the cost of performance. TTAnalyze was used for monitoring the emulated environment to collect the behavior of the sample and produce an analysis report. The authors then compared the analysis report with Kaspersky's description of the malware. The produced analysis report matched the description to a great extent.

2.5.5 Packer

L L Chuan, C L Yee, M Ismail, K Jumari proposes an implementation that combines a packer detector, packer unpacker, static heuristic analysis, and emulation for dynamic heuristic analysis [32]. The system starts with examining the entry point of the sample and determines if the sample has been packed and with which packer, then the sample is unpacked based on the detected packer signature. Afterwards a static heuristic analysis module is used to collect the software signatures and checks if they are in the signature database. If the signature exists in the database then the sample is seen as malicious otherwise it is sent to the emulator. The emulated environment disassembles the code dynamically and performs just-in-time compilation for the host OS. The signatures are extracted and compared with the signature database, based on the outcome the sample is marked as malicious or benign.

Chapter 3

Research Design

3.1 Research Motivation

Malicious programs are destructive and can cause both financial and personal damage for the victims. The obfuscation techniques of malware are becoming more sophisticated and as a result signature-based and static analysis anti-malware tools are not detecting all of the malicious programs. Malware can be included in a legitimate program which can cause confusion in static and signature-based anti-malware tools which may lead to the malware avoiding detection. Therefore there is a need for an anti-malware tool which can detect new and obfuscated malware.

3.2 Aims and Objectives

The aim of the study is to develop a heuristic dynamic malware analysis tool that can detect unknown malware and generate as few false positives and false-negatives as possible. Furthermore we aim for the generated patterns to be representative of malware behavior in general. The objective of the thesis is to determine if an implementation using Cuckoo and decision tree can manage to produce even better results than the state-of-the-art due to the high amount of data that Cuckoo collects about a program's execution.

3.3 Research Question

RQ1: How does the proposed implementation using Cuckoo and decision tree compare with the state-of-the-art antivirus tools in terms of accuracy when classyfing software?

3.4 Research Method

The material for background research was gathered using the snowballing method described by C. Wohlin [42]. Snowballing is a method for systematic literature

study which consists of two main steps. The first step is to gather a start set which is done by picking keywords and deciding on a search string to use in database searches. The resulting articles are then evaluated and the relevant ones are selected based on a set of rules called the inclusion criteria. Based on the keywords, a search string was constructed which was used in database search.
The second part is iterative and consists of backward and forward snowballing. Backward snowballing is to examine all the references of an article to find relevant texts to use in the study. Forward snowballing refers to finding new articles by looking at articles citing the examined articles. This is done iteratively for new articles identified in either backward or forward snowballing. The process is completed when no more relevant articles are found and all identified articles have been examined.

For this study the reference database Inspec was used. A search was performed and relevant articles have been selected based on the inclusion criteria, title, and abstract.
Keywords: *heuristic analysis, dynamic analysis, static analysis, malware detection, malware monitoring*
Search string used for startset: *(heuristic OR heuristic analysis OR heuristic method) AND (static OR dynamic OR monitoring) AND (virus OR malware)*
Inclusion criteria:

- Articles written after 1990
- Articles written in English, Swedish, or Polish
- Articles related to malware analysis
- Journal articles, conference articles, or conference proceedings.

Chapter 4

Implementation of the System

In this section we describe the design of the system and how it works. First we present an overview of the system architecture. Next section describes the components of the system in depth and their relations with each other. Then the order of execution of the components is presented. In the end we describe why we have chosen those components that our system consists of.

4.1 System Architecture

The implementation is constructed using existing tools as well as custom developed applications. The system can be split up into two parts, collection, and analysis. The collection part is handled by Cuckoo and our own developed applications. The analysis part is managed by a learning algorithm called Decision tree. See figure 4.1 for an overview of the system architecture.

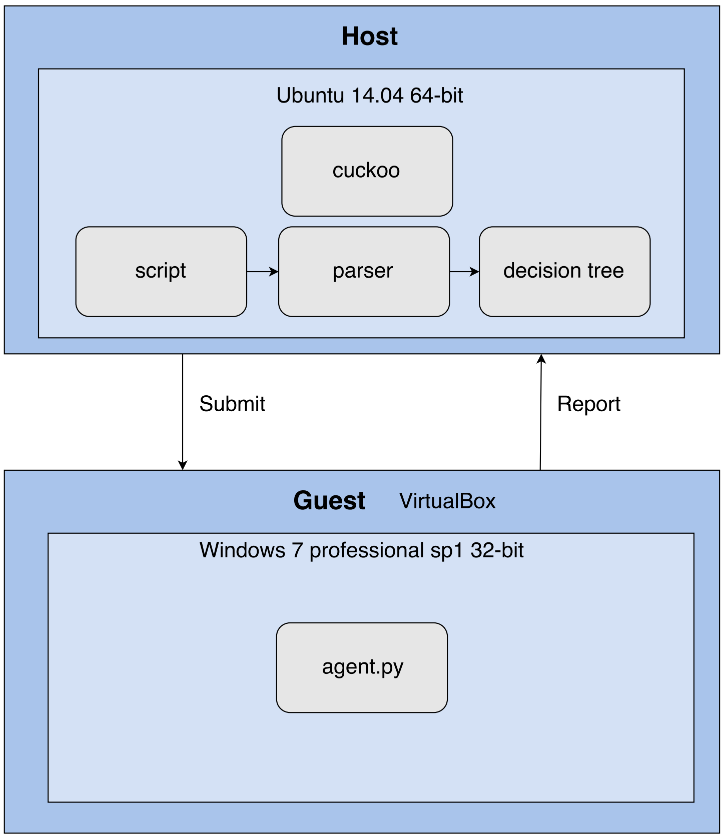

Figure 4.1: Overview of the system architecture

4.2 System Components

4.2.1 Cuckoo Sandbox

Cuckoo Sandbox v2.0-rc1 is a behaviour collection tool which consists of two parts. The execution part consists of running a sample in a virtual environment and reporting the behaviour to the host using the supplied reporting program. The behaviour is collected by using a method called API hooking(descrived in background) although Cuckoo only records what the analysed sample does and does not negate the damage, therefore a virtual environement is needed. The second part is analysis where the collected behaviour is parsed into a json file. After the execution of the sample the virtual machine is restored automatically to a snapshot which is taken before the execution [4].

4.2.2 Virtual Environment

In the implementation, virtualbox v5.0.16 has been used to host the virtual environment [13]. The virtual machine is running Windows 7 32-bit which has UAC and Windows-firewall disabled to enable malicious activities. Python has been installed to enable communication to the Cuckoo [10]. The virtual machine is isolated so it can only communicate with the host.

4.2.3 Parser

A parser was implemented in programming language C to collect data from the json report file generated by Cuckoo. To work with json files in C we use cJSON, a lightweigth open source json library for C [3]. The parser works with one report file at the time and generates a temporary file containing all the selected behavioral attributes of the subject presented in chronological order. The selected attributes are API calls and related data which are used in the classification algorithm to classify the samples as either malicious or benign. The selected attributes are described in table 4.1. *See Appendix 3 for parser code.*

Type	**Name**	**Description**
Calls	API	The name of the Windows API-call
Arguments	dirpath	Path to a directory
	function_name	Name of the function which is calling the API
	filepath	Path to a file
	filepath_r	Path to a file
	hostname	Target address for a connection
	library	Library loaded
	module	Module loaded, .dll files
	module_name	Name of the module loaded
	port	Port used for a connection
	regkey	Path of a registry key
	regkey_r	Path of a registry key
	sectionname	PE binary section
	url	Target address on the web
	value	Values submitted with the API-call
flags	flags	Flags used when making the API-call

Table 4.1: Parsed attributes

4.2.4 Shell Script

The shell script automates the execution of the parser for each malicious and benign subject. The parsed reports are then used to generate the necessary files for the decision tree in the correct format. *See Appendix 4 for the shell scripts.*

4.2.5 Decision Tree

The decision tree is constructed using C5.0 which is a tool for generating classifiers. C5.0 works with two types of files, ".names" files, and ".data" files. The names file contains a list of all the attributes of a subject that will be used in classification and the possible classes of subjects. The attributes need to have a name and a description of what values it can be assigned. In the implementation the attributes can be either true(t) or false(f) since we are only interested in the existence of attributes and not the frequencies. The second required file type are data files, there is one data file for each subject and they contain the values of the attributes described in the names file [2]. To train the decision tree a behaviour analysis of 100 malicious and 100 benign programs have been performed. The behaviour files are handled by the parser and shell script. The malicious programs used for training are collected from VX Heaven and the benign programs are collected from Softonic [16][12]. Additional software with a known class (malicious or benign) can be used with the decision tree after the initial training to potentially increase the accuracy of the prediction. *See Appendix 1 for samples used in training.*

4.3 Order of Execution

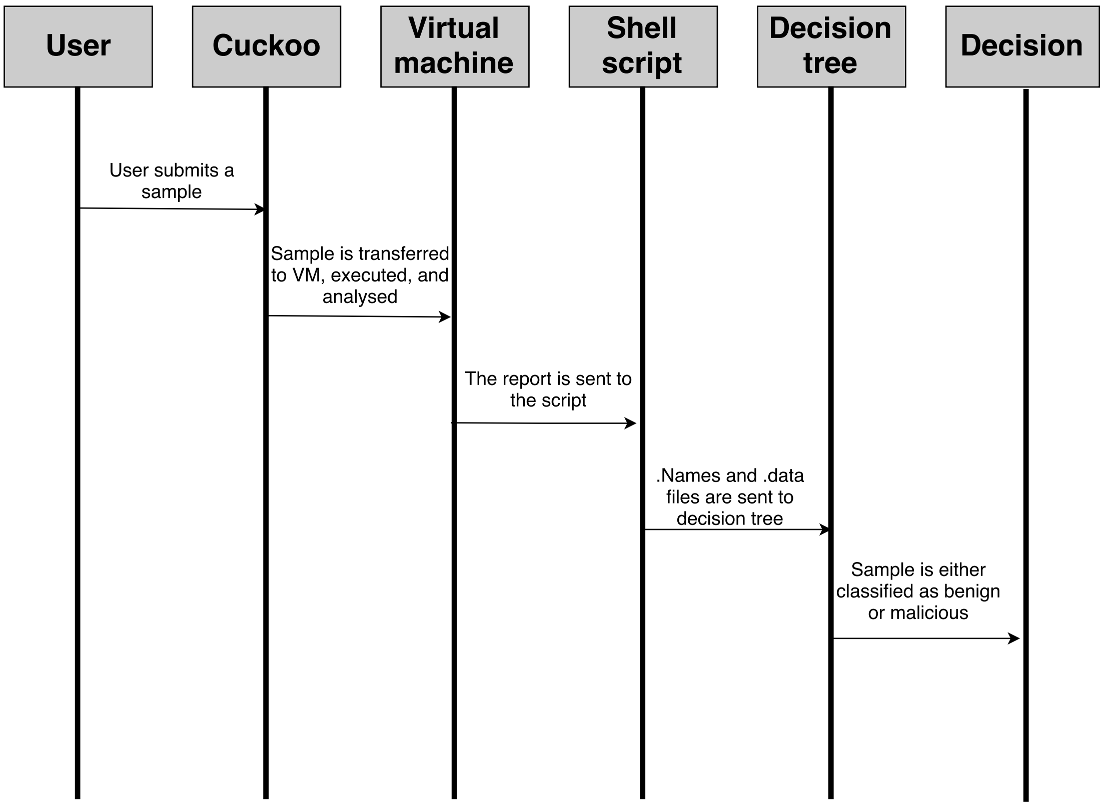
User
Cuckoo
Virtual machine
Shell script
Decision tree
Decision
User submits a sample
Sample is transferred to VM, executed, and analysed
The report is sent to the script
.Names and .data files are sent to decision tree
Sample is either classified as benign or malicious

Figure 4.2: Order of execution for one sample

The execution of the system starts with submitting the samples to Cuckoo. After the sample submission Cuckoo creates a list of applications to be analysed. The analysis begins with Cuckoo restoring the virtual machine to a previously captured snapshot and then it starts the virtual machine. When the virtual machine is started Cuckoo transfers the sample to the virtual machine and the application is executed. The executed application is analysed and a report file is produced with the behaviour specifications. Afterwards the shell script is executed that passes the report file to the parser which extracts selected behaviour attributes. Names file and data file are produced and decision tree can start classification. Based on knowledge gained from training decision tree decides whether the analysed sample is malicious or benign. All of the explained steps can be seen in figure 4.3.

4.4 Motivation of Selected System Components

The programs which are included in the system have been selected mainly because of two reasons, firstly all the selected components are open source and available to anyone, secondly the selected components are effective in their corresponding workfields. Cuckoo is already a powerful tool for dynamically collecting a software's behaviour and is constructed for analysis of malware. Furthermore, no heuristic malware detection software have been implemented using Cuckoo to collect the behaviour data so implementing and analysing a system doing so is interesting for the research field, in the related work CWSandbox was used instead although it is not open source. C5.0 is a complete implementation for generating decision trees and evaluating test cases, it was selected because it is written in C programming language which enabled us to get a better understanding of the algorithm. CJSON library have been selected because of its simplicity and short execution time, all other libraries which were tested were either inefficient or could not handle large files.

Chapter 5

Experiment Design

5.1 Experiment Scoping

The objects studied in the experiment are malware analysis tools. The purpose of the experiment is to determine whether the proposed implementation will perform better than the state-of-the-art in regards to accuracy. The experiment will be performed as a quasi-experiment because it is problematic to randomly select malicious and benign samples for this experiment.

5.2 Experiment Planning

5.2.1 Context Selection

The experiment will be performed off-line by the authors. The studied phenomena is general since it involves malware and it addresses a real problem which is malware analysis.

5.2.2 Variable Selection

The experiment has only one independent variable. The independent variable is an analysis tool that is used for evaluating executables. The analysis tools which are evaluated: the proposed implementation, Avast, Kaspersky, and Malwarebytes [1][5][8]. State-of-the-art software that implement a heuristic detection method have been selected to enable comparison with the proposed implementation. We have chosen to include three software products to represent the state-of-the-art. The amount of software to use in the experiment was decided by balancing between the amount of software which are necessary to represent the state-of-art and time constraints of the study. The state-of-the-art software are the latest versions available at the time of the experiment: Avast v11.2.2261, Kaspersky v16.0.0614(d), and Malwarebytes v2.2.1.1043. The dependant variable is the accuracy of the analysis tool which is defined as: (TP+TN)/(TP+TN+FP+FN).

5.2.3 Subject Selection

The subjects used in the experiments are executables which are either benign or malicious. The malicious samples are collected from an online repository VX Heaven [16], VX Heaven state that the malicious samples are actually malicious. The benign samples are collected from a reliable online source Softonic as well as applications which are included in Windows 7 [12]. The collected benign samples are scanned by reliable online anti-malware tool Virustotal to ensure they do not contain any malicious activity [14]. The chosen samples are distinct from the training samples to ensure more reliable results. *See appendix 2 for a list of selected subjects.*

5.2.4 Design Type

The experiment will have one factor with one treatment for each of the examined analysis tools.
The same subjects will be tested against all treatments to prevent incorrect results which may be caused by differences in specific subject characteristics. In addition, subjects are divided into two groups malicious and benign to allow measurement of the accuracy.

5.2.5 Instrumentation

To conduct the experiment we have set up one computer to host a virtual machine. The virtual machine is hosted by virtualbox v5.0.16 [13]. The proposed implementation will be run on the host and state-of-the-art will be run inside the virtual environment since more anti-malware programs are available on Windows operating system. See table 5.1 for specifications of the systems.

	Guest	Host
OS	Windows 7 professional sp1 32-bit	Ubuntu 14.04 64-bit
RAM	4 GiB	15,6 GiB
CPU	1 Core of “Intel Core i7-5557U 3.10GHz”	Intel core i7-5557U 3.10GHz x4

Table 5.1: Specifications of Guest and Host

5.2.6 Validity Discussion

Regarding the conclusion validity there needs to be a measurable difference in accuracy between every analysis tool, otherwise a relationship between the treatment and the outcome would not be observable. If there is no diffrence in accuracy between the analysis tools then more samples should be included in the experiment.
Some state-of-the-art analysis tools combine different detection methods, we need

to ensure that the detection method is comparable to the proposed implementation. The proposed implementation will be trained and tested with separate sets of softwares to assure that the test subjects are unknown to the implementation. However this can not be done for the state-of-the-art as it can not be controlled if the subjects have been used in any heuristic training or signature generation. This means that the measured accuracy of the state-of-the-art may not represent the accuracy when analysing completely unknown malware.
The state-of-the-art consist of commercial software and only three of them will be used in the study. This could mean that the selected softwares are not representative of the state-of-the-art in general.
The study looks at the accuracy of analysis tools. Whether the measured accuracy is correct is dependent on the analysed benign and malicious software. The analysed software must be of the correct class meaning the benign programs should not be malicious. The reliability of the measured accuracy will also increase if the amount of analysed software is increased. In addition, a larger training set for the decision tree would further increase the validity of the experiment.

5.3 Experiment Operation & Data Collection

5.3.1 Experiment Operation

Experiment will be performed by following the guidelines:

1. Malware and benign software are collected from previously described sources.

2. Benign samples are scanned using Virustotal to ensure they are not malicious.

3. All subjects are analysed by one of the analysis tools and the results are recorded. The process is repeated for each treatment.

4. Accuracy is calculated for each of the analysis tools.

5. Accuracy of each analysis tool is evaluated.

5.3.2 Data Collection

Data is collected by manually running the analysis tools with all subjects and the result of the analysis is collected. All results are then compared to the class of the specific subjects (malicious or benign) and the results are recorded as FP, TP, FN, or TN and the accuracy of each analysis tool is calculated. For each analysis tool a table will be used to record the results which consists of four columns.

Chapter 6 Results

In this section results from the experiment are presented. The results are divided into separate sections for each of the analysis tools. Malicious and benign subjects are separated in two tables to clearly illustrate the relation between true positives(TP) and false negatives(FN) as well as true negatives(TN) and false positives(FP). The tables show the name of the analysed software as well as the result generated from the sample analysis.

6.1 Results for the proposed implementation

Malicious subject name	Analysis result	TP/FN
Virus.Win32.Etap	Malware	TP
Virus.Win32.HLLC.Asive	Malware	TP
Virus.Win32.HLLP.Vampore	Malware	TP
Virus.Win32.HLLW.Veedna	Malware	TP
Virus.Win32.Mohmed.4607	Malware	TP
Virus.Win32.Plutor.b	Benign	FN
Virus.Win32.Folcom.b	Benign	FN
Virus.Win32.Rufoll.1432	Malware	TP
Virus.Win32.Seppuku.4827	Malware	TP
Virus.Win32.Slicer.poly	Malware	TP
Virus.Win32.Stepar.g	Malware	TP
Virus.Win32.VB.bq	Malware	TP
Virus.Win32.VB.fs	Malware	TP
Virus.Win32.VB.jn	Malware	TP
Virus.Win32.Delf.u	Malware	TP
Virus.Win32.Parite.b	Benign	FN
Virus.Win32.Savior.1740	Malware	TP
Virus.Win32.Higway.b	Malware	TP
Virus.Win32.Xorer.ab	Malware	TP
Virus.Win32.Yerg.9571	Malware	TP

Table 6.1: Experiment results for the proposed implementation(Malware)

From the table 6.1 we can clearly see that most malware samples have been classified correctly. We can see that malicious samples which were classified incorrectly are *Plutor.b*, *Folcom.b*, and *Parite.b*.

Benign subject name	Analysis result	TN/FP
AirDroid_Desktop_Client_3rdmarket.exe	Benign	TN
calibre-2.55.0.ms	Benign	TN
DiscordSetup.exe	Benign	TN
DittoSetup_3_21_50_0.exe	Benign	TN
Everything-1.3.4.686.x86-Setup.exe	Benign	TN
FFSetup3.8.0.0.exe	Benign	TN
FileZilla_3-14-1_win64-setup.exe	Benign	TN
FreeraserSetup.exe	Benign	TN
Keypass-1.31-setup.exe	Benign	TN
KindleForPC-Installer-1.15.43.061.exe	Benign	TN
MyPublicWifi.exe	Benign	TN
Rainmeter-3.3.1.exe	Benign	TN
Thunderbird Setup 45.0.exe	Benign	TN
install_virtualdj_pc_v8.1.2857.msi	Benign	TN
kdewin-installer-gui-1.0.0.exe	Benign	TN
kochbuch-1.7.1.exe	Benign	TN
mirc732.exe	Benign	TN
msgr11us.exe	Benign	TN
superunitconverter2_setup.exe	Benign	TN
Fences2-cnet-setup.exe	Malware	FP

Table 6.2: Experiment results for the proposed implementation(Benign)

Table 6.2 shows that almost all the benign samples have been classified correctly. Only *Fences2-cnet-setup.exe* has been missclassified by the proposed implementation.
From the previously presented tables we can summarize the detection rate and calculate accuracy for the proposed implementation.
TP rate: 17
TN rate: 19
FP rate: 1
FN rate: 3
Accuracy of the proposed implementation:
(TP+TN)/(TP+TN+FP+FN)=(17+19)/(17+19+1+3)=0,9.

6.2 Results for Malwarebytes

Malicious subject name	Analysis result	TP/FN	TP/FN Prop. impl.
Virus.Win32.Etap	Benign	FN	TP
Virus.Win32.HLLC.Asive	Benign	FN	TP
Virus.Win32.HLLP.Vampore	Benign	FN	TP
Virus.Win32.HLLW.Veedna	Benign	FN	TP
Virus.Win32.Mohmed.4607	Benign	FN	TP
Virus.Win32.Plutor.b	Benign	FN	FN
Virus.Win32.Folcom.b	Malware	TP	FN
Virus.Win32.Rufoll.1432	Benign	FN	TP
Virus.Win32.Seppuku.4827	Benign	FN	TP
Virus.Win32.Slicer.poly	Benign	FN	TP
Virus.Win32.Stepar.g	Benign	FN	TP
Virus.Win32.VB.bq	Benign	FN	TP
Virus.Win32.VB.fs	Malware	TP	TP
Virus.Win32.VB.jn	Benign	FN	TP
Virus.Win32.Delf.u	Benign	FN	TP
Virus.Win32.Parite.b	Malware	TP	FN
Virus.Win32.Savior.1740	Benign	FN	TP
Virus.Win32.Higway.b	Benign	FN	TP
Virus.Win32.Xorer.ab	Malware	TP	TP
Virus.Win32.Yerg.9571	Benign	FN	TP

Table 6.3: Experiment results for the state-of-the-art Malwarebytes(Malware)

Table 6.3 shows a high amount of FN. 16 malicious samples have been missclassified by Malwarebytes compared to proposed implementation's 3. Worth noting is that *Folcom.b* and *Parite.b* which have been missclassified by the proposed implementation have been classified correctly by Malwarebytes.

Benign subject name	Analysis result	TN/FP	TN/FP Prop. impl.
AirDroid_Desktop_Client_3rdmarket.exe	Benign	TN	TN
calibre-2.55.0.ms	Benign	TN	TN
DiscordSetup.exe	Benign	TN	TN
DittoSetup_3_21_50_0.exe	Benign	TN	TN
Everything-1.3.4.686.x86-Setup.exe	Benign	TN	TN
FFSetup3.8.0.0.exe	Benign	TN	TN
FileZilla_3-14-1_win64-setup.exe	Benign	TN	TN
FreeraserSetup.exe	Benign	TN	TN
Keypass-1.31-setup.exe	Benign	TN	TN
KindleForPC-Installer-1.15.43.061.exe	Benign	TN	TN
MyPublicWifi.exe	Benign	TN	TN
Rainmeter-3.3.1.exe	Benign	TN	TN
Thunderbird Setup 45.0.exe	Benign	TN	TN
install_virtualdj_pc_v8.1.2857.msi	Benign	TN	TN
kdewin-installer-gui-1.0.0.exe	Benign	TN	TN
kochbuch-1.7.1.exe	Benign	TN	TN
mirc732.exe	Benign	TN	TN
msgr11us.exe	Benign	TN	TN
superunitconverter2_setup.exe	Benign	TN	TN
Fences2-cnet-setup.exe	Benign	TN	FP

Table 6.4: Experiment results for the state-of-the-art Malwarebytes(Benign)

The results from the table 6.4 show that all the benign samples have been classified correctly by Malwarebytes compared to the proposed implementation where 1 benign sample was classified as malicious.
From the table 6.3, and 6.4 we can summarize the detection rate and calculate accuracy for Malwarebytes.
TP rate: 4
TN rate: 20
FP rate: 0
FN rate: 16
Accuracy of Malwarebytes:
(TP+TN)/(TP+TN+FP+FN)=(4+20)/(4+20+0+16)=0,6.

6.3 Results for Avast

Malicious subject name	Analysis result	TP/FN	TP/FN Prop. impl.
Virus.Win32.Etap	Malware	TP	TP
Virus.Win32.HLLC.Asive	Malware	TP	TP
Virus.Win32.HLLP.Vampore	Malware	TP	TP
Virus.Win32.HLLW.Veedna	Benign	FN	TP
Virus.Win32.Mohmed.4607	Malware	TP	TP
Virus.Win32.Plutor.b	Malware	TP	FN
Virus.Win32.Folcom.b	Malware	TP	FN
Virus.Win32.Rufoll.1432	Malware	TP	TP
Virus.Win32.Seppuku.4827	Malware	TP	TP
Virus.Win32.Slicer.poly	Benign	FN	TP
Virus.Win32.Stepar.g	Malware	TP	TP
Virus.Win32.VB.bq	Benign	FN	TP
Virus.Win32.VB.fs	Benign	FN	TP
Virus.Win32.VB.jn	Malware	TP	TP
Virus.Win32.Delf.u	Benign	FN	TP
Virus.Win32.Parite.b	Malware	TP	FN
Virus.Win32.Savior.1740	Malware	TP	TP
Virus.Win32.Higway.b	Malware	TP	TP
Virus.Win32.Xorer.ab	Malware	TP	TP
Virus.Win32.Yerg.9571	Malware	TP	TP

Table 6.5: Experiment results for the state-of-the-art Avast(Malware)

By analysing the table 6.5 we can see that the malicious samples which have been missclassified by Avast, *HLLW.Veedna*, *Slicer.poly*, *VB.bq*, *VB.fs*, and *Delf.u* have been classified correctly by the proposed implementation. The missclassified malicious samples *Plutor.b*, *Folcom.b*, and *Parite.b* by the proposed implementation have been classified correctly by Avast.

Benign subject name	Analysis result	TN/FP	TN/FP Prop. impl.
AirDroid_Desktop_Client_3rdmarket.exe	Benign	TN	TN
calibre-2.55.0.ms	Benign	TN	TN
DiscordSetup.exe	Benign	TN	TN
DittoSetup_3_21_50_0.exe	Benign	TN	TN
Everything-1.3.4.686.x86-Setup.exe	Benign	TN	TN
FFSetup3.8.0.0.exe	Benign	TN	TN
FileZilla_3-14-1_win64-setup.exe	Benign	TN	TN
FreeraserSetup.exe	Benign	TN	TN
Keypass-1.31-setup.exe	Benign	TN	TN
KindleForPC-Installer-1.15.43.061.exe	Benign	TN	TN
MyPublicWifi.exe	Benign	TN	TN
Rainmeter-3.3.1.exe	Benign	TN	TN
Thunderbird Setup 45.0.exe	Benign	TN	TN
install_virtualdj_pc_v8.1.2857.msi	Benign	TN	TN
kdewin-installer-gui-1.0.0.exe	Benign	TN	TN
kochbuch-1.7.1.exe	Benign	TN	TN
mirc732.exe	Benign	TN	TN
msgr11us.exe	Benign	TN	TN
superunitconverter2_setup.exe	Benign	TN	TN
Fences2-cnet-setup.exe	Benign	TN	FP

Table 6.6: Experiment results for the state-of-the-art Avast(Benign)

As seen in the table 6.6, all the benign samples have been classified correctly compared to proposed implementation missclassification of *Fences2-cnet-setup.exe*.
From the previously presented tables we can summarize the detection rate and calculate accuracy for Avast.
TP rate: 15
TN rate: 20
FP rate: 0
FN rate: 5
Accuracy of Avast:
(TP+TN)/(TP+TN+FP+FN)=(15+20)/(15+20+0+5)=0,875.

6.4 Results for Kaspersky

Malicious subject name	Analysis result	TP/FN	TP/FN Prop.impl.
Virus.Win32.Etap	Malware	TP	TP
Virus.Win32.HLLC.Asive	Malware	TP	TP
Virus.Win32.HLLP.Vampore	Malware	TP	TP
Virus.Win32.HLLW.Veedna	Malware	TP	TP
Virus.Win32.Mohmed.4607	Malware	TP	TP
Virus.Win32.Plutor.b	Malware	TP	FN
Virus.Win32.Folcom.b	Malware	TP	FN
Virus.Win32.Rufoll.1432	Malware	TP	TP
Virus.Win32.Seppuku.4827	Malware	TP	TP
Virus.Win32.Slicer.poly	Malware	TP	TP
Virus.Win32.Stepar.g	Malware	TP	TP
Virus.Win32.VB.bq	Malware	TP	TP
Virus.Win32.VB.fs	Malware	TP	TP
Virus.Win32.VB.jn	Malware	TP	TP
Virus.Win32.Delf.u	Malware	TP	TP
Virus.Win32.Parite.b	Malware	TP	FN
Virus.Win32.Savior.1740	Malware	TP	TP
Virus.Win32.Higway.b	Malware	TP	TP
Virus.Win32.Xorer.ab	Malware	TP	TP
Virus.Win32.Yerg.9571	Malware	TP	TP

Table 6.7: Experiment results for the state-of-the-art Kaspersky(Malware)

As we can see from the table 6.7 all the malicious samples have been classified correctly by Kaspersky compared to missclassification of *Plutor.b*, *Folcom.b*, and *Parite.b* by the proposed implementation.

Benign subject name	Analysis result	TN/FP	TN/FP Prop. impl.
AirDroid_Desktop_Client_3rdmarket.exe	Benign	TN	TN
calibre-2.55.0.ms	Benign	TN	TN
DiscordSetup.exe	Benign	TN	TN
DittoSetup_3_21_50_0.exe	Benign	TN	TN
Everything-1.3.4.686.x86-Setup.exe	Benign	TN	TN
FFSetup3.8.0.0.exe	Benign	TN	TN
FileZilla_3-14-1_win64-setup.exe	Benign	TN	TN
FreeraserSetup.exe	Benign	TN	TN
Keypass-1.31-setup.exe	Benign	TN	TN
KindleForPC-Installer-1.15.43.061.exe	Benign	TN	TN
MyPublicWifi.exe	Benign	TN	TN
Rainmeter-3.3.1.exe	Benign	TN	TN
Thunderbird Setup 45.0.exe	Benign	TN	TN
install_virtualdj_pc_v8.1.2857.msi	Benign	TN	TN
kdewin-installer-gui-1.0.0.exe	Benign	TN	TN
kochbuch-1.7.1.exe	Benign	TN	TN
mirc732.exe	Benign	TN	TN
msgr11us.exe	Benign	TN	TN
superunitconverter2_setup.exe	Benign	TN	TN
Fences2-cnet-setup.exe	Benign	TN	FP

Table 6.8: Experiment results for the state-of-the-art Kaspersky(Benign)

From the table 6.8, we can see that all the benign samples have been classified correctly compared to proposed implementation missclassification of *Fences2-cnet-setup.exe*.
From the previously presented tables, table 6.7, and table 6.8, we can summarize the detection rate and calculate accuracy for Kaspersky.
TP rate: 20
TN rate: 20
FP rate: 0
FN rate: 0
Accuracy of Kaspersky:
(TP+TN)/(TP+TN+FP+FN)=(20+20)/(20+20+0+0)=1,0.

Fold	Errors
1	7,7%
2	15,4%
3	0,0%
4	23,1%
5	7,7%
6	0,0%
7	0,0%
8	0,0%
9	0,0%
10	0,0%
Mean	5,4%
SE	2,6%

Table 6.9: Results from 10-fold cross-validation

6.5 Evaluation of decision tree

The decision tree was evaluated using 10-fold cross-validation using the training samples. The samples are divided into ten blocks with an equal amount of malicious and benign samples. A classifier is generated using nine of the blocks and the classifier is evaluated with the remaining block (hold out data). The process is repeated leaving out one new block each time until all the samples have been used for classification. Results produced during cross-fold validation can be seen in table 6.9, the table shows error rate which is the rate of missclassifications generated when classfiying the hold out data, mean of the error rate, and SE which stands for standard error of the mean.

6.6 Summary of the results

The results have shown that all of the state-of-the-art software had none false positives while the proposed implementation had 1. The proposed implementation had higher accuracy on the experiment samples than Avast and Malwarebytes, see table 6.10. Results also indicate that the missclassified samples by state-of-the-art differ from the missclassified samples by the proposed implementation.

Tool:	Accuracy(%):
Kaspersky	100%
The proposed implementation	90%
Avast	87,5%
Malwarebytes	60%

Table 6.10: Accuracy of the tools

Chapter 7
Analysis and Discussion

In this section the results of the experiment are discussed and the proposed implementation is evaluated against the state-of-the-art to answer the research question. The accuracy of implementations in related work are compared to the proposed implementation. In addition, the advantages and disadvantages of the proposed implementation are discussed.

The purpose of this study was to evaluate the proposed implementation against the state-of-art as well as determining if an implementation using cuckoo and decision tree can be effective for finding new malware. The classifier has been evaluated through 10-fold cross-validation and an experiment together with the state-of-the-art. During cross-validation the mean error rate has been measured to 5,4%. The accuracy of the classifier is calculated as follows: 1-0,054 = 0.94,6 which means an accuracy of 94,6%. During some of the folds the error rate was much higher than the others, for example during 4th fold the error rate was 23,1%. This high error rate is probably because there is an uneven distribution of samples with similar behavior between the test group and the training group. The accuracy of classifier during the 10-fold cross-validation is higher than during the experiment. The accuracy is not directly comparable since the number of samples, as well as the proportion of samples used in training and testing differ between cross-validation and the experiment. However, the measured accuracy during cross-fold validation is close to the accuracy measured in the experiment which supports the validity of the results.
In the experiment the implementation has the second best accuracy of the tested tools. The measured accuracy of the implementation was 90%, the misclassifications were more commonly false negatives with 15% of malware classified as benign and 5% of the benign samples classified as malicious. Worth noting is that the samples which were used in training differ from the samples used in the experiment. Malwarebytes had the lowest accuracy of all the tested tools. Only 4 out of 20 malicious samples have been classified correctly. Worth noting is that 2 out of the 4 correctly classified malicious samples by Malwarebytes have been misclassified by the proposed implementation. This means that the detection method of Malwarebytes differs from the detection method of the proposed

implementation. Although Malwarebytes implement their own heuristic analysis they could not detect half of the malicious samples which is surprising. Malwarebytes do state that *"Three proprietary technologies—signature, heuristics, and behavior—automatically guard you and your online experience from malware that antivirus products don't find. Real-time protection detects and shields against the most dangerous forms of malware."* [9] so it is possible that they are focusing on finding new malware and in the experiment known malware were used. Regarding Avast it had an accuracy of 87,5% compared to the proposed implementations 90%. Avast classified 15 out of 20 malicious samples correctly and all the samples which were misclassified by Avast were classified correctly by the proposed implementation. The samples which were misclassified by Avast have little behaviour data which may not have been enough for Avast to classify them as malicious, especially if Avast wants to avoid any false positives. It is unexpected that Avast and Malwarebytes did not classify all the malicious samples correctly, the samples used in the experiment are available online for anyone to download so Avast and Malwarebytes could have taken signatures of them. It is worth noting that the misclassified samples by the proposed implementation have been classified correctly by both Avast and Malwarebytes, this could mean that the proposed implementations training did not include enough samples of those malware types or malware with similar behavior. Kaspersky antivirus managed to get an accuracy of 100% but since the malware samples were collected from an online resource it is likely that Kaspersky has signatures of the samples. The recorded accuracy of state-of-the-art is not representative of accuracy while analysing unknown samples since we do not know if the state-of-the-art have taken signatures of the samples. In addition, the method used by the state-of-the-art for classification of samples is unknown, meaning that both signature and heuristics analysis could have been used in classification. We have ensured that the samples which were used in the experiment were unknown to the proposed implementation by having separate sets of samples for training and the experiment. *See appendix 1 for training samples and appendix 2 for experiment samples.*

For the state-of-the-art none of the examined tools generated any false positives. For the creators of anti-malware products the goal is to detect all malware to protect the customer's computer. From the user's perspective any false positives can be irritating when trying to execute a benign software and a high rate of false positives could make the tool unusable. It is important when designing a heuristic malware detection tool to consider how strict the detection algorithm should be and what rate of false positives are acceptable.

Previous research in dynamic heuristic analysis have achieved similar levels of accuracy, one of the solutions presented in related work, Argus got accuracy ranging from 72.52% to 89,9% which is slightly lower than the proposed implementation. In this case it must be taken into consideration that the measurements were made with a different and larger set of samples compared to the proposed implementation [43].

There are some points of the proposed implementation that should be discussed. A disadvantage of the proposed implementation which needs to be considered is analysis, some of the software generated only a small amount of behaviour data. Particularly some of the malware made only a couple of API calls before terminating which makes it difficult for the decision tree to make accurate classifications based on small amounts of data. This could mean that the proposed implementation would not perform well in detecting malware that alters their behaviour when they run in virtual environments. However if a software tries to alter their behaviour in a virtual environment, that behaviour can be used as a section in the decision tree which can be applicable to detect malware. Another drawback with the implementation that needs to be brought up is the execution time. Training the implementation requires execution of a large set of samples in a virtual environment which can vary greatly in execution time between the samples. Secondly the generated data is very large so parsing and extracting the correct data for the decision tree as well as generating the tree is very time-consuming. The time to classify software is faster and based on the time to classify the samples in the experiment the execution time is acceptable for running in a lab environment. To use the implementation as an antivirus before executing a program would be too slow to be practical. The implementation could be used together with other antivirus solutions to identify new malware and generate signatures.

Chapter 8
Conclusions and Future Work

In the study we propose an implementation for classification of unknown malware. The proposed implementation consists of Cuckoo sandbox, decision tree, and custom developed applications. Furthermore we evaluate the proposed implementation against state-of-the-art through a quasi-experiment where 20 benign and 20 malicious programs are examined by the tools. The proposed implementation is found more accurate than two out of three state-of-the-art tools with an accuracy of 90%. The proposed implementation may at this stage not be ready for commercial use although this study can be used as a framework for future research and commercial implementations. Based on the experiment results we can draw a conclusion that an implementation using Cuckoo together with decision tree can perform well in classifying malicious and benign software.

We are confident that the results are a good demonstration of the proposed implementations accuracy, the results could be made more statistically secure by conducting an experiment with a larger set of samples and perform a cross-validation in the experiment for both the proposed implementation and for the state-of-the-art. In this study we were not able to do cross-validation when examining state-of-the-art due to time restrains so it should be considered for future work. The accuracy of the implementation could potentially be increased with a larger training set. Since the decision tree only splits a node if new valuable information for classification can be gained, any irrelevant data will be filtered out and all additional data gained from a larger training set would be beneficial. An additional approach to improve the accuracy of the proposed implementation is to collect more behavioral data of software. In particular to collect data from malware that alter their behaviour when executing in virtual environments. One possible solution is to implement measures to prevent malware from knowing that they are executing in virtual environments. Alternatively static behaviour collection could generate more data than dynamic in these cases, so a static collection method could be used in addition to Cuckoo. Another improvement can be made in the future with a signature-based detection to improve detection of previously known malware.

References

[1] Avast, free antivirus. https://www.avast.com/sv-se/index. [Online; accessed 26-April-2016].

[2] C5.0 decision tree. https://www.rulequest.com/see5-unix.html. [Online; accessed 11-April-2016].

[3] cjson, json parsing library for c. https://github.com/kbranigan/cJSON. [Online; accessed 11-April-2016].

[4] Cuckoo sandbox. https://www.cuckoosandbox.org/. [Online; accessed 24-February-2016].

[5] Kaspersky lab, antivirus software. http://www.kaspersky.com/se/. [Online; accessed 26-April-2016].

[6] Kaspersky security bulletin 2015. https://securelist.com/files/2015/12/Kaspersky-Security-Bulletin-2015_FINAL_EN.pdf. [Online; 02-February-2016].

[7] Malware definition. https://en.wikipedia.org/wiki/Malware. [Online; accessed 07-April-2016].

[8] Malwarebytes, free anti-malware and internet security. https://www.malwarebytes.org/. [Online; accessed 26-April-2016].

[9] Malwarebytes, free anti-malware and internet security. https://www.malwarebytes.org/antimalware/. [Online; accessed 04-June-2016].

[10] Python. https://www.python.org/. [Online; accessed 05-April-2016].

[11] Qemu, open source processor emulator. http://wiki.qemu.org/Main_Page. [Online; accessed 30-March-2016].

[12] Softonic. http://en.softonic.com/windows. [Online; accessed 05-April-2016].

[13] Virtual box. https://www.virtualbox.org/. [Online; accessed 05-April-2016].

[14] Virustotal. `https://www.virustotal.com/`. [Online; accessed 05-April-2016].

[15] Vmware. `http://www.vmware.com/`. [Online; accessed 01-April-2016].

[16] Vx heaven. `http://vxheaven.org/`. [Online; accessed 24-February-2016].

[17] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic Analysis of Malicious Code. *Journal in Computer Virology*, 2(1):67–77, May 2006.

[18] C. Burgess, F. Kurugollu, S. Sezer, and K. McLaughlin. Detecting packed executables using steganalysis. In *2014 5th European Workshop on Visual Information Processing (EUVIP), 10-12 Dec. 2014*, 2014 5th European Workshop on Visual Information Processing (EUVIP). Proceedings, page 5 pp. IEEE, 2014.

[19] M. Carpenter, T. Liston, and E. Skoudis. Hiding virtualization from attackers and malware. *IEEE Security & Privacy*, 5(3):62–5, May 2007.

[20] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium, 4-8 Aug. 2003*, 12th USENIX Security Symposium, pages 169–86. USENIX Assoc., 2003.

[21] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant. Semantics-aware malware detection. In *Proceedings. 2005 IEEE Symposium on Security and Privacy, 8-11 May 2005*, Proceedings. 2005 IEEE Symposium on Security and Privacy, pages 32–46. IEEE Comput. Soc., 2005.

[22] T. Dube, R. Raines, G. Peterson, K. Bauer, M. Grimaila, and S. Rogers. Malware Type Recognition and Cyber Situational Awareness. In *2010 IEEE Second International Conference on Social Computing (SocialCom 2010). the Second IEEE International Conference on Privacy, Security, Risk and Trust (PASSAT 2010), 20-22 Aug. 2010*, Proceedings of the 2010 IEEE Second International Conference on Social Computing (SocialCom 2010). the Second IEEE International Conference on Privacy, Security, Risk and Trust (PASSAT 2010), pages 938–43. IEEE Computer Society, 2010.

[23] T. Dube, R. Raines, G. Peterson, K. Bauer, M. Grimaila, and S. Rogers. Malware target recognition via static heuristics. *Computers & Security*, 31(1):137–47, February 2012.

[24] T.E. Dube, R.A. Raines, M.R. Grimaila, K.W. Bauer, and S.K. Rogers. Malware Target Recognition of Unknown Threats. *IEEE Systems Journal*, 7(3):467–77, September 2013.

[25] Thomas E. Dube. A Novel Malware Target Recognition Architecture for Enhanced Cyberspace Situation Awareness. Technical report, September 2011.

[26] Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining : concepts and techniques*. Elsevier/Morgan Kaufmann, Waltham, Mass., 2012.

[27] Jau-Hwang Wang, P.S. Deng, Yi-Shen Fan, Li-Jing Jaw, and Yu-Ching Liu. Virus detection using data mining techinques. In *Proceedings. IEEE 37th Annual 2003 International Carnahan Conference on Security Technology, 14-16 Oct. 2003*, Proceedings. IEEE 37th Annual 2003 International Carnahan Conference on Security Technology (IEEE Cat. No.03CH37458), pages 71–6. IEEE, 2003.

[28] Jianguo Ding, Jian Jin, P. Bouvry, Yongtao Hu, and Haibing Guan. Behavior-based proactive detection of unknown malicious codes. In *2009 Fourth International Conference on Internet Monitoring and Protection (ICIMP 2009), 24-28 May 2009*, 2009 Fourth International Conference on Internet Monitoring and Protection (ICIMP 2009), pages 72–7. IEEE, 2009.

[29] Jianyong Dai, R. Guha, and Joohan Lee. Efficient virus detection using dynamic instruction sequences. *Journal of Computers*, 4(5):405–14, May 2009.

[30] R. Koike, N. Nakaya, and Y. Koi. Development of system for the automatic generation of unknown virus extermination software. In *2007 International Symposium on Applications and the Internet, 15-19 Jan. 2007*, 2007 International Symposium on Applications and the Internet, page 7 pp. IEEE Comput. Soc., 2007.

[31] Jeremy Z. Kolter and Marcus A. Maloof. Learning to Detect Malicious Executables in the Wild. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 470–478, New York, NY, USA, 2004. ACM.

[32] Lee Ling Chuan, Chan Lee Yee, M. Ismail, and K. Jumari. Design and development of a new scanning core engine for malware detection. In *2012 18th Asia-Pacific Conference on Communications (APCC), 15-17 Oct. 2012*, 2012 18th Asia-Pacific Conference on Communications (APCC), pages 770–4. IEEE, 2012.

[33] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2):40–5, March 2007.

[34] M Morgenstern and Hendrik Pilz. Useful and useless statistics about viruses and anti-virus programs. In *Proceedings of the CARO Workshop*, 2010.

[35] Alfredo Andres Omella. Methods for virtual machine detection. 2006.

[36] Ravinder R. Ravula, Kathy J. Liszka, and Chien-Chung Chan. Learning Attack Features from Static and Dynamic Analysis of Malware. In Ana Fred, Jan L. G. Dietz, Kecheng Liu, and Joaquim Filipe, editors, *Knowledge Discovery, Knowledge Engineering and Knowledge Management*, number 348 in Communications in Computer and Information Science, pages 109–125. Springer Berlin Heidelberg, October 2011. DOI: 10.1007/978-3-642-37186-8_7.

[37] Sanjam Singla, Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Detecting and classifying morphed malwares: A survey. *International Journal of Computer Applications*, 122(10), 2015.

[38] G.G. Sundarkumar and V. Ravi. Malware detection by text and data mining. In *2013 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), 26-28 Dec. 2013*, 2013 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), page 6 pp. IEEE, 2013.

[39] S. Treadwell and Mian Zhou. A heuristic approach for detection of obfuscated malware. In *2009 IEEE International Conference on Intelligence and Security Informatics (ISI), 8-11 June 2009*, 2009 IEEE International Conference on Intelligence and Security Informatics (ISI), pages 291–9. IEEE, 2009.

[40] Jau-Hwang Wang, P. S. Deng, Yi-Shen Fan, Li-Jing Jaw, and Yu-Ching Liu. Virus detection using data mining techinques. In *IEEE 37th Annual 2003 International Carnahan Conference on Security Technology, 2003. Proceedings*, pages 71–76, October 2003.

[41] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security & Privacy*, 5(2):32–9, March 2007.

[42] Claes Wohlin. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pages 38:1–38:10, New York, NY, USA, 2014. ACM.

[43] Yongtao Hu, Liang Chen, Ming Xu, Ning Zheng, and Yanhua Guo. Unknown malicious executables detection based on run-time behavior. In *2008 Fifth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), 18-20 Oct. 2008*, volume vol.4 of *2008 Fifth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, pages 391–5. IEEE, October 2008.

[44] D. Zenkin. Fighting against the invisible enemy. Methods for detecting an unknown virus. *Computers & Security*, 20(4):316–21, 2001.

[45] Bo-yun Zhang, Jian-ping Yin, Jin-bo Hao, Ding-xing Zhang, and Shu-lin Wang. Using support vector machine to detect unknown computer viruses. *International Journal of Computational Intelligence Research*, 2(1):100–104, 2006.

Chapter 9

Appendix

9.1 Appendix 1

Table 9.1: **Samples used for training**

Name	**Class**
32bit_Win7_Win8_Win81_R275.exe	Benign
7z920.exe	Benign
advanced-systemcare-setup.exe	Benign
Apache_OpenOffice_4.1.1_Win_x86_install_en-US.exe	Benign
aresregular240_installer.exe	Benign
audacity-win-2-1-2.exe	Benign
avast_free_antivirus_setup_online.exe	Benign
avc-free.exe	Benign
AVSMediaPlayer.exe	Benign
BitComet_1.40_x86_setup.exe	Benign
Bluetooth-QuickInstaller-1.0.0.983.exe	Benign
CamStudio_Setup_v2-6b_r294.exe	Benign
ccsetup514.exe	Benign
cdbxp_setup_4.5.5.5571.exe	Benign
Connectify2016Installer.exe	Benign
dap10_full.exe	Benign
debutsetup.exe	Benign
driver_booster_setup.exe	Benign
DriverEasy_Setup.exe	Benign
DropboxInstaller.exe	Benign
DTLiteInstaller.exe	Benign
dvdshrink32setup.exe	Benign
dxwebsetup.exe	Benign
eMule0.50a-Installer.exe	Benign
ExcelViewer.exe	Benign

Continued on next page

Table 9.1 – continued from previous page

Name	Class
Facebook-Pro.exe	Benign
fast-video-converter-3.8.0.4.exe	Benign
Firefox-Setup-45-0_EN.exe	Benign
FoxitReader734_enu_Setup_Prom.exe	Benign
freepdfreader.exe	Benign
frostwire-6.2.3.windows.fusion.exe	Benign
GeoGebra-Windows-Installer-5-0-223-0.exe	Benign
gimp-2.8.16-setup.exe	Benign
GOMPLAYERGLOBALSETUP.EXE	Benign
gu5setup.exe	Benign
hjsplit.exe	Benign
HSS-5.2.2-install-hss-805-ext.exe	Benign
icq_rfrset.exe	Benign
icytower_install.DM.exe	Benign
ifunbox_setup.exe	Benign
IKEA_Home_Planner_FY10.exe	Benign
instagram-app.exe	Benign
Install JDownloader.exe	Benign
instsf449.exe	Benign
iTunesSetup.exe	Benign
JAD8014_BASIC.exe	Benign
Kies3Setup.exe	Benign
K-Lite_Codec_Pack_1205_Full.exe	Benign
KMPlayer_4.0.6.4.exe	Benign
LenovoSHAREitSoftonic.exe	Benign
LGPCSuiteIV_Setup.exe	Benign
LineInst.exe	Benign
MecaNet.exe	Benign
MKVPlayerSetupD.exe	Benign
msgr11us.exe	Benign
MT_Install.exe	Benign
New_PC_Studio_1.5.1.10064_2.exe	Benign
Nokia_PC_Suite_ALL.exe	Benign
npp.6.8.7.Installer.exe	Benign
office2007sp3-kb2526086-fullfile-en-us.exe	Benign
OperaSetup.exe	Benign
OriginThinSetup.exe	Benign
paint.net.4.0.9.install.exe	Benign

Continued on next page

Table 9.1 – continued from previous page

Name	Class
PDFCreator-2_3_0-Setup.exe	Benign
PhotoScape_V3.6.2.exe	Benign
PowerPointViewer.exe	Benign
putty.exe	Benign
QuickTimeInstaller.exe	Benign
RazerCortexSetup_7.1.14.12241.exe	Benign
rcsetup152.exe	Benign
revosetup.exe	Benign
RocketDock-v1.3.5.exe	Benign
SafariSetup.exe	Benign
setup_2.9.5.exe	Benign
SetupDWGTrueView2013_32bit.exe	Benign
SkypeSetup.exe	Benign
Songr_2_0_2378_Setup.exe	Benign
SP27608.exe	Benign
Spotify Installer.exe	Benign
SteamSetup.exe	Benign
SUPERsetup.exe	Benign
SweetHome3D-5.2-windows.exe	Benign
TeamSpeak3-Client-win32-3.0.16.exe	Benign
TeamViewer_Setup.exe	Benign
tuxpaint-0.9.21-win32-installer.exe	Benign
u1504.exe	Benign
UCBrowser_V5.6.11651.1011_windows_pf101_(Build16040516).exe	Benign
uTorrent_3-4-5-build-41372.exe	Benign
VDownloader.exe	Benign
ViberSetup.exe	Benign
vlc-2.2.1-win32.exe	Benign
vpsetup.exe	Benign
winamp5666_full_en-us.exe	Benign
windroy_20140113.exe	Benign
wlsetup-web.exe	Benign
wpsetup.exe	Benign
yet_another_cleaner_sfto.exe	Benign
YouWave-Android-Free-3-30.exe	Benign
YTDSetup.exe	Benign
ZuneSetupPkg.exe	Benign
Virus.Win32.Adson.1651	Malicious

Continued on next page

Table 9.1 – continued from previous page

Name	Class
Virus.Win32.Agent.ak	Malicious
Virus.Win32.Aidlot	Malicious
Virus.Win32.Akez	Malicious
Virus.Win32.Alcaul.j	Malicious
Virus.Win32.Aldebaran.8365.b	Malicious
Virus.Win32.Aliser.7825	Malicious
Virus.Win32.Alman.b	Malicious
Virus.Win32.Andras.7300	Malicious
Virus.Win32.Anuir.3818	Malicious
Virus.Win32.AOC.2044	Malicious
Virus.Win32.Apathy.5378	Malicious
Virus.Win32.Apparition.b	Malicious
Virus.Win32.Arcer	Malicious
Virus.Win32.Arianne.1052	Malicious
Virus.Win32.Aris	Malicious
Virus.Win32.Artelad.2173	Malicious
Virus.Win32.Asorl.b	Malicious
Virus.Win32.Atav.1939	Malicious
Virus.Win32.Auryn.1155	Malicious
Virus.Win32.Autoec	Malicious
Virus.Win32.AutoIt.j	Malicious
Virus.Win32.AutoRun.abz.ajs	Malicious
Virus.Win32.Autorun.bk	Malicious
Virus.Win32.AutoWorm.3072	Malicious
Virus.Win32.Awfull.3254	Malicious
Virus.Win32.Badda.5137	Malicious
Virus.Win32.Bakaver.b	Malicious
Virus.Win32.Barum.1536	Malicious
Virus.Win32.Bee	Malicious
Virus.Win32.Belial.b	Malicious
Virus.Win32.Belod.b	Malicious
Virus.Win32.Benny.3223	Malicious
Virus.Win32.Bika.1857	Malicious
Virus.Win32.Blackcat.2537	Malicious
Virus.Win32.Blakan.2308	Malicious
Virus.Win32.Bluback.1376	Malicious
Virus.Win32.Bobep	Malicious
Virus.Win32.Bolzano.2122	Malicious

Continued on next page

Table 9.1 – continued from previous page

Name	Class
Virus.Win32.Brof.b	Malicious
Virus.Win32.Bube.b	Malicious
Virus.Win32.Butter	Malicious
Virus.Win32.Bytesv.1445	Malicious
Virus.Win32.Cabanas.MsgBox	Malicious
Virus.Win32.Calm.1819	Malicious
Virus.Win32.Cargo.11935	Malicious
Virus.Win32.Cerebrus.1482	Malicious
Virus.Win32.Champ.5430	Malicious
Virus.Win32.Chatter	Malicious
Virus.Win32.Chiton	Malicious
Virus.Win32.Cmay.1222	Malicious
Virus.Win32.Compan.b	Malicious
Virus.Win32.Crosser	Malicious
Virus.Win32.Crypto.b	Malicious
Virus.Win32.CTX.6886	Malicious
Virus.Win32.Dahorse	Malicious
Virus.Win32.Damm.1796	Malicious
Virus.Win32.Datus	Malicious
Virus.Win32.DeadCode.b	Malicious
Virus.Win32.Deemo	Malicious
Virus.Win32.Delf.ab	Malicious
Virus.Win32.Dicomp.8192.b	Malicious
Virus.Win32.Dictator.2304	Malicious
Virus.Win32.Dion.b	Malicious
Virus.Win32.Dislex	Malicious
Virus.Win32.Ditto.1488	Malicious
Virus.Win32.Dock.b	Malicious
Virus.Win32.Donut	Malicious
Virus.Win32.Doser.4183	Malicious
Virus.Win32.Downloader.ac	Malicious
Virus.Win32.Dream.4916	Malicious
Virus.Win32.Driller	Malicious
Virus.Win32.Drivalon.1876	Malicious
Virus.Win32.Drol.5337.b	Malicious
Virus.Win32.Dropet.790	Malicious
Virus.Win32.Drowor.b	Malicious
Virus.Win32.Dudra.5632	Malicious

Continued on next page

Table 9.1 – continued from previous page

Name	**Class**
Virus.Win32.DunDun.1396	Malicious
Virus.Win32.Eclipse.b	Malicious
Virus.Win32.Elkern.dam	Malicious
Virus.Win32.Emotion.b	Malicious
Virus.Win32.Enar	Malicious
Virus.Win32.Enumiacs.6656	Malicious
Virus.Win32.Eva.g	Malicious
Virus.Win32.Flechal	Malicious
Virus.Win32.FlyStudio.b	Malicious
Virus.Win32.Fofox.2470	Malicious
Virus.Win32.Freebid	Malicious
Virus.Win32.Gaybar	Malicious
Virus.Win32.Grum.l	Malicious
Virus.Win32.Halen.2339	Malicious
Virus.Win32.Harrier	Malicious
Virus.Win32.Hawey	Malicious
Virus.Win32.HLLC.Shinex	Malicious
Virus.Win32.HLLC.Susan	Malicious
Virus.Win32.HLLO.GreenDay	Malicious
Virus.Win32.HLLP.Ghostdog.b	Malicious
Virus.Win32.HLLW.Kill	Malicious
Virus.Win32.Horope.l	Malicious
Virus.Win32.Infinite.1661	Malicious

9.2 Appendix 2

Table 9.2: Samples used for experiment

AirDroid_Desktop_Client_3rdmarket.exe	Benign
calibre-2.55.0.msi	Bengin
DiscordSetup.exe	Benign
DittoSetup_3_21_50_0.exe	Bengin
Everything-1.3.4.686.x86-Setup.exe	Benign
FFSetup3.8.0.0.exe	Bengin
FileZilla_3-14-1_win64-setup.exe	Benign
FreeraserSetup.exe	Bengin
Keypass-1.31-setup.exe	Benign
KindleForPC-Installer-1.15.43.061.exe	Bengin
MyPublicWifi.exe	Benign
Rainmeter-3.3.1.exe	Bengin
Thunderbird Setup 45.0.exe	Benign
install_virtualdj_pc_v8.1.2857.msi	Bengin
kdewin-installer-gui-1.0.0.exe	Benign
kochbuch-1.7.1.exe	Bengin
mirc732.exe	Benign
msgr11us.exe	Bengin
superunitconverter2_setup.exe	Benign
Fences2-cnet-setup.exe	Bengin
Virus.Win32.Etap	Malware
Virus.Win32.HLLC.Asive	Malware
Virus.Win32.HLLP.Vampore	Malware
Virus.Win32.HLLW.Veedna	Malware
Virus.Win32.Mohmed.4607	Malware
Virus.Win32.Plutor.b	Malware
Virus.Win32.Folcom.b	Malware
Virus.Win32.Rufoll.1432	Malware
Virus.Win32.Seppuku.4827	Malware
Virus.Win32.Slicer.poly	Malware
Virus.Win32.Stepar.g	Malware
Virus.Win32.VB.bq	Malware
Virus.Win32.VB.fs	Malware
Virus.Win32.VB.jn	Malware
Virus.Win32.Delf.u	Malware
Virus.Win32.Parite.b	Malware
Virus.Win32.Savior.1740	Malware
Virus.Win32.Higway.b	Malware
Virus.Win32.Xorer.ab	Malware
Virus.Win32.Yerg.9571	Malware

9.3 Appendix 3

parser.c

```
#include <stdio.h>
#include <stdlib.h>
#include "cJSON.h"

void parseApiData(char *filename)
{
    FILE *f;
    long len;
    char *data;
    cJSON *json;
    cJSON *format;
    cJSON *formatTmp;
    char *stuff;
    int i = 0;
    int j = 0;
    int nrOfApiCalls = 0;

    f = fopen(filename, "rb");
    fseek(f, 0, SEEK_END);
    len = ftell(f);
    fseek(f, 0, SEEK_SET);
    data = malloc(len+1);
    fread(data, 1, len, f);
    fclose(f);

    json=cJSON_Parse(data);
    format = cJSON_GetObjectItem(json, "behavior");
    format = cJSON_GetObjectItem(format, "apistats");
    int processes = cJSON_GetArraySize(format);
    for(i = 0; i < processes; i++)
    {
         formatTmp = cJSON_GetArrayItem(format, i);
         nrOfApiCalls = cJSON_GetArraySize(formatTmp);
         for(j = 0; j < nrOfApiCalls; j++)
         {
                 printf("%s\n", cJSON_GetArrayItem(
                    formatTmp, j)->string);
         }
    }

    cJSON_Delete(json);
}

void createNameFile(char *filename)
{
    FILE *f;
    FILE *out2 = fopen("tempdata", "w");
    if(out2 == NULL)
    {
    printf("fail\n");
    exit(1);
    }
    long len;
    char *data;
    cJSON *json;
    cJSON *format;
    cJSON *formatTmp;
    cJSON *formatCalls;
    cJSON *formatProcess;
    cJSON *formatArguments;
    char *stuff;
    int i = 0;
    int j = 0;
    int k = 0;
    int nrOfApiCalls = 0;
    int nrOfFlags = 0;


    f = fopen(filename, "rb");
    fseek(f, 0, SEEK_END);
    len = ftell(f);
    fseek(f, 0, SEEK_SET);
    data = malloc(len+1);
    fread(data, 1, len, f);
    fclose(f);



    json=cJSON_Parse(data);
    format = cJSON_GetObjectItem(json, "behavior");
    format = cJSON_GetObjectItem(format, "processes");
    int processes = cJSON_GetArraySize(format);

    for(i = 0; i < processes; i++)
    {
         formatProcess = cJSON_GetArrayItem(format, i);
         formatCalls = cJSON_GetObjectItem(
            formatProcess, "calls");
         nrOfApiCalls = cJSON_GetArraySize(formatCalls)
            ;


         for(j = 0; j < nrOfApiCalls; j++)
         {
                 formatTmp = cJSON_GetArrayItem(
                    formatCalls, j);
                 fprintf(out2, "%s:\n",
                    cJSON_GetObjectItem(formatTmp, "api
                    ")->valuestring);
                 formatArguments = cJSON_GetObjectItem(
                    formatTmp, "arguments");
                 if(cJSON_HasObjectItem(formatArguments
                    , "dirpath"))
                 {
                fprintf(out2, "%s:\n",
                  cJSON_GetObjectItem(formatArguments,
                  "dirpath")->valuestring);
                 }
                 if(cJSON_HasObjectItem(formatArguments
                    , "function_name"))
                 {
                         fprintf(out2, "%s:\n",
                            cJSON_GetObjectItem(
                            formatArguments, "
                            function_name")->valuestring
                            );
                 }
                 if(cJSON_HasObjectItem(formatArguments
                    , "filepath"))
                 {
                         fprintf(out2, "%s:\n",
                            cJSON_GetObjectItem(
                            formatArguments, "filepath")
                            ->valuestring);
```

```
}
if(cJSON_HasObjectItem(formatArguments
   , "filepath_r"))
{
        fprintf(out2, "%s:\n",
           cJSON_GetObjectItem(
           formatArguments, "filepath_r
           ")->valuestring);
}
if(cJSON_HasObjectItem(formatArguments
   , "hostname"))
{
        fprintf(out2, "%s:\n",
           cJSON_GetObjectItem(
           formatArguments, "hostname")
           ->valuestring);
}
if(cJSON_HasObjectItem(formatArguments
   , "library"))
{
        fprintf(out2, "%s:\n",
           cJSON_GetObjectItem(
           formatArguments, "library")
           ->valuestring);
}
if(cJSON_HasObjectItem(formatArguments
   , "module"))
{
        fprintf(out2, "%s:\n",
           cJSON_GetObjectItem(
           formatArguments, "module")->
           valuestring);
}
if(cJSON_HasObjectItem(formatArguments
   , "module_name"))
{
        fprintf(out2, "%s:\n",
           cJSON_GetObjectItem(
           formatArguments, "
           module_name")->valuestring);
}
if(cJSON_HasObjectItem(formatArguments
   , "port"))
```

```
{
        fprintf(out2, "port%d:\n",
           cJSON_GetObjectItem(
           formatArguments, "port")->
           valueint);
}
if(cJSON_HasObjectItem(formatArguments
   , "regkey"))
{
        fprintf(out2, "%s:\n",
           cJSON_GetObjectItem(
           formatArguments, "regkey")->
           valuestring);
}
if(cJSON_HasObjectItem(formatArguments
   , "regkey_r"))
{
        fprintf(out2, "%s:\n",
           cJSON_GetObjectItem(
           formatArguments, "regkey_r")
           ->valuestring);
}
if(cJSON_HasObjectItem(formatArguments
   , "sectionname"))
{
        fprintf(out2, "%s:\n",
           cJSON_GetObjectItem(
           formatArguments, "
           sectionname")->valuestring);
}
if(cJSON_HasObjectItem(formatArguments
   , "url"))
{
        fprintf(out2, "%s:\n",
           cJSON_GetObjectItem(
           formatArguments, "url")->
           valuestring);
}
if(cJSON_HasObjectItem(formatArguments
   , "value"))
{
        fprintf(out2, "%d:\n",
           cJSON_GetObjectItem(
                                formatArguments , "value")->
                                valueint);
                    }
                    formatArguments = cJSON_GetObjectItem(
                       formatTmp , "flags");
                    nrOfFlags = cJSON_GetArraySize(
                       formatArguments);
                    for(k = 0; k < nrOfFlags; k++)
                    {
                            fprintf(out2, "FLAGS: %s:\n",
                               cJSON_GetArrayItem(
                               formatArguments , k)->
                               valuestring);
                    }
            }
    }

}

int main(int argc, char *argv[])
{
    if(argc != 2)
    {
         printf("USAGE <reportfile>\n");
         exit(1);
    }
    else
    {
         createNameFile(argv[1]);
    }
    printf("done\n");
    return 0;
}
```

9.4 Appendix 4

Analyze.sh

```
#!/bin/bash

for f in *.json; do
```

```
     ./createName.sh $f
done

for f in *.json; do
     ./createData.sh $f
done
```

createName.sh

```
#!/bin/bash
./parser $1
echo "" > tempdata1
while read line
do
echo "$line" | tr -d '[:,.|]' >> tempdata1
done<tempdata

echo "done"
arg=$1
newname=${arg%.*}
echo "nyttnamn: $newname"

cat tempdata1 | sort -u > tempdata2
cp tempdata2 data/tempdata2$newname

while read line
do
echo "$line:    f, t." >> result.names
done<tempdata2

cat result.names | sort -u > tempnames
while read line
do
echo $line >> result.names
done<tempnames

rm tempdata
rm tempnames
rm tempdata1
```

createData.sh

```
#!/bin/bash
```

```
echo "" > tempdata3
temp=0
arg=$1
newname=${arg%.*}
filename="data/tempdata2$newname"

if [[ ($(wc -c $filename) > 2) ]]; then
while read line
do
line2=${line/':    f, t.'/}
if grep -qxe "$line2" $filename; then
echo -n "t" >> tempdata3
else
echo -n "f" >> tempdata3
fi
echo -n "," >> tempdata3
done <result11.names

cat tempdata3 | sed '$s/.$//' > data/$newname.data
if [[ $newname == "b"* ]]; then
echo ",B" >> data/$newname.data
else
echo ",M" >> data/$newname.data
fi

echo $newname
rm data/tempdata2$newname
rm tempdata3
else
echo "empty file"
fi
```

createDataForExperiment.sh

```
#!/bin/bash
./parser $1
echo "" > tempdata1
while read line
do
echo "$line" | tr -d '[:,.|]' >> tempdata1
done<tempdata

echo "done"
```

```
arg=$1
newname=${arg%.*}
echo "nyttnamn: $newname"

sort tempdata1 | uniq -u > tempdata2
cp tempdata2 tempdata2$newname

echo "" > tempdata3
temp=0
filename="tempdata2$newname"

if [[ ($(wc -c $filename) > 2) ]]; then
while read line
do
line2=${line/':    f, t.'/}
#result=$(grep "'$line2'" $filename)
#if [ "$result" == "" ]; then
if grep -qxe "$line2" $filename; then
echo -n $line2 >> $newname
fi
done <final.names

echo $newname
#rm data/tempdata2$newname
rm tempdata3
else
echo "empty file"
fi
```