Comparing the utility of User-level and Kernel-level data for Dynamic Malware Analysis

A thesis submitted in partial fulfilment

of the requirement for the degree of Doctor of Philosophy

Matthew Nunes

October 2019

Cardiff University
School of Computer Science & Informatics

A copy of this document in various transparent and opaque machine-readable formats and related software is available at `http://orca.cf.ac.uk`.

To my family (blood or otherwise)
for their love and support.

Abstract

Dynamic malware analysis is fast gaining popularity over static analysis since it is not easily defeated by evasion tactics such as obfuscation and polymorphism. During dynamic analysis, it is common practice to capture the system calls that are made to better understand the behaviour of malware. System calls are captured by hooking certain structures in the Operating System. There are several hooking techniques that broadly fall into two categories, those that run at user-level and those that run at kernel-level. User-level hooks are currently more popular despite there being no evidence that they are better suited to detecting malware. The focus in much of the literature surrounding dynamic malware analysis is on the data analysis method over the data capturing method. This thesis, on the other hand, seeks to ascertain if the level at which data is captured affects the ability of a detector to identify malware. This is important because if the data captured by the hooking method most commonly used is sub-optimal, the machine learning classifier can only go so far. To study the effects of collecting system calls at different privilege levels and viewpoints, data was collected at a process-specific user-level using a virtualised sandbox environment and a system-wide kernel-level using a custom-built kernel driver for all experiments in this thesis.

The experiments conducted in this thesis showed kernel-level data to be marginally better for detecting malware than user-level data. Further analysis revealed that the behaviour of malware used to differentiate it differed based on the data given to the classifiers. When trained on user-level data, classifiers used the evasive features of malware to differentiate it from benignware. These are the very features that malware

uses to avoid detection. When trained on kernel-level data, the classifiers preferred to use the general behaviour of malware to differentiate it from benignware. The implications of this were witnessed when the classifiers trained on user-level and kernel-level data were made to classify malware that had been stripped of its evasive properties. Classifiers trained on user-level data could not detect malware that only possessed malicious attributes. While classifiers trained on kernel-level data were unable to detect malware that did not exhibit the amount of general activity they expected in malware. This research highlights the importance of giving careful consideration to the hooking methodology employed to collect data, since it not only affects the classification results, but a classifier's understanding of malware.

Acknowledgements

This thesis has been a marathon in the making and simply would not be possible without the generous help I've received from a number of people.

Firstly, I'd like to thank my primary supervisor, Pete Burnap. He gave me the freedom to research the domain I was interested in, but maintained a level of control to ensure that I did not get lost, never to return. He also endured my sense of humour and even occasionally encouraged it! Which is more than I can say for my wife.

I'd like to thank the additional supervisors I was blessed with, Omer Rana and Philipp Reinecke. Their guidance always came when I needed it most.

Speaking of wives, I'd like to thank Rebecca. She always took an interest in my research and listened as I described the many challenges I faced. In addition, she took upon herself a large burden of chores and household tasks that were rightly mine so that I could focus on my thesis. Chores, which, I can look forward to returning to after submitting (as she frequently reminds me). However, it goes without saying that her support was crucial in producing this thesis. Nevertheless, I've been told to say it.

I'd like to thank my colleagues. I'd like to thank Kaelon Lloyd, my honorary supervisor. His contributions helped accelerate the pace of my research and learning. Being in his presence was a privilege that he ensured was not lost on me. I'd like to thank Bob, for the countless hours he spent proof-reading and educating me on vector graphics, despite its irrelevance to my field. Nyala, for her statistical knowledge and shared interests. Beryl, for her extensive knowledge on our shared interests. Lauren for con-

sistently reminding me of the temperature in the office. Adi, my brother. Pete Sueref, for never being afraid to ask the difficult questions; questions that were irrelevant to my research, but, nonetheless, important.

I'd like to thank my family for their support and patience. They selflessly cooperated with my hectic schedule and abysmal memory when it came to family matters.

Finally, there are many more people whose support I would not have managed without, so I would like to just say a big thank you!

Contents

List of Publications

This thesis includes work introduced in the following publication:

- Matthew Nunes, Pete Burnap, Omer Rana, Philipp Reinecke, and Kaelon Lloyd. Getting to the root of the problem: A detailed comparison of kernel and user level data for dynamic malware analysis. *Journal of Information Security and Applications*, 48:102365, 2019.

List of Figures

List of Tables

List of Acronyms

ASCII American Standard Code for Information Interchange

ASLR Address Space Layout Randomisation

API Application Programming Interface

CSV Comma-Seperated Values

CTMC Continuous Time Markov Chains

DBI Dynamic Binary Instrumentation

DLL Dynamic Link Library

FPR False Positive Rate

GUI Graphical User Interface

HMM Hidden Markov Model

IAT Import Address Table

I/O Input/Output

IRP I/O Request Packet

JADE Java Agent DEvelopment framework

JVM Java Virtual Machine

JIT Just-In-Time

MSR Model Specific Register

OS Operating System

PE Portable Executable

Q-Q Quantile-Quantile

SM-SC Self-Modifying and Self-Checking

SSDT System Service Descriptor Table

SVM Support Vector Machine

TPR True Positive Rate

UI User Interface

VM Virtual Machine

VMI Virtual Machine Introspection

VMM Virtual Machine Monitor

Chapter 1

Introduction

Over the last two decades, worldwide Internet usage has grown at a remarkable rate. As of 2018 there were 3.8 billion users worldwide [11]. All of these users are able to share data with one another with minimal effort. While this provides countless opportunities, it also can be problematic. This is because the breakdown of borders also allows malicious software, otherwise known as 'malware', to spread without hindrance between users. Understandably, this provides the potential for malware to have a considerable impact in the world. In the past, the payload of malware would take the form of a practical joke making the user very aware of its presence [61]. For example, when a computer was infected with the *Cascade virus*, it would cause the characters on the screen to drop to the bottom of the screen [2]. While this was an annoyance, it was not explicitly harmful. Malware has evolved considerably since then. With the introduction of Internet banking and the increase in the amount of sensitive information being stored online, malware authors have far greater incentive to steal data. To complement that, malware stopped making its presence known to the victim and even attached itself to legitimate programs to trick the user into downloading it [106]. However, as malware was getting more complex, paradoxically, it was also getting easier to create through the introduction of toolkits capable of automatically producing malware with just a click such as the Virus Creation Laboratory (VCL) [17] and PS-MPC [13]. To counter the growing threat of malware, security measures such as two-factor authentication became the norm. In response, malware authors varied their tactics. One of the more successful methods by which malware authors obtain capital is through the

use of ransomware. Ransomware is a class of malware that prevents a user from accessing a core component on their machine and demands a payment for the release of that component [101]. The locked component in question can range from a user's personal files to the entire machine. Ransomware falls into two categories; Locker and Crypto [138]. Locker ransomware simply blocks access to the installed Operating System (OS), typically by altering the boot loader in order to force the computer to boot into a constrained interface that only allows the user to enter a code to release their machine (which is provided after the exchange of money in some form). Crypto ransomware generally encrypts the files in a user's home directory and demands a payment from the user in exchange for the decryption key. Locker ransomware typically locks the user's machine without actually altering any of their files, making it trivial for a user to get access to their original files. In contrast, crypto ransomware can be challenging to recover from, particularly if a sophisticated encryption algorithm is used. This strain of malware alone has caused financial losses in the millions [184].

Malware's impact is not limited to financial. In 2010 the potential threat of malware was further emphasised with the discovery of *Stuxnet* [85]. Unlike malware before it, Stuxnet did not attempt to cause financial damage or steal information, it was created to cause physical damage. Specifically, it was a piece of malware written to target and damage industrial control systems [85]. It did not stop there as the discovery of Stuxnet inspired a number of other malware families such as Duqu and Flame [45]. Currently malware is growing at alarming rates, with 350,000 new samples released everyday [31].

Malware analysts initially responded to the threat of malware by employing simple static analysis techniques. Static analysis refers to the analysis of a sample (usually its code) without ever actually running the sample. An example of such a technique is doing a simple string scan to find identifying byte patterns whilst maintaining some flexibility by allowing for a certain number of mismatches [241]. The identifying signature created from static analysis is syntactic in nature. The advantage of this is that it

is relatively quick to scan a sample which is imperative given the speed at which new malware is produced. The Sapphire worm, for example, was able to infect 90% of its targets within 10 minutes [170]. However, static analysis struggled significantly with the introduction of toolkits such as the Mutation Engine (ME) [3] that could automatically obfuscate code within malware by adding junk code for example. The release of such toolkits made obfuscation techniques much more commonplace in malware [39]. To overcome this, techniques such as smart scanning were used where ineffective instructions (such as NOP instructions) were removed/ignored when scanning the malware sample [241]. The biggest challenge came when malware authors started encrypting their samples. Malware analysts relied on the malware author using a recognisable encryption algorithm with an insecure key or that the decryption routine within the sample was easily decipherable. Unless a sample is decrypted, it is difficult for an antivirus to find identifying bytes in its code [162, 171]. A prime example was Lexotan32 [21]. Though it has been around since 2002, only 12.6% of samples were identified by 40 virus scanners in 2009 [151]. In addition, though static analysis has the benefit of speed when it comes to producing signatures, it lacks robustness. The storm worm took advantage of this. Its writer produced many short lived variants on a daily basis [119]. Antivirus companies struggled to keep up with it as for each variant, they needed to produce another signature [128]. Subsequently as the amount of obfuscation being added to malware grew, it became clear to analysts that static analysis contained limitations that could not be overcome.

Therefore, *behavioural analysis* was proposed as a remedy to the shortcomings of static analysis. Behavioural or dynamic analysis involves running the binary and observing its behaviour. Since code obfuscation does not alter the general behaviour of malware, behavioural detection is not affected by such techniques. Further, since the behaviour of malware is analysed, the signatures produced are much more generic (since there are some behavioural patterns common to many malware samples) [128]. However, with the introduction of dynamic analysis, malware started employing features to evade dynamic analysis as well. The main aim with these techniques was to avoid displaying the

malicious behaviour when being analysed. This would prevent analysts from making representative signatures of the malicious sample. An example of a method malware could use to achieve this is by not executing for 24 hours after it has been run. This is effective because typically dynamic analysis does not run for more than a few minutes (given the amount of malware being produced), therefore, delaying execution prevents an analyst from observing any malicious behaviour. Alternatively malware could attempt to detect peculiarities in the environment it's being executed in since dynamic analysis is normally performed on a virtual or emulated machine, and if malware detects artefacts indicative of a virtual environment, it can simply not run. Additionally, malware can attempt to subvert the data collection mechanism used within dynamic analysis to capture a sample's behaviour. The most popular way to capture a sample's behaviour is by collecting the system calls made. A system call is a request made to the OS for some functionality. This could include reading or writing to a file or displaying something on the screen, amongst many other things. Ultimately, for a sample to do anything significant, it needs to use system calls. Therefore if malware can prevent having its calls monitored and logged, it can hide its behaviour from the monitoring tool. System calls are captured using whats known as a *hook*. A hook modifies the standard execution pathway by inserting an additional piece of code into the pathway [208]. There are a number of methods by which hooks can be performed. Broadly speaking, those methods fall into two categories, those that run in *user-mode* and those that run in *kernel-mode*. The terms user and kernel mode are labels assigned to specific Intel x86 *privilege rings* built into their microchips. Privilege rings relate to hardware enforced access control. Traditionally, there are four privilege rings and they range from ring 0 to ring 3 [219]. Windows only uses two of these rings, ring 0 and ring 3. Ring 0 has the highest privileges and is referred to as kernel mode (this is the privilege most drivers run at) by the Windows OS. Ring 3 has the least privileges and is referred to as user mode (and is the level of privileges that most applications run at) [210]. The focus of this research is on Windows because it is still the most targeted OS by malware as reported in [30, 100, 239].

User-mode hooks tend to only record system/API calls made by a single process since they usually hook one process at a time, whilst kernel-mode hooks are capable of recording calls made by all the running processes at a global, system level. This is an important difference as malware may choose to inject its code into a legitimate process and carry out its activities from there (where it is less likely to be blocked by the firewall). Alternatively, malware could divide its code into a number of independent processes as proposed by [202] so that no single process in itself is malicious, but collectively, they succeed in achieving a malicious outcome. Therefore the choice of hooking methodology could affect the quality of the data gained. Another difference between kernel and user level hooks is that each one hooks into a different Application Programming Interface (API). For example, one type of kernel level hook intercepts calls within the SSDT whose calls are similar to those found in the native API, which is mostly undocumented, whilst user mode hooks typically hook the Win32 API which is documented [174]. Although methods in the Win32 API essentially call methods in the native API, there may be some methods in the native API that are unique to it (since it is only supposed to be used by Windows developers) [49], likewise, there are some user level methods that do not make calls into the kernel. Therefore, it is of paramount importance that the difference in utility between data collected at each level is objectively studied so that analysts can make an informed choice on which type of data collection method to use.

As is evident, if malware intends to prevent a monitoring tool from capturing it's behaviour via system calls, the technique it chooses to use will differ depending on which hooking methodology it intends to evade [220]. Consequently, if a piece of malware is focused on avoiding a particular type of hooking methodology, it is likely that any analysts using that methodology to monitor malware will see a very different picture to those using another methodology. Though the data collection method may seem trivial, the choice of method can have a significant impact on results as evasive methods are not uncommon; in fact, one study found evasive behaviour in over 40% of the samples they analysed [65].

From studying the existing literature, it has been made evident that the majority of the literature captures *user level* calls as shown in table 2.1 in chapter 2. This suggests that researchers believe that user level data has more utility than kernel level data, or that they are yet to consider there to be a significant difference between either type of data for the purposes of detecting malware. Although there are kernel level tools available, they are not as popular as user level tools. Thus, given the aforementioned evasion concerns and fundamental differences in each class of hooking methodology, one of the motivations of this thesis is to study the differences in data collection at kernel and user level, and consider whether it effects a machine learning method's ability to classify the data.

As malware has undergone a number of transformations over the years, and continues to evolve and adapt, particularly with regard to evasion techniques, it is not feasible to expect experts to continuously update heuristics for detecting malware. Therefore, both static and dynamic analysis have moved towards using machine learning classifiers to detect malware since they can automatically update themselves when presented with new samples. Machine learning methods are not without weaknesses though, it's important that their learning is monitored to ensure that the features they are using to differentiate malicious from benign are sensible and not due to chance. This is important because if they are identifying malware using a feature not common amongst malware but common amongst the dataset they have been trained with, they are likely to perform poorly when placed in a real environment. For example, it is likely that malware would exhibit many more evasive properties within a training/virtual environment than when encountered in a real environment. Therefore, it is not enough to view a classifier's accuracy when assessing it's performance, it must be dissected in order to understand its reasoning.

1.1 Contributions

The main contribution from this thesis is the study of the differences in data collection at kernel and user level, and its effect on a machine learning algorithm's ability to classify the data. This motivates the hypothesis that the features of malware that are used to differentiate it from benignware[1] will differ based on the data capturing method used. This will provide insights into the utility of the different forms of data collected from a machine when observing potentially malicious behaviour. Furthermore, given that the majority of malware is likely to first assess the environment it is running in before exhibiting malicious behaviour, and that machine learning classifiers are typically trained on a few minutes of activity, this research evaluates whether machine learning classifiers are identifying malware through their evasive and anti-vm behaviour as opposed to malicious behaviour. This is particularly important in the cyber-security domain where the focus tends to be on the data analysis method over the data capturing method.

In order to test the hypothesis, a *Kernel Driver* was implemented that hooks the entire SSDT with the exception of one call as its internal behaviour prevented it from being hooked safely. A tailor-made kernel driver had to be used since many of the existing tools that hook the SSDT only monitor calls in a specific category (such as calls relating to the file system or registry) and provide no objective justification as to why they chose the calls they did (if they even make that information available). Therefore, the kernel driver used in this research hooks all the calls in the SSDT to ensure that no subtle details regarding malware behaviour are missed and in order to make an objective recommendation on the most important calls to hook when detecting malware. The kernel driver used is also unique in that it collects the SSDT data at a global system-wide level as opposed to a local process-specific level. In doing this, this research can answer the question of whether collecting data at a global level assists in detecting malware or whether it is simply adding noise. In order to gather user level data to compare with the driver, Cuckoo Sandbox is used since it is the most popular malware analysis

[1] Benignware refers to any software that behaves as advertised without any malicious intent

tool operating at a user level (as shown in Table 2.1 in chapter 2). The data gathered from the driver and Cuckoo is then used to experiment with state of art machine learning techniques to better understand the implications of monitoring machine activity from different perspectives. Alongside the general insights gained from classifying the data, specialised feature ranking methods are employed to provide insights concerning the behaviour of malware that is utilised by the classifiers in order to distinguish it. In the interests of transparency and reproduce-ability, the source code of the kernel driver has been made available at [176] and the data from the experiments available at [175]. The driver can be installed on any system running Windows XP 32-bit as this was the platform targeted. However it can also easily be extended to run on Windows 7.

The research questions that this thesis aims to answer are the following:

RQ1 Does data collected at different privilege levels during dynamic malware analysis affect classification results?

RQ2 Is data collected at a global level more beneficial for dynamic malware analysis than that collected at a local level?

RQ3 How does the understanding of malware differ at a kernel and a user level?

RQ4 Does the traditional Dynamic Malware Analysis process create a bias in the data collected and subsequently classified?

RQ5 How much malicious behaviour can a malware sample exhibit before it is detected?

RQ6 Are high-level languages such as Java suitable for emulating malware to test system call monitoring tools?

RQ7 How can the dynamic malware analysis process be amended to prevent unintended security flaws from emerging?

In answering these questions, the following contributions are made:

C1 This thesis contributes an extensive survey and review of dynamic malware analysis tools used or proposed in the literature. Such a survey (despite its importance) has never been conducted before, particularly in such depth. This survey provides information such as the progress made with regards to each hooking methodology, the most popular hooking methodology, and the most popular tool for dynamic analysis.

C2 This thesis contains the first objective comparison on the effectiveness of kernel- and user-level calls for the purposes of detecting malware.

C3 This research assesses the usefulness of collecting data for malware detection at a global system-wide level as opposed to a local individual process level, giving novel insights into data science methods used within malware analysis.

C4 This research assesses the benefits, or otherwise of combining kernel and user level data for the purposes of detecting malware.

C5 This research studies and identifies the features contributing to the detection of malware at kernel and user level and the number of features necessary to get similar classification results, providing valuable knowledge on the forms of system behaviour that are indicative of malicious activity.

C6 This research assesses whether popular classifiers can generalise to detect ransomware that does not contain the most distinguishing features that were found in chapter 3.

C7 This research assesses whether kernel-level or user-level data is better at generalising towards malware that does not contain the distinguishing features found in chapter 3.

C8 This research determines the sensitivity of classifiers trained in the traditional dynamic malware analysis process to changes in system calls made.

C9 This research contributes a driver that hooks all but one call in the SSDT and gathers calls at a global level.

C10 The findings from this research are generalised to inform the general dynamic malware analysis process.

1.2 Thesis Structure

The outline for the remainder of this thesis is as follows:

Chapter 2 — Background: This section contains the motivation for the remainder of the thesis. It provides a description of static and dynamic analysis; the motivation for moving towards dynamic analysis over static analysis; a summary of the data collection methods within dynamic analysis; the favoured methodology and a summary of the progress made for each methodology; and a summary of the place of machine learning in the dynamic analysis process, the most common algorithms employed, and their limitations. The contribution found in this chapter is **C1**.

Chapter 3 — Comparison of User-level and kernel-level data for dynamic malware analysis: This chapter compares the utility of user level and kernel level data for detecting malware in the traditional dynamic malware analysis process. In addition, the features contributing most towards the results are analysed to determine how the classifiers are making their predictions. This chapter contributes **C2**, **C3**, **C4**, **C5** and **C9**.

Chapter 4 — Assessing the effectiveness of classifiers to detect non-evasive malware: Building on the previous chapter, this chapter assesses the robustness of classifiers that are created using the traditional dynamic malware analysis process. To assist with this, a Java based program called Amsel is used to emulate malware lacking evasive properties. This chapter contributes **C6** and **C7**.

Chapter 5 — Testing the robustness of emulated ransomware results: Inspired by the results of the previous chapter, this chapter assess the correctness of emulated malware that is written in high-level languages such as Java. In addition, this chapter determines how sensitive classifiers trained on system calls are to small changes in system calls. This chapter contributes **C8**.

Chapter 6 — Discussion: This chapter generalises the lessons learned from chapter 3, chapter 4 and chapter 5 to provide recommendations regarding how dynamic malware analysis should be carried out going forward. This chapter also discusses the limitations with this work and the next steps. The contribution in this chapter is **C10**.

Chapter 7 — Conclusion: This chapter summarises the research conducted in this thesis.

Chapter 2

Background

This chapter introduces the field of malware analysis and describes the state of the art tools in the field. Importantly, this chapter contains our first contribution:

C1 *This thesis contributes an extensive survey and review of dynamic malware analysis tools used or proposed in the literature. Such a survey (despite its importance) has never been conducted before, particularly in such depth. This survey provides information such as the progress made with regards to each hooking methodology, the most popular hooking methodology, and the most popular tool for dynamic analysis.*

2.1 The Problem of Malware

Malware, short for Malicious Software, is the all-encompassing term for unwanted software such as Viruses, Worms, and Trojans. The problem of malware is significant; AVTEST register 350,000 new malware samples every day and recorded a total of 885.24 million malware samples in 2018 [31]. The prime target for malware in 2018 was the Windows OS, with 64% of the samples targeting it [30]. Therefore the focus of this research is on defending Windows against malware. The volume of malware produced for this OS alone is far too much for a human analyst to analyse manually. This also creates an incredible challenge for antivirus companies since their solutions are only as secure as their databases are up-to-date [148]. Therefore, there is a need for

solutions capable of automatically analysing and classifying unseen malware samples without being overly reliant on signature databases or heuristics.

Malware can be analysed in one of two ways; statically and dynamically. The main difference between the two methods is that static analysis involves studying a binary without executing it. Whereas dynamic analysis consists of executing the binary and analysing its behaviour while it is running [227].

2.2 Static Analysis

Static code analysis involves studying the suspicious file and looking for patterns in its structure that might be indicative of malicious behaviour without ever actually running the file. In most cases, the file being studied during static analysis is the compiled file, since malware authors seldom share the source code of the binary [80, 266]. Therefore, the typical features that are extracted during static analysis include system calls, strings, header information, opcodes and byte n-grams [266]. System calls are calls made to the OS when a binary requires it to perform some operation for it such as opening and writing to a file. Strings refers to all the strings extracted from a sample by interpreting every byte of the binary as a string. While this can produce a lot of noise, it can also reveal a lot about a file such as the function names used by the malware author, names of directories and even IP addresses. Header information refers to the structural information of the sample-specific to its file-type. Opcodes (short for Operation Codes) refers to the assembly instructions within the sample. Similarly, byte n-grams are sequences of n bytes extracted from the binary and used as features. The most popular tool used to extract this information is IDA Pro [116].

The main benefit of performing static analysis is efficiency. Since feature extraction only consists of going through the code of the binary, analysis is rapid. This is ideal for real-world scenarios where security solutions need to operate in real-time. It is for this reason that static analysis is frequently used in Antivirus solutions [148]. An-

other benefit of static analysis is that the analyst can observe the entirety of the sample, whereas, with dynamic analysis, the portion of the sample analysed is the portion displayed for the duration of the execution [266]. The final main benefit provided by static analysis is safety. The chances of cross-contamination occurring during static analysis are extremely slim since the binary file does not need to be run to be analysed.

However, static analysis has been losing popularity due to its inability to deal with obfuscated and polymorphic malware [171, 162]. Polymorphic malware is malware that encrypts itself and changes the key it uses to encrypt itself every time it propagates. Some of the polymorphic malware samples even alter the decryption routine used on each propagation [20]. Obfuscation refers to strategies used by malware authors to make their code seem more benign. Obfuscation techniques include code integration, code transposition, dead code insertion, instruction substitution, register reassignment, and subroutine reordering [269]. Broadly speaking, these techniques alter the code (source or compiled) so that the malware sample's signature changes significantly while ensuring that the malware sample behaves in the same way. The main challenge with statically analysing obfuscated or polymorphic malware is that unless the obfuscation technique or encryption algorithm can be detected and reversed, the information extracted from the binary file is likely to be heavily skewed [162]. The problem is further compounded by the fact that it is relatively easy for malware authors to add polymorphism and obfuscation to their samples due to the prevalence of tools available to automate the task [266]. One study found that 92% of malware samples contained obfuscation of some sort [53].

The problems with static analysis have been studied in great depth in the literature. [171] showed that commercial antivirus solutions were unable to deal with malware with very simple obfuscation applied to it. To remedy this [72] proposed semantics-aware static analysis. Semantics aware analysis defines a blueprint for general behaviours (such as a decryption loop) that can be compared to specific instructions within a binary to check for a match. This has the benefit of not being reliant on specific re-

gisters or instructions since it is looking for behaviours rather than exact code matches. This makes it resilient to obfuscation techniques such as register reassignment (where the registers used in malware are changed to evade detectors). However, [171] showed that even semantic aware static analysis could be evaded by adding complex obfuscations to important constants so that their values could only be determined at run-time. Recently, [75] compared the performance of static, dynamic, and hybrid (combination of static and dynamic data) techniques of gathering data when classifying malware, and found that a fully dynamic approach produced the best classification results. As a result, research (including this thesis) is now focused on dynamic analysis due to its potential. While there is still some research in semantic-aware static analysis, given that semantic-aware static analysis attempts to recognise the behaviour of malware, it is undoubtedly more reliable to simply observe the behaviour of malware as dynamic analysis does.

2.3 Dynamic Analysis

Dynamic behavioural analysis involves running the binary in a controlled environment, such as on an emulator, or Virtual Machine (VM), and searching for patterns of OS calls or general system behaviours that are indicative of malicious behaviour. Behavioural analysis has gained popularity over static analysis since it runs malware in its preferred environment making it harder to evade detection completely. Dynamic analysis can be deployed as part of an anti-malware solution much like static analysis. However, it can also be used to complement traditional analysis techniques when those techniques are unable to confidently classify a sample.

Due to the fact that dynamic analysis involves running malware samples, there is a risk of cross-contamination. To mitigate that risk, samples tend to be run within an isolated/semi-isolated environment, where it is easy to revert the system back to a

clean state when necessary. This can be achieved using virtualisation or emulation. Virtualisation allows a user to run multiple machines, referred to as 'Virtual Machines' on the same hardware [36]. A Virtual Machine Monitor (VMM) or Hypervisor is responsible for sharing resources between each of the VMs. The key with virtualisation is that instructions are run on the hardware itself [149]. Whereas with emulation, the hardware or OS is implemented in software [149]. Each method has its advantages and disadvantages. Virtualisation provides the benefit of good performance since instructions are run on the actual hardware [149]. Emulation, on the other hand, allows an analyst to gather detailed information regarding the execution of a sample since it is implemented in software.

Finally, in order to conduct behavioural analysis, data relating to the sample's behaviour must be extracted and logged during or after its execution. The type of data that can be useful to gather includes CPU load, memory usage, disk accesses, and system calls. The most popular mechanism in the literature for understanding malware's behaviour during execution is through capturing the calls made to the OS, i.e. system calls [203]. In broad terms, to capture system calls, a tool must create a *hook* into the OS or monitored process. A hook modifies the standard execution pathway by inserting an additional piece of code into the pathway [208]. This is done to interrupt the normal flow of execution that occurs when a process makes a system call in order to document the event. There are several methods to intercept system calls in Windows and these fall into two general categories: those that run in user mode and those that run in kernel mode [208].

The terms 'user' and 'kernel' mode are labels assigned to specific Intel x86 privilege rings built into their microchips. Privilege rings relate to hardware enforced access control. There are four privilege rings and they range from ring 0 to ring 3 [219]. Windows only uses two of those rings, ring 0 and ring 3. Ring 0 has the highest privileges and is referred to as kernel mode (this is the privilege most drivers run at) by the Windows OS. Ring 3 has the least privileges and is referred to as user mode (and

is the privilege level that most applications run at) [210]. The main purpose of this stringently enforced access control is to protect user applications from modifying parts of memory belonging to the Operating System and causing a complete system crash. However, this also means that anything running in kernel mode has complete access to system memory and therefore must be designed with the utmost care [210].

Though system calls can be invoked at user-mode, their functionality is implemented in kernel-mode. An example of the normal pathway for a system call in Windows is shown in figure 2.1. From user mode, a process may call createFileA, createFileW, NtCreateFile, or ZwCreateFile, however, ultimately, they all lead to the NtCreateFile method in the SSDT. In order to provide the requested functionality, the processor must move from Ring 3 (user level) to Ring 0 (kernel level). It does this by issuing the sysenter instruction. Although createFileA has been shown to call NtCreateFile/ZwCreateFile in figure 2.1, strictly speaking, it calls createFileW. However, as they are provided by the same library, they are shown at the same level. Figure 2.1 also shows that, in the case of createFile, to get the same information in user-mode as kernel-mode, more methods need to be hooked.

Figure 2.1: System call visualisation

Within user-mode and kernel-mode there are a number of methods by which system calls can be intercepted. This is shown in figure 2.2.

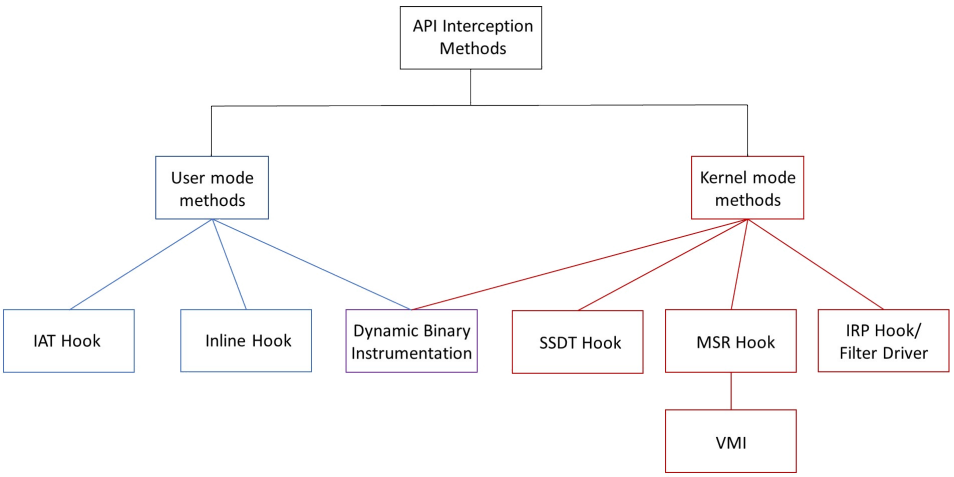

Figure 2.2: Methodologies by which system calls can be intercepted

There are a plethora of tools used to perform dynamic analysis. All the tools used in the literature are shown in table 2.1. Therefore table 2.1 excludes any novel tools that have not yet been used/tested by other papers. Table 2.1 contains six columns; "Name" which is the name of the tool, "Description" which describes the hooking methodology used by the tool, "Kernel Hook" which is marked if the tool employs a hook at kernel level, "User Hook" which is marked if the tool employs a hook at user level, "Functions Hooked" which mentions the categories of functions hooked, and "Used By" which lists the papers that used that tool. For each tool mentioned in Table 2.1, if the tool was available online, it was tested in order to understand how it was intercepting API-calls. Where the tool was not available, documentation was used to determine the type of hook being used. To limit the length of the table, Table 2.1 only contains tools that had been used at least once in the literature (i.e., at least one entry in their "Used By" column).

Name	Description	Kernel Hook	User Hook	Functions Hooked	Used By
API Monitor [27]	Capable of hooking every method in the Windows API		x	All methods in Windows API	[25] [251] [26] [66]
APIMon [1]	Uses EasyHook [5] to perform inline hooking		x	Hooks all user-level APIs	[79]
Buster Sandbox Analyser	Not documented how it gathers API calls.		x	File system changes, registry changes, & port changes [58]	[75] [254]
CaptureBAT [15]	Uses filter drivers	x		Monitors registry, file, and process activities	[238] [221]

Continuation of table

Cuckoo Sandbox [111]	Leading open-source dynamic malware analysis system [111]. Uses inline hook to hook Windows API calls	x	A variety of methods in the Windows API including file, network, process, and thread operations, to name but a few	[115] [70] [245] [213] [97] [69] [87] [201] [152] [153] [93] [114] [67] [196] [150] [237] [46] [96] [77] [90] [189] [274] [160] [261] [42] [141] [37] [147] [260] [222] [126] [243] [132] [226] [121] [129] [247] [242] [35]

Continuation of table				
CWSandbox [259]	Uses in-line code hooks [259]	x	File operations, registry changes, Dynamic Link Library (DLL)s loaded (by malware), memory operations, process operations, network operations, and services and kernel drivers loaded. [259]	[206] [199] [200] [157] [102] [205]
Deviare [4]	Hooking engine that is also integrate-able with many programming languages	x	Entire Win32 API	[95]

Continuation of table				
Ether [78]	Virtual Machine Introspection (VMI) solution focused on being undetectable by malware (known for achieving good transparency). Utilises Xen hypervisor and Intel VT [250] to provide hardware virtualisation	x	Not mentioned	[169] [75] [185] [173]
HookMe	Uses Microsoft's Detours [124] to perform in line hooking	x	Not documented	[244] [113]

Continuation of table					
Malpimp [8]	Based on pydbg (pure Python debugger)		x	Unclear, but does not seem to hook all user-level APIs.	[86]
Micro analysis System (MicS) [127]	Executes in a real (not virtual) environment and uses IAT hooking		x	Unclear	[136]
NtTrace [10]	Tool that uses inline hooking		x	ntdll.dll	[130]
Osiris [60]	VMI solution using a modified version of QEMU [43]. Also provides a simulated network environment	x	x	Monitors 187 user-level API calls and 73 kernel level API calls	[169]

Continuation of table						
StraceNT [7]	Inspired by strace on Linux. Uses IAT hooking		x	All user-level Windows APIs	[172] [257]	[187]
Sysinternals Process Monitor [209]	Gathers data using kernel driver (file system filter driver) [210]	x		Registry activity, file system activity, network activity, process & thread activity, and Process Profiling activity	[245] [238] [163] [233]	[94] [48] [265]
TEMU [267]	Extensible complete-system, fine-grained analysis platform	x	x	Everything (can perform complete system analysis)	[263] [249]	[268]

Continuation of table					
TTAnalyze (used in Anubis (Analysis of unknown binaries) Sandbox [40] [80])	Uses QEMU [43] to perform software emulation. Monitors API calls through Just-In-Time (JIT) compilation [38]	x	x	File, Registry, Process, Network & Service [38] [80]	[89] [161] [146] [142] [108] [145]
WinAPIOverride [19]	Free tool to monitor API calls made by processes		x	All user-level Windows APIs	[215] [214]

Table 2.1 shows that the majority of the tools used in the literature hook at the user-level. The most popular tool of all is Cuckoo Sandbox [111] which employs an inline hook to intercept API-calls. The most popular tool for monitoring at a kernel level is Sysinternal's Process Monitor [209]. However, two of the most well-known tools for monitoring at a kernel level, TTAnalyze and CWSandbox, are now commercial products called LastLine and ThreatAnalyzer [178]. In the following sections, we describe each hooking methodology in more detail.

2.3.1 IAT Hook

An *IAT* hook modifies a particular structure in a Portable Executable (PE) file. The PE file format refers to the structure of executable files, Dynamic Link Libraries (DLLs), and similar files in Windows [195]. IAT hooks exploit a feature of the PE file format, the imports that are listed in a PE file after compilation. Typically the IAT contains a list of the external DLLs and functions that a program requires from the OS. When the PE file is loaded into memory, each function listed in the IAT is mapped to an address in memory. An IAT hook modifies the address in memory so that the import points to an alternative piece of code rather than the required function [154, 49, 71]. Usually, after the alternative piece of code has done what it needs (such as log the call), it calls the real import on behalf of the original code. Detecting the presence of an IAT hook is quite trivial as one only needs to check that the addresses in the IAT actually point to the correct module [118]. In addition, if malware wants to hide the names of the functions it is using, it can simply import the LoadLibrary and GetProcAddress functions supplied by Kernel32.dll to import DLLs and load the required functions on demand. Due to its limitations, IAT hooks are not often used in the literature. One well known tool employing an IAT hook is MicS [127], an automated malware analysis system. In addition, an IAT hook is also used by STraceNT, a tool built to mimic the functionality provided by the Unix tool 'strace'.

2.3.2 Inline Hook

An inline hook modifies the target function's code within memory by adding a jump to another piece of code belonging to an analysis engine for example. Once the analysis engine has finished doing what it needs to (such as logging the call), it jumps back to the original code in a process known as *trampolining* [124, 118]. When performing this hook, the analysis engine must be careful not to overwrite important functionality in the target function. The easiest way to avoid this is by overwriting the *preamble* of a function as generally the first five bytes of a function in Windows is always the same. The reason for this is that it allows the OS to be *hot patched*. Hot patching refers to when fixes are made to a module within memory meaning that a system restart is not required to enact the change [118]. Inline hooks suffer similar weaknesses to IAT hooks in that they can be detected quite easily by scanning the code of the system functions in memory and checking if they match the code on disk. However, inline hooks are the most popular hooking methodology employed in the literature and are used by the most popular tools, Cuckoo Sandbox [111] and CWSandbox [259]. Its uptake was helped by the fact that Microsoft published the library they used to perform inline hooking (in the context of hot-patching) known as Detours [124].

2.3.3 MSR Hook

An *MSR* hook essentially hooks the sysenter instruction. More specifically, it involves changing the value of a processor-specific register referred to as the SYSENTER_EIP_MSR register. This register normally holds the address of the next instruction to execute when sysenter is called (which is called every time a system call is made). Therefore if this value is altered, the next time the sysenter instruction is called, the new value in the register will be the next instruction that is executed (which in this case can point to the analysis engine). Since an MSR hook modifies a processor specific register, developers need to ensure that they modify the registers on each processor (since most

systems nowadays contain multiple processors) [210]. There are few examples of an MSR hook being used as a standalone method in the literature. Usually, it is employed in the context of VMI solutions.

VMI solutions refer to those in which the malware analysis engine resides at the same level as the Hypervisor or VMM. They tend to use *Breakpoints* or *Page Faults* to perform MSR hooks [99]. Breakpoints are placed in 'interesting' locations and whenever one is reached, the VMM and thereby malware analysis engine are notified.

Since VMI solutions operate at the same level as the Hypervisor, this can provide benefits such as the ability to monitor a VM without having a large presence on the VM (and thereby making it harder for malware to detect the presence of the analysis engine). The difficulty with monitoring at this level is that a "semantic gap" must be bridged in some way. The semantic gap refers to the fact that when monitoring at the VMM layer, much of the data available is very low level (such as register values). This data is not at a level of granularity that is easy to interpret. Therefore, in order to bridge that, solutions use a number of techniques to convert these values to more abstract values. For example, as mentioned previously, VMI solutions use a variation of the MSR hook whereby instead of placing the address of the analysis solution into the SYSENTER_EIP_MSR register, an invalid value is placed into that register. As a result, every time a system call is made and sysenter is called, a page fault will occur. This will in turn lead to the VMEXIT instruction being called which will pass control to the VMI tool (since it operates at the same level as the hypervisor). The VMI tool must then examine the value of the EAX register in order to find out the system call made. Since monitoring system calls in this manner can have a significant impact on performance, VMI tools usually limit their monitoring to a particular process. To achieve this, the tool must monitor for any changes in the CR3 register. The *CR3 register* contains the base address of the page directory of the currently running process, therefore, if the page directory address of the process of interest is known, then system calls can be filtered to only those emanating from the process of interest.

Unlike the previous two hooking methods, the VMI solutions in the literature are quite varied in how they monitor malware and how they attempt to address some of the issues inherent to VMI. The most well known VMI tool is TTAnalyze [38]. TTAnalyze executes malware in an emulated environment (QEMU [43]) as opposed to a virtual one. [38] argue that an emulated environment is harder for malware to detect since a real system can be mimicked perfectly. However, this comes at the expense of performance, as samples are executed significantly slower. Another well known tool in this domain is *Panorama* [268]. Panorama is built on top of TEMU [235] (the dynamic analysis component of BitBlaze [235] that can perform whole-system instruction-level monitoring), and performs fine-grained *taint analysis*. Taint analysis refers to the act of monitoring any data touched by the executable being analysed. The name stems from the fact that data touched by the executable is subsequently considered 'tainted'. [235]'s contribution lies in the fine-grained taint tracking it performs, even recording keystrokes among many other things. Ether [78] is also a popular tool employing VMI that differs by exploiting Intel VT [250] which enables in-built processor support for virtualisation and provides a significant performance boost when running a VM. *Ether* is also particularly focused on not being detectable by malware and, as such, has very little presence on the guest machine. *Osiris* [60] is similar to Ether, however, it manages to perform an even more complete analysis by also monitoring any processes the original process injects its code into. [155] propose *DRAKVUF* which focuses more on reducing the presence of an analysis engine from the guest machine as normally there is some code present on the guest to run the process being monitored or help the VMI solution with the analysis. However, DRAKVUF employs a novel method to execute malware using process injection and therefore doesn't require any additional software to be present on the guest. In addition, it monitors calls at both user and kernel level. [188] take a different approach to VMI by using invalid opcode exceptions instead of breakpoints to intercept system calls. Invalid opcode exceptions are raised if system calls are disabled and a system call is then called. This, they argue, performs better. In addition, their monitoring solution is not paired with a hypervisor but exploits a vul-

nerability ([211]) to virtualise a live system, forgoing the need for a reboot to install the monitoring solution.

While it's clear that significant progress has been made with VMM solutions, there is still a delay overhead incurred from the mechanism (breakpoints/page faults) that is typically used to monitor API-calls. Ether, a well-known tool in this genre, was shown to have approximately a 3000 times slowdown [264]. This, among other things, makes it easier for malware to detect the presence of a monitoring tool by simply timing how long it takes to execute an instruction. Furthermore, while some solutions have managed to remove much of the presence of the analysis component from the machine being monitored, this has the unfortunate effect of making it even more challenging to bridge the semantic gap.

2.3.4 DBI

Instrumentation refers to the insertion of additional code into a binary or system for the purpose of monitoring behaviour. *Dynamic instrumentation* implies that this occurs at runtime [230]. Dynamic Binary Instrumentation is usually implemented using a JIT compiler. In DBI, code is executed in basic blocks, and the code at the end of each block is modified so that control is passed to the analysis engine where it can perform a number of checks, such as whether a system call is being executed [54, 198]. Two of the most popular frameworks for achieving dynamic instrumentation in Windows are DynamoRIO [54] and Intel Pin [166].

The main limitation in solutions using JIT compilation is Self-Modifying and Self-Checking (SM-SC) since DBI solutions can be detected by the modifications they make to the code. Therefore, SPiKE [253] was proposed as an improvement to such tools since it uniquely did not use a JIT compiler, but breakpoints in memory. Specifically, it employs "stealth breakpoints" [252], that retain many of the properties of hardware breakpoints, but don't suffer from the limitation that pure hardware breakpoints do of

only allowing the user to set between two and four. Through using such breakpoints, it is harder to detect the presence of the monitoring tool and the tool is more immune to SM-SC code. Reportedly, this even brought a performance gain. [198] built their solution, Arancino, on top of Intel Pin which is focused on countering all known anti-instrumentation techniques that are employed by malware to evade detection.

The problems that solutions in this space suffer from is performance and remaining undetectable by malware. Though [198] make a considerable effort towards improving this, they admit their solution is unlikely to be undetectable.

2.3.5 SSDT Hook

SSDT hooks modify a structure in kernel memory known as the System Service Descriptor Table (SSDT). The SSDT is a table of system call addresses that the OS consults when a process invokes a system call in order to locate the call. An SSDT hook replaces the system call addresses with addresses to alternative code [208] [49]. SSDT hooks have been used in a number of solutions proposed in the literature [158, 143, 109, 59], however, the tools created have never hooked the entirety of the SSDT and therefore the full potential of such a hook has never been truly studied. This may be because while SSDT hooks provide the benefit of giving unprecedented access to the internals of the kernel (allowing one to access system calls and argument values), they are not supported by Windows and therefore are very challenging to implement (requiring a great deal of reverse engineering). In addition, not only are SSDT hooks implemented as kernel drivers, but they require a developer to alter parts of memory belonging to the OS, therefore there is no room for error as any errors lead to immediate system crashes.

2.3.6 IRP Hook/Filter Driver

In *IRP hooking*, the analysis engine intercepts another driver's IRPs [49, 208]. IRPs, are used to communicate requests to use I/O to drivers. This is similar to filter drivers which are drivers that essentially sit on top of another driver for a device intercepting all the IRPs intended for that driver [256]. Filter drivers do not directly communicate with the hardware but sit on top of lower-level drivers and intercept any data that comes their way. The most well-known tools using filter drivers are Procmon [209] and CaptureBAT [15]. Other examples in the literature where filter drivers are used are [125] and [271].

The limitation with using filter drivers is that they cannot intercept the same breadth of API-calls that other hooking methodologies can. They focus on the major operations in particular categories (such as file system and registry).

2.3.7 Dynamic Analysis Limitations

Dynamic analysis is not without its weaknesses, it is common for malware to hide its malicious behaviour if it detects that it is being analysed. A study of 4 million samples found that 72% of the samples contained techniques to detect that they were being run in a virtual environment [50]. There are a number of methods by which malware can do this, however, essentially, all of them boil down to attempting to detect features unique to the dynamic analysis process. An example of a method by which malware does this is through detecting the environment it is being run in. Since dynamic analysis is frequently carried out in a virtual or emulated environment, malware can look for artefacts unique to those environments (in a process known as fingerprinting [179]) and alter its behaviour accordingly. An example of this would be a malware sample looking for drivers or devices specific to a VM or emulator [98]. Another method exploits the fact that in dynamic analysis, typically, a binary is not run for more than

a few minutes, therefore, in order to evade detection, malware could simply delay performing malicious activities for a few minutes (or even 24 hours). There are a plethora of additional techniques by which malware can detect whether it is being analysed or not, all of which are extensively documented in numerous places: [24, 179, 225, 65, 57, 98, 50].

In addition to those, more novel methods have been proposed that take advantage of the manner in which calls are gathered and then analysed. [202] noticed that most analysis tools classified processes as malicious or benign one process at a time. Therefore, they divided a chosen malicious sample into a number of processes that individually would not be malicious, however, together, these processes could cooperate to achieve the malicious outcome. They analysed the divided malicious sample using 43 different Antiviruses and seven dynamic analysis tools (including Anubis [40], JoeBox [55], and Norman Sandbox [234]) and found that it evaded detection in every case. [167] automate the theory of the technique employed by [202] by producing a tool that when given the source of a malicious sample is able to split it into a number of samples, specifically splitting the source whenever a potentially incriminating system call is used. In addition, the tool added the required communication code between the samples created. The resulting malware produced by their tool was tested on CWSandbox [259] and Norman Sandbox [234] and succeeded in evading analysis. [236] evade system call analysing tools by only ever calling a single system call from their malicious process that tells a custom-made driver the actual system call the process wants called. Since the driver runs at kernel mode, it can then call the system call directly (bypassing any monitoring tools). In doing this, any tools gathering system calls only observe a single system call coming from the malicious process.

2.3.8 Discussion

Unlike static analysis, in dynamic analysis the number of methods by which the same information can be extracted (i.e. system calls) is significant. Therefore, the first de-

cision an analyst must make when performing dynamic analysis is the method by which they are going to extract system calls. The argument for hooking in user mode is that the code analysing the sample is "closer" to the application being analysed since the APIs hooked are the ones that developers are encouraged to use. More technically speaking, within Windows, kernel-mode hooks tend to hook into what's known as the native API, which is mostly undocumented, whilst user mode hooks typically hook the Win32 API which is documented since it is what Windows encourages developers to use [174]. Therefore, the advantage of hooking into the Win32 API is that the analyst is likely to observe the methods that the sample was programmed to call. Whereas, with the native API, the analyst is likely to observe the methods that are called by the methods the sample calls. Therefore, the calls made at the Win32 API are likely to be easier to interpret since they provide more details. In addition, user-mode hooks lend themselves naturally to hooking and monitoring a single process whereas with kernel-mode hooks, additional code must be written in order to limit the information to the process under investigation. This is illustrated in figure 2.3. Figure 2.3, though simplistic, shows why a kernel-mode hook is better suited to monitoring at a global, multi-process level. Obviously, there are many features missing from figure 2.3 (such as communication with the Hardware), however, the main purpose of the diagram is to show why kernel- and user-level hooks observe a different picture of the system. Ultimately, despite it's advantages, the main limitation with user-mode hooks is that they operate at the same privilege level as the process being examined and therefore, are much easier for the process being examined to evade and feed misinformation.

The main argument for hooking at kernel mode is that due to the heightened privilege, it is a lot harder for a malware sample to evade the analysis code. Further, while user-level hooks can only hook a single process, a kernel-level analysis tool is capable of observing much more since it has a system-wide view. This is an important difference as malware may choose to inject its code into a legitimate process and carry out its activities from there (where it is less likely to be blocked by the firewall). Alternatively, malware could divide its code into a number of independent processes as

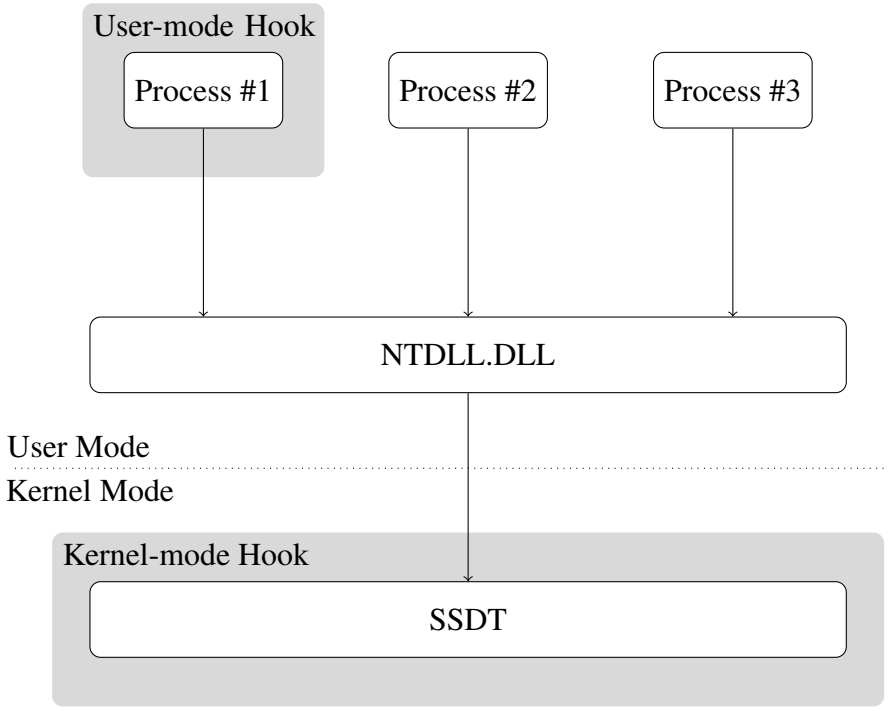

Figure 2.3: The different views of kernel and user-mode hooks illustrated

proposed by [202] so that no single process in itself is malicious, but collectively, they succeed in achieving a malicious outcome. Therefore the choice of hooking methodology could affect the quality of the data gained. In addition, hooking the native API has it's advantages since malware may decide to directly call methods in the kernel to evade user-level hooks. While methods called in the native API may not be as easy to label as a particular behaviour, the advantage of hooking into the native API is that the analyst mainly sees the important methods.

Regardless of the benefits of each data gathering method, it is clear that the manner in which data is gathered is more than likely to affect the data obtained. Even methods gathering data at the same privilege level may gather different data since each method has its own weakness and is evaded differently [220]. This can be a significant concern given the pervasiveness of evasive techniques in malware. Its important to emphasise, that when the word "evasive" is used in this thesis, it is specifically referring to the behaviours of malware that are used to prevent the data capturing component (hooking

methodology) from capturing accurate information regarding the behaviour of malware whilst it is being run. Therefore, if two methods are gathering data at different privilege levels, hooking into different APIs, and collecting data from different viewpoints (local vs global), they are undoubtedly going to get very different views of the malware sample and the system. Despite this, the focus in the literature of malware analysis tends to be on the method used to classify the data rather than the actual data itself. While the detection method is important, unless the data gathered is optimal, the detection method can only go so far. This is also important because if the literature only treats the detection method as the variable within malware analysis, it will be difficult to assess whether an improvement in results is due to the novel detection method used or the manner in which the data was gathered. Furthermore, while the majority of the literature uses user-level tools to gather data, there is no evidence for their supremacy or otherwise over kernel-level tools, despite the significant differences between them. This provides the motivation for the first three research questions:

RQ1 Does data collected at different privilege levels during dynamic malware analysis affect classification results?

RQ2 Is data collected at a global level more beneficial for dynamic malware analysis than that collected at a local level?

RQ3 How does the understanding of malware differ at a kernel and a user level?

2.4 Classifying System Calls

Once system calls have been gathered, the patterns within them need to be extracted and converted into rules that can be used to distinguish malicious from benign. While this can be performed manually by experts, the sheer volume of malware being produced makes this impossible. To obtain complete coverage over all malware samples produced, an expert would have to analyse each new malware sample within

1.6 seconds [218]. Therefore, rather than manually extracting patterns to identify malware from system calls, the process must be automated using machine learning.

In order to do this, the system calls that have been gathered must be be represented in a form that can be understood by machine learning classifiers. The most common method is to convert the data into numeric form. This can be as simple as representing a sample as a frequency histogram showing how many times each system call was made by that sample. Alternatively, it can be more complex, taking context into consideration by using call sequences [186]. Regardless, once the data is represented numerically, it can be passed to a machine learning classifier. Machine learning refers to the process by which a machine automatically learns how to perform a task (such as distinguishing malware from benignware). While machines can be manually programmed to perform a task, some tasks can be very difficult to solve, and are therefore difficult to teach a machine to perform in a robust manner that generalises well. Machine learning can help with this.

Classification is the machine learning process in which a machine learning algorithm is provided with input data and it predicts the category that the data belongs to based on patterns within the data. In this case, the input data is the API calls made and the output from the classifier is the label, 'malicious' or 'benign'. There are a wide range of classifiers that a malware analyst can use to analyse malware. They fall into two general categories, supervised, and unsupervised. With supervised algorithms, the classifier is trained on labelled data. In this case, that means that the classifier is told which data comes from benign and which comes from malicious samples. With unsupervised algorithms, the classifier is not told the class that the data belongs to beforehand, it is up to the classifier to decide which data comes from which sources. For the purposes of this study, however, this research will only focus on supervised classifiers since they are better suited to this specific task, particularly given that the number of classes is already known.

The focus of this thesis is to not only assess if there is a difference in the classifica-

tion results but to understand the features causing the difference. It is for this reason that deep learning is disregarded for the study. Deep learning is a machine learning algorithm inspired by the structure of the brain and consists of a large array of neural networks (similar to neurons in the brain) [104]. Deep learning algorithms are extremely complex and while deep learning produces promising results, the main aim of this research is not to obtain exceptional results, but to better understand the difference in the results. Given these conditions, the main classifiers used within dynamic malware analysis are summarised in table 2.2. Table 2.2 lists each classifier used within the recent literature of dynamic malware analysis, provides a short description of the classifier, and lists the papers that have used that classifier.

Table 2.2: Popular classifiers in Malware analysis

Classifier	**Description**	**Used By**
AdaBoost [91]	AdaBoost is a collection of weak classifiers (frequently Decision Trees) on which the data is repeatedly fitted with adjusted weights (usually weighting misclassified samples more heavily) until, together, the classifiers produce a suitable classification score or a certain number of iterations are complete.	[244, 130, 133, 126, 273]
Decision Trees [52]	create if-then rules using the training data which they then use to make decisions on unseen data.	[244, 89, 25, 169, 95, 130, 120, 126, 132]
Gradient Boosting [92]	A more general version of AdaBoost that uses a collection of weak learners (Decision Trees) and at each stage adds a new learner to model by fitting the new learner on the previous learner's errors	[223, 126]

Continuation of table		
Hidden Markov Model (HMM)s	HMMs predict the values of some hidden state using only the current state (and nothing before it)	[132]
Logistic Regression	One of the simplest classification algorithms. It attempts to separate the data using a linear boundary and therefore can only be used if the output variable is categorical.	[126, 223]
Support Vector Machine (SVM) [74]	SVMs separate the data by finding the hyperplanes that maximise the distance between the nearest training points in each class.	[244, 89, 25, 169, 95, 248, 130, 223, 165, 132, 226, 275, 242]
K-Nearest Neighbours	K-Nearest Neighbours picks representative points in each class and when presented with a new observation calculates its proximity to the points and assigns it to whichever is closest.	[130]
Naive Bayes [168]	Uses Bayes' Theorem to predict the probability that a sample belongs to each category (malicious or benign) and then assigns it the category with the highest probability. It's described as naive due to the fact that it assumes that features are independent [41]	[130, 275]

Continuation of table		
Random Forest [51]	Random Forest, like AdaBoost, is a collection of classifiers, and, like AdaBoost, the classifiers are all decision trees. However, AdaBoost tends to employ shallow decision trees while Random Forest tends to use deep decision trees. Random Forest splits the dataset between all the decision trees and then averages the result.	[95, 213, 115, 244, 133, 120, 223, 165, 126, 197, 232, 132]

2.4.1 Limitations of Machine Learning

While machine learning equips a machine to automatically distinguish malicious from benign without having to manually construct heuristics, it can still be exploited by attackers in the form of *adversarial attacks*. An adversarial attack is a technique in which confidently classified samples are altered using small, but tactical perturbations in order to cause the classifier to incorrectly classify the sample with confidence. Adversarial attacks first emerged in the field of image recognition, where the alterations made to images was so slight that there was no observable difference between the original and altered images, however, state of the art classifiers incorrectly classified them with extremely high confidence [240]. Adversarial attacks work by estimating the decision boundaries of the classifier and then selectively altering input samples using the smallest number of perturbations necessary so that they fall outside the decision boundary. They can be white-box attacks, in which the attacker has complete access to the classifier, its hyper-parameters, and the input samples it was trained on. Alternatively, they can be black-box attacks where the attacker does not have access to the internals of the classifier but can still view the final classification decision it makes [207, 270]. With white-box attacks, it is relatively trivial to find the classifier's decision boundary

due to all the information available to the attacker. However, with black box attacks, the attacker must create a *surrogate classifier* that is trained on the classification decisions made by the original classifier being attacked. Samples are then modified to evade the surrogate classifier in the hope that they will also evade the classifier being attacked [207]. These attacks have been quite successful as adversarial samples have been found to be transferable between classifiers trained to make the same decision [182]. The success of adversarial attacks in general is attributed to the linear behaviour of some classifiers in high dimensions [103]. Within the field of image recognition, adversarial attacks are performed through the use of minor perturbations to pixel values in images. In malware analysis, the general trend is to alter the API-calls called. Attackers must take care when altering API-calls made by malware as they could unintentionally alter the behaviour such that it no longer executes. Therefore, most of the literature does not subtract or remove system calls made, rather they only add calls to avoid altering any of the malware's existing behaviour. However, attackers still need to be vigilant when adding calls to the feature space, as if a call to ExitProcess is added, it would immediately end execution when called thereby significantly altering the malicious sample's behaviour.

[47] focus on adversarial attacks against Linear SVMs and Neural Networks using malware embedded in PDF files. They consider two attack scenarios, one in which the attacker has perfect knowledge of the model being attacked and one in which the attacker has a limited knowledge of the model being attacked. They use gradient descent as their attack strategy but they bias it by adding a 'mimicry component'. The mimicry component pushes the gradient descent towards the largest cluster of legitimate samples. They found that regardless of the information available to the attacker regarding the target model, they were able to evade it with near identical probability. In addition, they were able to evade Linear SVM models with as few as five to ten modifications to a malicious file, whereas neural networks were slightly more robust.

[110] apply adversarial attacks to the field of malware analysis by attacking a neural

network trained to detect malware that targets the Android platform. Specifically, they train a neural network to obtain the current state-of-the-art performance that has been achieved when classifying the DREBIN dataset [28] — a dataset consisting of malware for Android. The features used are API-calls gathered statically from malware which are represented as a binary vector where '1' means the corresponding call was imported whilst a '0' indicates otherwise. Adversarial samples are crafted by adding features once they have deciphered which feature, when modified, would produce the most change in the classifier's output. They identify these features using the method employed by [183] who take the derivative of the trained neural network with respect to its input features. Through this they manage to make 63% of the previously detectable malware samples undetectable.

[122] propose MalGAN, a adversarial neural network, which takes malware samples and produces adversarial samples that can evade classifiers. They perform a black box attack, assuming that access to the machine learning classifier's internals is not available. They tested MalGAN on a number of classifiers, namely, Random Forest, Logistic Regression, Decision Tree, Support Vector Machine, Multi-Layer Perceptron, and a voting based ensemble of these classifiers. They manage to alter malware samples such that the accuracy of many of the classifiers fell from above 90% to 0%.

2.4.2 Discussion

It is clear that classifiers are not impenetrable, therefore, it's imperative to understand how the classifiers used are working and what their predictions are based on. This is the motivation of the fourth and fifth research questions:

RQ4 Does the traditional dynamic malware analysis process create a bias in the data collected and subsequently classified?

To answer this question, this research studies the features that the classifiers favour when distinguishing malicious from benign (as stated in RQ3). This is done for both the kernel-level and user-level data in order to better understand the differences in viewpoints. This will also provide an understanding of the picture of malware that is built up by classifiers. Using that, this research will test whether classifiers will be able to correctly classify malware falling outside that picture. For example if classifiers are distinguishing malware largely through its Internet usage, what happens when it encounters malware that does not use the Internet? Alternatively, given the prevalence of evasive features in malware, if solutions and classifiers are robust enough to detect malware with evasive features, does that come at the cost of being able to detect other features of malware that have nothing to do with evasion? This is investigated in the fifth research question:

RQ5 How much malicious behaviour can a malware sample exhibit before it is detected?

2.5 Emulating Malware

To answer RQ5, an emulated malware generator is used to create malware that is malicious but does not possess properties of malware that the classifiers rank highly. However, malware emulation is an extremely understudied field, with very few published solutions.

Malware emulation/simulation suites are used for one of two purposes. To educate a user to recognise malware, or to test an anti-virus [105]. One of the first educational suites was the Virus Simulation Suite [117] written by Joe Hirst. It simulated the visible and audible symptoms of malware. Virlab [84] is another well known educational malware simulator. It simulates and visualises the spread of DOS viruses for users. More recently, Spamulator [33] was created to educate students on spyware and bulk-mailing spam. This was extended to also simulate drive-by download attacks [34].

Threat Tracer [137] was proposed to demonstrate the risks of Advanced Persistent Threats (APT). APTs are sophisticated attacks with a long term goal [63]. Threat Tracer was recently extended to also simulate the Mirai Botnet [262]. However these solutions represent the extent of malware simulators for educational purposes.

The literature on malware simulators for the testing of Anti-Viruses is also quite scarce. The Rosenthal Virus Simulator [83] was the first in this category. It is capable of producing harmless programs that contain virus signatures. Trojan Simulator [164] goes slightly further, simulating a property of malware that ensures it is run every time the machine is powered on. However, again, it simulates no malicious symptoms. More recently, MalSim [156] was proposed. MalSim, written in Java Agent DEvelopment framework (JADE) [44], is capable of simulating a rich set of malware variants in addition to generic behaviours seen in malware. However it is careful not to do any actual harm to the system. Unfortunately malware simulators that do not do any harm to the system are quite limiting since the full extent of a malware detector cannot be tested. For this reason, none of the proposed solutions are suitable in this research. The solution chosen is Amsel [190], a Java-based malware emulator. It is described in more detail in chapter 4.

One of the observations made from reviewing malware emulators is that Java [107] is a common choice as a programming language [190, 156, 137, 262]. Java is a general purpose programming language that is platform independent. In keeping with that, Java provides the developer with generic methods, which the Java Virtual Machine (JVM) then translates to more specific methods depending on the platform that the program is being run on. The JVM is what every Java application runs inside (with a new instance created for each application) [255]. In addition to running the Java program specified, the JVM performs a number of jobs on behalf of the developer. For example, the JVM carries out what is known as “Garbage Collection”. Garbage collection refers to the free-ing of any memory no longer used by the Java program [255]. In lower-level languages (such as C), tasks such as these would have to be performed manually by

the programmer, and if the task is not carried out correctly, it can lead to errors that are challenging to debug. Therefore, Java and the JVM in general allows the programmer to focus more on implementing the functionality of the program without having to worry about the minutiae of how that functionality is achieved.

While Java may provide convenience for programmers and lead to fewer errors in code [194], there are questions around its suitability to emulate malware for the dynamic malware analysis process. This is because, when using Java, the developer has very little control over the exact system calls being made. Therefore, though Java can be used to faithfully reproduce the effects of a malware infection, the developer has little control over how the effects are executed. This is essential when monitoring solutions are monitoring at a low level of abstraction. Furthermore, besides the lack of control on how the specifics of the functionality are executed, the calls made by the JVM are also mixed in with the calls made by the program being monitored (emulated malware in this case). This means that calls made for benign purposes are mixed in with calls made for a malicious purpose. For example, the garbage collector must periodically check for any memory to free. This can throw off a machine learning classifier trained on system call data. Conversely, in the C programming language, this must be performed manually by the developer. As a result, the developer can minimise the amount of interference from tasks such as these. Therefore this research also tests the robustness of Java as a malware emulator. This is summarised in RQ6:

RQ6 Are high-level languages such as Java suitable for emulating malware to test system call monitoring tools?

Finally, the findings made in this thesis are evaluated and generalised into principles, that, if followed when performing dynamic malware analysis, will ensure that the results produced from the process are much more robust. This is summarised in the final research question:

RQ7 How can the dynamic malware analysis process be amended to prevent uninten-

ded security flaws from emerging?

Chapter 3

Comparison of User-level and Kernel-level data for dynamic malware analysis

3.1 Introduction

In this chapter, data is collected on malware at the kernel and user level, and then passed through a number of classifiers. The aim of this is to determine if there is a significant difference in the ability of the same classifiers to detect malware when provided with data at different privilege levels. Through doing this, the following research questions will be answered in this chapter:

RQ1 *Does data collected at different privilege levels during dynamic malware analysis affect classification results?*

RQ2 *Is data collected at a global level more beneficial for dynamic malware analysis than that collected at a local level?*

RQ3 *How does the understanding of malware differ at a kernel and a user level?*

In answering these questions, the following contributions are made:

C2 *This thesis contains the first objective comparison on the effectiveness of kernel and user level calls for the purposes of detecting malware.*

C3 *This research assesses the usefulness of collecting data for malware detection at a global system-wide level as opposed to a local individual process level, giving novel insights into data science methods used within malware analysis.*

C4 *This research assesses the benefits, or otherwise of combining kernel and user level data for the purposes of detecting malware;*

C5 *This research studies and identifies the features contributing to the detection of malware at kernel and user level and the number of features necessary to get similar classification results, providing valuable knowledge on the forms of system behaviour that are indicative of malicious activity;*

C9 *This research contributes a driver that hooks all but one call in the SSDT and gathers calls at a global level.*

The remainder of this chapter is structured as follows; the 'Method' section describes the experiments carried out, the 'Results' section describes and interprets the results obtained from those experiments, and the 'Conclusion' section summarises the findings.

3.2 Method

In order to conduct the experiments required for this study, 2500 malicious samples were obtained from VirusShare [18] and 2500 benign samples were obtained from SourceForge [14] and FileHippo [6]. This sample size correlates with dataset sizes used in previous literature [245, 70, 75, 97]. The categories of malware collected for the experiments are shown in table 3.1. Extracting categories for malware is a challenging task as there is no agreed naming convention for malware [135]. In order to

obtain this information, VirusTotal [246] was used. VirusTotal scans any files a user submits to it using over 60 different Antiviruses. It then reports the findings (amongst a plethora of other information) from each of the antivirus products. If a file is found to be malicious, VirusTotal shows the label attached to the file by each of the antivirus products. Developers can access this information via VirusTotal's API. One of the observations made during this research is that the Antiviruses within VirusTotal do not often agree on the label they assign to malware. Furthermore, the naming conventions used also tend to differ. Therefore, in order to get the most balanced view of the categories of malware, the labels given to a malware sample by each antivirus are used. The labels don't just identify the category of the malware sample, but also it's family, the Operating System it's compatible with, the file format, the programming language, the variant, and any additional information. Therefore, in order to obtain the category of each malware sample, the label had to be split on a selection of punctuation characters such as ':' and '.' since each antivirus product used such characters to separate each piece of information. Then a number of heuristics were used to remove information such as the platform or programming language and isolate the category assigned to the malware sample by each antivirus. Finally, the category assigned to the malware sample by the majority of vendors was used as the final category. Benign samples were also run through VirusTotal to ensure that they were not malicious.

Category	Quantity
Trojan	1846
Virus	458
Worm	86
Rootkit	34
Ransomware	23
Adware	22
Keylogger	2
Spyware	2

Table 3.1: Quantity of each category of malware in the dataset

Due to the lack of agreement over the use of each of the terms in table 3.1, it is difficult to objectively define each category. However, in general, the following is meant by each category:

Trojan: Malicious samples disguised as benign samples are referred to as Trojans. A user typically willingly downloads the infected file believing it to only perform the benign function it advertises [134].

Virus: Viruses differ from Trojans in that they usually obtain access to the victim's machine via a vulnerability. On infecting a host, they usually attempt to infect additional files on the victim's machine [134].

Worm: A worm is a self-replicating program capable of infecting multiple machines via a network connection (unlike viruses). Worms are often employed in targeted attacks where a particular user is the intended victim [134].

Rootkit: A rootkit is rarely found on its own but as part of another piece of malware. This is because the sole purpose of a rootkit is to hide the presence of a malware sample. In addition, it is responsible for ensuring that the attacker continues to have access to the machine. In order to hide the attackers presence from the antivirus, rootkits

sometimes employ hooking techniques to control the output/feedback being given to the antivirus by the OS. Therefore, it is quite common for rootkits to work in kernel-mode due to the privileges it brings. This is important because if a monitoring solution is only monitoring at a user-level, anything bypassing user-mode and going straight to kernel-mode will not be recorded [49].

Ransomware: Ransomware takes a user's files or machine hostage by preventing the user from accessing them and then demands payment (usually in the form of a crypto-currency) for their safe release.

Adware: Unlike many forms of malware, Adware make their presence known to a user by bombarding them with advertisements [32]. Adware is delivered in a number of ways, including as a hidden add-on to a program a user installs or as a drive-by download [68].

Keylogger: Keyloggers silently record all keys pressed by a user and the application they were pressed in. Keyloggers can come in the form of a hardware device or be implemented in software. Within software, they can be implemented at user-mode or kernel-mode.

Spyware: Spyware is known for its silent invasive monitoring of user activity and behaviour. Spyware is frequently paired with Adware and therefore has similar delivery mechanisms to Adware.

To gather calls made at a kernel level, a Windows Kernel Driver was written to hook all but one kernel call in the SSDT since none of the tools available currently provide this. The only call the driver does not hook, NtContinue, was not hooked due to the fact that hooking it produced critical system errors. A bespoke kernel driver had to be created for this task since many of the existing tools that hook the SSDT only monitor calls in a specific category (such as calls relating to the file system or registry) and have no objective justification as to why they monitor those calls. Therefore, a bespoke kernel driver was written to hook all the calls in the SSDT to ensure that it can detect any

subtle details regarding malware behaviour. This also allows it to provide a more objective recommendation on the most important calls to hook when detecting malware. Hooking all system calls is quite challenging as it requires exceptional performance on the part of the kernel driver since as soon as its loaded, it is inundated with calls made to the kernel. Therefore, if it cannot handle these rapidly, the system will crash. In addition, due to the number of times some system calls are made, memory use must be limited as much as possible, even ensuring each string is not taking up more space then needed. Errors caused to the system by the driver are relatively easy to spot since even the smallest of errors cause a blue screen of death. However, these can be quite difficult to locate in the driver source code since the details about the cause of the error are only made available through a memory dump. The driver was further stress-tested using *Driver Verifier* [82]. Driver Verifier is a tool supplied by Windows that runs a number of tests on a selected kernel-mode driver. Driver Verifier can check for errors such as memory leaks and insufficient error handling (particularly with regards to resource usage) amongst many other things. If Driver Verifier discovers an error, it throws a system error (Blue Screen of Death), and provides details of the error in a memory dump. In order to ensure correctness, the driver created in this research was run through all the tests provided by Driver Verifier. In addition, the driver has been written carefully to ensure that it does not gather data regarding its own behaviour, but only behaviour external to it. The driver is unique in that it collects the SSDT data at a global system-wide level as opposed to a local process-specific level. This has been done to determine whether collecting data at a global level assists in detecting malware or whether it is simply adding noise. Therefore, the data from the tool can be used to predict whether the machine's state is malicious or not.

While SSDT hooks have been used in drivers previously, they have not had as comprehensive a coverage of calls as the kernel driver used in this research has. [158] employed an SSDT hook to automatically build infection graphs and construct signatures for their system, AGIS (Automatic Generation of Infection Signatures). AGIS then monitors a program to see if it contravenes a security policy and matches a signa-

ture. Therefore, it only focuses on calls from a specific process and ignores all other calls. [143] propose BareBox to counter the problems associated with malware capable of detecting that it is being run in a virtual environment. BareBox runs malware in a real system and is capable of restoring the state of a machine to a previous snapshot within four seconds. BareBox monitors what the authors perceive to be important system calls using an SSDT hook. However, as the number of devices attached to the machine increase, the time it takes BareBox to restore the system to a benign state increases considerably. [109] propose BehEMOT (Behavior Evaluation from Malware Observation Tool) which analyses malware in an emulated environment first, then in a real environment if it does not run within the emulated environment. They use an SSDT hook to monitor API calls relating to certain operations. However, by performing analysis on a real environment, BehEMOT suffers a similar problem to BareBox in relation to restoration time. Furthermore, the focus with BehEMOT seems to be producing human-readable and concise reports after each analysis and therefore, only small-scale tests were conducted on a handful of samples.

As mentioned previously, the kernel driver created for this research differs from other solutions using SSDT hooks in that they only log calls made to certain API calls by certain processes. The bespoke kernel driver logs all calls (except one) by all processes in order to determine their utility in classification. TEMU is the only tool to offer similar functionality, however, where it differs is that it runs in an emulated environment (which is easier for malware to detect [57]) and is focused on providing instruction-level details as opposed to high-level system calls.

To gather user level data, a third party tool that is readily available was used since there are already well established solutions providing this. Specifically, the tool used is the one that is most frequently mentioned in the existing literature - Cuckoo (specifically, Cuckoo 2.0.3). Cuckoo is a sandbox capable of performing automated malware analysis. Cuckoo provides a whole host of features from simulated user interactions

with the desktop to VM hiding techniques to prevent malware from detecting its environment. In addition, since Cuckoo is open source, it integrates with a number of other tools such as inetsim [123], volatility and many others. When given a sample, Cuckoo can return the VirusTotal [246] results, a memory dump, the network capture and system calls, amongst many other things. For the purposes of this research, only the system calls returned are used. As mentioned previously, Cuckoo intercepts system calls using an inline hook.

The experiments for this research were carried out on a virtual machine with Windows XP SP3 installed. The reason for choosing Windows XP was that writing a kernel driver, particularly one delving in undocumented parts of Windows, is frustratingly challenging. However, this is made slightly easier in Windows XP due to the fact that it has slowly become more documented through reverse engineering. Another reason for choosing XP is that all 64 bit systems are backwards compatible with 32 bit binaries [112] and the most commonly prevailing malware samples in the wild are also 32 bit [62] (with not a single 64-bit sample appearing in the top ten most common samples). As of 2016, AVTEST found that 99.69% of malware for Windows was 32 bit [31]. The reason for the popularity of 32 bit malware samples over 64 bit is that its scope is not limited to one architecture. Therefore, given the current prevalence of 32 bit malware, it did not seem that using Windows XP would make the results any less relevant especially since the method used could be repeated on other versions of Windows and it would simplify the already challenging engineering task. The host OS was Ubuntu 16.04 and the Hypervisor used was VirtualBox [12]. Both the host and guest machine had a connection to the Internet. In order to ensure fairness and to provide automation, the simulated user interaction features present within Cuckoo were implemented for the kernel driver. Aside from the sample being investigated and simulated user behaviour, the only other processes running on the guest machine were the standard Windows processes. Figure 3.1 shows the system diagram used for this research describing the entire experimental process in order to obtain the results.

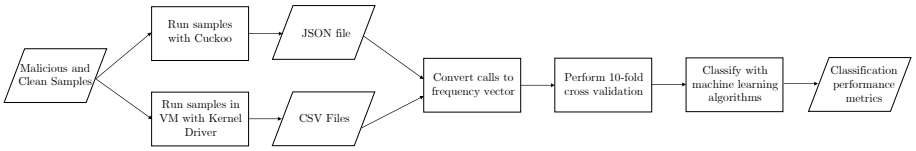

Figure 3.1: Workflow Diagram of the proposed system's pipeline

The kernel driver creates one Comma-Seperated Values (CSV) file for each system call. A new line is written to each file every time the system call associated with the file is called. The logs are placed in a directory under the Windows directory since that seemed the safest location given that programs are unlikely to modify files essential for the OS to function. In fact, it has also been known to be used by malware to hide its files due to the safety it provides [227]. After the analysis, a shared folder is used to transfer over the CSV files to the analysis machine. This folder is wiped after each run. Cuckoo uses a network connection to transfer over analysis files from the VM to the host machine, after which we transfer the JSON file to the analysis machine. The output produced from each of the monitoring tools is encoded using a frequency histogram of calls within a two minute period. This feature representation is used to fit a classification model for virus detection.

3.2.1 Initial Experiments

The transformed data (frequency histograms) from Cuckoo and the kernel driver was then classified using a selection of machine learning algorithms provided by scikit-learn [56]. The machine learning algorithms chosen were drawn from the existing literature, as the focus of this research was on the utility of different views of machine-level actions (user vs kernel) rather than new classification algorithms. The classification algorithms used were AdaBoost, Decision Tree, Linear SVM, Nearest Neighbours, and Random Forest. The reason these algorithms were chosen is that they are used widely in the literature as shown in table 2.2. In addition, Random Forest frequently achieved impressive results [95, 213, 115, 244] as has AdaBoost [244]. Finally, Nearest Neigh-

bours was chosen due to its simplicity in order to set a baseline.

For each classifier, the data was split using 10-fold cross-validation as it is also the standard in this field [46, 244, 25, 97]. 10-fold cross-validation (an implementation of k-fold cross-validation) is a statistical method for splitting data in a manner that minimises bias. 10-fold cross-validation randomly splits the data into 10 subsets and then trains the classifier on 9 subsets and tests it on 1. Each subset gets a chance to be the test set. In addition, to increase confidence in the results, 10-fold-cross-validation was run 100 times, providing 1000 classification results for each classifier. To measure a classifier's performance a number of metrics can be obtained. The metrics used in this research are Area Under the Receiver Operating Characteristic (ROC) Curve (AUC), Accuracy, Precision, and F-Measure since these are the metrics commonly reported in the literature [169, 115, 245, 25, 133] and they provide a complete view of the performance of the classifier without missing out on subtle details. To understand these measures in this context, it is important to define a few basic terms. For this research True Positives (TP) are interpreted as malicious samples that are correctly labelled by the classifier as malicious. False Positives (FP) are benign samples that are incorrectly predicted to be malicious. True Negatives (TN) are benign samples that are correctly classified as benign. False Negatives (FN) are malicious samples that are incorrectly classified as benign. With regards to the actual measures used, AUC relates to ROC curves. ROC curves plot True Positive Rate (TPR) against False Positive Rate (FPR). FPR is the fraction of benign samples misclassified as malicious, while TPR represents the proportion of malicious samples correctly classified. A ROC curve shows how these values vary as the classifier's threshold is altered. Therefore the AUC is a good measure of a classifier's performance. Accuracy can be described as the sum of all the correct predictions (malicious and benign) divided by the sum of all the predictions. Precision refers to the correctly labelled malware divided by the sum of the correctly labelled malicious samples, and the benign samples incorrectly labelled ($\frac{\mathrm{TP}}{\mathrm{TP+FP}}$). This gives the proportion of correctly labelled malware in comparison to all samples labelled as malware. Recall is the correctly labelled malicious samples divided

by correctly labelled malicious samples, and malicious samples incorrectly labelled as benign ($\frac{\text{TP}}{\text{TP + FN}}$). This gives the proportion of malicious samples that are correctly identified. Precision has been included since false positives are a common issue in malware detection. Recall was not included for brevity and since it can be quickly deduced from the F-Measure (which is included) which is the harmonious mean of precision and recall.

In order to confirm whether the differences in classification results were statistically significant or due to randomness, the 1000 classification results (specifically AUC values) obtained from running 10-fold cross-validation 100 times were utilised. These values were plotted using Q-Q Plots against a normal distribution. The Q-Q plot provides a visual comparison of a dataset's distribution with a chosen theoretical distribution. In this case the theoretical distribution being compared against is the normal distribution. Provided the Q-Q plots show the data as being normally distributed, the required prerequisites for using Welch's T-Test [258] are satisfied. Welch's T-Test tests whether two populations have equal means. In this case Welch's T-Test is used to determine whether the differences between the classification results from Cuckoo and the kernel driver are statistically significant or not (with the significance level, α, set to 5%). Welch's T-Test was chosen due to its robustness and widespread recommendation in the literature [76, 212].

In addition, in order to gain insight into whether collecting data at a global level is more beneficial for classifying malware, the API calls logged by the kernel driver were reduced to just those coming from the process that was being monitored (and any child processes that it spawned). It's important to note that though the data from the kernel driver is limited to mimic the scope of the data that Cuckoo provides, it will still not provide the same data as Cuckoo. This is due to a number of reasons. Importantly, the localised kernel driver is still monitoring a different API to Cuckoo. Therefore, there are a number of Windows calls that the driver is able to observe that Cuckoo is not able to and vice versa. Furthermore, rootkits typically operate at kernel-mode and

therefore would still only be visible to the kernel driver. Therefore it will be useful to determine how the effectiveness of the data from the kernel driver differs when its scope is limited.

Finally, the data from Cuckoo and the kernel driver was combined and then classified. This was done to determine if the combination of user- and kernel-level data would improve classification results.

To further understand the data recorded from the kernel and user level, and confirm whether the features being used differ depending on the data collection method used, the features were ranked by importance using two metrics, the independent feature ranking metric and the inbuilt feature ranking metric, for the classifier that had the best classification results.

3.2.2 Independent Feature Ranking

For the independent feature ranking metric, the classifier is only given the data from one feature (or API-call) at a time. The classifier therefore uses only one feature to differentiate malicious and benign. The AUC scores obtained from each feature are noted. This method can give an indication of the strength of individual features. Where it lacks, however, is in its ability to account for the relationship between features. For example, a feature on its own may not be that strong, but when paired with another, may be very strong. Therefore, to account for that, an additional feature ranking method is used.

3.2.3 Inbuilt Feature Ranking

This feature ranking method ranks features using each classifier's inbuilt feature ranking mechanism. This ranking mechanism works in different ways depending on the classifier used. For Decision Trees scikit-learn uses the *Gini importance* as described

here [52]. The same is true for Random Forests and AdaBoost since they are composed of a multitude of Decision Trees. The only difference being that, as they are composed of multiple Decision Trees, the importance is averaged over each one. Finally, with Linear SVMs, the coefficients assigned to each feature is used to rank them. In the case of K-Nearest Neighbour, there is no inbuilt feature ranking mechanism, therefore, it is not included in this measure.

3.2.4 Global Feature Ranking

The independent and inbuilt feature ranking mechanism show the most influential features one classifier at a time. While this is useful in showing how each classifier individually chose to recognise malware, where it lacks is in providing a overarching narrative showing the commonalities between each of the classifiers. This can be particularly difficult to manually determine if many classifiers are used. Therefore to better understand the commonalities between classifiers when it came to distinguishing malware, a new aggregate measure was created.

The purpose of the aggregate measure is to rank features across all the classifiers for both the inbuilt and independent feature ranking methods. This would highlight which features are robust since the previous measure only shows the top ten for a chosen classifier — which could arguably be skewed in its favour. The aggregate measure was calculated as follows. For each classifier, the features were ranked according to the score they were given by the independent or inbuilt feature ranking method. Then, the rank was plotted on the x-axis from 0 (the best rank) to the total number of API-calls (the worst rank) for each feature. On the y-axis was a score from 0 to 1 and at each rank $\frac{1}{\text{number of classifiers}}$ was added to the score. Once this was done, the area under the curve was found which represented the total strength of the features across all classifiers. This global feature ranking method can be used with any local feature ranking method. Figure 3.2 shows an example of this global feature ranking method on a dataset with 250 features. In figure 3.2, the feature in question has got the ranks 0, 20, 50, and

200 in the four classifiers it was used with. At each rank, the value has gone up by 1/4 (since there are four classifiers). If a feature was ranked as the most useful feature across all classifiers, its ranks would be 0, 0, 0, and 0, and therefore the area under the curve for it is 1 as shown in figure 3.3.

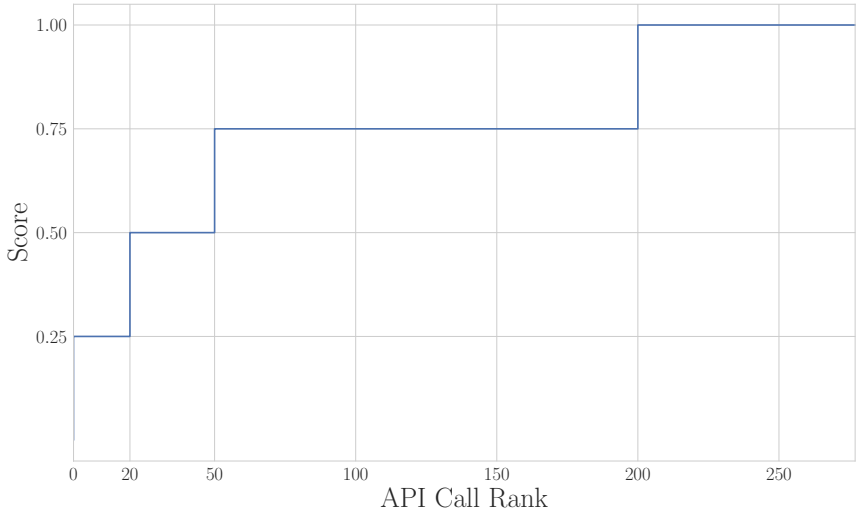

Figure 3.2: Example graph of feature ranking mechanism

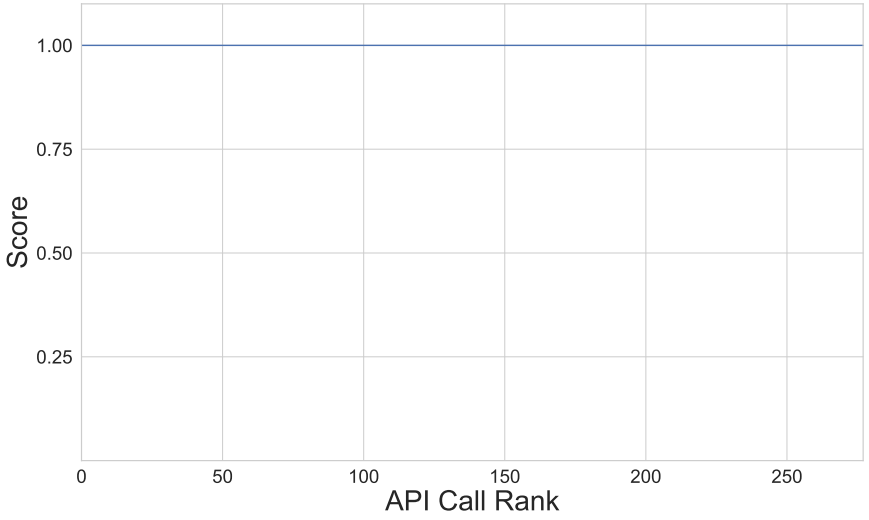

Figure 3.3: Example graph of feature ranking mechanism with perfect score

The pseudocode of the global feature rank method that is used to find the global rank of a particular feature is detailed in Algorithm 1.

Algorithm 1: Find global rank for a given feature

Input : An integer array $ranks\_obtained$ containing the ranks obtained by a feature from each classifier

Output: The global rank (auc) for the feature

1 curve_array $\leftarrow [0] \times number\_of\_features$;
2 **for** $rank \in$ ranks_obtained **do**
3 curve_array$[rank] \leftarrow$ curve_array$[rank] + \frac{1}{number_of_classifiers}$;
4 **end**
5 **for** $i = 1$ **to** $number\_of\_features$ **do**
6 curve_array$[i] \leftarrow$ curve_array$[i]$ + curve_array$[i-1]$;
7 **end**
8 total $\leftarrow$ `sum(`curve_array`)`;
9 auc $\leftarrow \frac{total}{number\_of\_features}$;
10 return auc;

The function is provided with an array sorted in ascending order which contains the ranks that has been assigned to a feature by each classifier used according to some feature ranking methodology (such as inbuilt or independent). As seen in line 1, $curve\_array$ is an array of size $number\_of\_features$ in which every element is initialised to 0. In lines 2 and 3, the rank is used as an index into the array, and the value at that index is added to $\frac{1}{number\_of\_classifiers}$. The $curve\_array$ is then looped through and every element in the array is equal to its current value added to the previous element's value. At this point the $curve\_array$ will represent curves such as those shown in figures 3.2 and 3.3. Finally, the average of $curve\_array$ is found which represents the feature's global rank.

3.2.5 Feature Ranking Evaluation

In order to verify that both of the feature ranking methods were selecting features that are optimal, and that the results they produced could be relied on, an additional experiment was conducted. In this experiment, the AUC was calculated using only the top ‘x’ features where ‘x’ was gradually increased from 10 by increments of 10 up to the total number of features. This will also show the minimum amount of features necessary to obtain similar classification results to those obtained when using all the features.

3.2.6 Additional Data Analysis

To gain insight into the differences in behaviour between malware and benignware, the data is studied further using simple analysis techniques. To begin with, the mean frequency with which each call is made by malware and benignware is plotted and compared. This will reveal differences in both malicious and benign behaviour as well as differences in Cuckoo and the kernel driver’s data collection methods.

In addition to studying the frequently called features, the features exclusive to malware or benignware are also studied. In order to glean this data, a binary feature vector is created where ‘1’ represents a feature being present and ‘0’ represents a feature being absent. For each malicious sample, its call-histogram is iterated through, and for each call that is called at least once by a malicious sample, a ‘1’ is added to the binary feature vector. The same is done for the benignware data. Finally the binary feature vector for malicious samples is subtracted from the binary feature vector for benign samples. The resulting vector will contain ‘1’ for features present in benign samples but not malicious samples and ‘-1’ for features present in malicious samples not present in benign samples. If the unique feature only appears in a few samples (less than 15), it is ignored as it is considered an outlier. This comparison is performed for the both the Cuckoo and kernel data independently. In addition, this analysis is carried out to

understand the differences between the different data collection methods (such as local and global).

3.2.7 Sample Size Verification

In order to verify that the sample size chosen was suitable, the initial experiments described above were conducted using different sample sizes. Specifically, 10-fold cross-validation was conducted with the classifiers being used in this chapter, however, the classifiers were gradually given larger subsets of the whole dataset. The sample size was increased from 100 samples up to over 2000 in increments of 100. For each sample size, 10-fold cross-validation was repeated 100 times and the average AUC was recorded. The AUC values were plotted in order to observe when the curve plateaued for each classifier.

3.3 Results

In this section, the results from classifying the data collected at a kernel and user level are described. In order to understand the contributing factors to the results, additional experiments are conducted using modified forms of the data. Feature ranking is used (among other things) to analyse the ten most significant features in order to gain a better understanding of what the machine learning algorithms are using to identify malware.

3.3.1 Initial Experiments

The results from classifying data collected using the kernel driver at a global level and data collected from Cuckoo are shown in Table 3.2

Machine Learning Algorithm	Kernel Driver				Cuckoo			
	AUC	Accuracy	Precision	F-Measure	AUC	Accuracy	Precision	F-Measure
AdaBoost	0.983	94.1	0.934	0.941	0.973	91.8	0.911	0.920
Decision Tree	0.944	92.3	0.906	0.925	0.943	87.8	0.918	0.913
Linear SVM	0.945	90.3	0.873	0.906	0.932	86.9	0.835	0.870
Nearest Neighbour	0.964	90.3	0.896	0.903	0.942	86.2	0.877	0.863
Random Forest	**0.986**	**95.2**	**0.960**	**0.944**	**0.984**	**94.0**	**0.958**	**0.942**

Table 3.2: Comparison of classification results of data from Cuckoo and kernel driver.

On the whole, the results show that the data from the kernel driver is marginally better for the purposes of differentiating between benign and malicious states regardless of the machine learning algorithm used. The algorithm with the best performance for both the kernel driver and Cuckoo was Random Forest, obtaining an AUC of 0.986 and 0.984, and an accuracy of 95.2 and 94.0 respectively. In addition, it was found that on average, 93% of the samples were given the same label regardless of the data used by the best performing classifier (Random Forest). This shows that while there was agreement on a large number of samples, there were still some samples where data from one was better than the other for detecting malware.

In order to verify whether the difference between the kernel and Cuckoo classification results are statistically significant and not just occurring by chance, Welch's T-Test was performed on the AUC values as described earlier. A prerequisite for using Welch's T-Test is that the data must be normally distributed. This was verified using Q-Q plots as shown in Figure 3.4.

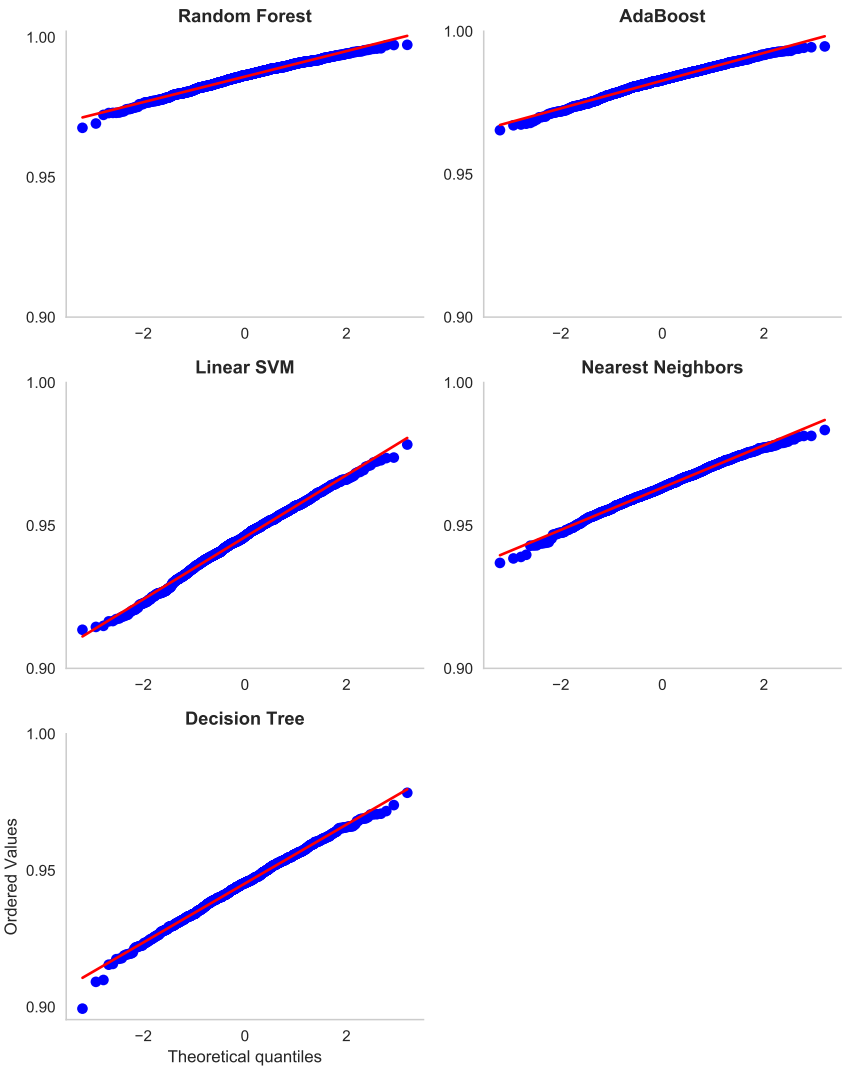

Figure 3.4: Q-Q Plots of AUC values from kernel data

The Q-Q plots show the distribution of the AUC values and how closely (or otherwise) it relates to the normal distribution (shown as a red line). The plots show that the AUC values barely deviate from the normal distribution, and therefore, Welch's T-Test would be an appropriate test to observe if the difference between the kernel and Cuckoo values are statistically significant. Given that the Q-Q plots for the Cuckoo data were very similar, they are not shown here for brevity.

In Welch's T-Test, the null hypothesis is that the means are equal (i.e., H_0: $\boldsymbol{\mu}_1 =$

$\boldsymbol{\mu}_2$), and therefore the alternative hypothesis is that the means are not equal (i.e., H_a: $\boldsymbol{\mu}_1 \neq \boldsymbol{\mu}_2$). The threshold α value was set to 0.05 as it is an appropriate level for the experiments. Therefore if the p-value returned from performing Welch's T-Test is less than α, the null hypothesis can be rejected. Table 3.3 shows the results of performing Welch's T-Test on the AUC values from each classifier.

Machine Learning Algorithm	p-value
AdaBoost	$1.80 \times e^{-208}$
Decision Tree	$1.41 \times e^{-6}$
Linear SVM	$8.41 \times e^{-78}$
Nearest Neighbour	$9.29 \times e^{-290}$
Random Forest	$2.29 \times e^{-10}$

Table 3.3: p-values returned from Welch's T-Test using AUC values

As Table 3.3 shows, the p-values returned are considerably lower than the threshold, 0.05. Therefore, the null hypothesis is rejected meaning that the means of the kernel and Cuckoo AUC values for each classifier are not the same. This shows that, at a significance level of 0.05, the difference between the kernel and Cuckoo results are statistically significant and not just due to chance. Therefore, it can be concluded that the data collected at the kernel level produces slightly better classification results than that collected at a user level.

3.3.1.1 Independent Feature Ranking

In order to further understand and confirm the differences between the data gathered by Cuckoo and the kernel driver, the top ten features using the independent and inbuilt feature ranking methods are compared. Table 3.4 compares the top ten features (in order of score) using the independent feature ranking method for Cuckoo and the kernel driver. The feature importance is shown only for Random Forest since it had the best

performance. While it would have been ideal to show a comparison of all the calls rather than simply the top ten, due to space restrictions, it was limited to ten.

Cuckoo	**Kernel Driver**
GetSystemMetrics	NtQueryDebugFilterState
LoadResource	NtEnumerateKey
FindResourceExW	NtQueryFullAttributesFile
NtQueryInformationFile	NtReleaseSemaphore
SetFileTime	NtEnumerateValueKey
NtUnmapViewOfSection	NtReadVirtualMemory
NtOpenSection	NtSetInformationProcess
NtWriteFile	NtSetValueKey
FindResourceA	NtOpenEvent
CreateDirectoryW	NtNotifyChangeKey

Table 3.4: Top ten features using independent feature ranking with Random Forest.

From table 3.4, it can be seen that there are no features in common between Cuckoo and the kernel driver within the top ten using the independent feature ranking method. This suggests that both views used very different indicators to distinguish malware. In terms of the actual methods in the top ten for each tool, Cuckoo contains some highly specific calls such as SetFileTime (to set MAC (modify, access, and create) times on a file) and GetSystemMetrics (to get information about the system). The presence of SetFileTime is not surprising as it is often used by malware to conceal its accesses of a file (and thereby conceal its malicious activity) [227]. GetSystemMetrics is used by malware to evaluate whether it is running in a virtual environment or a real one (since virtual machines tend to have low memory and storage). NtUnmapViewOfSection (and NtMapViewOfSection) can also be used to evade detection as malware can use it to replace the code of a legitimate process in memory with its code so that the legitimate process runs its code. This could be the reason why the kernel driver monitoring at

a global level performed better than Cuckoo monitoring at a local level as it was able to capture this behaviour better. The top ten also includes some methods relating to resources (LoadResource and FindResourceExW), malware tends to hide its payload inside the resource section of a PE file, and therefore these methods would be used to extract it into memory. What is also noticeable in Cuckoo's top ten is a mix of calls from the native API (usually starting with Nt) and Win32 API. An example of that is NtQueryInformationFile, used to obtain information about a file. The reason for malware using this method over an equivalent Win32 call is that it provides more information.

On the other hand, the kernel driver contains relatively generic calls relating to the registry, threading, memory, events, and processes. However, there are a few interesting calls on the kernel side. There is the method NtSetInformationProcess, which has been known to be used by malware to disable Data Execution Prevention (DEP). DEP is a protection in memory which prevents malware from running code in non-executable sections of memory [22]. Another method in the top ten likely to be directly related to malware is NtNotifyChangeKey. This is used by a process to ask Windows to notify it whenever any changes are made to the registry. This could be used by malware to monitor what is being done on the system or even prevent any changes to the keys that it created. On the whole, it's clear that the vast majority of features favoured by classifiers to distinguish malware in the Cuckoo data are the evasive features of malware, whereas the kernel driver uses differences in the general behaviour of malware to distinguish it from benignware.

3.3.1.2 Inbuilt Feature Ranking

Table 3.5 shows the top ten features using the inbuilt feature selection method.

Cuckoo	Kernel Driver
GetSystemMetrics	NtWriteFile
FindResourceA	NtFlushVirtualMemory
LdrGetProcedureAddress	NtReadFile
LoadResource	NtUnlockFile
NtReadFile	NtOpenMutant
NtQueryInformationFile	NtLockFile
SetFileTime	NtNotifyChangeDirectoryFile
GetFileAttributesW	NtOpenEvent
NtOpenSection	NtDeleteAtom
NtUnmapViewOfSection	NtQueryValueKey

Table 3.5: Top ten features using inbuilt feature ranking with Random Forest

Much of the discussion about the top ten features in Cuckoo for table 3.4 applies to the features of Cuckoo in table 3.5. However, unlike table 3.4, there is one method in common between the kernel and Cuckoo features, NtReadFile. This suggests that this feature is important regardless of the perspective from which data is being gathered. Another interesting observation is that there are seven methods in common between Cuckoo's independent (Table 3.4) and inbuilt feature ranking (Table 3.5). This suggests that many of the contributing features in Cuckoo's case can be used alone to detect malware (which is worth considering when selecting feature representation methods). Due to this, many of the observations made about Cuckoo's top ten in Table 3.4 apply here (such as Cuckoo focusing more on malware's evasive behaviour over the general behaviour of malware). Aside from this, Cuckoo's top ten in Table 3.5 also contains LdrGetProcedureAddress. This is important as it can be used by malware to evade static analysis and dynamic heuristic analysis by loading all the routines it needs at runtime and therefore malware can achieve all that it intends to with only that method linked at compile time.

On the kernel side, there is one method in common between the inbuilt and independent feature ranking method, NtOpenEvent. This is no surprise as this method can be used to interact with Windows Events which malware could use to ensure it is run every day, for example. In general, the top tens for the kernel data for both tables are more focused on differences in general process behaviour with fewer methods directly related to specific behaviour exhibited by malware. However, there are a few exceptions.

One such exception is NtNotifyChangeDirectoryFile, a completely undocumented method. This method is used by a process to ask Windows to notify it when any changes occur in a directory. Malware could be using it to simply monitor system activity and protect itself or to attach itself to any file moves. However, another possible reason is that this method is responsible for a well publicised vulnerability [9] that could be used to expose parts of kernel memory and defeat Address Space Layout Randomisation (ASLR). NtNotifyChangeDirectoryFile is not the only undocumented method in the top ten; NtDeleteAtom and NtOpenMutant are also completely undocumented by Windows. This could explain why the kernel data was able to better distinguish malware from benignware as it is able to capture behaviour that cannot be captured at user level. Aside from that, the differences in general process behaviour are being used to detect malware.

In conclusion, tables 3.4 and 3.5 demonstrate that Random Forest utilises different behavioural aspects to identify malware . While Cuckoo and the kernel driver generally monitor equivalent calls, the fact that the observed rankings are different suggests that the scope (local or global) of the calls is an important factor. Another contributing factor could be that malware evades or detects the inline API hooking technique used by Cuckoo but not the SSDT hooking method employed by the driver (since it requires a more sophisticated approach to evade).

3.3.1.3 Call Frequencies

To gain a better understanding of the data and the differences in each data collection method, the mean frequency with which each system call was called was plotted as histograms in figures 3.5 and 3.6. In order to keep the graph neat, each system call name has been replaced by an index, hence each number on the x-axis represents a single system call.

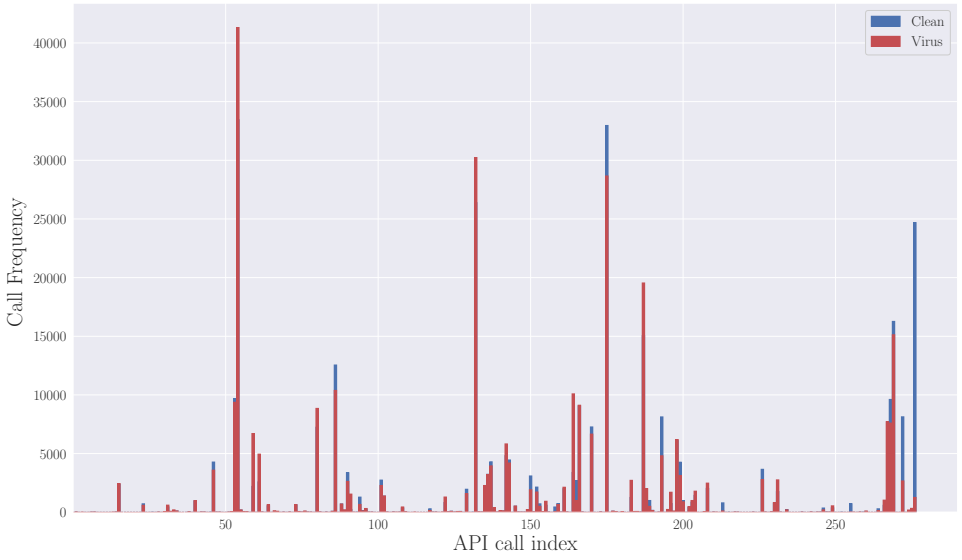

Figure 3.5: Mean call frequency (y-axis) for each call (x-axis) as recorded by the kernel driver.

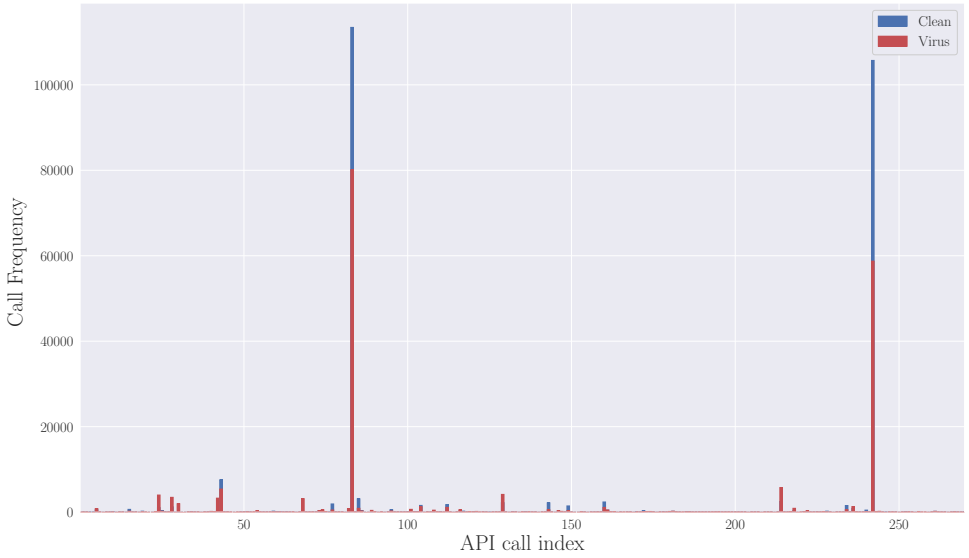

Figure 3.6: Mean call frequency (y-axis) for each call (x-axis) as recorded in Cuckoo.

These graphs show some of the overarching differences between each data collection method. The data from Cuckoo only has two real peaks, interestingly, for both those peaks, the mean value for malware is lower than that for benignware. On the other hand, the kernel data has more peaks due to the fact that it is collected globally and therefore representing all processes running on the system.

In order to better interpret the graphs, the top ten most frequently occurring calls in malware and benignware are listed for Cuckoo and the kernel driver. Table 3.6 shows the top ten for Cuckoo.

Benignware	Malware
NtReadFile	NtReadFile
SetFilePointer	SetFilePointer
GetSystemTimeAsFileTime	NtClose
NtWriteFile	GetSystemTimeAsFileTime
NtClose	RegCloseKey
NtAllocateVirtualMemory	NtDelayExecution
RegCloseKey	RegOpenKeyExA
RegOpenKeyExW	NtCreateFile
RegQueryValueExW	RegQueryValueExA
LdrGetProcedureAddress	GetAsyncKeyState

Table 3.6: Most frequently occurring features in benignware and malware for Cuckoo.

Table 3.6 contains many features in common between benignware and malware (NtReadFile, SetFilePointer, NtClose, GetSystemTimeAsFileTime and RegCloseKey), and some interesting differences. NtDelayExecution is among the most frequent system calls used by malware which is unsurprising since it is commonly used by malware to hide it's behaviour. Malware calls this method to sleep for a duration of time before finally executing. This can be quite effective since most analysis tools only tend to run for a few minutes at most [179]. Given how frequently it is called, this could suggest that the malware samples being used are not showing much of their malicious behaviour. Another method that is known to be used by malware is GetAsyncKeyState. This allows malware, or, more specifically, keyloggers, to poll the state of keys to determine what keys have been pressed [227].

Another interesting observation from table 3.6 is that malware calls the methods RegOpenKeyExA and RegQueryValueExA with high frequency, whereas, benignware calls RegOpenKeyExW and RegQueryValueExW with a high frequency. These methods

are essentially the same, as, within Windows, methods ending with 'A' refer to the ASCII version of the call, whereas methods ending with 'W' refer to the Unicode version of the call. The ASCII version of every call eventually calls the Unicode version. To gain further insight into the extent of this difference in system calls, the frequency with which all ASCII calls were used by benignware and malware have been plotted as histograms. The same has been done for Unicode calls. This is shown in figures 3.7 and 3.8.

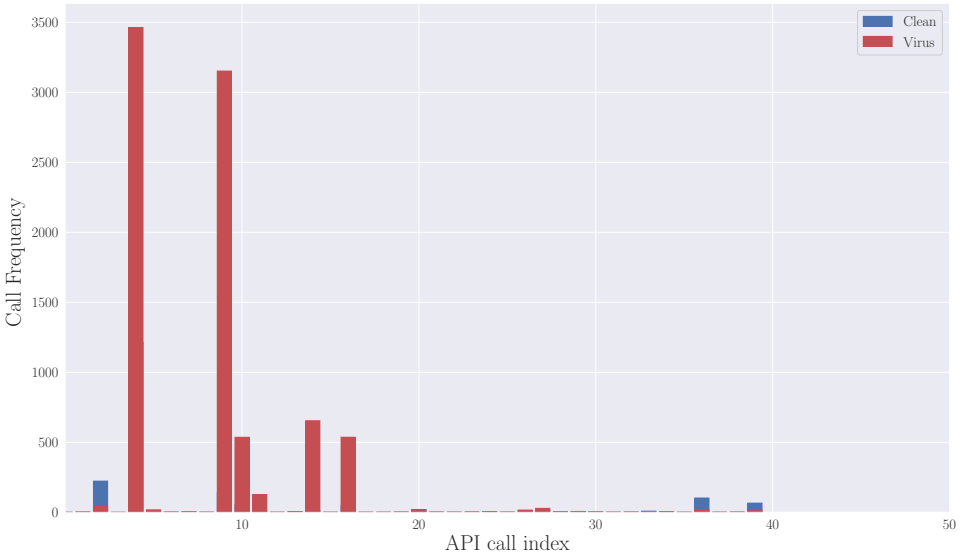

Figure 3.7: Mean call frequency (y-axis) for ASCII calls (x-axis) from Cuckoo data.

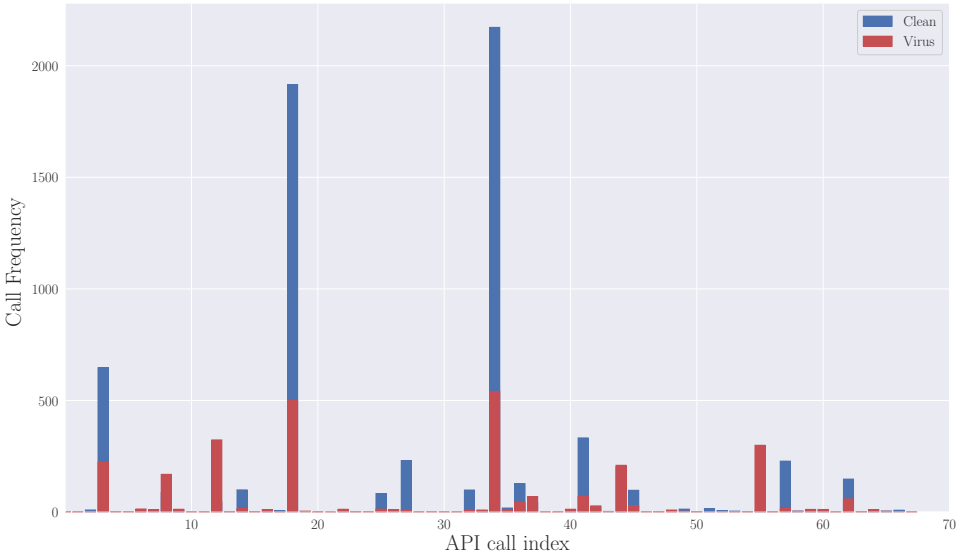

Figure 3.8: Mean call frequency (y-axis) for Unicode calls (x-axis) from Cuckoo data.

Figures 3.7 and 3.8 show that in general, malware tends to use the ASCII versions of Windows methods whereas benignware tends to opt for the Unicode versions of methods. There are many possible reasons for this, Windows recommends developers use Unicode versions of system calls since Unicode supports more characters providing support for more languages. Therefore, developers of legitimate applications are likely to use Unicode versions of system calls since they are more extensible. However, malware authors are unlikely to be concerned with supporting multiple languages and ASCII strings tend to be simpler to use and more familiar.

Table 3.7 shows the ten most frequently occurring features for benignware and malware as seen by the kernel driver.

Benignware	Malware
NtClose	NtClose
NtQueryPerformanceCounter	NtOpenKey
NtOpenKey	NtQueryPerformanceCounter
NtYieldExecution	NtQueryValueKey
NtWaitForSingleObject	NtWaitForSingleObject
NtQueryValueKey	NtDeviceIoControlFile
NtDeviceIoControlFile	NtQueryInformationProcess
NtClearEvent	NtClearEvent
NtWaitForMultipleObjects	NtQueryInformationToken
NtQueryInformationToken	NtDelayExecution

Table 3.7: Most frequently occurring features in benignware and malware for the kernel driver.

There is less variation in top ten for benignware and malware here with eight features in common. The two differing features of malware, NtDelayExecution and NtQueryInformationProcess, are known to be used by malware. NtDelayExecution was discussed previously as it was also in the most frequent features for Cuckoo. NtQueryInformationProcess is commonly used by malware to detect if its being debugged [88]..

3.3.1.4 Exclusive Features

Table 3.8 shows the system calls in malware samples not found in benign samples for the kernel data and the number of samples they appeared in. The benignware data did not contain any calls that were never called in the malicious data from the kernel driver. In addition, the Cuckoo data did not contain any calls that were exclusive to either benignware or malware and above the threshold defined in section 3.2.6.

System call	No. of samples
NtGetWriteWatch	64
NtResetWriteWatch	62

Table 3.8: System calls in malware not present in benignware for kernel driver data.

Though there are not many features that are unique to malware, these two features emphasise an interesting feature of malware behaviour. NtGetWriteWatch/NtResetWriteWatch can be used by malware to check if they are being debugged. It provides a mechanism by which malware can monitor sections of memory they use and detect if anything else tries to access it. Therefore, it is not unusual to see that it in the malware data.

In conclusion, the additional data analysis has also suggested that anti-analysis features are prominent in the data gathered on malware. Even a few have been seen in the kernel data.

3.3.2 Localised Kernel Data Results

Currently it is still unclear if the kernel data's results were assisted by the fact that the data is being collected on a global scale and observing all processes. To gain further clarification regarding whether collecting the data at a global level assisted the classification process, the kernel data was limited to the data produced by the process being analysed and any processes it created. The results from this are shown in Table 3.9.

From Table 3.9, it can be seen that the classification results have decreased when collecting data from the kernel driver at a local, process-specific, level. For example, with Random Forest the AUC has decreased from 0.986 to 0.978 and the accuracy from 95.2% to 92.3%. The differences between global and local kernel data were also

Machine Learning Algorithm	Localised Kernel Driver			
	AUC	Accuracy (%)	Precision	F-Measure
AdaBoost	0.962	89.6	0.902	0.891
Decision Tree	0.901	83.8	0.855	0.825
Linear SVM	0.884	82.0	0.893	0.788
Nearest Neighbour	0.934	86.6	0.875	0.858
Random Forest	0.978	92.3	0.944	0.921

Table 3.9: Classification results of data from the kernel driver focusing on the process under investigation.

found to be statistically significant. Therefore, it is evident that collecting data at a kernel level is not the only contributing factor to the improved classification results, the data must also be collected at a global-level to obtain better classification results. It's also interesting to note that, at a significance level of 0.05, the classification results from localised kernel data are statistically significantly lower than the Cuckoo results as well. This shows that if data is going to be collected at a process-specific level, user-level hooks provide more value since they will also observe many of the process' interactions that did not reach the kernel. In addition, this shows that simply collecting at a kernel privilege is not enough. The scope of the collection (local vs global) is also important. It may be possible to improve the localised kernel results slightly by attempting to detect when malware injects its payload into benign software and runs it from there. However, that data would be captured by a global kernel capture and therefore the results are unlikely to improve beyond the global kernel results.

To gain further insight into the differences between a classifier's view of the system at a local kernel level and a global kernel level, the top ten features of the local kernel data are compared with those obtained previously from the global kernel data. As with the global kernel data, the random forest classifier was used since it was the best performing classifier.

3.3.2.1 Independent Feature Ranking

Table 3.10 compares the results of the independent feature ranking method on the local kernel data and the global kernel data.

Local Kernel Driver	Kernel Driver
NtQueryValueKey	NtQueryDebugFilterState
NtQueryKey	NtEnumerateKey
NtQueryAttributesFile	NtQueryFullAttributesFile
NtQueryInformationProcess	NtReleaseSemaphore
NtEnumerateValueKey	NtEnumerateValueKey
NtQueryVirtualMemory	NtReadVirtualMemory
NtProtectVirtualMemory	NtSetInformationProcess
NtQueryDebugFilterState	NtSetValueKey
NtOpenKey	NtOpenEvent
NtOpenFile	NtNotifyChangeKey

Table 3.10: Top ten features using independent feature ranking on local kernel data with Random Forest.

There are quite a few similarities between the top ten features of the local kernel data and global kernel data. They have two methods in common, NtEnumerateValueKey and NtQueryDebugFilterState. In addition, they both have three methods relating to the Windows registry and one method relating to processes. The local kernel data has two methods regarding files in comparison to the global kernel data's one method. The same is true for the methods regarding virtual memory. The local kernel data contains the method NtProtectVirtualMemory in its top ten. This method is normally used by software to prevent triggering a major exception. The call itself is used to mark a page in memory so that when it is accessed, an exception is triggered. That page is then placed at the bottom of the stack so that software is prevented from popping an empty stack. However, this can also be used as an anti-debug trick. Malware can mark a page

as protected using NtProtectVirtualMemory, trigger an exception by accessing it, and then watch to determine if a debugger intercepts the exception.

The main difference between the local and global kernel data is that there is more diversity in the global kernel data's top ten as it contains two methods in categories not captured by the local kernel data's top ten. These are NtReleaseSemaphore and NtOpenEvent. Semaphores are sometimes used by malware to avoid reinfecting its victim. This is achieved by creating a semaphore with a unique name and then when assessing if a victim has already been infected, malware just needs to check for the presence of that semaphore object [241]. Another observation that can be made is that while each has the same amount of methods relating to the registry, there is one in the global kernel data which is particularly important with regards to detecting malware that is not present in the local kernel data. That is NtNotifyChangeKey. NtNotifyChangeKey can be used to monitor and prevent any changes to specific keys. This can be used by malware to ensure that nothing else tampers with its data.

3.3.2.2 Inbuilt Feature Ranking

Table 3.11 compares the results of the inbuilt feature ranking method on the local kernel data and the global kernel data.

Localised Kernel Driver	Kernel Driver
NtReadFile	NtWriteFile
NtQueryVirtualMemory	NtFlushVirtualMemory
NtOpenEvent	NtReadFile
NtOpenMutant	NtUnlockFile
NtQueryValueKey	NtOpenMutant
NtFlushVirtualMemory	NtLockFile
NtUnlockFile	NtNotifyChangeDirectoryFile
NtAllocateVirtualMemory	NtOpenEvent
NtClose	NtDeleteAtom
NtQueryInformationProcess	NtQueryValueKey

Table 3.11: Top ten features using inbuilt feature ranking with Random Forest

In table 3.11 there are even more similarities with a total of six methods in common (NtReadFile, NtOpenEvent, NtOpenMutant, NtQueryValueKey, NtFlushVirtualMemory, NtUnlockFile). An interesting method on the local kernel side is NtQueryInformationProcess. This method is commonly used by malware to detect if its being debugged [88]. A notable absence in the local kernel top ten is NtNotifyChangeDirectoryFile, which, as discussed previously can be used to monitor changes to a directory. This method is likely to be used by a kernel-mode rootkit. Since rootkits tend to be installed silently (via process injection, for example), the local kernel method will not detect its creation and therefore not monitor its activity.

Therefore, it can be observed that limiting the kernel data to that produced by the process under investigation and its children has the effect of reducing the diversity in the malware behaviour observed. The local kernel data misses out on any activity that results from malware injecting its code into another process. In addition, limiting the kernel data has increased the probability of classifiers using the more explicit evasive features of malware to detect it. As a result, the local kernel data is not able to produce

as strong classification results as the global kernel data.

3.3.2.3 Exclusive Features

To further understand the data that is lost at a local level, the features of malware that were present in the global data but not the local data were analysed. This is shown in table 3.12.

System call	No. of samples
NtCreateToken	352
NtExtendSection	52
NtMakeTemporaryObject	2457
NtQuerySystemInformation	2456
NtSetEvent	2455

Table 3.12: System calls in malware recorded at global kernel level but not local level.

As can be seen, there are a number of features that are not captured in the data gathered at the local kernel level. Perhaps, the most crucial of them is NtQuerySystemInformation. This call is frequently used by malware to get information about the system such as a list of running processes [159]. It can even be used by rootkits to hide a malware sample from the user [118]. Another system call that would not be unusual to see in malware is NtCreateTokens. Each process and thread in Windows has a token that determines its privileges. Therefore it is quite common for rootkits to modify these tokens to give a malware sample elevated privileges [118].

The missing features provide insight into some of the important behaviour not captured by the local kernel driver. This further explains why the local kernel driver data was unable to obtain similar results to the global kernel driver data.

3.3.3 Combined User and kernel data results

Since limiting the data from the kernel driver did not improve results, and given that Cuckoo and the kernel driver seemed to fail on different samples, the next step was to combine the data from Cuckoo and the kernel driver to determine if the classification results are improved by the combination of data. The results of this are also shown in Table 3.13

Machine Learning Algorithm	Cuckoo and Kernel Driver			
	AUC	Accuracy (%)	Precision	F-Measure
AdaBoost	0.990	94.9	0.956	0.960
Decision Tree	0.954	92.4	0.924	0.936
Linear SVM	0.952	91.5	0.916	0.915
Nearest Neighbour	0.960	90.3	0.873	0.888
Random Forest	0.990	96.0	0.962	0.942

Table 3.13: Classification results from combining Cuckoo and kernel data

Table 3.13 shows that combining data from both tools produces classification results that are slightly stronger for the purposes of malware classification with an AUC of 0.990 for both AdaBoost and Random Forest. The only classifier with reduced results was K-Nearest-Neighbours suggesting that it struggles to classify data beyond a certain number of dimensions. Again, as with all the data, the differences shown in this table (improvements or otherwise) are statistically significant. Therefore, this further validates the claim that there is a difference in the data between Cuckoo and the kernel driver since the results would not have improved had this not been the case.

3.3.4 Feature Ranking Evaluation

To confirm the correctness of both of the feature ranking methods employed throughout this research, simple feature reduction (described in the method section) was performed using the feature ranking methods. The results of this are shown in figures 3.9 and 3.10. These graphs were created for both the data from the kernel driver, and the data from the Cuckoo driver. However, since the graphs were a very similar shape, for brevity's sake, only the graphs for the data from the kernel driver are shown.

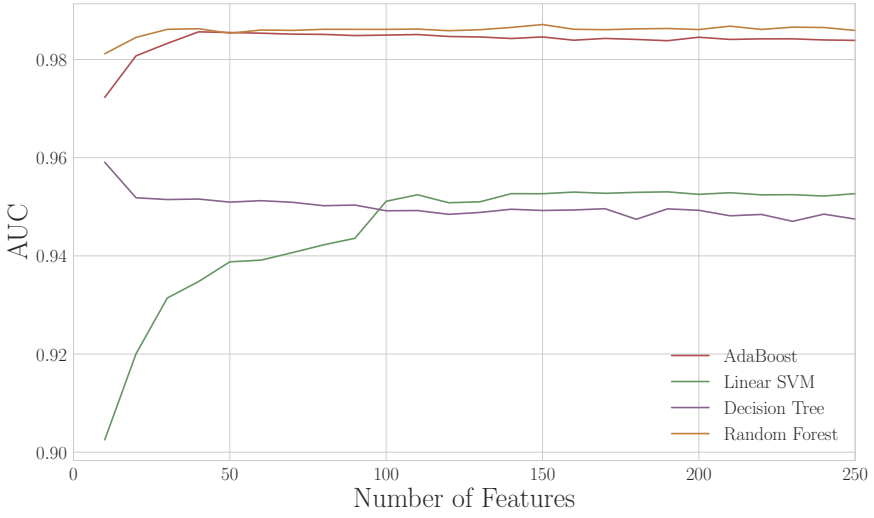

Figure 3.9: Feature selection using inbuilt feature selection method

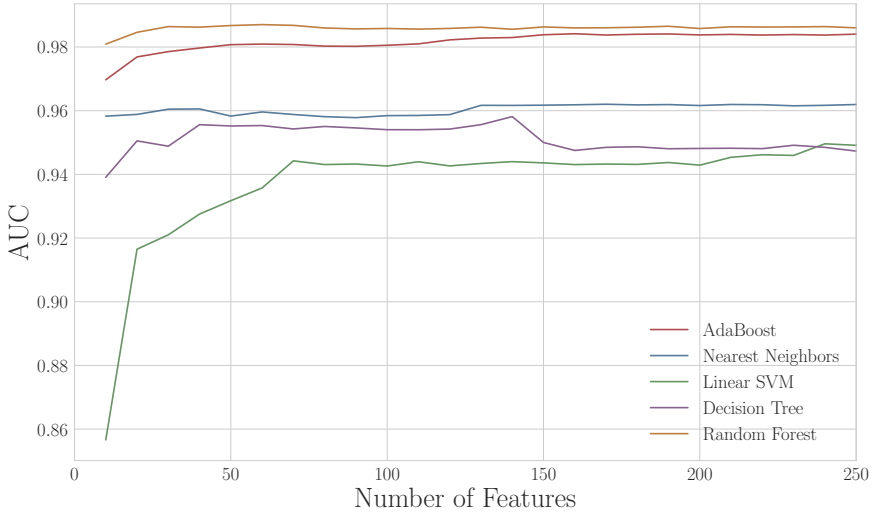

Figure 3.10: Feature selection using independent feature selection method

For most of the plots in figures 3.9 and 3.10 the AUC is at its lowest with just ten features, however, as the number of features that the machine learning algorithms use increases, the AUC increases until it reaches its peak at around 50 features. After 50 features, the introduction of new features does not add any more useful information, thereby reducing or not contributing to the difference in the AUC. This highlights that the feature ranking method is correctly deciphering which features are important. In addition, it shows that in most cases no more than 50 API-calls need to be hooked for similar results.

3.3.5 Global Feature Ranking

Finally, the global feature ranking metric was applied to get a concise yet comprehensive view of the features of malware that were consistently considered important by all classifiers. The results from applying the global feature ranking for both the inbuilt and independent feature selection methods are shown in Tables 3.14 and 3.15.

Cuckoo	Kernel Driver
GetSystemMetrics	NtReleaseSemaphore
NtQueryInformationFile	NtLockFile
LoadResource	NtUnlockFile
RegQueryValueExW	NtEnumerateKey
NtUnmapViewOfSection	NtWriteFile
NtDuplicateObject	NtOpenMutant
RegOpenKeyExW	NtReadFile
RegCloseKey	NtOpenThreadToken
NtOpenSection	NtReplyWaitReceivePortEx
NtWriteFile	NtQueryVirtualMemory

Table 3.14: Top ten features using independent feature selection considering all classifiers.

Cuckoo	Kernel Driver
NtOpenSection	NtFlushVirtualMemory
InternetCloseHandle	NtOpenMutant
LoadResource	NtFilterToken
SetUnhandledExceptionFilter	NtUnlockFile
SetFileTime	NtAccessCheckByTypeAndAuditAlarm
LdrLoadDll	NtQueryVirtualMemory
CreateActCtxW	NtDeleteAtom
getaddrinfo	NtWriteFile
LdrGetDllHandle	NtReadFile
LdrGetProcedureAddress	NtCompleteConnectPort

Table 3.15: Top ten features using inbuilt feature selection considering all classifiers.

These tables show which features perform best across all the classifiers that were used. This provides a clearer picture of which features are extremely strong when it comes to differentiating malware from benignware. With regards to the Cuckoo data, table 3.14 contains some of the features used to evade detection that are also in table 3.4 (GetSystemMetrics, NtUnmapViewOfSection, and NtOpenSection). There are also resource related methods (LoadResource) and the native API method (NtQueryInformationFile) that was observed in table 3.4. Of the new methods, NtDuplicateObject is interesting because it is used by malware to evade antivirus heuristics, as antiviruses would expect malware to call the more commonly used DuplicateHandle to duplicate a process handle to kill or inject into it and would therefore be less likely to flag a call to NtDuplicateObject as suspicious [224]. From this it can be concluded that, regardless of the classifier used, when trained on data from Cuckoo, malware will be largely recognised using its evasive features.

Cuckoo's top ten in table 3.15 places an even stronger emphasis on the evasive behaviour of malware. For example, LdrLoadDll, LdrGetDllHandle and LdrGetProcedureAddress are in the top ten and are known to be used by malware to load DLLs dynamically in order to import methods from them. This can also be used to avoid being detected by IAT hooks. In addition, the method SetUnhandledExceptionFilter in the Cuckoo top ten is also used as an anti-debugging trick by malware as this method is used to specify a function to be called in the event of an exception occurring that is not handled by any exception handler. However, the function specified will only be called if the process that raised the exception is not being debugged. Therefore, malware can register a function to deliver its payload and then throw an exception, and if the process is being debugged, that function will not be called, and hence the malware will not display its malicious behaviour [88]. SetFileTime, which has been described previously, is also used to curb suspicions. Finally, NtOpenSection, as mentioned previously, can be used to embed malicious code in a benign process (however, it can also be used for benign purposes). Therefore, as can be seen, much of the top ten for Cuckoo in table 3.15 suggest that classifiers utilise the evasive behaviour of malware to detect it.

On the kernel side, each table contains methods from a wide range of categories (such as file-system, threading, networking etc.), making it more general than the top ten calls in the Cuckoo data. While many of the methods in these tables are likely to be used by malware, they are not used solely by malware (as would be expected from a tool monitoring at a global level). On the whole, it can be seen that with the Cuckoo data, malware is detected through the techniques it uses to detect a monitoring or virtual environment, whereas, with the data from the kernel driver, malware is differentiated from benignware through how its general behaviour differs from the norm.

3.3.6 Sample Size Verification

To confirm that the dataset size used was suitable, the experiments described in section 3.2.7 were conducted. The results are shown in figure 3.11.

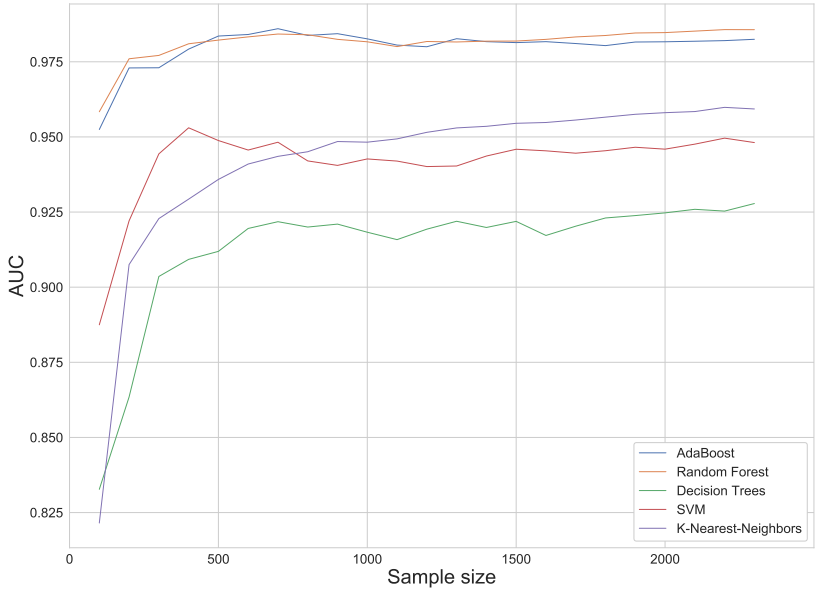

Figure 3.11: How the AUC responds as sample size is increased

Figure 3.11 shows that after 1000 samples, the AUC values almost completely plateau.

This suggests that after this point, adding more samples will not have a significant effect on the classification results. Therefore, it is reasonable to conclude that 2000 samples is more than enough.

3.4 Conclusion

Motivated by a hypothesis that kernel level API calls and user level API calls do not produce the same classification results, experiments were conducted to study the differences. This was achieved by collecting data at different privilege levels within the Windows Operating System. Data was collected at a user level using Cuckoo, and at the kernel level using a custom made kernel driver since there are no existing tools that hook all the calls in the SSDT on a global scale. The data collected was classified using several state-of-the-art machine learning algorithms to determine whether collecting data at different levels altered classification results. The results showed kernel data to be statistically significantly better for all classification algorithms despite the fact that user level methods are significantly more popular in the literature. Random Forest performed the best with an accuracy of 94.0% for Cuckoo and 95.2% for the kernel driver. In addition, limiting the kernel data to that produced by the process under observation (and its subprocesses) had a negative impact on the classification results suggesting that the collection of data at a global, system-wide level aided the classification process. The strongest classification results were observed by combining the data from Cuckoo (user level) with that from the kernel driver; achieving an AUC of 0.990 and accuracy of 96.0% for Random Forest.

In order to understand why the differences in data collection methods had contributed to the different classification results, feature ranking was conducted for Random Forest and collectively for all classifiers used, and it was found that the features focused on by classifiers differed depending on the data used. The main observation from this was that monitoring on a process specific level as Cuckoo does caused the machine learn-

ing algorithm to detect malware using its evasive properties. Whereas, when trained on data obtained from monitoring at a global, kernel level, the machine learning algorithm used the more general behaviour of malware (and processes in general) to distinguish it from benignware. Limiting the kernel data to that produced by the process under investigation and its sub-processes did not produce as strong a performance as the global kernel data since it missed malicious activity carried out through process injection. In addition, when the data was limited, classifiers placed more emphasis on the evasive features to recognise malware. The differences resulting from collecting data at different privilege levels highlighted the benefit gained from collecting data at a kernel level (or both levels) in order to detect malware and the importance of the literature carefully detailing the data collection method that has been used since the results are affected by it. Table 2.1 shows that while there exists a plethora of well established tools for collecting data at a user level, there are only a handful of established tools to collect data at a kernel level, and fewer still that are freely available. While the driver used in this research is specific to Windows XP, the main contributions of this research (a comparison of user and kernel level calls) will apply to future releases of Windows. In conclusion, this chapter conducted the first objective, evidence-based comparison of kernel level and user level data for the purposes of malware classification and found that more thought must be given to the data collection method when conducting dynamic malware analysis as the most optimal tool does not yet exist.

Chapter 4

Assessing the effectiveness of classifiers to detect non-evasive malware

4.1 Introduction

One of the observations from the previous chapter was that malware possesses a significant amount of evasive/anti-VM/anti-debug properties. Paradoxically, the classifiers were using the very existence of those features to detect malware. Therefore, this chapter assesses the degree to which this biases the classifiers' results. This is the inspiration behind the fourth research question:

RQ4 *Does the traditional Dynamic Malware Analysis process create a bias in the data collected and subsequently classified?*

If found to be true, this question raises many issues. One being that if classifiers trained in the traditional dynamic malware analysis process are placed in a real environment, there is a possibility that many malware samples would go undetected since they would not exhibit as many anti-VM features once they realise they are in a real environment. To provide an example, many malware samples do not run if there is no internet connection, as is commonly the case in an analysis environment. Therefore, if data from an analysis environment without an internet connection is used to train a classifier, the classifier would probably recognise malware by its inactivity. Subsequently, when

that classifier is placed in a real environment, where it is highly likely that an internet connection would exist, malware will go unnoticed [140].

This chapter aims to assess whether there is a threshold number of malicious symptoms that malware must exhibit to avoid detection from classifiers trained through the standard dynamic malware analysis process. This is summarised in the fifth research question:

RQ5 *How much malicious behaviour can a malware sample exhibit before it is detected?*

In answering these questions, this chapter will provide the sixth and seventh contribution of this research:

C6 *This research assesses whether popular classifiers can generalise to detect ransomware that does not contain the most distinguishing features that were found in chapter 3.*

C7 *This research assesses whether kernel-level or user-level data is better at generalising towards malware that does not contain the distinguishing features found in chapter 3.*

4.2 Background

The literature highlighted in Chapter 2 makes it clear that there are a significant number of methods that malware can utilise to evade classifiers; furthermore, as an increasing number of evasive methods are identified and defended against, malware authors respond by adding new evasive techniques. As it becomes more common for malware samples to contain evasive features, the classifiers trained on data obtained from running malware for a few minutes will only recognise malware from its evasive attributes as opposed to its malicious properties. Therefore, while most of the literature looks

at adding evasive features to malware to make it undetectable, this chapter attempts to make malware undetectable by removing evasive behaviour. When referring to malware in this chapter, the category of malware being alluded to is *Ransomware*.

The reason for focusing on ransomware in this chapter is due to the surge in its popularity recently. In 2017, Symantec reported a 46% increase in the number of ransomware variants. In addition, successful ransomware attacks have been quite costly, the notable attack on the NHS by the ransomware variant called WannaCry was estimated to cost £92 million [29]. Another reason to focus on ransomware in this study is that its malicious behaviour is relatively simple to automate (without requiring human intervention) and is very well documented. The tool used to simulate the malicious symptoms of ransomware is called Amsel.

4.2.1 Amsel

Amsel [190] is a tool written in Java that is designed to simulate malware for research purposes. Amsel is essentially composed of two libraries, the symptom injectors and the models. The symptom injectors library consists of various malicious symptoms that a user may want to emulate. Potential symptoms include the generation of suspicious network traffic to the encryption of files on the host. Symptom injectors do not have to be used in isolation but can be combined to create a complete attack chain. For example, a complete attack chain may consist of connecting to a server, stopping a running process, running a new process and then reconnecting to a server. The models library consists of stochastic models, in particular, Continuous Time Markov Chains (CTMC). The purpose of this library is to decide how long Amsel should spend in each stage/symptom of the attack chain. It also determines how long to wait between each symptom. Each symptom in itself may have random elements controlled by the model library. For example, if one of the symptoms is to send network traffic to a server, the models library can be used to define the size of the traffic in each iteration. Amsel provides a user with complete control when creating a kill chain; nevertheless, a user

may choose to relinquish some of that control if the addition of randomness makes for a more accurate representation of an attack. The user can specify the exact order of symptoms in an attack chain, or the user can assign a probability with which each step may be taken and then leave it to Amsel to create the final kill chain. This has the advantage of adding an element of randomness each time Amsel is run, as, in some cases, Amsel may skip a step, or change the order in which steps are taken for each run. This allows a user to thoroughly test the robustness of their security system and determine if it can detect an attack regardless of the sequence. It is this mix of structure with controlled randomness that allows for very realistic modelling of actual attacker behaviour. More information regarding Amsel can be found at [191], [192] and [193].

4.3 Method

4.3.1 Initial Experiments

As the aim of this chapter is to assess if there are flaws in the traditional dynamic malware analysis process, the first step is to conduct the standard dynamic analysis experiments using real ransomware and benignware. To begin with, 2500 ransomware samples were collected from VirusShare [18] and 2500 benign files were collected from SourceForge [14] and FileHippo [6]. The ransomware samples used are all crypto-ransomware as opposed to locker ransomware. The reason for this is that the locker variants do not represent much of a threat since their actions are easily reversible [204]. Besides, they are relatively simple to detect since they almost immediately make their presence known to the user. Furthermore, crypto-ransomware are more commonly used [216]. The crypto-ransomware used was largely labelled as Trojan:W32/Ransom [16].

Once the samples were collected, they were run for two minutes in a virtual machine running Windows XP SP3. To increase the likelihood of the ransomware exhibiting

malicious behaviour, real files (such as documents, presentations, images, and videos etc.) were placed in the user's home directory since this would make the environment look more realistic. When a sample was run, the system calls it made were monitored and extracted at both user-level and kernel-level separately. User-level system calls were gathered using Cuckoo Sandbox [111]. Kernel-level calls were gathered using the custom-built kernel driver described in the previous chapter. The calls made by each sample were then represented numerically as frequency histograms. Before passing the data to the classifiers, the frequency histograms were normalised using L1-normalisation to reduce noise and overfitting. This was done separately for both Cuckoo and the kernel driver. The classifiers that were used for this chapter are the same as those used in the previous chapter (AdaBoost, Decision Tree, Linear SVM, Nearest Neighbours, and Random Forest) with the addition of Gradient Boost. The reason for including Gradient Boost is the impressive results it has obtained in previous research focused solely on ransomware [223, 272]. The classifiers were initially trained and tested on the calls from the real ransomware and benignware using 10-fold-cross-validation. The results from this are reported using the same metrics as in the previous chapter, namely, AUC, accuracy, precision, and F-measure. This information will give a clear picture of the classifiers' performance when it comes to differentiating ransomware from benignware.

4.3.2 Amsel Experiments

After analysing the results from the initial experiments, the next step is to observe how those same classifiers perform when detecting ransomware emulated by Amsel. The functionality required of Amsel for the experiments in this chapter is relatively simplistic to reduce the possibility of the classifiers being biased by additional behaviours. The only symptom used was the file encryption symptom since that is the main malicious symptom of ransomware that solutions try to prevent. The behaviour of this symptom is to encrypt files in the directory specified (including all sub-directories).

The encryption algorithm used by default is a simple XOR operation. There are several parameters within the symptom's settings that can be altered; however, for the experiments carried out in this chapter, only one parameter is altered between each run, the interarrival time. This parameter determines how long the emulated malware sample should wait between encrypting each file in the directory specified. The interarrival time was gradually incremented from 1.0×10^{-6} to just below 60 seconds. The size of the increment was 0.01 initially. After reaching 1 second, the increment was increased to 2 seconds. The reason for only altering a single parameter is that it makes it a lot easier to interpret the results from the classifiers (since there is only one variable). The reason the interarrival time, in particular, was chosen is that, depending on its value, it makes it possible to emulate malicious and evasive behaviour.

When the time between encrypting each file, a.k.a the interarrival time, is set to a lower value, the emulated ransomware is encrypting more frequently and thereby exhibiting malicious behaviour much more frequently. Whereas when the interarrival time is at higher values, the emulated malware is idle for more extended periods which, when observing system calls, would look very similar to evasive malware. When the time between encrypting files is set to its highest value (57 seconds), Amsel will only encrypt two files at most before analysis is complete. While it is true that all evasive behaviour cannot be described by idleness, it encompasses a wide variety of evasive behaviours. This is because the goal of most evasive techniques is to stall execution. Therefore, a lack of behaviour is often the result of most evasive techniques. To provide an example, the system call NtDelayExecution can be used by malware to delay executing its payload.

To ensure that the results obtained are not due to chance, each unique emulated sample (unique because of the interarrival time value it is given) is replicated ten times. This provides more than enough samples per unique time value to ensure that the results are consistent rather than an accident. It is also not so high that the experiments become infeasible due to the constraints of time. Finally, as with the traditional dataset, each

emulated sample is run on the same virtual machine, and the system calls it makes are recorded by Cuckoo and the kernel driver, and converted into frequency histograms. Since the goal of these experiments is to evaluate how well classifiers trained on real ransomware and benignware can detect malicious symptoms, the emulated ransomware dataset is treated as an unseen test set. As a result, the classifiers are trained on all the real ransomware and benignware data, and then those same classifiers are made to classify the emulated ransomware. Since some of the classifiers used have a random element (such as Decision Trees), each classifier is trained and tested 1000 times and the mean accuracy is reported. The overall experimental process described so far is summarised in figure 4.1.

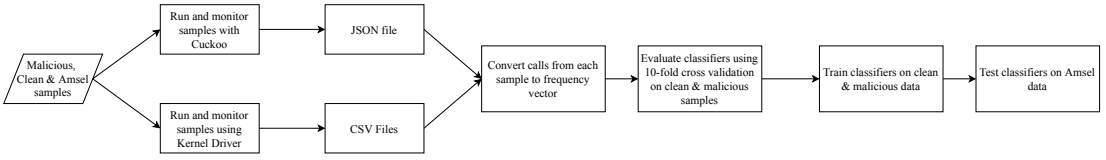

Figure 4.1: System Diagram

4.3.3 Misclassified Samples

An important aspect of this chapter is to determine the rate of encryption at which emulated ransomware goes undetected and to ascertain if that value differs depending on whether Cuckoo or the kernel data is used. This can be calculated using the prediction results that are produced from the "Amsel Experiments" section (4.3.2). In the previous subsection, the classifiers were tested against all emulated ransomware samples 1000 times. Using the data obtained from that, the mean prediction value for each emulated ransomware sample can be found separately for each classifier. The reason that the mean value must be used is that some classifiers show slightly different results on every run due to the fact that they make use of a random element (Random Forest, for example). Therefore, a sample may be correctly classified in one run but incorrectly classified in the next even though the exact same data was used for both. Therefore,

if the mean prediction value for a sample is below 0.5, that sample can be considered incorrectly classified (since '1' represents malicious and '0' represents benign as in the previous chapter). Once the incorrectly classified samples have been found, they can be correlated to the interarrival time (i.e. time between encrypting each file). This information is important as if it is found that a classifier trained on real ransomware and benignware is better at detecting the emulated ransomware when it frequently exhibits idle behaviour, it would suggest that a simple way to evade classifiers trained using the traditional dynamic malware analysis process is to create malware without evasive features. On the other hand, if the classifier is better at detecting the emulated ransomware when it exhibits its malicious features (which in this case is file encryption) more frequently, this would suggest that the classifier is recognising malware by its malicious activity (or activity in general).

4.3.4 Call Categories

A simple method through which data from multiple sources can be better understood is by grouping the data into categories and comparing the distribution of categories in each source. With system call data this presents a few challenges. There is no generally accepted standard on the categories of system calls or the categories that each call belongs to. Another challenge is that some calls can belong to multiple categories. For example NtClose can be used to close a file handle or process handle amongst many more. Furthermore, the list of all possible categories will differ between kernel- and user-level data. In order to conduct the call category analysis as fairly as possible, the categories for the Cuckoo data were taken from one of the configuration files in the Cuckoo source code. For the kernel driver, there is the added challenge that some calls are not documented at all and are therefore difficult to categorise. Therefore, the categories for the kernel calls were taken from the "Windows NT/2000 Native API Reference" [174]. This is the most comprehensive documentation available of the kernel calls. The calls not listed in either category were manually placed into a

category after careful research.

After the categories and the calls belonging to each category had been decided, the data from Cuckoo and the kernel driver was split into the benign, malicious and emulated data for both. For each call (NtCreateProcess, for example), the number of times it was called by all samples belonging to a class (e.g., benign) was added to the category the call belonged to (Process, for example). Once this was performed for each call, the total number of times each category was called within each class (benign, malicious and emulated) was produced and visualised as pie charts.

4.4 Results

4.4.1 Initial Results

Tables 4.1 and 4.2 show the results from classifying real ransomware and the accuracy obtained when those same, trained classifiers are used to detect Amsel for the Cuckoo and kernel data.

Machine Learning Algorithm	Ransomware				Amsel
	AUC	Accuracy (%)	Precision	F-Measure	Accuracy (%)
AdaBoost	0.992	96.6	0.958	0.959	40.5
Decision Tree	0.959	95.7	0.976	0.950	76.7
Gradient Boost	0.996	97.3	0.969	0.967	64.0
Linear SVM	0.861	78.1	0.840	0.687	0.867
Nearest Neighbour	0.969	91.1	0.921	0.889	1.33
Random Forest	0.994	96.8	0.976	0.961	35.0

Table 4.1: Classification results using Cuckoo data

Machine Learning Algorithm	Ransomware				Amsel
	AUC	Accuracy (%)	Precision	F-Measure	Accuracy (%)
AdaBoost	0.996	97.7	0.981	0.973	4.0
Decision Tree	0.969	97.0	0.964	0.964	67.0
Gradient Boost	0.997	98.2	0.990	0.979	0.540
Linear SVM	0.557	57.4	0.0	0.0	0.20
Nearest Neighbour	0.975	92.1	0.926	0.905	3.87
Random Forest	0.995	97.7	0.990	0.973	48.6

Table 4.2: Classification results using kernel data

When differentiating real ransomware from benignware, the kernel data is shown to be more useful for the task. Though the differences in results are small, they are similar to what is expected, given the results from the previous chapter. The only classifier that performs better using the Cuckoo data is Linear SVM. When using the kernel data, Linear SVM obtains a precision (and therefore F-measure) of 0. Taking into consideration the formula for precision, this means that Linear SVM classified all samples in the kernel data as benign. This suggests that the kernel data is not as linearly separable as the Cuckoo data. That being said, Linear SVM obtains the lowest performance in comparison to the other classifiers for both sets of data. On the other hand, the classifier that obtained the strongest performance is Gradient Boost, obtaining accuracy values of 98.2% and 97.3% for the kernel and Cuckoo data.

The last column in both tables shows the accuracy obtained when classifying the emulated ransomware. These results do not bear much resemblance to the results obtained when classifying real ransomware. On the Cuckoo side, the only classifiers with a mildly respectable performance are Decision Tree with an accuracy of 76.7% and Gradient Boost (64%). For the kernel data, the only classifier with an accuracy higher than 50% is Decision Tree (67%). As can be seen, when classifying emulated ransomware, the Cuckoo data seems to have produced a better performance than the data from

the kernel driver. To obtain a better understanding of the results from classifying the emulated ransomware, they have been illustrated in figure 4.2.

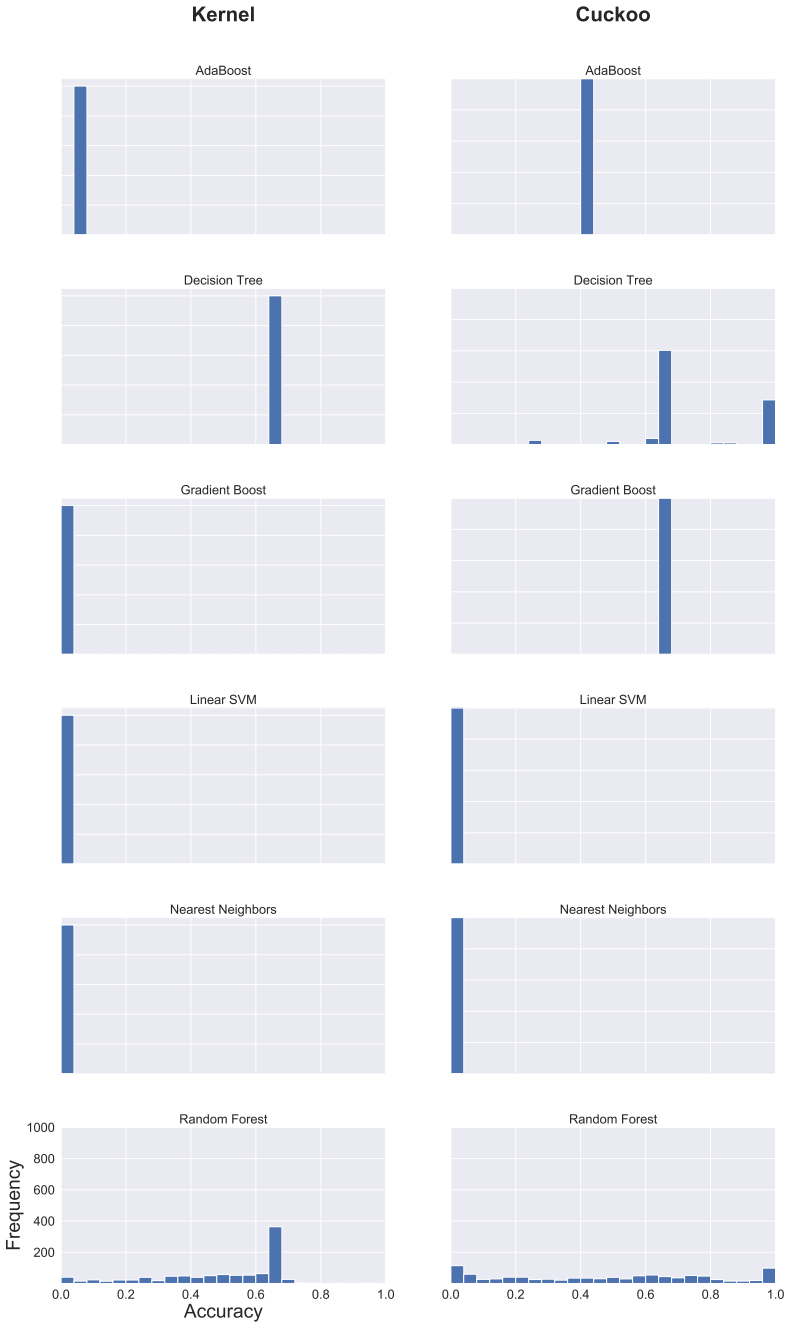

Figure 4.2: Results of classifying Amsel data 1000 times

The graphs above show the results from testing each of the classifiers on the emulated

ransomware data 1000 times (as described previously). For most of the classifiers, the accuracy obtained from each run was the same. However, when using the Cuckoo data, Decision Tree and Random Forest are the only classifiers where the results vary significantly. This is likely due to the random element within these two classifiers that guides how the trees are constructed. Remarkably, Decision Tree is able to get 100% accuracy on occasion when using the Cuckoo data. However, as the results vary so widely, that cannot be relied on. The modal value reveals a bit more information regarding the classifier's performance since it shows the most likely value that would be obtained. The modal values for Decision Tree and Random Forest using the Cuckoo data are 68% and 0.07%.

With regards to the kernel data, figure 4.2 shows that only Random Forest has a spread of results. Its modal accuracy value is 67% . Therefore, when comparing modal values, there is little difference between the Cuckoo and kernel data. In addition, while the Cuckoo data is seemingly more effective than kernel data when it comes to detecting the emulated ransomware, neither have produced strong results. Therefore, the next step is to ascertain the emulated ransomware samples that the classifiers failed to correctly classify. By determining which emulated samples were incorrectly classified, it will be possible to ascertain whether the classifiers are better at detecting emulated malware when the interarrival time is higher or lower.

4.4.2 Misclassified Samples

In order to visualise the results per sample, two histograms had to be created due to the large spread in times used for the experiments. For each new emulated ransomware sample, the interarrival time was incremented very gradually between 0 and 1 second. After 1 second, the magnitude of the increment for each new sample was increased. Figures 4.3 and 4.4 show how Decision Tree performed when classifying emulated ransomware samples with times ranging from 1.0×10^{-6} to 2 seconds for the Cuckoo and kernel data. The reason for selecting Decision Tree is that it obtained the

best performance for both the kernel and Cuckoo data when classifying the emulated ransomware produced from Amsel.

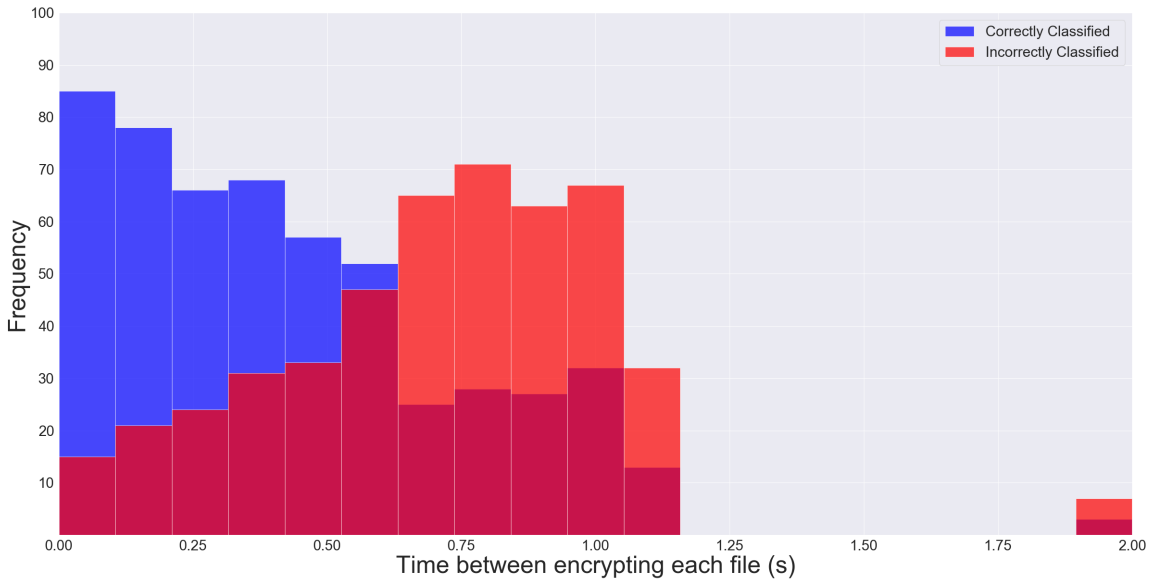

Figure 4.3: Results of classifying Amsel data (with time between encryption <=2s) obtained by Cuckoo.

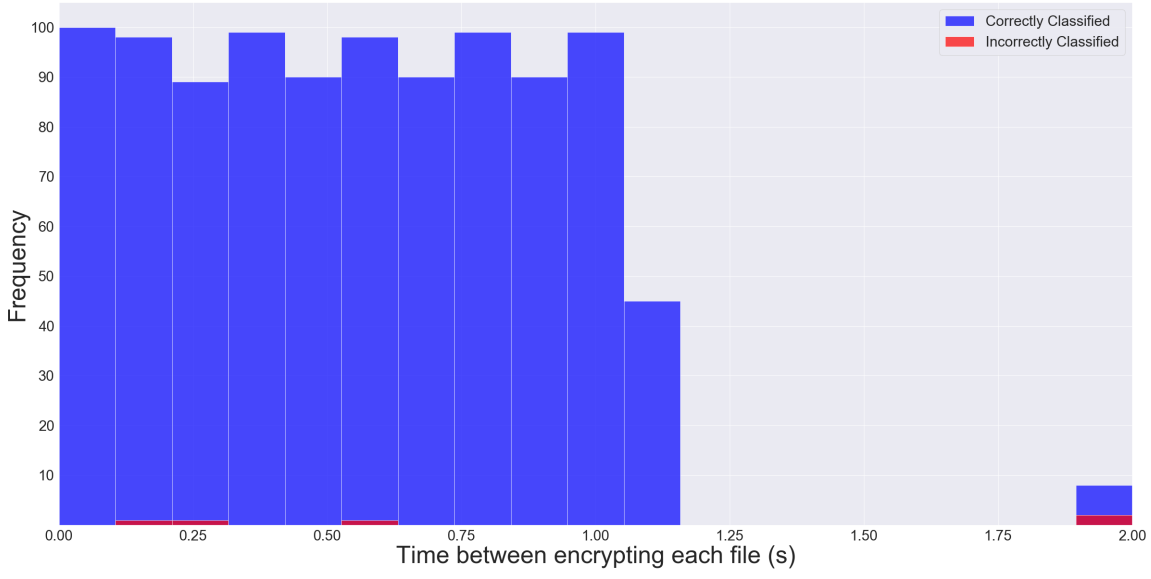

Figure 4.4: Results of classifying Amsel data (with time between encryption <=2s) obtained by the kernel driver.

Figure 4.3 shows the results using Cuckoo data. It can be seen that initially at extremely low time intervals between encrypting each file, Decision Tree is able to identify some, but not all emulated ransomware samples. However, as the time approaches two seconds, the classifier's performance in detecting emulated ransomware using the Cuckoo data decreases significantly.

Figure 4.4 shows the results from Decision Tree attempting to detect emulated ransomware using the kernel data. When using the kernel data, Decision Tree is able to correctly classify the emulated malware samples as malicious in almost every instance.

Figures 4.5 and 4.6 show the remaining results from classifying emulated ransomware samples with the time between encrypting each file (interarrival time) ranging from 2 seconds to 57 seconds.

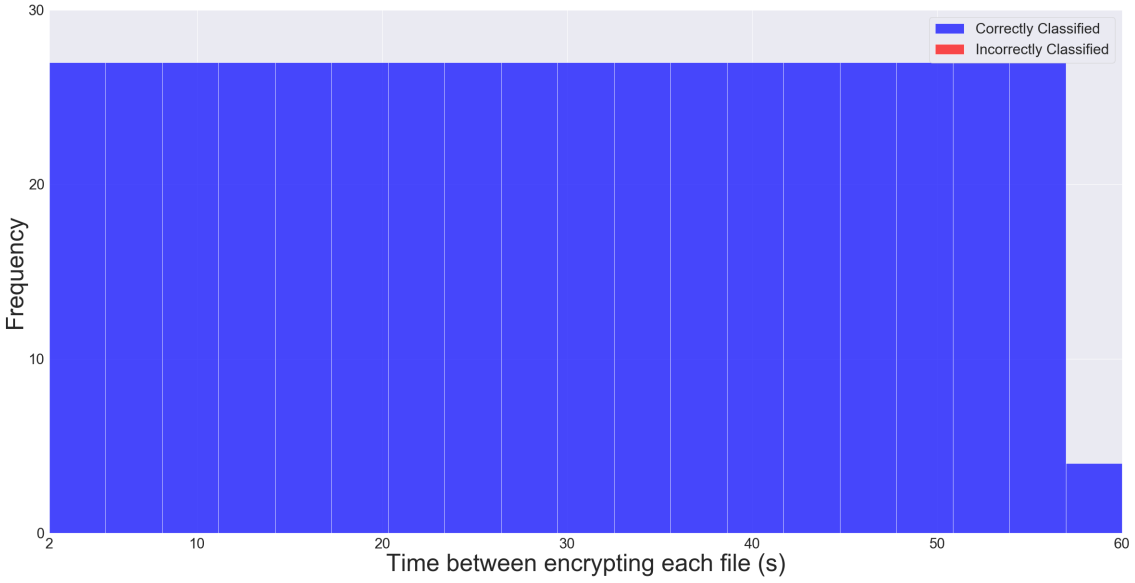

Figure 4.5: Results of classifying Amsel data (with time between encryption >2s) obtained by Cuckoo.

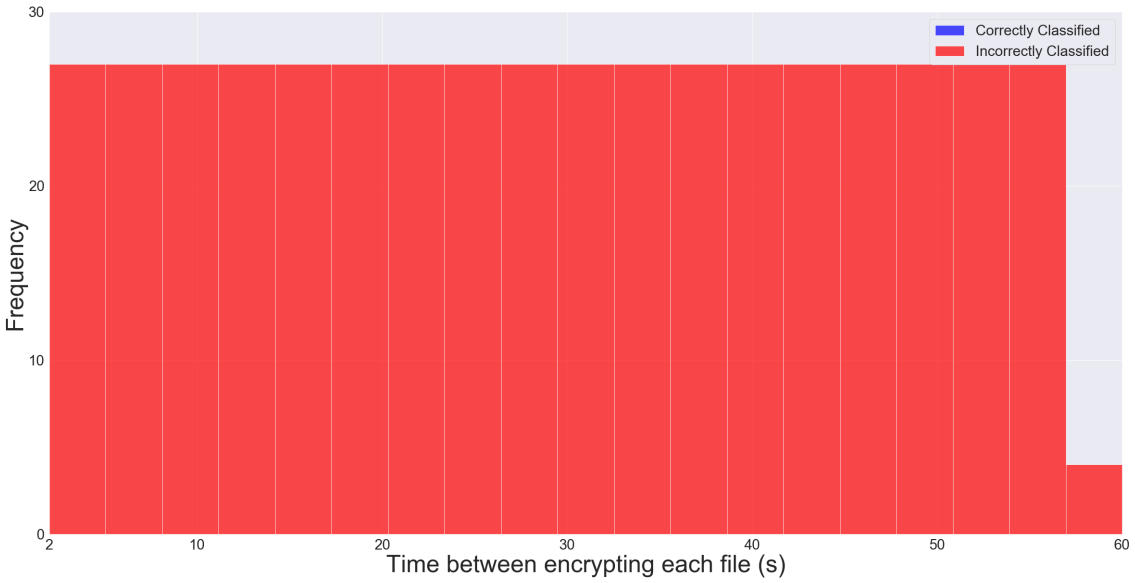

Figure 4.6: Results of classifying Amsel data (with time between encryption >2s) obtained by the kernel driver.

Figure 4.5 shows a very different picture to the previous results, as, in this case Decision Tree is able to detect all the emulated ransomware samples using the Cuckoo data. This suggests that the ability of the classifier to detect emulated ransomware (when trained on real ransomware and benignware) improves as the evasiveness in its behaviour increases.

Figure 4.6 is the complete opposite of figure 4.5 visually. When using the Cuckoo data, Decision Tree's performance improved as the evasiveness of the emulated samples increased, whereas, as figure 4.6 shows, the opposite occurred when using the kernel data. This shows that when a classifier is attempting to detect malware using global kernel-level data, the presence of malicious activity (as opposed to inactivity) is required for the classifier to correctly classify it. This is, in part, to do with the fact that the kernel driver is monitoring at a global level. Therefore as the activity of the malicious sample reduces, it is much more likely to fade into the background (and get lost amongst the general system activity).

The results in this section have shown that the Decision Tree classifier obtains a stronger performance using the Cuckoo data, provided that the delay between encrypting each file is 2 seconds or more. Below 2 seconds, the kernel data is better suited to identifying ransomware. This suggests that when monitored using Cuckoo (or another user-level monitoring tool), the likelihood of ransomware being identified increases as the amount of evasiveness in its behaviour increases. The opposite is true for the kernel-level data.

4.4.3 Hyper-parameter Results

Given that Gradient Boost was the best performing classifier on the training data, and that it is quite closely related to Decision Tree (as it is composed of multiple regression trees), its vastly different results when tested on the emulated ransomware was unexpected. The same can be said for Random Forest. Due to the fact that Gradient Boost and Random Forest are composed of Decision Trees, they have many hyper-parameters in common with Decision Tree. Therefore, a brief analysis of the hyper-parameters in common is performed in order to gain more insight into the reasons behind the differences. In all experiments for this thesis, the default parameters for each classifier in sklearn are used. Of the parameters that Decision Tree and Gradient Boost have in common, the only one where the default value differs is that of 'max_depth'.

As its name suggests, max_depth defines how deep a tree can be. The deeper the tree, the more minutiae it can capture, however, this also puts it at greater risk of over-fitting. For Decision Trees, this parameter is set to "None" by default which means the tree is expanded as much as possible (since there is no limit). With Gradient Boost, this parameter is set to '3' by default. Therefore, for this experiment we changed the value of max_depth for Gradient Boost to match the default value for Decision Tree.

Likewise, Random Forest only differs with Decision Tree on one shared hyper-parameter, 'max_features'. This parameter dictates the number of features that should be considered when making a split. For example, if a dataset consists of 100 features and

max_features is set to 20, when the classifier is making a decision on which feature to split on, it will consider 20 random features and choose the best amongst them. Again, this can be set to lower values to guard against overfitting. For Decision Trees the default value of this parameter is "None", which means that at each split, all features are considered for the split. For Random Forest this value is set to the square root of the number of features. Therefore, for this experiment, the value of this parameter in Random Forest is altered to match that of the value in Decision Tree.

The results from classifying emulated ransomware using the new hyper-parameter values for the Cuckoo and kernel data are shown in tables 4.3 and 4.4.

Classifier	**Amsel Accuracy**
Decision Tree	76.7
Gradient Boost	78.5
Random Forest	75.0

Table 4.3: New emulated ransomware results on Cuckoo data after modifying hyper-parameters.

Classifier	**Amsel Accuracy**
Decision Tree	67.0
Gradient Boost	67.0
Random Forest	67.0

Table 4.4: New emulated ransomware results on kernel data after modifying hyper-parameters.

As can be seen in tables 4.3 and 4.4, by simply modifying one hyper-parameter for Gradient Boost and Random Forest the results have been rectified so that they are equivalent to that of Decision Tree. However, given the purpose of the hyper-parameters, it also suggests that the behaviour exhibited by the emulated ransomware is only detected if the classifiers are set to capture the subtler details within the training data. Both must

be set to their maximum possible values as it were in order to ensure that the classifiers correctly classify the emulated ransomware.

This also highlights the importance of careful hyper-parameter tuning, as the impact it can have on the results is very significant. The modified versions of Gradient Boost and Random Forest obtained the same results on the training data as their unmodified counterparts. However, with regards to the testing data, as can be seen, the results were affected.

4.4.4 Cuckoo Feature Ranking Results

To further understand the reasons behind the results, it can be helpful to look at the features that the best performing classifier considers the most important. Given that the best performing classifier on real ransomware (Gradient Boost) is not the same as the classifier that performs best at detecting emulated ransomware (Decision Tree), the best features for both are studied for the kernel and Cuckoo data. For the sake of brevity, only the top ten features are discussed here. The feature ranking method used here is the inbuilt feature ranking method described in the previous chapter in section 3.2.3. For completeness, the top ten features of all classifiers used are shown in table 4.5 (barring Nearest Neighbours since it has no inbuilt feature ranking method), however, the focus of this section is on Decision Tree and Gradient Boost. Table 4.5 shows the top ten features for the Cuckoo data.

AdaBoost	Decision Tree	Gradient Boost	Linear SVM
NtTerminateProcess	FindResourceA	FindResourceA	GetSystemMetrics
NtUnmapViewOfSection	CreateDirectoryW	CreateDirectoryW	NtReadFile
CoInitializeEx	NtProtectVirtualMemory	NtProtectVirtualMemory	LdrGetProcedureAddress
RemoveDirectoryA	NtOpenKey	NtOpenSection	NtTerminateProcess
CreateActCtxW	__exception__	WriteProcessMemory	LdrGetDllHandle
NtProtectVirtualMemory	WriteProcessMemory	CreateActCtxW	GetUserNameExW
LdrGetProcedureAddress	CoUninitialize	NtTerminateProcess	CreateToolhelp32Snapshot
GetDiskFreeSpaceExW	FindResourceExW	NtOpenKey	NtOpenSection
GetSystemMetrics	CreateProcessInternalW	LdrGetProcedureAddress	GetVolumePathNameW
NtQueryDirectoryFile	LdrGetDllHandle	NtUnmapViewOfSection	NtSetValueKey

Table 4.5: Top ten features using the inbuilt feature ranking method for AdaBoost, Decision Tree, Gradient Boost, and Linear SVM when considering the data from Cuckoo .

With regards to the classifiers that did not obtain a strong performance, Linear SVM placed a lot of importance on specific evasive calls that Decision Tree and Gradient Boost ignored such as GetUserNameExW, CreateToolhelp32Snapshot and GetVolumePathNameW. GetUserNameExW and GetVolumePathNameW are used to detect the use of specific virtualisation tools. Whilst, CreateToolhelp32Snapshot is used to obtain a list of running processes that malware can then terminate (if they are anti-virus processes) or inject its code into [227]. On the other hand, AdaBoost does not give as much focus to evasive calls and contains only one additional evasive call in its top ten, GetDiskFreeSpaceExW. This is used by malware to obtain the hard-disk space to determine if its running in a VM (since VMs tend to have smaller hard disks than real machines).

Focusing on Gradient Boost and Decision Tree, there are five features in common between the two classifiers in table 4.5. The reason for Decision Tree's superior performance over Gradient Boost is not immediately clear from just the features. Decision Tree contains a few features that are known to be used by malware for evasive purposes. WriteProcessMemory, for example, is commonly used by malware to write malicious

code into a benign process' memory space [227]. In addition, the calls NtProtectVirtualMemory and LdrGetDllHandle can be used for evasive purposes as discussed in the previous chapter (in sections 3.3.2.1 and 3.3.5). Aside from that, the calls relate to resources, processes and the registry.

Besides the features in common, Gradient Boost contains a few additional features that can be used by malware to evade detection. The combination of NtOpenSection and NtUnmapViewOfSection was seen in the previous chapter (section 3.3.1.1). Obviously there are many legitimate uses for it, but it can also be used by malware to insert code into another process' memory. In addition, LdrGetProcedureAddress, can be used to dynamically load an external call. This would evade any tool only looking at the statically declared calls, or those monitoring using an IAT hook. The lack of file-related calls in both classifiers' top tens suggests that excessive levels of file activity is not the distinguishing characteristic being observed of ransomware (as would have been expected). This explains why the Cuckoo data was unsuitable for detecting emulated ransomware when the frequency of encryption was high.

However, despite the differences, it is not obvious as to why Decision Tree performed better than Gradient Boost when it comes to detecting the emulated ransomware. Both classifiers only have one file related call in their top ten and therefore, it would seem, that they have equal chance of detecting the emulated ransomware. In order to better understand the reason for the differences in detecting the emulated ransomware, the top ten features of the optimised version of Gradient Boost were extracted and analysed. Interestingly, those top ten were exactly the same as the top ten features of Decision Tree. Therefore to better understand the reason for the difference, the differences in frequencies for each feature in the top ten are plotted for benignware, malware, and emulated ransomware in a bar chart. This is shown in figures 4.7 and 4.8.

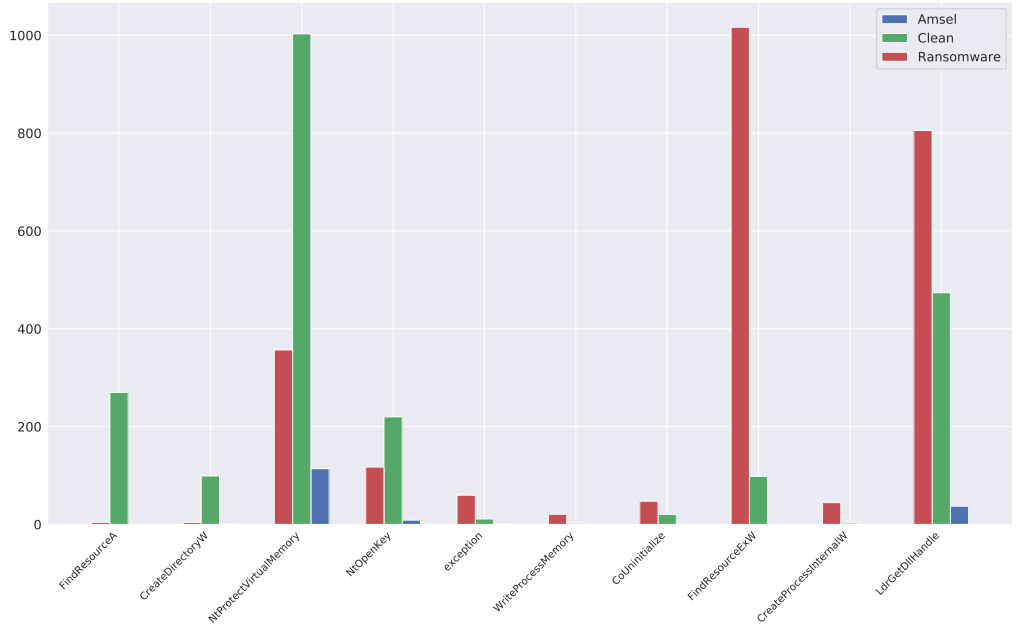

Figure 4.7: Frequencies (y-axis) of the top ten features (x-axis) of Decision Tree in order (from left to right) for malicious, clean, and Amsel data from Cuckoo.

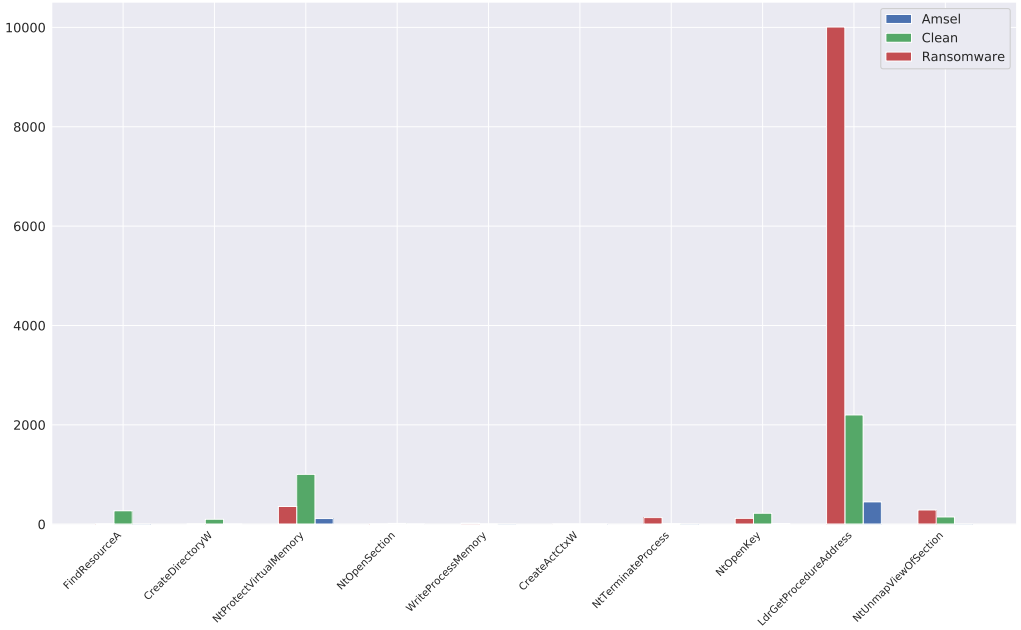

Figure 4.8: Frequencies (y-axis) of the top ten features (x-axis) of Gradient Boost in order (from left to right) for malicious, clean, and Amsel data from Cuckoo.

The graph for Decision Tree is dominated by NtProtectVirtualMemory which is heav-

ily used by beningnware. Gradient Boost is dominated by LdrGetProcedureAddress, which is commonly used by malware to dynamically load calls and evade analysis. Interestingly, Amsel seems to use LdrGetProcedureAddress frequently. This is a side-effect from using Java to emulate ransomware. Java is likely to opt for dynamic linking since it is platform independent and uses a JIT compiler.

However, the presence of features with extremely high frequencies makes it difficult to view the trends in the remaining features. Therefore, to better understand the differences between Decision Tree and Gradient Boost, the features that are in common and dominant within the top ten are removed and the graphs redrawn. This is shown in figures 4.9 and 4.10.

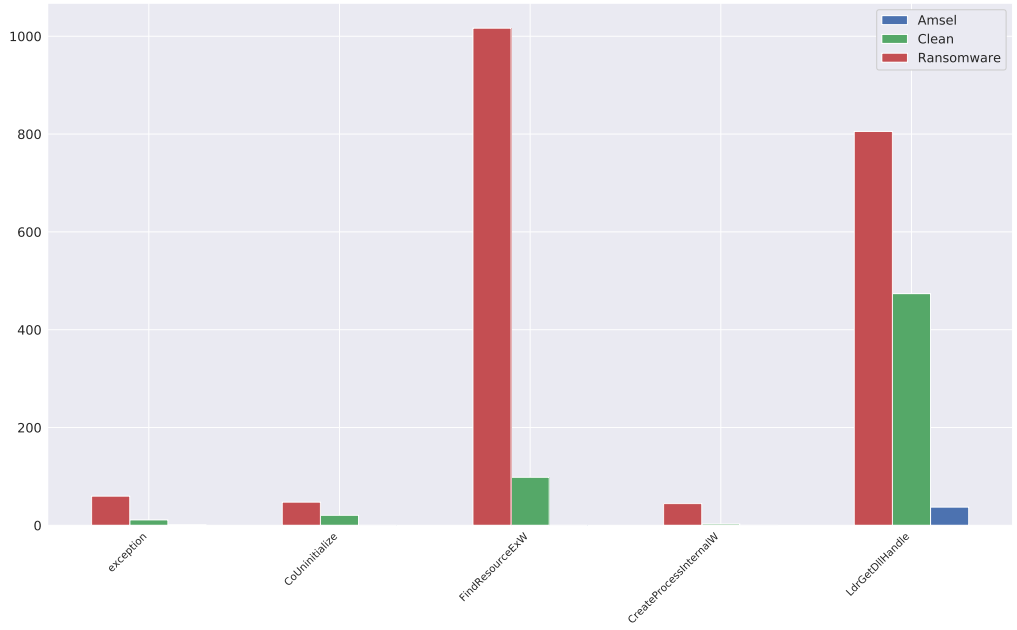

Figure 4.9: Frequencies (y-axis) of the unique features (x-axis) within the top ten of Decision Tree in order (from left to right) for malicious, clean, and Amsel data from Cuckoo.

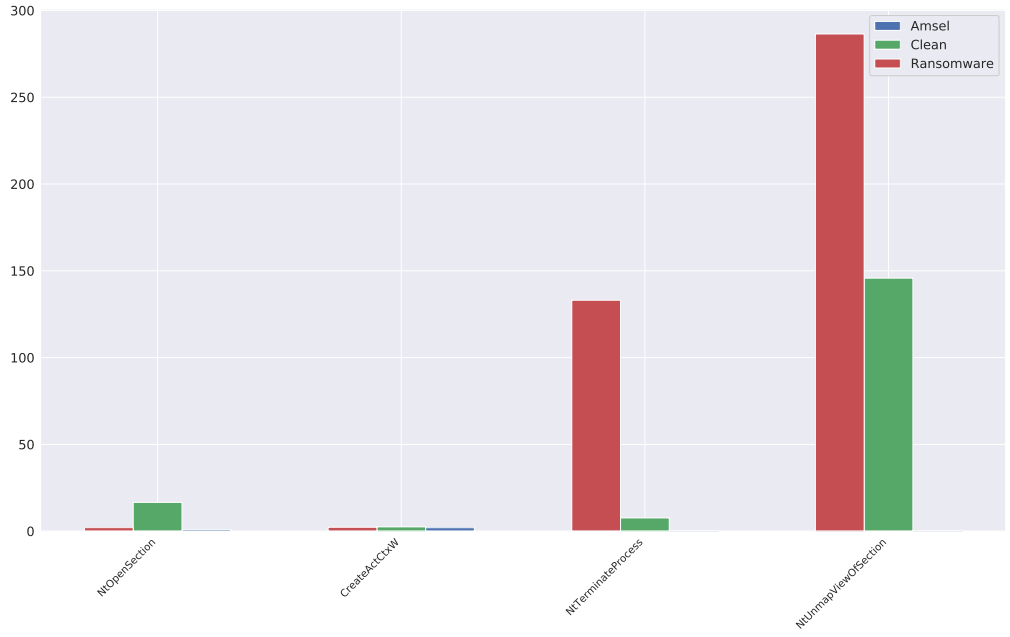

Figure 4.10: Frequencies (y-axis) of the unique features (x-axis) within the top ten of Gradient Boost in order (from left to right) for malicious, clean, and Amsel data from Cuckoo.

As can be seen figure 4.10 contains features that separate benign and malicious well, however, many of them are not used by the emulated ransomware, therefore they are unlikely to assist Gradient Boost in classifying it. On the other hand, some of the unique features of Decision Tree (which are the features Gradient Boost uses when max_depth is altered) in figure 4.9 are used by the emulated ransomware. Interestingly, the emulated ransomware makes use of LdrGetDllHandle. This is again due to the fact that it is written in Java and Java tends to opt for dynamic linking.

The top ten features have shown that Cuckoo is very dependent on evasive features for identifying ransomware (as in the previous chapter). Therefore, it is primarily identifying emulated ransomware from the evasive behaviour it displays. In addition, it can be seen that Decision Tree's superior performance in detecting emulated ransomware is in part due to some unforeseen functionality shown by the JVM, rather than any actual malicious behaviour. In particular, the high rank given to LdrGetProcedureAddress by the Gradient Boost and its use by the JVM seems to have impacted its performance.

4.4.5 Kernel Feature Ranking Results

Moving on to the kernel data, table 4.6 shows the top ten features using the inbuilt feature ranking method for all classifiers used (except Nearest Neighbours). As with the Cuckoo data, the focus in this section is on Decision Tree and Gradient Boost.

AdaBoost	Decision Tree	Gradient Boost	Linear SVM
NtOpenObjectAuditAlarm	NtOpenObjectAuditAlarm	NtOpenObjectAuditAlarm	NtYieldExecution
NtFlushVirtualMemory	NtRequestWaitReplyPort	NtRequestWaitReplyPort	NtCompactKeys
NtOpenMutant	NtCreateDebugObject	NtSetInformationThread	NtCompareTokens
NtQuerySystemTime	NtQueryAttributesFile	NtFlushVirtualMemory	NtCompressKey
NtReadFile	NtReadFile	NtAccessCheck	NtCreateDebugObject
NtRequestWaitReplyPort	NtRaiseHardError	NtReadFile	NtCreateJobSet
NtWriteFile	NtUnmapViewOfSection	NtCreateDebugObject	NtCreateKeyedEvent
NtCompareTokens	NtPulseEvent	NtYieldExecution	NtDebugActiveProcess
NtAcceptConnectPort	NtCreateIoCompletion	NtOpenMutant	NtDebugContinue
NtAccessCheckByType	NtQueryValueKey	NtWriteFile	NtDeleteBootEntry

Table 4.6: Top ten features using inbuilt feature ranking for AdaBoost, Decision Tree, Gradient Boost and Linear SVM when considering the data from the kernel driver.

From table 4.6, it is immediately clear why Linear SVM did not perform well. Many of the features focused on are quite obscure and not known to be used by malware or benignware. AdaBoost, on the other hand, has more in common with Decision Tree and Gradient Boost. Uniquely, it contains NtQuerySystemTime, an operation that can be used to detect delays caused by the presence of debuggers. Interestingly, although AdaBoost ranks NtReadFile and NtWriteFile highly, this does not seem to help its classification results.

From comparing Decision Tree and Gradient Boost, it can be seen that they four features in common (NtOpenObjectAuditAlarm, NtRequestWaitReplyPort, NtCreateDebugObject and NtReadFile). NtCreateDebugObject is an interesting addition to both top tens as it can be used in a very powerful anti-debug trick to detect a debugger [23]. In fact,

NtCreateDebugObject had a very low mean frequency, however, it was still considerably larger for malware than benignware. The remaining methods in common relate to networking (NtRequestWaitReplyPort), file handling (NtReadFile), and monitoring changes (NtOpenObjectAuditAlarm). Three of the calls in the top ten for Decision Tree relate to file handling (NtQueryAttributesFile, NtReadFile and NtCreateIoCompletion), whereas only two calls in the top ten of gradient boost relate to file handling (NtReadFile and NtWriteFile). When using the kernel data, the classifiers are clearly much more likely to focus on file-related behaviour than when data from Cuckoo is used.

In order to further understand the reasoning for the differences in classification results, the mean frequencies of the top ten calls are plotted for both Gradient Boost and Decision Tree. As with the Cuckoo data, when the max_depth hyper-parameter is altered, the top ten features of Gradient Boost are exactly the same as Decision Tree. The frequency plots are shown in figures 4.11 and 4.12.

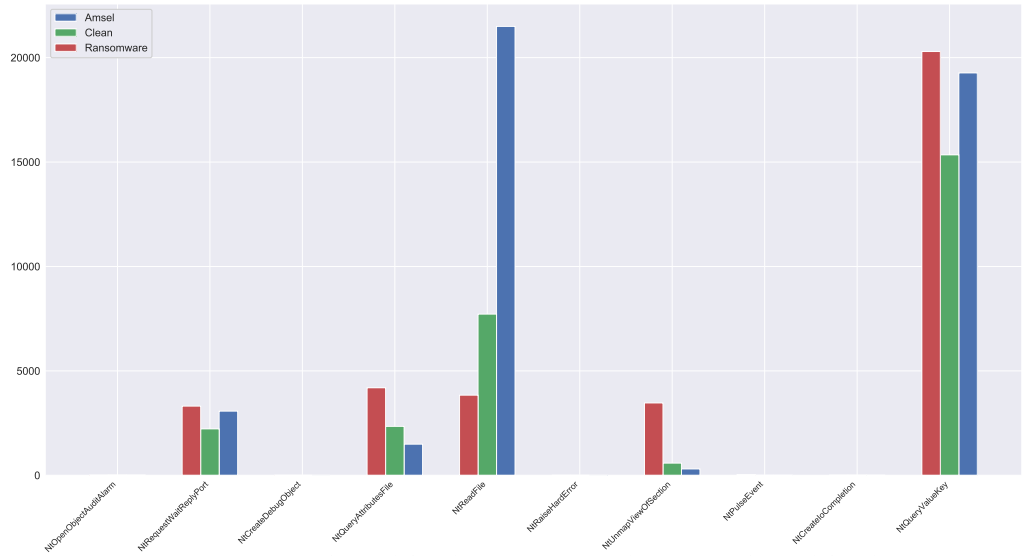

Figure 4.11: Frequencies (y-axis) of the top ten features (x-axis) of Decision Tree in order (from left to right) for malicious, clean, and Amsel data from kernel driver.

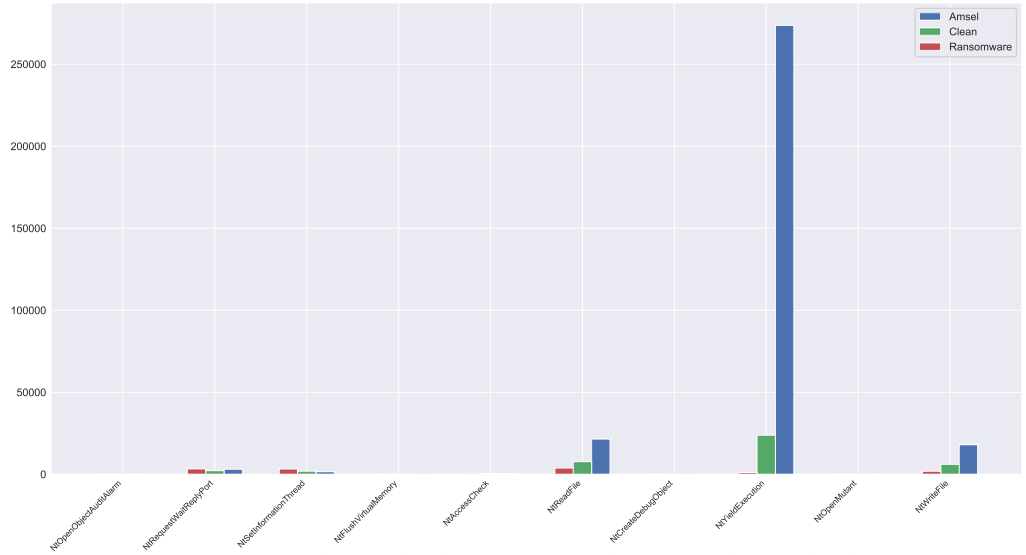

Figure 4.12: Frequencies (y-axis) of the top ten features (x-axis) of Gradient Boost in order (from left to right) for malicious, clean, and Amsel data from kernel driver.

Figure 4.11 shows the mean frequencies of the top ten features for the kernel data. The most prominent call, NtQueryValueKey is used heavily by both ransomware and emulated ransomware. Again this is likely the Java Virtual Machine that checks the registry for various configuration parameters. This is also the reason behind the presence of NtRequestWaitReplyPort in the emulated ransomware. This call is used by the JVM to listen for connections.

Moving onto the calls relating to file-handling, an interesting phenomenon is that NtReadFile is not called as frequently by ransomware as benignware. This is significant since many solutions use file activity as an indicator of ransomware [217, 138, 73, 120, 223, 131, 139, 180]. However, ransomware calls NtQueryAttributesFile much more frequently than benignware. This call is used to obtain information about a file and can be used by ransomware for a number of purposes. One possibility is as an anti-VM trick since when sandboxes are created, they tend to be populated with files that are never again modified. Therefore this call can be used to examine files on the system for those properties. Another possibility is that ransomware is using the call to select what files to encrypt, since ransomware tends not to encrypt every file, but just

those it deems invaluable to the user.

In figure 4.12, the main call that eclipses the others is NtYieldExecution. This is used to stop execution of the current thread and start the execution of another. It is most prominently used by the emulated ransomware. The presence of this call is a side-effect from the emulated ransomware "sleeping" for a certain amount of time as NtYieldExecution is used to stop executing the current thread and start executing a new one. The high importance given to this call is very likely partially responsible for Gradient Boost's poor performance detecting emulated ransomware since real ransomware does not use it much. Interestingly, as with NtReadFile, NtWriteFile is used even more rarely by ransomware. In order to get an understanding of the behaviour of the remaining features in the top ten, rather than simply removing the common features, the features whose behaviourial trend is already obvious from these graphs are removed in order to observe the trend in the less prominent features. The features that have been removed are: NtRequestWaitReplyPort, NtReadFile, NtWriteFile, NtYieldExecution, NtQueryAttributesFile, NtUnmapViewOfSection and NtQueryValueKey. The new graphs are shown in figures 4.13 and 4.14

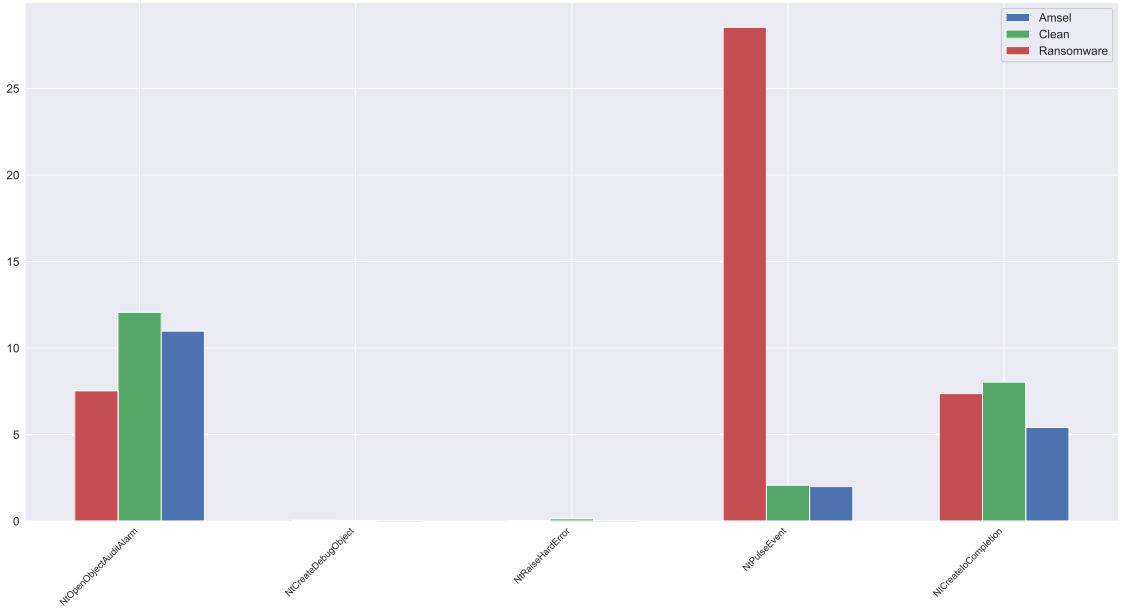

Figure 4.13: Frequencies (y-axis) of the less prominent features (x-axis) within the top ten of Decision Tree in order (from left to right) for malicious, clean, and Amsel data from the kernel driver.

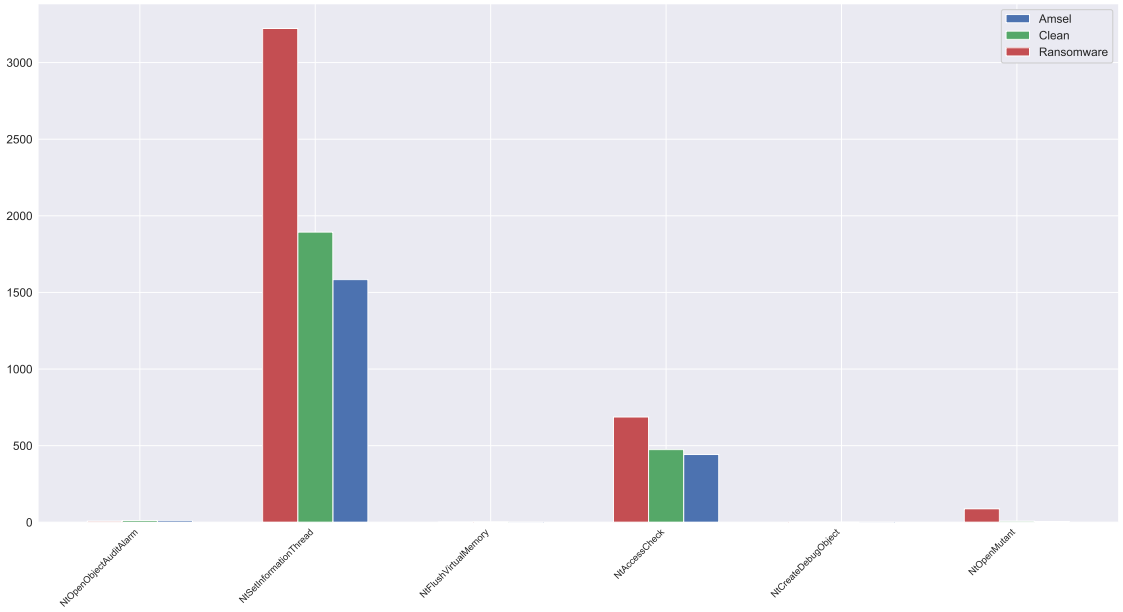

Figure 4.14: Frequencies (y-axis) of the less prominent features (x-axis) within the top ten of Gradient Boost in order (from left to right) for malicious, clean, and Amsel data from the kernel driver.

Figure 4.13 shows the trends in call frequencies for some of the features for Decision Tree. Again, the file-related call NtCreateIoCompletion is called more frequently by benignware than ransomware. In figure 4.14, the feature NtSetInformationThread stands out. Ransomware uses this call significantly more than benignware. One of the parameters in this call contains the attribute ThreadHideFromDebugger, which, as its name suggests can be used by malware to hide its actions from any debugger [88, 50]. However, the emulated ransomware does not use it much, thereby making it more difficult for gradient boost to correctly classify it.

The results from analysing the top ten features used by Decision Tree and Gradient Boost show that the classifiers are still partially reliant on evasive features. However, there is a larger proportion of file related calls compared to the Cuckoo top ten and also a smattering of calls from other categories. Therefore, the classifiers are not completely dependent on evasive features in order to detect ransomware. This is why Decision Tree is more likely to correctly classify the emulated ransomware with shorter interarrival times when using kernel data. Gradient Boost performs worse than Decision Tree on the emulated ransomware for a number of reasons, namely, its overemphasis on evasive techniques (NtSetInformationThread) and interference from the JVM assisting Decision Tree's results (NtRequestWaitReplyPort and NtQueryValueKey). In addition, the call NtYieldExecution, which is used both by the JVM and emulated ransomware to encrypt files at regular time intervals seems to have played a part in damaging Gradient Boost's ability to detect emulated ransomware.

4.4.6 Combined Data Results

Having carefully studied the ability of classifiers to detect the emulated ransomware using kernel-level and user-level data separately, the next approach is to determine how well classifiers perform when the data is combined. This is particularly relevant as one of the findings from the previous chapter (chapter 3) was that the combination of user and kernel-level data produced the best classification results. Therefore the aim in this

subsection is to see if that remains true when the dataset is different and whether the combination of data improves the performance when detecting emulated ransomware. The results from these experiments are shown in table 4.7 and figure 4.15.

Machine Learning Algorithm	Ransomware				Amsel
	AUC	Accuracy (%)	Precision	F-Measure	Accuracy (%)
AdaBoost	0.998	98.6	0.985	0.979	0.07
Decision Tree	0.968	97.2	0.955	0.958	1.85
Gradient Boost	0.999	98.7	0.990	0.980	88.5
Linear SVM	0.539	66.0	0.0	0.0	0.0
Nearest Neighbour	0.973	91.6	0.908	0.871	32.4
Random Forest	0.996	98.0	0.987	0.969	43.0

Table 4.7: Classification results using Cuckoo and kernel data combined

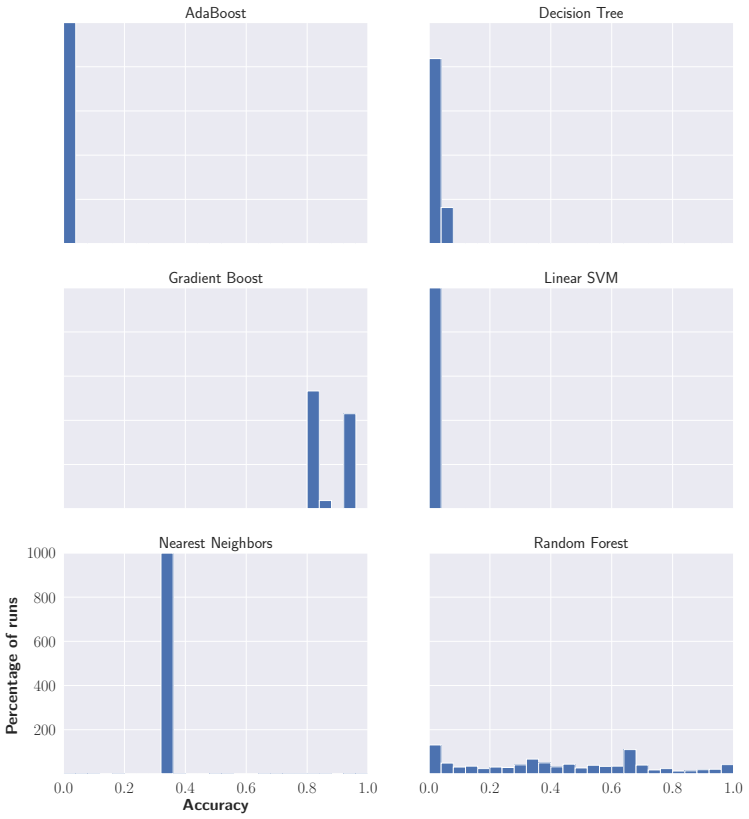

Figure 4.15: Results of classifying Amsel data using the Cuckoo and kernel data together 1000 times.

Table 4.7 shows that combining the kernel and Cuckoo data improves the results in distinguishing real ransomware from benignware, as expected. When it comes to detecting emulated ransomware, the results differ significantly from the trend seen in the previous set of results. Table 4.7 shows that while the maximum accuracy obtained in detecting emulated ransomware has improved significantly by combining the Cuckoo and kernel data (from 76% to 88%), the classifier that obtains the highest accuracy has changed from Decision Tree to Gradient Boost. Unlike the previous results, the only classifier that has obtained an accuracy detecting the emulated ransomware that is higher than 50% is Gradient Boost. Even Decision Tree, which on the Cuckoo and kernel data separately is the strongest, performs poorly when the data is combined.

4.4.6.1 Feature Ranks

To try understand some of the reasoning behind the stark difference in results between Decision Tree and Gradient Boost, the top ten features were found using the inbuilt feature ranking method. These are shown in table 4.8.

Decision Tree	Gradient Boost
FindResourceA	NtOpenObjectAuditAlarm (*Kernel*)
NtOpenObjectAuditAlarm (*Kernel*)	FindResourceA
CreateDirectoryW	NtReadFile
NtQueryValueKey (*Kernel*)	CreateDirectoryW
WriteProcessMemory	GetSystemMetrics
NtLockFile (*Kernel*)	WriteProcessMemory
NtWriteFile (*Kernel*)	NtFlushVirtualMemory (*Kernel*)
NtDeleteKey (*Kernel*)	NtYieldExecution (*Kernel*)
NtReadRequestData (*Kernel*)	NtQueryValueKey (*Kernel*)
GetFileAttributesExW	NtProtectVirtualMemory

Table 4.8: Top ten features using inbuilt feature ranking for Decision Tree and Gradient Boost when considering the data from the kernel driver and Cuckoo combined. The calls that came from the kernel data have been labelled as such.

Many of the features in table 4.8 are features that have been encountered previously. Since the number of features have now doubled, it would seem from the top ten features that Decision Tree is overfitting to the data (hence why it cannot correctly classify the emulated ransomware). This is evidenced by calls like NtLockFile, NtReadRequestData and NtDeleteKey which are not known to be good distinguishing features for ransomware. In addition, they are not heavily used (or used at all) by the emulated ransomware. To further clarify the reasons behind the results, the frequency of the features not in common have been plotted. In addition, NtYieldExecution has been removed from the Gradient Boost plot since, as seen previously, it tends to dominate the

plot making it difficult to see the trends in other calls. The trends are shown in figures 4.16 and 4.17.

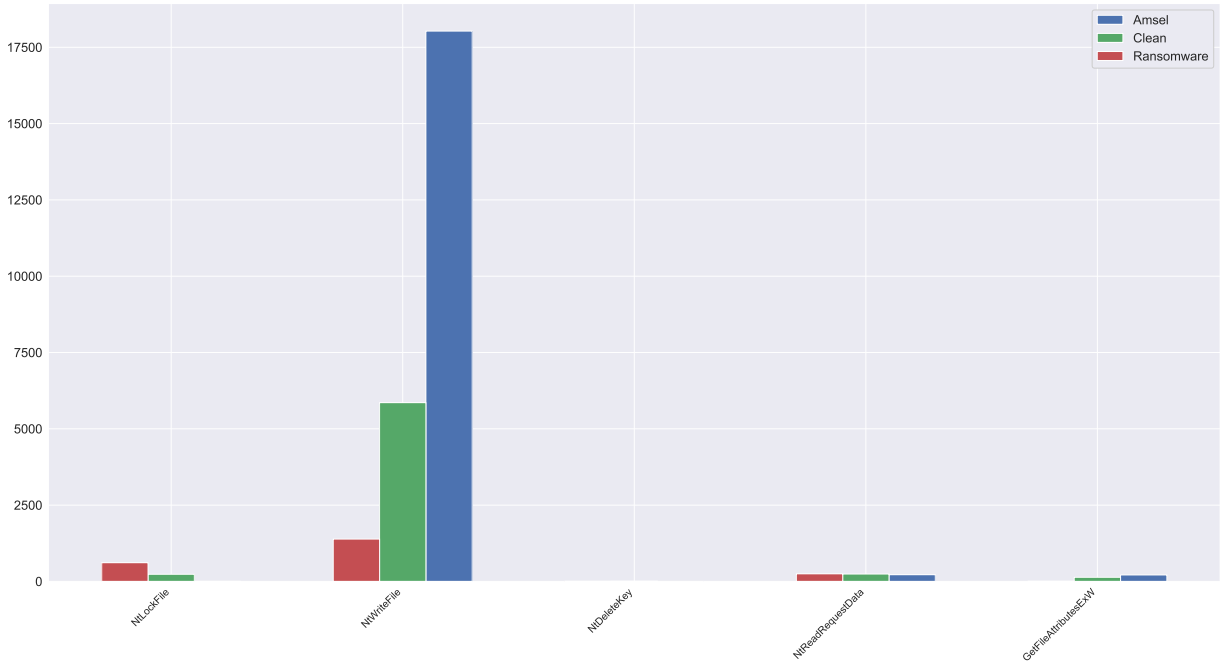

Figure 4.16: Frequencies (y-axis) of the unique features (x-axis) within the top ten of Decision Tree in order (from left to right) for malicious, clean, and Amsel data from the combination of the kernel driver and Cuckoo.

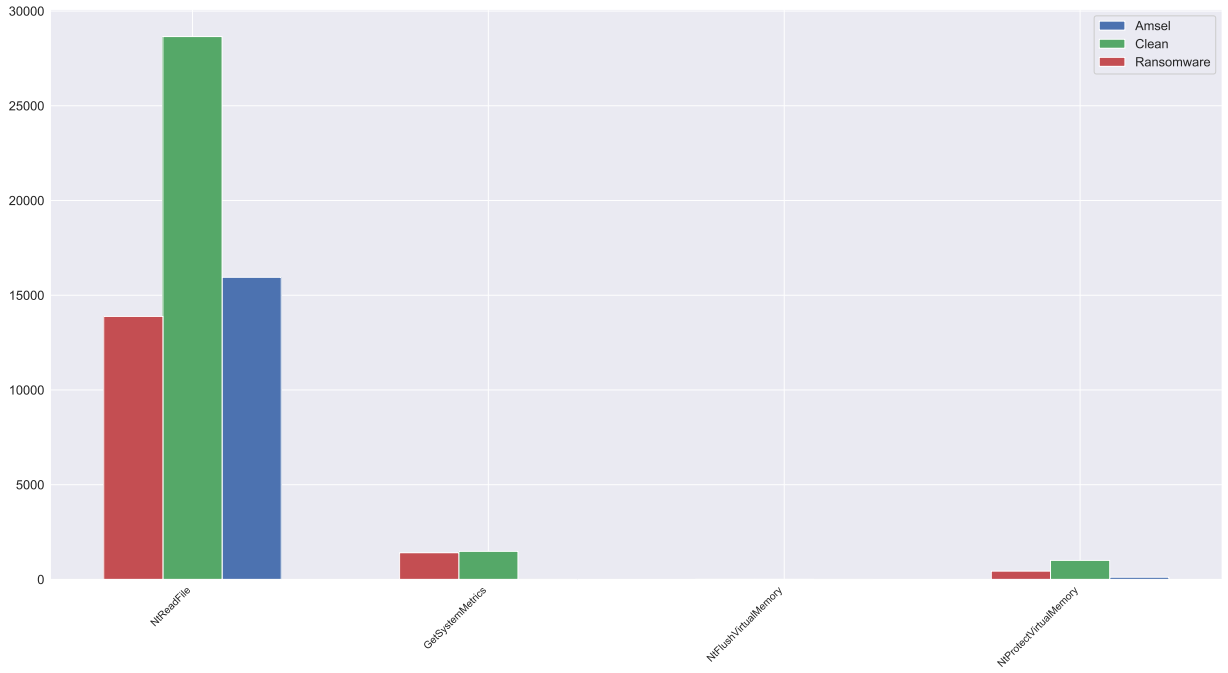

Figure 4.17: Frequencies (y-axis) of the unique features (x-axis) within the top ten of Gradient Boost in order (from left to right) for malicious, clean, and Amsel data from the combination of the kernel driver and Cuckoo.

The most noticeable feature in figure 4.17 is NtReadFile. The call shown in this figure is the one recorded by Cuckoo. What is remarkable about this call is that the frequency with which it has been called by benignware far exceeds that of emulated ransomware. Furthermore, the frequency with which it was called on average by emulated ransomware is very similar to that of actual ransomware. In comparison in figure 4.16 the most prominent feature is NtWriteFile. Here, the frequency with which it was called by emulated ransomware far exceeds that of ransomware and is closer to that of benignware (but still far beyond it). This gives additional insight as to why Gradient Boost outperformed Decision Tree when it comes to detecting emulated ransomware.

4.4.6.2 Misclassified Samples

In order to get a better understanding of the results from the best performing classifier, the classification results per emulated ransomware sample are plotted. For each sample, the time between encrypting each file is extracted and plotted. This is described in more detail in section 4.3.3 - Misclassified Samples.

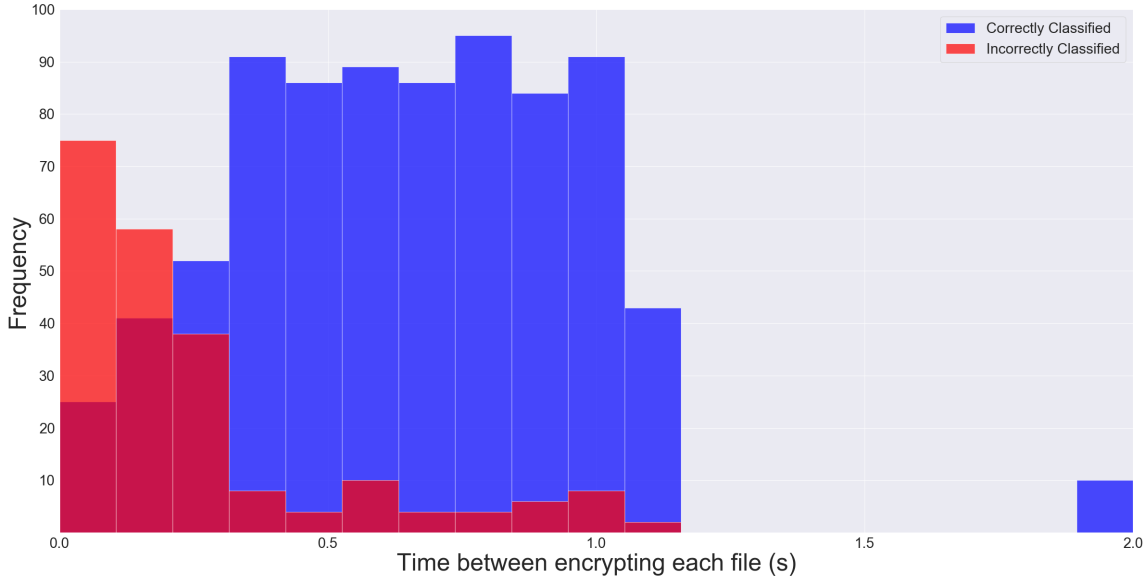

Figure 4.18: Results of classifying Amsel samples (with time between encryption <=2s) using the combined data of Cuckoo and the kernel driver.

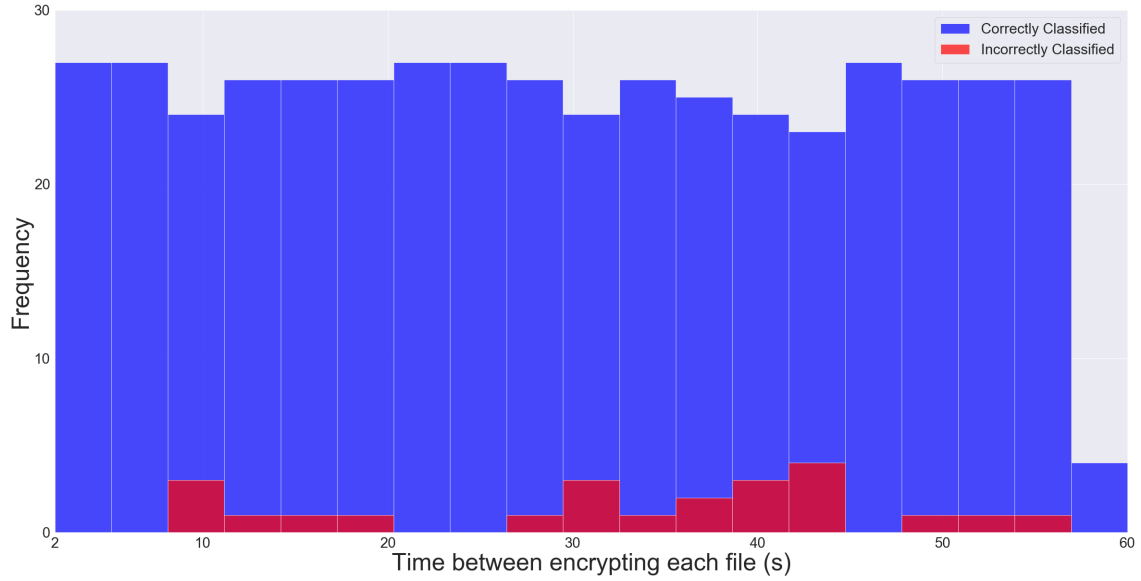

Figure 4.19: Results of classifying Amsel samples (with time between encryption >2s) using the combined data of Cuckoo and the kernel driver.

As can be seen from figures 4.18 and 4.19, in general, gradient boost is able to identify the emulated ransomware as malicious regardless of the time spent between encrypting each file. However, interestingly, when the time intervals are very short (below half a second), gradient boost is much more likely to classify the sample as benign. This could be due to the fact that many benign tools that use similar calls to ransomware (such as 7zip and WinRaR) tend to encrypt/zip the files requested as quickly as possible. Therefore, the results are not necessarily problematic since it protects from false positives. However, it is important to remember that ransomware could utilise this knowledge to evade traditional classifiers.

The combination of Cuckoo and the kernel driver data can be seen to be more effective when it comes to detecting real and emulated ransomware. The classifiers still rank evasive features highly, however, additional features are taken into account, in particular, file-handling related calls, forming a more holistic picture of ransomware. Gradient Boost showed the best performance classifying emulated ransomware, struggling the most when the time between encrypting each file (interarrival time) was very

low (almost zero).

4.4.7 Call Category Frequency

In order to better understand the data that the classifiers are being fed, a study of the type of calls being made most frequently by ransomware, benignware, and emulated ransomware (Amsel) was conducted. The frequency with which calls in each category were made are depicted as pie charts. The spread of calls in the Cuckoo data are shown in the pie charts in figures 4.20, 4.21 and 4.22.

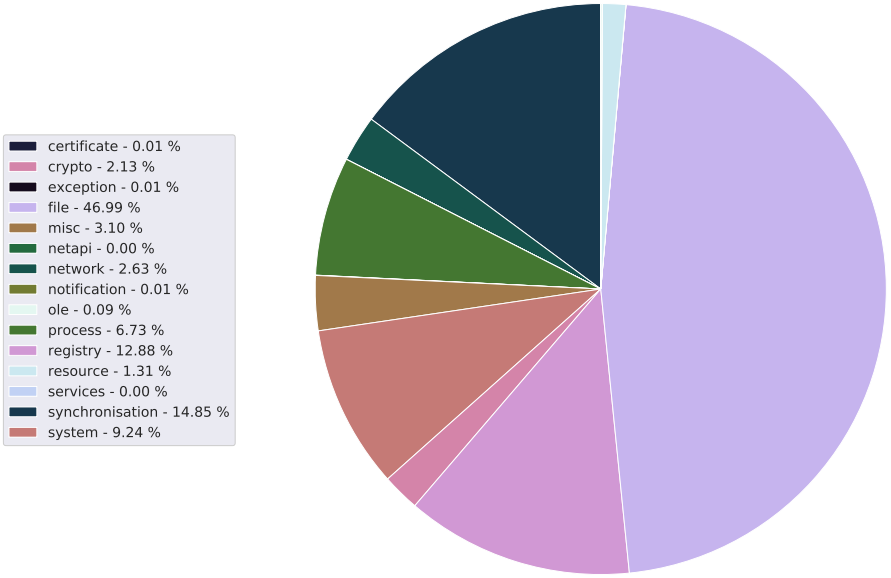

Figure 4.20: Distribution of call categories in data recorded by Cuckoo for clean samples.

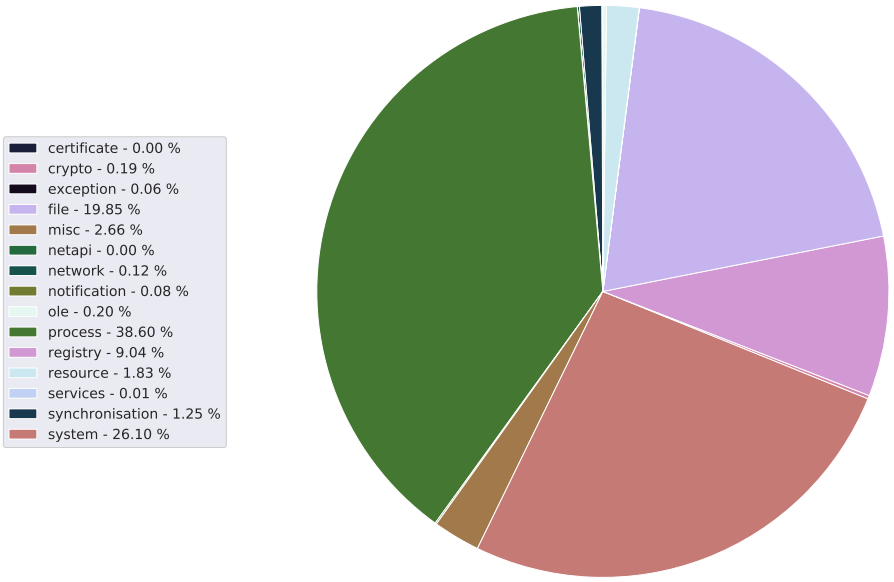

Figure 4.21: Distribution of call categories in data recorded by Cuckoo for ransomware samples.

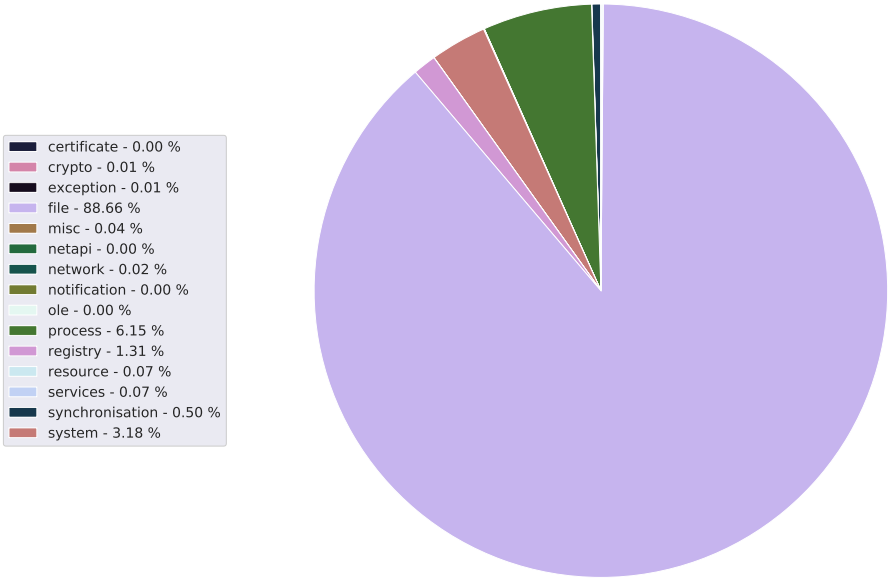

Figure 4.22: Distribution of call categories in data recorded by Cuckoo for emulated malware samples.

An interesting observation when comparing figures 4.20 and 4.21 is that 46% of the calls relate to file handling for benignware while only 19% of the calls relate to file handling for ransomware. Given that the main aim of ransomware is to encrypt a user's files, the expectation is that it would contain a larger proportion of file calls. While that may be the case in a real, unmonitored environment, a large proportion of what a dynamic analysis tool observes when analysing malware is evasive behaviour (as seen previously). The remaining categories of calls commonly used by benignware relate to synchronisation and registry usage. Ransomware, on the other hand, consists largely of calls relating to processes. 50% of the calls made in the process category for ransomware come from one call, ReadProcessMemory. This can be used by malware to read the memory of another process [227]. The purpose of ReadProcessMemory is further confirmed when looking at the next most frequently made call in the process category for ransomware, Process32NextW. This is used by malware to cycle through a list of running processes (usually obtained by calling CreateToolhelp32Snapshot) to find a process to inject its code into. The last major category of calls made by ransomware samples is system. In system, 50% of the calls from ransomware belong to LdrGetProcedureAddress. This call was also in the top ten features for Gradient Boost as it has been known to be used to evade specific analysis mechanisms.

The category of calls used by emulated malware samples is largely dominated by calls made to the file system. Hence, it can be seen that the emulated samples are, for the most part, behaving in an expected manner.

The distribution of call categories within the kernel data is shown in figures 4.23, 4.24, and 4.25.

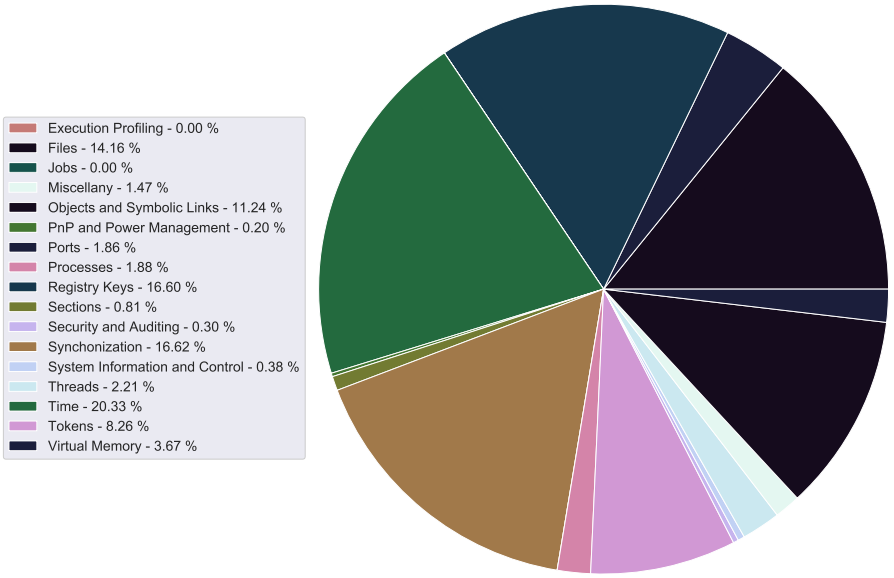

Figure 4.23: Distribution of call categories in the data recorded by the kernel driver for clean samples.

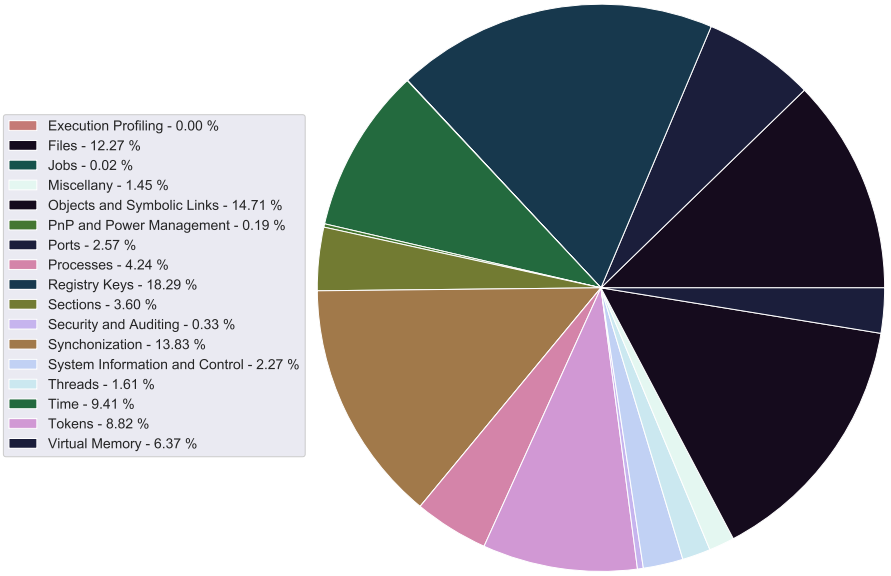

Figure 4.24: Distribution of call categories in the data recorded by the kernel driver for ransomware samples.

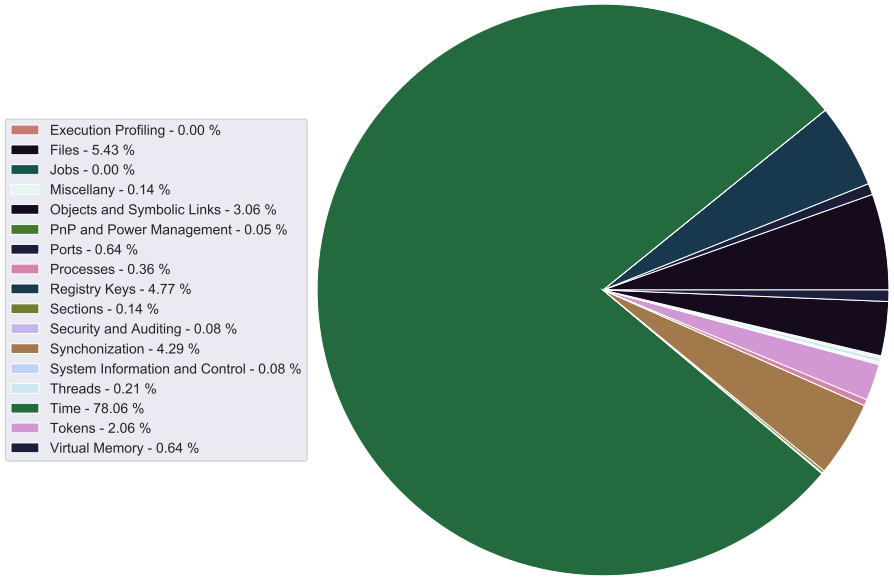

Figure 4.25: Distribution of call categories in the data recorded by the kernel driver for emulated malware samples.

The pie charts of the call categories for the malicious and benign samples is quite evenly distributed. This is partly down to the fact that the kernel driver records the calls for all processes in the system. The largest difference between figures 4.23 and 4.24 is within the "time" category. The call in this category that contributes the most to this difference is NtYieldExecution which has been seen previously (4.4.5). This difference is possibly due to the fact that the main reason ransomware (and malware in general) is likely to use the calls in this category is to detect whether they are running in a virtual/emulated environment. Though there are many legitimate uses for the calls in this category, they do not interest malware authors as much as general software developers. In addition, calls relating to synchronisation are used more frequently by benignware as would be expected given the previous observation.

Another category with an interesting difference is "Sections". Whereas 3% of the calls relate to sections in malware, only 0.8% of the calls relate to sections in benignware. Looking deeper into the difference, it's evident that it is due to the fact that malware

makes much more use of NtMapViewOfSection and NtUnmapViewOfSection (whose purpose has been described in section 3.3.1.1). On the other hand, the top two calls in this category for benignware is NtMapViewOfSection and NtCreateSection. This is because benignware is more likely to actually create its own sections, rather than attempt to inject itself into another process' memory.

The "processes" category contributes 4% of the calls for the average ransomware sample and only 1% for the average benignware sample. The reason for this is that ransomware (and malware in general) uses NtQueryInformationProcess much more frequently than benignware since it can serve as anti-debugging method. While NtQueryInformationProcess contributes 66% to the processes category in ransomware, it only contributes 53% to the equivalent for benignware.

Finally, the pie chart for emulated malware in figure 4.25 bears a strong resemblance in shape to the equivalent chart from Cuckoo in figure 4.22. However, there is a major difference. With Cuckoo the main category of calls were file related. On the kernel side, the most dominant category of calls relate to time. The main reason for this is that the two calls contributing to the most to the time category - NtQueryPerformanceCounter and NtYieldExecution - are not monitored by Cuckoo. This means that Cuckoo misses out on a major element of a program's behaviour. However, it can be argued that it helps in stopping classifiers from learning unnecessary behaviours that can lead to over-fitting. In fact, removing those two features from the kernel dataset improves the base random forest accuracy (before hyper-parameter tuning) from 48.6% to 51.0% when detecting emulated malware. With Gradient Boost, the accuracy jumps from 0.540% to 67.1%. These calls are used very heavily by the emulated ransomware to schedule the encryption of each file.

The call category data has further confirmed the pervasiveness of evasive techniques in the ransomware data in addition to the relative lack of file handling calls. An interesting observation is between the pie charts of the Cuckoo data and kernel data with regards to the emulated ransomware data. The reason for the difference is mainly down

to the fact that Cuckoo does not monitor some calls that the kernel driver does. These calls can be considered important for detecting malware by some analysts (particularly NtQueryPerformanceCounter), however, they can also be seen to be at risk of causing overfitting. In addition, these visualisations have highlighted some unintended consequences due to the manner in which the emulated ransomware was implemented, some of which would be difficult to overcome completely.

4.4.8 Cuckoo Missing Calls

In most of the experiments performed in this thesis, the UI calls monitored by Cuckoo are ignored. This is so that the comparison between the kernel driver and Cuckoo are fair (since the UI calls are not monitored on the kernel side). In addition, if a classifier is identifying malware by its UI activity (or lack of), it is unlikely to be robust since malware can easily alter its UI activity as it is not its primary concern. This is best evidenced in the change in results for the Cuckoo data in detecting the emulated ransomware as shown in table 4.9 and figure 4.26.

Machine Learning Algorithm	Ransomware				Amsel
	AUC	Accuracy (%)	Precision	F-Measure	Accuracy (%)
AdaBoost	0.992	96.6	0.959	0.959	1.80
Decision Tree	0.953	95.4	0.941	0.950	35.4
Gradient Boost	0.995	97.4	0.966	0.967	45.8
Linear SVM	0.863	75.3	0.860	0.687	1.07
Nearest Neighbour	0.969	91.4	0.927	0.889	1.47
Random Forest	0.994	97.0	0.975	0.961	60.0

Table 4.9: Classification results using Cuckoo data with UI features

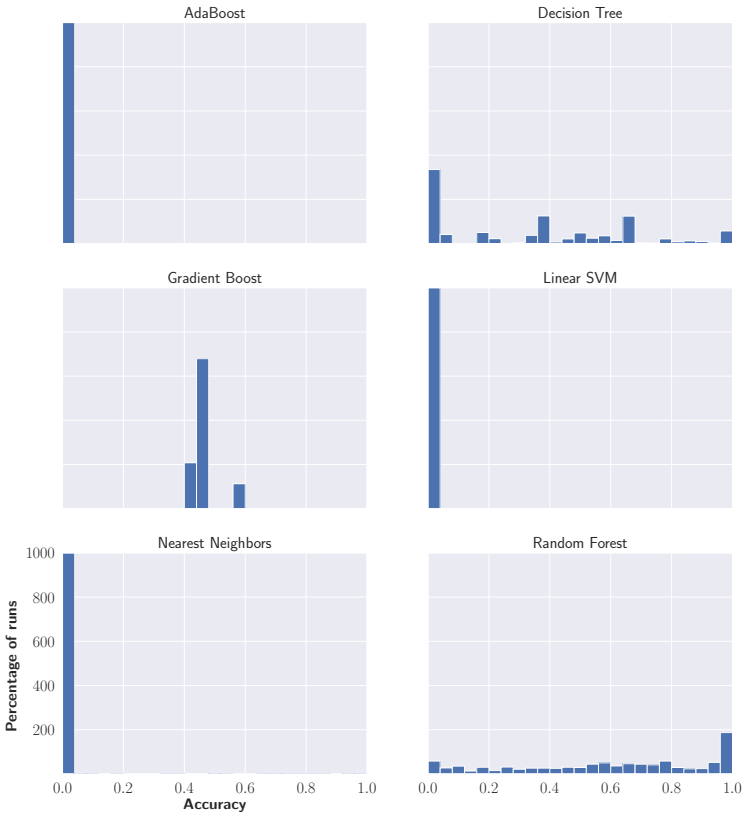

Figure 4.26: Results of classifying Amsel data from Cuckoo 1000 times

In distinguishing real ransomware from benignware using the additional calls provided by Cuckoo, it can be seen from table 4.9 that there is barely any difference in the results. Therefore, the additional calls neither add nor take away anything with regards to the training data. However, when it comes to detecting the emulated ransomware, the drop in results is significant. Without the additional features, Decision Tree was the best performing classifier on the emulated ransomware data from Cuckoo with an accuracy of 77%. In addition, two classifiers were able to obtain an accuracy higher than 50%. When the UI features are added, however, only Random Forest is able to get above 50% on the Amsel data from Cuckoo with an accuracy of 60%. This is significantly worse than the original classification results.

This highlights the importance of feature selection, particularly with regards to the Cuckoo data. When hooking at user-level, there is an abundance of calls available to hook, some of which can be very specific. Therefore, care must be taken when choosing which calls to hook to ensure that the calls chosen are not likely to result in over-fitting as that can be easily evaded. This also shows that though certain calls were removed from the default set provided by Cuckoo for this research, it has not had a negative impact on the results.

4.4.8.1 Feature Ranks

In order to confirm that the UI calls are responsible for the significant dip in results when detecting emulated ransomware, the top ten features using the inbuilt and independent feature ranking methods were extracted. They are shown in table 4.10

Independent Feature Ranking	**Inbuilt Feature Ranking**
DrawTextExA	DrawTextExA
FindWindowA	FindWindowA
CreateDirectoryW	NtReadFile
FindResourceA	DrawTextExW
LdrGetProcedureAddress	FindResourceA
SetWindowsHookExA	NtOpenSection
NtReadFile	LdrGetProcedureAddress
DrawTextExW	CreateDirectoryW
SetFileTime	NtTerminateProcess
NtOpenSection	GetForegroundWindow

Table 4.10: Top ten features for Random Forest on all Cuckoo data

As can be seen the top ten is now dominated by UI calls. Three calls in the independent top ten (DrawTextExA, FindWindowA and DrawTextExW) and four in the inbuilt top ten (DrawTextExA, FindWindowA, DrawTextExW and GetForegroundWindow) relate

to UI. The decision making process is now centred around UI activity. This can be easily evaded by malware, particularly Trojan horses which tend to attach themselves to legitimate programs and therefore would not differ at all from benignware with regards to UI calls.

4.5 Conclusion

The aim of this chapter was to assess whether the dynamic analysis process is biased by the change in behaviour malware displays when under investigation. To help with this, a tool called Amsel was used to generate samples that emulate the malicious behaviour of ransomware without any evasive properties. To begin with, the initial experiments were conducted using real ransomware and benignware. These revealed that the data from the kernel driver is more effective than that from Cuckoo for differentiating ransomware from benignware. The classifier that showed the best performance was Gradient Boost, obtaining an accuracy of 97.3% on the Cuckoo data and 98.2% on the kernel data. Using Amsel, emulated ransomware samples were created. These same trained classifiers were then made to classify data from running 1500 variations of emulated ransomware. Samples only varied on one parameter, the interarrival time or the time spent waiting between encrypting each file. This time, the best performing classifier was Decision Tree, obtaining an accuracy of 76.7% on the Cuckoo data and 67.0% on the kernel data. Further analysis revealed that, when trained on Cuckoo data, Decision Tree showed the strongest performance when the interarrival time was higher. In other words, when Decision Tree was trained on data from Cuckoo, it was more likely to classify a sample as ransomware as the time it spent encrypting files was reduced. The opposite was true for the kernel data. It was also found that by modifying a single hyper-parameter of gradient boost and random forest, they were able to get the same performance detecting emulated ransomware as that obtained by decision tree. The hyper-parameter in question increased the risk of both classifiers over-fitting, thereby suggesting that malicious behaviour of real ransomware is not the

most distinguishing behavioural trait seen in ransomware when under analysis.

Through studying the ten most useful features for Decision Tree and Gradient Boost, it was possible to identify the reasons behind the results. When identifying ransomware using the Cuckoo data, the primary attribute used by classifiers to detect ransomware were its evasive attributes. In addition, the reason for Decision Tree performing better than Gradient Boost using the Cuckoo data was, in part, due to a unintended behavioural side effect of the emulated ransomware. Using the kernel data, the classifiers placed more emphasis on file-handling calls explaining why Decision Tree performed better when the emulated ransomware had higher rates of encryption. Likewise, Decision Tree's superiority to Gradient Boost on emulated ransomware was also due to unintended behavioural traits present in Amsel's design.

As with the previous chapter, the best performance came from combining the user and kernel-level data. However, this time, the best performing classifier on real ransomware and emulated ransomware was Gradient Boost. Its accuracy detecting real ransomware was 98.7%, and its accuracy detecting emulated ransomware was 88.5%. With the increase in features, Decision Tree overfitted to the training data, which, in this case, did not help it in detecting emulated ransomware.

This research has shown that the dynamic analysis process, as is currently carried out, encourages classifiers to identify malware using the evasive properties that they show. This presents some risks, since, in a real environment, malware may not show as many evasive properties once it establishes it is running in a real environment. In addition, all it would require of a malware author to evade classifiers trained using the traditional dynamic malware analysis approach is to remove all evasive behaviour from their malware. Another observation made from this research is the impact on performance that the hyper-parameter values can have.

Despite not setting out to do so, this research discovered some of the limitations with using Java to develop malware simulators to run on Windows. Its compatibility with multiple OSes makes it a popular choice, even for malware simulators [156]. Though it

allows for the quick development of complex algorithms, the JVM's interference with the system call data gathered is too large to be ignored. This highlights the sensitivity of system call data and the importance of ensuring any additional tools present on the monitoring system have little to no impact on the system call data gathered.

Finally, these experiments have highlighted the importance of taking due consideration when selecting features to monitor. This will define what the classifiers use to identify malware. As shown, if allowed to identify malware through the use of UI features, classifiers are likely to. This would make them extremely susceptible to adversarial attacks. Therefore, feature selection must be performed with the malicious behaviour in mind that needs to be blocked.

Chapter 5

Testing the robustness of emulated ransomware results

The results in the previous chapter were affected by the interference from the JVM. Therefore, the purpose of this chapter is to determine the results that would be obtained if the experiments carried out in the previous chapter used an emulator that does not suffer the same shortcomings. By using the same emulator but written in a different language (C), it will be possible to either reject or further confirm the conclusions made in chapter 4. This will provide the answer to the sixth research question:

RQ6 *Are high-level languages such as Java suitable for emulating malware to test system call monitoring tools?*

In addition, through answering this question, this chapter tests the robustness of classifiers to detect malware that is functionally exactly the same but different with regards to the system calls used. This provides the eighth contribution:

C8 *This research determines the sensitivity of classifiers trained in the traditional dynamic malware analysis process to changes in system calls made.*

5.1 Method

The method employed in this chapter is exactly the same as that used in chapter 4 as it is an extension of it. The only difference in this chapter is that the emulated ransomware samples are written in C. The theory behind using C is that C provides more control over the system calls made by the program. In addition, the developer can specifically choose the system calls made.

Therefore, as mentioned previously, the functionality of Amsel employed in chapter 4 is implemented in C for this chapter. Much of the implementation is quite straightforward, as the program simply encrypts files in a specified directory and waits a specified amount of time between encrypting each file. However, in order to test the robustness of the trained classifiers, the delay between encrypting each file was implemented in two different ways. The first set of emulated ransomware used the `time` function provided by C. This function returns the number of seconds since January 1, 1970. It is used to implement the delay as follows:

```
1 delay = time(0) + secondsToWait;
2 while(time(0) < delay);
```

Listing 5.1: Delay implemented using a standard C method

The function `time()` is defined in time.h. Behind the scenes, `time` calls `GetSystemTimeAsFileTime` when run on Windows. The return value when `time(0)` is called is the number of seconds since January 1, 1970 at that point in time. The variable `secondsToWait` contains the user specified time to wait. This is added to the current time. After that, on line 2, a `while` loop is used to prevent progress until the current time exceeds the time in the future that it needs to wait until.

The second set of emulated ransomware was implemented using the delay function provided by Windows. In C, this function is called `Sleep` and it goes on to call the Windows system call `NtDelayExecution`. Implementing a delay with this is very straightforward:

```
Sleep(millisecondsToWait);
```

Listing 5.2: Delay implemented using a recommended Windows method

The reason for creating two types of emulated ransomware samples is to further test the robustness of the classifiers. The use of NtDelayExecution by malware is already documented [179]. It has also been encountered in this research in the top ten most frequent calls in tables 3.6 and 3.7. GetSystemTimeAsFileTime has not been encountered as much, although it appeared in the top ten most frequent calls for benignware and malware in table 3.6. Despite the fact that both system calls can be used to implement the same functionality, there is a possibility that the difference in behaviour of these calls will affect the classification accuracy of the emulated ransomware.

The experiments carried out in this chapter are identical to those carried out in the previous chapter. As with Amsel, 1500 emulated ransomware samples using the C time function were generated with the same spread of time delays as Amsel. Likewise, 1500 emulated ransomware samples using the Windows Sleep function were also generated. As before, the classifiers were trained on the real ransomware and benignware from chapter 4 and then separately tested on each group of emulated ransomware. These experiments were conducted for the Cuckoo and Kernel data.

5.2 Results

5.2.1 Standard C time method

The results from testing the classifiers on the emulated ransomware samples using the C time function are shown in table 5.1

Machine Learning Algorithm	Kernel Driver Accuracy (%)	Cuckoo Accuracy (%)
AdaBoost	83.1	0.0
Decision Tree	60.0	0.0
Gradient Boost	60.1	0.0
Linear SVM	0.0	0.0
Nearest Neighbour	6.07	2.27
Random Forest	48.6	3.54

Table 5.1: Classification accuracy of emulated ransomware with C time function using data from Cuckoo and the Kernel driver.

On the Cuckoo side, the results are considerably worse compared to the results when Java was used to emulate ransomware. Not a single classifier correctly classifies even 50% of the samples. On the other hand, using the kernel data, the classifiers obtain much better results. AdaBoost gets the best results with 83.1%. Therefore the most influential features are studied to determine what contributed to these results.

5.2.1.1 Influential features

To understand what contributed to the results, the most important features were analysed using the inbuilt feature ranking method described in section 3.2.3. Unlike chapter 4, the top 20 features are analysed since the emulated ransomware samples used in this chapter are very minimal with regards to the system calls they use. Given that AdaBoost is the best performing classifier for the kernel data, the top 20 features from this classifier are analysed. The same is used for the Cuckoo data since there were no outstanding classifiers. The first top ten features and the frequencies with which they were called are shown in figures 5.1 and 5.2

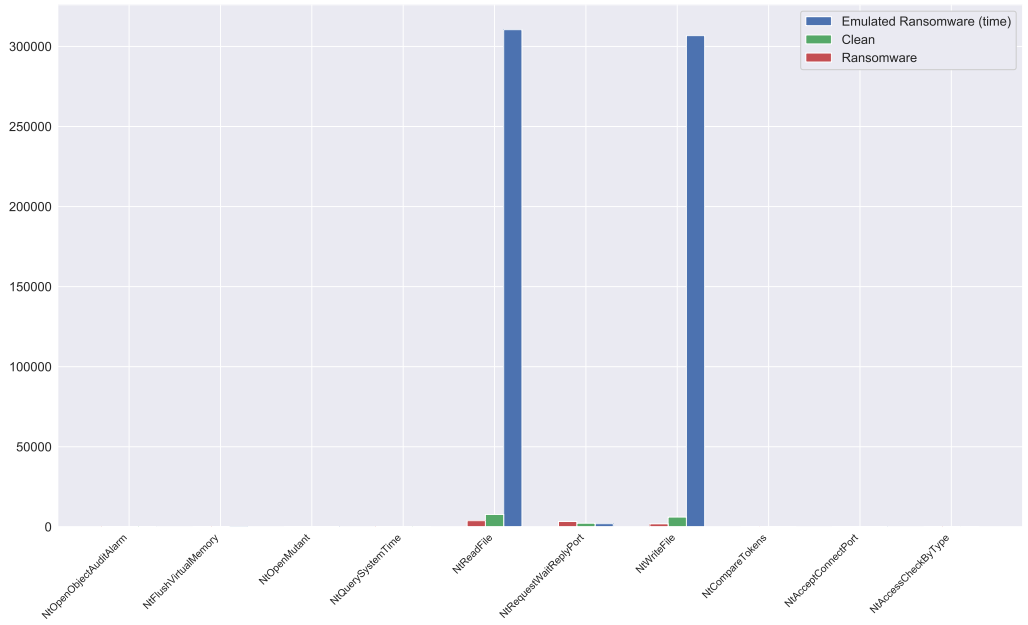

Figure 5.1: Frequencies (y-axis) of the top ten features (x-axis) of AdaBoost in order (from left to right) for malicious, clean, and emulated ransomware data (using C time) from the kernel.

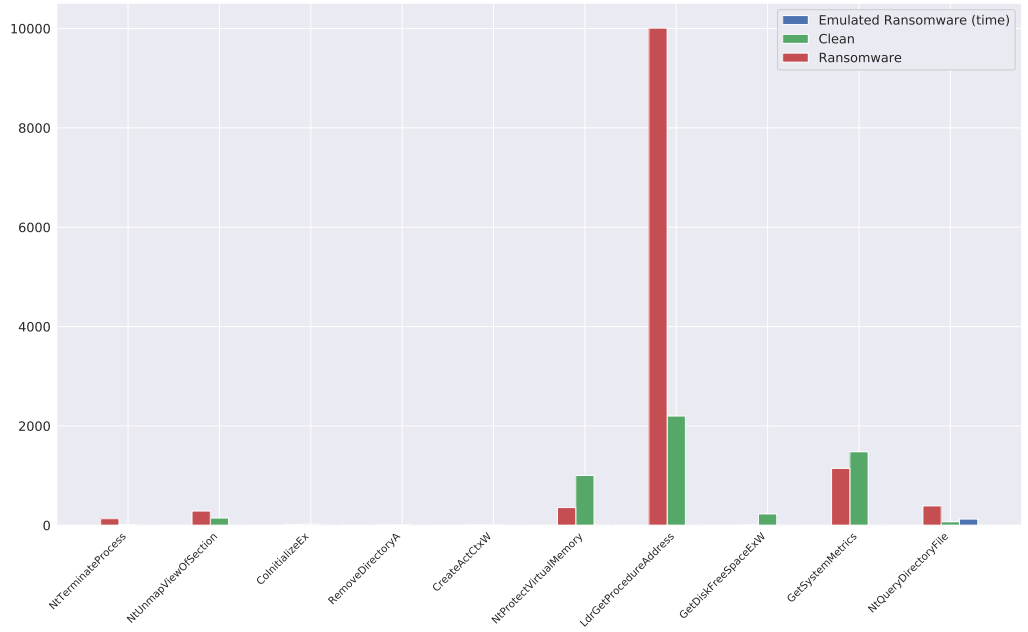

Figure 5.2: Frequencies (y-axis) of the top ten features (x-axis) of AdaBoost in order (from left to right) for malicious, clean, and emulated ransomware data (using C time) from Cuckoo.

Figure 5.1 shows the sheer volume of file-handling related calls made by the emulated ransomware. The two peaks in the call frequency are due to NtReadFile and NtWriteFile. The importance that AdaBoost has placed on file-handling calls when trained on the kernel data is also quite evident from figure 5.1. As observed previously, these two calls seem to be called more frequently by benignware than ransomware.

In contrast, figure 5.2 does not focus on features used by the emulated ransomware. The only feature in figure 5.2 that emulated ransomware uses is NtQueryDirectoryFile. This is also used by ransomware to obtain information regarding files and directories [227]. Besides that, the main focus in figure 5.2 is on features frequently used for evasive purposes (NtUnmapViewOfSection, LdrGetProcedureAddress, GetSystemMetrics).

Already, some important differences can be seen between the graphs here and those in chapter 4. In chapter 4, the calls LdrGetProcedureAddress and NtProtectVirtualMemory were also utilised by the emulated ransomware with a frequency similar to that employed by ransomware. However, the presence of these calls in the emulated ransomware was due to using Java as opposed to being intentional design choices.

Moving on to the next ten most influential features according to AdaBoost, figures 5.3 and 5.4 show their mean frequencies for the kernel data and Cuckoo data.

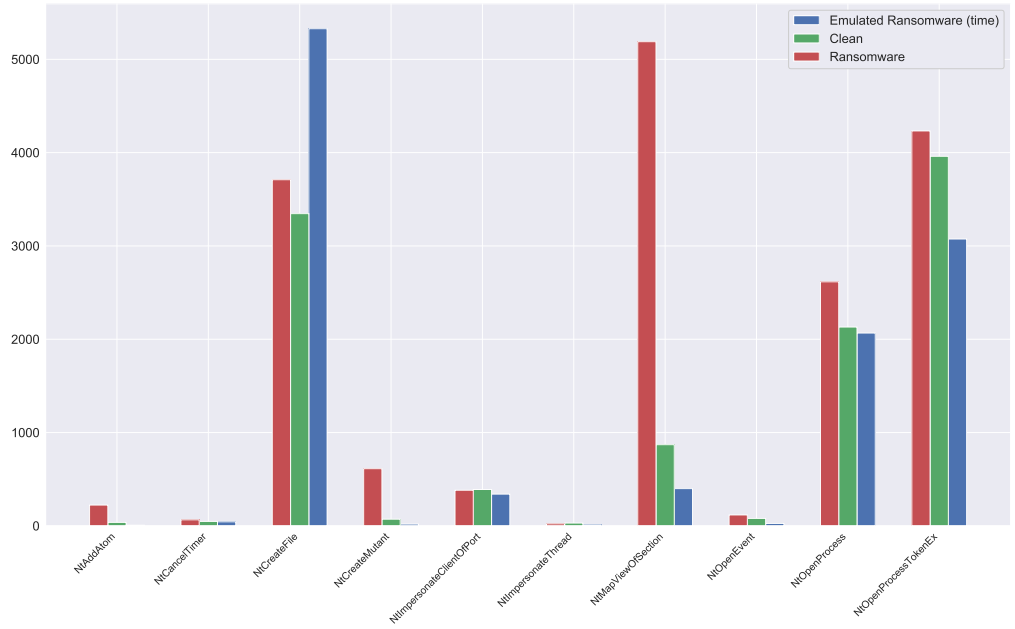

Figure 5.3: Frequencies (y-axis) of the features ranked 10-20 (x-axis) by AdaBoost in order (from left to right) for malicious, clean, and emulated ransomware (using C time) data from the kernel.

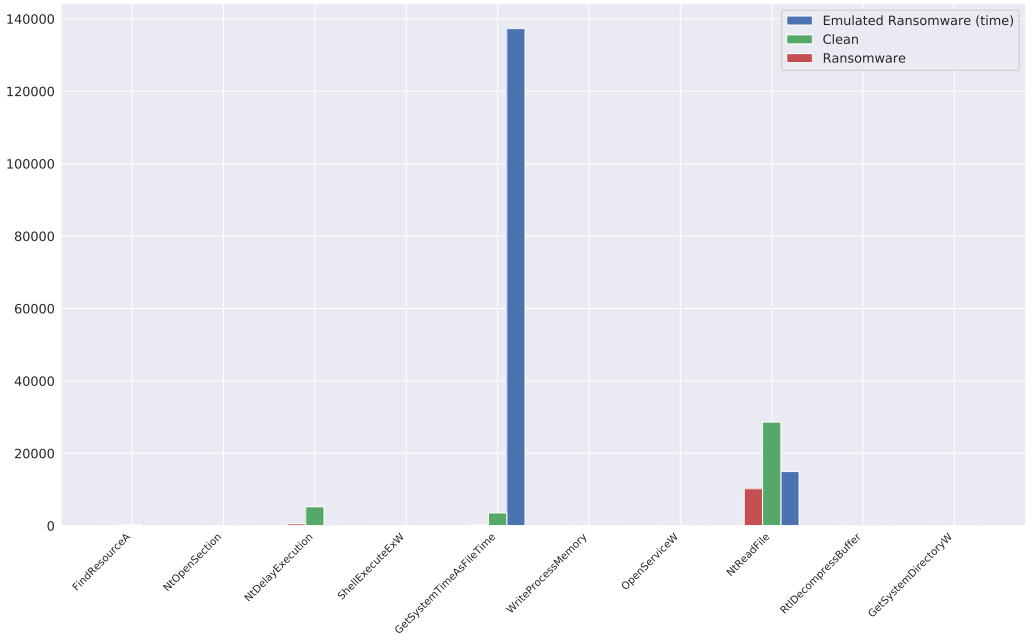

Figure 5.4: Frequencies (y-axis) of the features ranked 10-20 (x-axis) by AdaBoost in order (from left to right) for malicious, clean, and emulated ransomware (using C time) data from Cuckoo.

Figure 5.3 shows the next ten most important features according to AdaBoost when using the kernel data. Of note is NtCreateFile, which, unlike the previous file-handling calls, is called more frequently by ransomware than benignware, and even more frequently by the emulated ransomware. The frequency of the remaining features for emulated ransomware are not caused by its operation but are just due to the general system activity.

Figure 5.4 shows the next ten most important features when using the Cuckoo data. The most prominent peak comes from the emulated ransomware and is due to the time function that it calls (which goes on to call GetSystemTimeAsFileTime). The only other feature used by emulated ransomware is NtReadFile, whose call frequency is similar to that of real ransomware.

5.2.1.2 Misclassified samples

The classification results of the best performing classifier are further dissected to determine the time delay values at which the best performing classifier was able to identify emulated ransomware samples. Since classifiers were only able to produce an acceptable result using the kernel data, the Cuckoo data is not analysed here. The method employed to obtain this data is described in section 4.3.3. Figures 5.5 and 5.6 breakdown AdaBoost's performance based on the time delay between each file encryption.

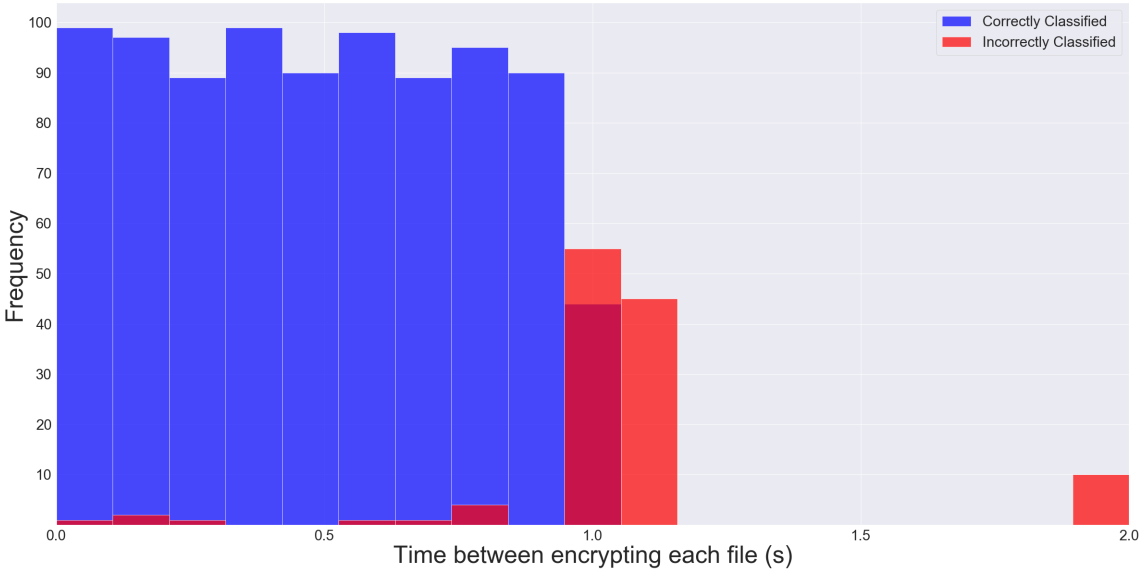

Figure 5.5: Results from AdaBoost classifying emulated ransomware samples using C time (with time between encryption <=2s) with the data from the kernel driver.

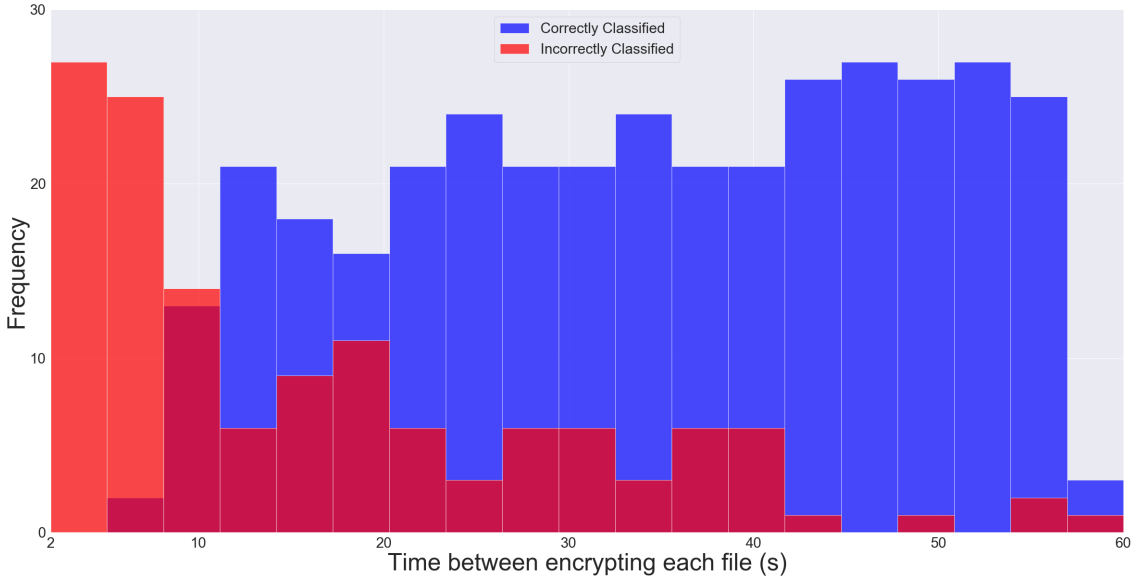

Figure 5.6: Results from AdaBoost classifying emulated ransomware samples using C time (with time between encryption >2s) using the data from the kernel driver.

The results show the misclassified samples to be spread across all the times. The main cluster of misclassified samples is between 1 and 10 seconds. AdaBoost seems to perform best at both extremes, either when the frequency of encryption is very high or very low. By correlating these plots with the top twenty features, it becomes clear that this is likely due to the fact that when the rate of encryption is average, the amount of file-handling calls correspond quite closely to those made by benignware.

5.2.1.3 Call category frequencies

Finally, to get an understanding of the data behind the results, the distributions of the categories of calls obtained from running the emulated ransomware are plotted into a pie chart for both Cuckoo and the kernel driver. The process involved in creating the pie charts is described in section 4.3.4. The distribution of call categories for the kernel data and Cuckoo data is shown in figures 5.7 and 5.8.

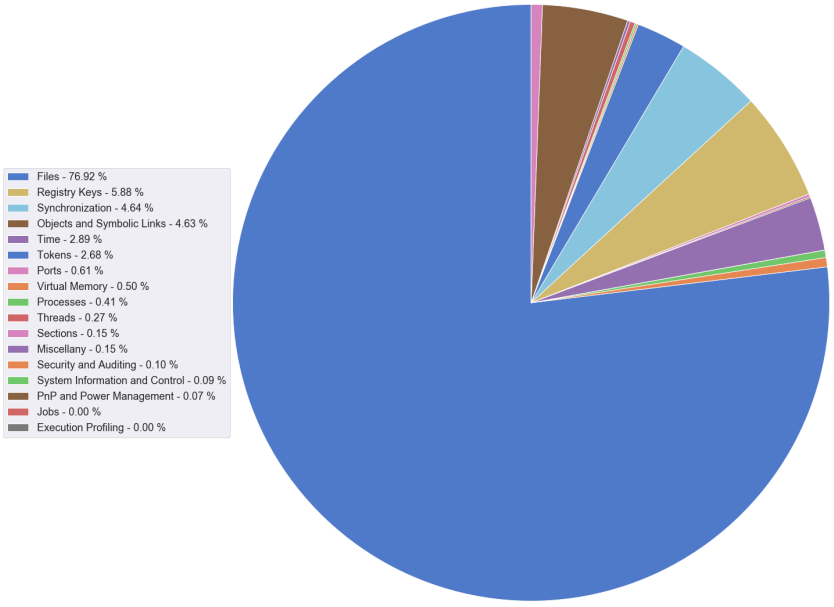

Figure 5.7: Distribution of call categories in data recorded by the kernel driver for emulated ransomware using C time function.

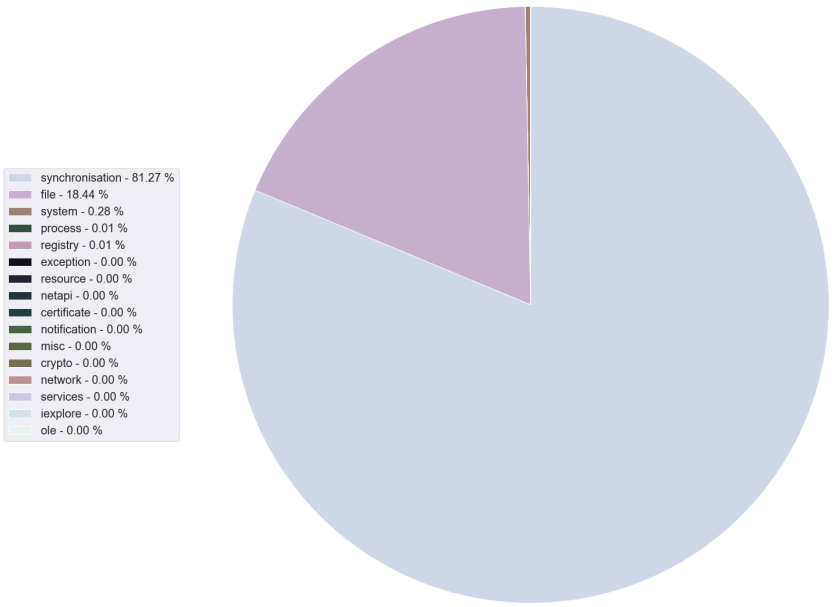

Figure 5.8: Distribution of call categories in data recorded by Cuckoo for emulated ransomware using C time function.

Figure 5.7 shows the call category distribution on the kernel side. The distribution of calls are dominated by calls to the file-system. This is not surprising given the volume of calls made by the emulated ransomware. Though the calls in the additional categories are also used, that is because the kernel driver is monitoring at a system-wide level.

Figure 5.8 shows the call category distribution on the Cuckoo side. There is even less distribution in the pie chart in this case. However, in this case, it is because it is dominated by synchronisation calls. Further analysis reveals that every single call in the synchronisation category comes from one system call, GetSystemTimeAsFileTime. The domination by synchronisation calls is unsurprising given that the time function needs to be called repeatedly to get the current time and compare with the projected end time within the while loop.

5.2.1.4 Combined data results

Finally, the data from Cuckoo and the kernel driver was combined to determine if the results would be aided by this. Due to the fact that only the kernel driver was able to produce a suitable performance, it would be unusual for the results to improve by combining the data since that would add more noise. Previously, the results improved from combining the data due to the fact that the classifiers misclassified different samples depending on whether they were given the Cuckoo or kernel driver data. In this case, since the classifiers were barely able to classify any samples correctly when using the Cuckoo data, it was unlikely that combining the data would help the results. In addition, the emulated ransomware in this chapter uses very few features, therefore the addition of more features was likely to add too much noise. To verify this hypothesis the classification experiments were performed using the data from Cuckoo and the kernel driver. The results are shown in table 5.2.

Machine Learning Algorithm	Accuracy
AdaBoost	21.3
Decision Tree	33.0
Gradient Boost	53.5
Linear SVM	0.0
Nearest Neighbour	1.17
Random Forest	53.7

Table 5.2: Classification accuracy of emulated ransomware with C time function using data from Cuckoo and the kernel driver.

As expected, the results from combining the Cuckoo and kernel driver data have produced weaker classification results than when the kernel data was used on its own. Though some classifiers suffered more than others with the addition of features, ultimately, none were able to obtain an accuracy higher than 54%.

The experiments carried out in this section reveal the lack of robustness in the classi-

fiers trained on system call data from Cuckoo. Even though the emulated ransomware demonstrated a mix of malicious and evasive symptoms, the classifiers using Cuckoo data were not even able to detect more than 10% of the samples. This is due to the fact that many of the features that were ranked highly by the classifiers were not used at all by the emulated ransomware. In addition, the feature used to create the time delay is not commonly used for this purpose by malware. On the other hand, the classifiers using the kernel data had more success, but were unable to get up to 90% accuracy. The classifiers did not perform as well when the amount of file-handling activity exhibited by the emulated ransomware matched that exhibited by benignware. An analysis of the data revealed that due to the manner in which the delay was implemented for the emulated ransomware, the amount of synchronisation calls far exceeded the number of file-handling calls recorded by Cuckoo. However, since these calls did not reach the kernel, the kernel data was dominated by file-handling calls.

5.2.2 Windows C Sleep method

In this section, the emulated ransomware using the Windows 'Sleep' method to implement delay is evaluated. Table 5.3 shows the results from classifying emulated ransomware using the Windows Sleep function to implement the delay between encrypting each file.

Machine Learning Algorithm	Kernel Driver Accuracy (%)	Cuckoo Accuracy (%)
AdaBoost	6.40	74.4%
Decision Tree	2.21	10.7%
Gradient Boost	2.33	0.0%
Linear SVM	0.0	0.0%
Nearest Neighbour	0.33	27.9
Random Forest	3.13	8.08

Table 5.3: Classification accuracy detecting emulated ransomware using Windows Sleep function (NtDelayExecution) using data gathered by Cuckoo and the kernel driver.

The results in table 5.3 show how a small change in the function used (but little change in functionality) can dramatically change the results. This time, the classifiers were unable to get more than 10% accuracy when using the kernel data. Whereas, using the data from Cuckoo, AdaBoost was able to obtain an accuracy of 74.4%. To get an understanding into why the results have changed so drastically, the top 20 features are analysed.

5.2.2.1 Influential features

As the top 20 features are obtained using best performing classifier (AdaBoost) on the training data (as described in section 3.2.3), they are the same as the top 20 features shown for the emulated ransomware using the C time function (in section 5.2.1.1). However, it is still useful to analyse the average frequencies of each call in the top 20 since they are likely to differ. The top ten for the kernel and cuckoo data are shown in figures 5.9 and 5.10.

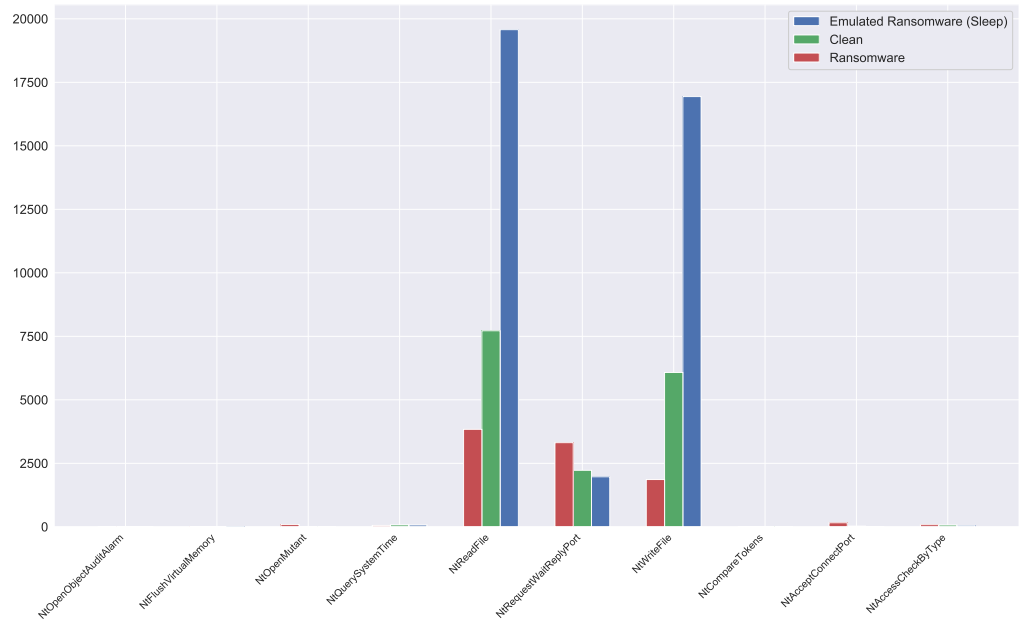

Figure 5.9: Frequencies (y-axis) of the top ten features (x-axis) of AdaBoost in order (from left to right) for malicious, clean, and emulated ransomware data (using Windows Sleep) from the kernel.

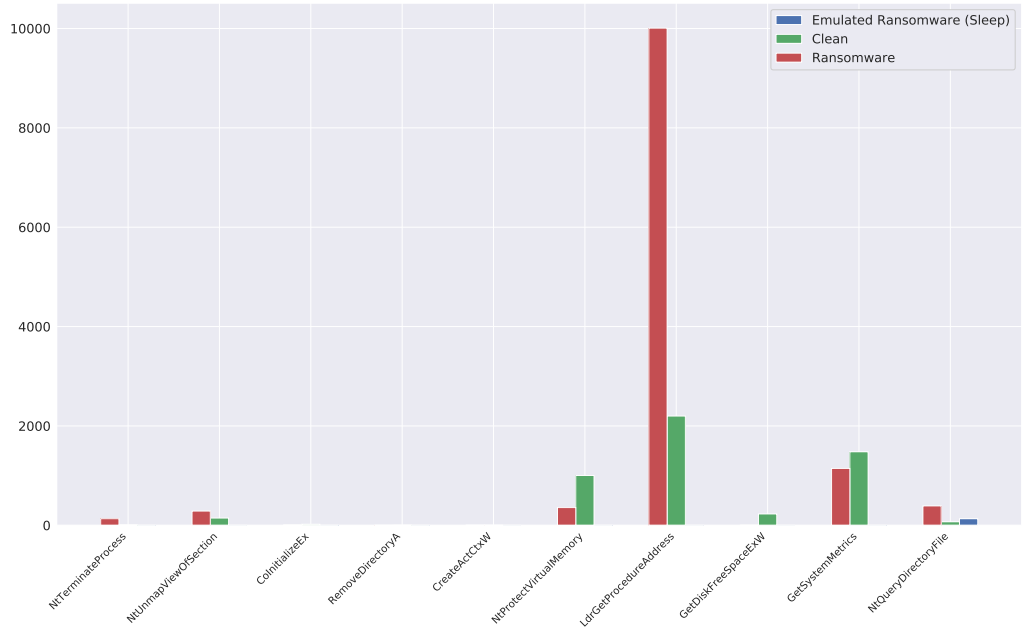

Figure 5.10: Frequencies (y-axis) of the top ten features (x-axis) of AdaBoost in order (from left to right) for malicious, clean, and emulated ransomware data (using Windows Sleep) from Cuckoo.

The distribution of calls in figure 5.9 is quite similar to that seen when the C time function was used. However, one difference is the amount of file-handling calls made. With the Windows Sleep function, the amount of file-handling calls made is considerably less. This is unusual given that the emulated ransomware using the Windows Sleep function should function in exactly the same manner as the emulated ransomware using the C time function. The only difference is in the method used to implement the time delay. After careful analysis, it became clear that the reason for this difference is within the time function itself. The C time function is documented to return the number of seconds since January 1, 1970, however, this is not guaranteed. It may return the number of milliseconds, for example. In addition, due to the fact that it is run on a virtual machine (that has just been restored from a snapshot) rather than directly on the hardware, it is even less likely to function as expected. Furthermore, the manner in which the delay is implemented with the time function is through the use of a while loop. This is not as efficient or accurate as the Sleep function. The reason being that the Sleep function suspends the running thread and interrupts the CPU when it requires attention again. Whereas the while loop is constantly executing on the CPU meaning that it's more likely to be affected by CPU load. The Sleep function, however, did call a hooked system call, so this would add some delay to the call being resolved which could have also impacted its accuracy. When testing an emulated ransomware sample with the C time function and giving it the same time delay value as an emulated ransomware sample using the Windows Sleep function, it was observed that occasionally they both had the same amount of file-related calls, but not always. The C time emulated ransomware sample occasionally made more file-related calls than the Windows Sleep emulated ransomware. This explains why the classifiers using data from the kernel driver did not detect the emulated ransomware with the Windows Sleep as effectively, since it did not make as many file-handling calls.

The frequencies for the top ten in the Cuckoo data in figure 5.10 are similar to that seen when the C time function was used. This is due to the fact that most of the functions in the top ten for Cuckoo are not used by the emulated ransomware. Therefore the next

ten features must be analysed to obtain a better understanding of the results.

Figures 5.11 and 5.12 show the next ten most influential features and the frequency with which they were called by malware, benignware, and the emulated ransomware using the Windows Sleep function.

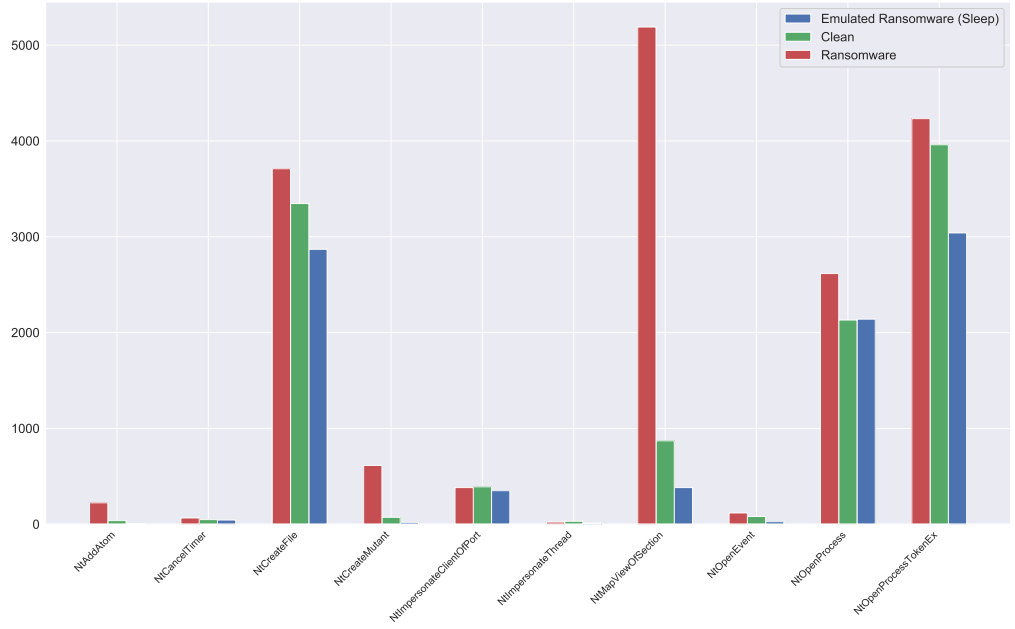

Figure 5.11: Frequencies (y-axis) of the features (x-axis) ranked 10-20 of AdaBoost in order (from left to right) for malicious, clean, and emulated ransomware data (using Windows Sleep) from the kernel.

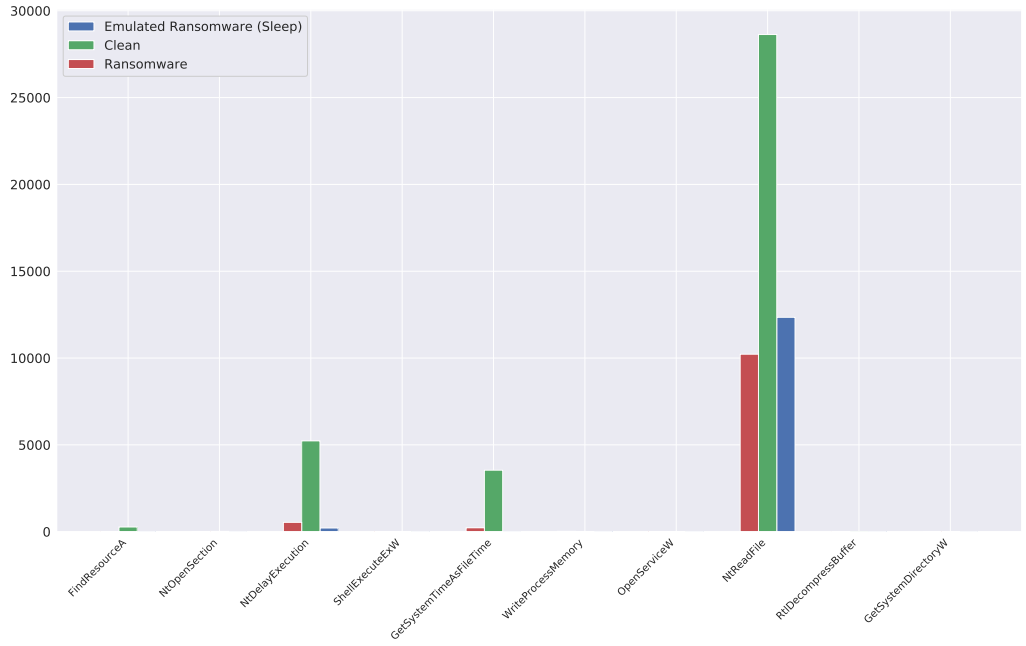

Figure 5.12: Frequencies (y-axis) of the features (x-axis) ranked 10-20 of AdaBoost in order (from left to right) for malicious, clean, and emulated ransomware data (using Windows Sleep) from Cuckoo.

Figure 5.11 again highlights the reduction in file-handling calls made by the emulated ransomware using the Windows Sleep function. The frequency of benignware is now closer to the frequency of emulated ransomware.

Figure 5.12 differs significantly from the emulated ransomware using C time (figure 5.4) with regards to the frequency with which the delay function is called. Obviously, the emulated ransomware using Windows Sleep does not call GetSystemTimeAsFileTime at all, rather, it calls NtDelayExecution (the function called by Sleep). Although, it calls it significantly less than benignware and ransomware on average. This is because the Sleep function only needs to be called once to suspend execution unlike the time function which is inside a while loop. Due to this, AdaBoost is able to correctly classify a significant portion of the emulated ransomware samples correctly. To understand what caused AdaBoost to incorrectly classify emulated ransomware samples, the classification results per sample are analysed.

5.2.2.2 Misclassified samples

The previous section showed how AdaBoost was able to detect the emulated ransomware samples. However, it is not clear why AdaBoost did not detect all the emulated samples. To understand this, the time delays of the samples that were correctly and incorrectly classified are plotted. This is shown in figures 5.13 and 5.14.

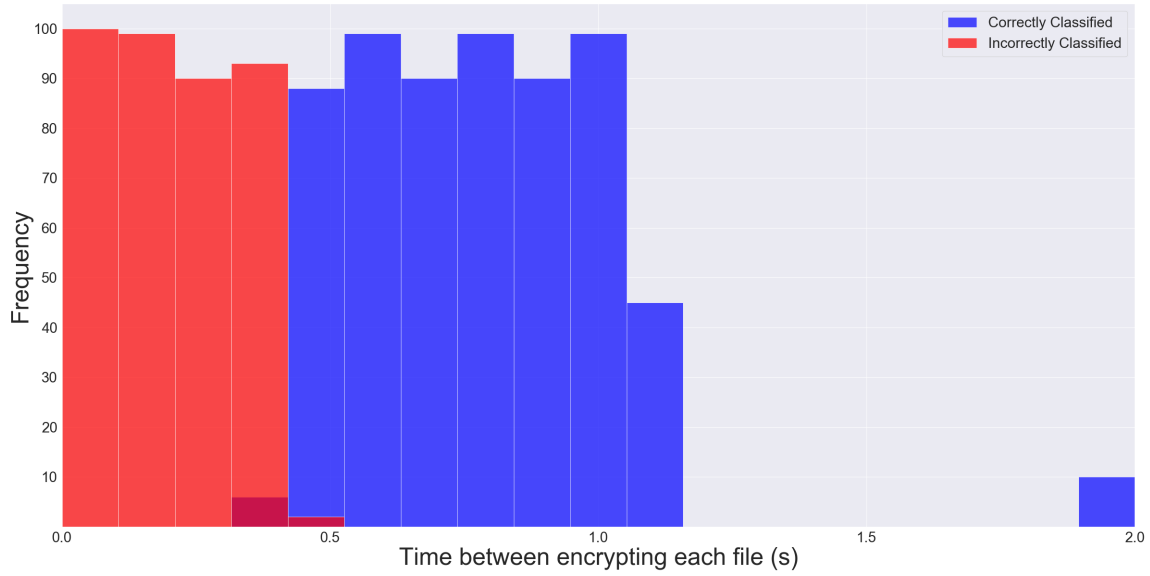

Figure 5.13: Results from AdaBoost classifying emulated ransomware samples using Windows Sleep function (with time between encryption <=2s) using the data from Cuckoo.

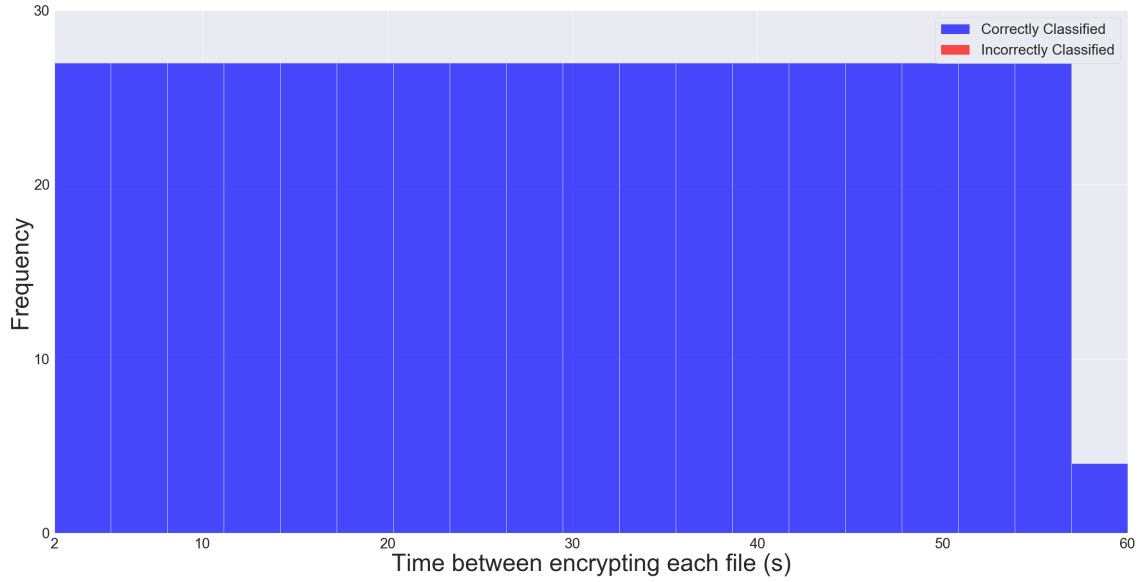

Figure 5.14: Results from AdaBoost classifying emulated ransomware samples using Windows Sleep function (with time between encryption >2s) using the data from Cuckoo.

From figure 5.13, it is immediately clear as to where AdaBoost misclassified samples. The samples that had a large proportion of file activity compared to evasive activity were not detected. Once the time delay between each encryption increased beyond half a second, AdaBoost was able to correctly classify the samples. In other words, as the evasive behaviour became more prominent in the Cuckoo data, the classification accuracy of AdaBoost improved. The results here are similar to those seen in figure 4.3 when Java was used to emulate ransomware. However, the difference here (figure 5.13) is that almost all samples with a delay below half a second went undetected, whereas in figure 4.3 some samples were detected.

5.2.2.3 Call category frequencies

To get an understanding of the data seen by the classifiers and how it differs to the data from the emulated ransomware using the C time function, the various categories of calls made by the average sample is visualised in the form of a pie chart for the kernel

and Cuckoo data in figures 5.15 and 5.16.

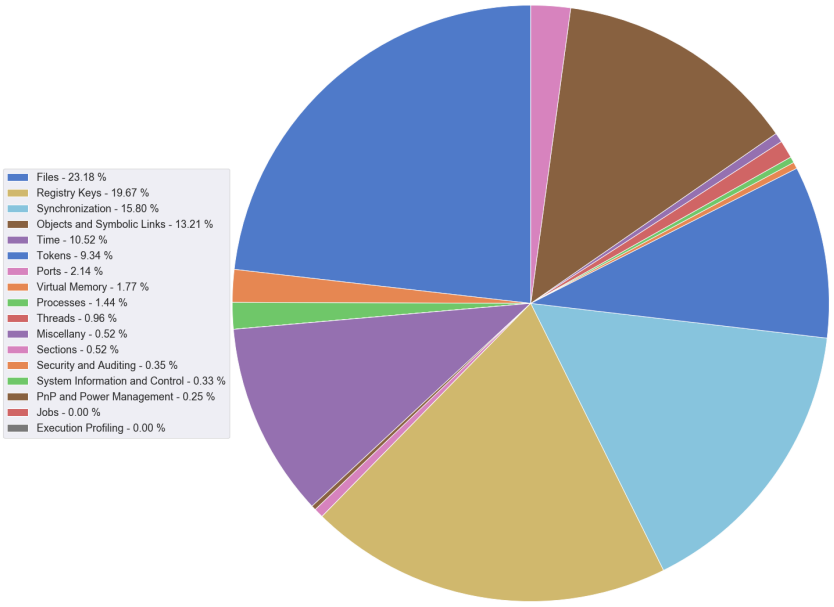

Figure 5.15: Distribution of call categories in data recorded by the kernel driver for emulated ransomware using the Windows Sleep function.

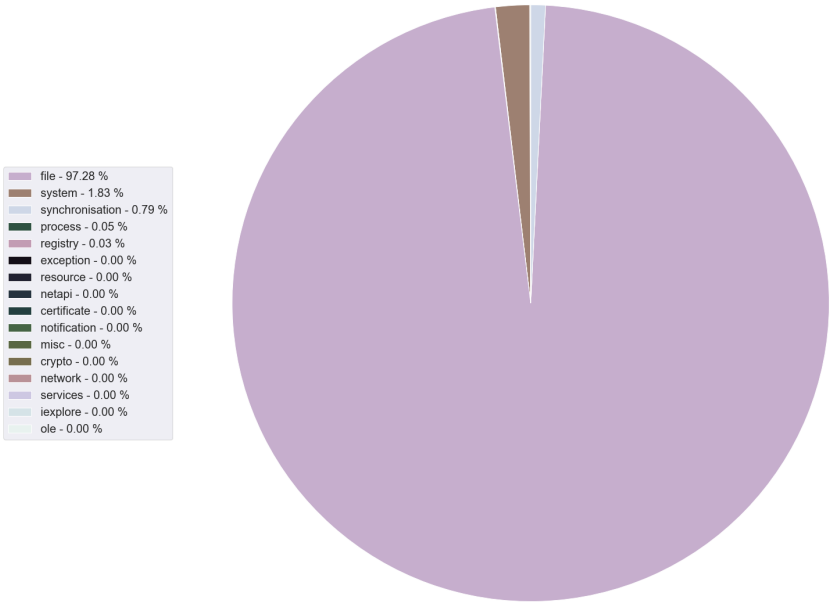

Figure 5.16: Distribution of call categories in data recorded by Cuckoo for emulated ransomware using the Windows Sleep function.

The distribution of the categories of calls in figure 5.15 for the kernel data shows a relatively even distribution (unlike figure 5.7). The files category is the largest in the pie chart, however, it does not show the same dominance as in figure 5.7 further showing why AdaBoost was unable to correctly classify the Windows Sleep emulated ransomware.

In the Cuckoo data, this time the dominant category is the file calls. Figure 5.16 shows file calls make up 97% of an emulated ransomware sample. This is to be expected since the call to Sleep does not need to be called as frequently as the C time call. In addition, in this case, all the calls in the synchronisation category came from NtDelayExecution.

5.2.2.4 Combined data results

Finally, as with the C time emulated ransomware, the data from Cuckoo and the kernel driver was combined. However, as before, it is not likely that the results will improve since in this case, only the Cuckoo data was able to obtain an acceptable level of accuracy. The results are shown in table 5.4.

Machine Learning Algorithm	Accuracy
AdaBoost	0.20
Decision Tree	9.76
Gradient Boost	1.2
Linear SVM	0.0
Nearest Neighbour	0.07
Random Forest	15.4

Table 5.4: Classification accuracy of emulated ransomware using Windows Sleep function from Cuckoo and Kernel driver.

As expected, the results are considerably worse when the data from Cuckoo and the kernel driver is combined. Therefore, this chapter has revealed that it is not always

appropriate to combine the Cuckoo data with the data from the kernel driver as it will not always lead to improved results.

Studying emulated ransomware using the Windows Sleep function as opposed to the C time function has revealed the lack of robustness within classifiers using system calls. The data from the kernel driver was found to be insufficient for detecting the emulated ransomware. While the data from Cuckoo was used to obtain an accuracy of 74.4%, this was largely helped by the Sleep function used. When emulated ransomware slept for very short time periods, and spent more time encrypting files, the Cuckoo data was not suitable for detecting the emulated ransomware.

5.3 Conclusion

This chapter set out to determine whether Java is a suitable medium to use to emulate malware and whether the classifiers trained on system calls are vulnerable to minor changes in calls used. It tested this by re-implementing the emulated ransomware used in chapter 4 in C. Two implementations were created that were identical except with regards to the function used to create the delay between encrypting each file. One implementation used the C standard time method to create the delay while the other used the Windows Sleep function to achieve it. The results showed that when the function used to implement the evasiveness/delay was the standard C time function, the emulated samples went completely undetected using the Cuckoo data. However, since the function commonly used by malware to implement evasiveness in that manner is the Windows Sleep function, one classifier using Cuckoo data was able to detect 74.4% of the samples. The only samples it did not detect were those where the amount of evasiveness was low relative to the file calls (or malicious behaviour). The results provided further evidence that the features being focused on within the Cuckoo data relate to the evasive behaviour of malware.

On the other hand, since the kernel data was focusing largely on the file activity, it

was not changes in the delay function that affected it. Rather, it was the changes that resulted from using a different delay function that affected it. When the emulated ransomware was using the Windows Sleep function, it did not display as much file activity as when the C time function was used. This is partially due to the manner in which the Windows Sleep is implemented. Therefore, when using the kernel data, AdaBoost was able to detect 83.1% of the emulated ransomware samples using the C time function.

Therefore, it can be concluded that when using system-call data from Cuckoo, the classifiers are particularly vulnerable to changes in the system calls used to achieve the same functionality. This is partly down to the low level of abstraction at which Cuckoo monitors programs. This is less of a problem in the kernel data where there are fewer redundant calls. A simple remedy for this would be to represent features that are practically identical as the same feature (for example, GetSystemDirectoryA and GetSystemDirectoryW).

Another remedy is to gather the data at both levels, so that each can offset the shortcomings of the other. However, unlike chapter 4, this chapter has found that it must not be blindly combined, or combined at all. Rather, each dataset should be consulted independently since combining the data can cause one to bring down the accuracy of the other.

Therefore this chapter has shown that while some of the general conclusions that were made when using Amsel in chapter 4 are similar to those made in this chapter, the interference from the JVM showed the classifiers to be much less vulnerable than they are. In addition, Java does not provide the flexibility and control required to test a classifier's vulnerability to small changes in system calls. Java is more suitable to test programs operating at higher levels of abstraction.

Chapter 6

Discussion

This thesis looked at some of the assumptions within the field of dynamic malware analysis to determine whether they needed rethinking. The findings indicate that the process requires some refining. Therefore this chapter proposes some general principles for dynamic malware analysis that have been extrapolated from the findings in the previous chapters. This is detailed in the final research question:

RQ7 *How can the dynamic malware analysis process be amended to prevent unintended security flaws from emerging?*

In answering this question, the final contribution of this thesis is provided:

C9 *The findings from this research are generalised to inform the general dynamic malware analysis process.*

In addition, this chapter discusses some of the limitations in this work as well as the future directions for this research.

6.1 General Principles for Dynamic Malware Analysis

This thesis has studied some of the basic assumptions in the dynamic malware analysis process and questioned the lack of theory behind them. In doing so, it has discovered

some inherent flaws in the traditional dynamic malware analysis pipeline. Therefore, to help future publications in this field, this section makes some recommendations to strengthen the traditional dynamic malware analysis process. Each recommendation is listed as a subsection.

6.1.1 Analyse the contributing features

When analysing the classification results from differentiating malware and benignware in chapter 3 (table 3.2) and chapter 4 (table 4.1 and 4.2), it would seem that the classifiers are extremely adept at detecting malware. However, from studying the contributing features (table 3.4, 3.5, 4.5 and 4.6) and subsequently the results from the experiments with emulated ransomware in chapter 4 and chapter 5 (table 4.1, 4.2, 5.1 and 5.3), it became clear that the classifiers are not as robust as the original results suggested. This realisation would not have occurred had the most influential features informing the classifiers not been analysed. Studying them highlighted the dependence that the classifiers had placed on evasive features. This also highlights the importance of having an understanding of the features being provided to the classifiers. For example, in knowing that GetSytemDirectoryA and GetSystemDirectoryW are essentially the same feature, the analyst can consider combining them to avoid overfitting. Without studying the features, a solution's robustness cannot be guaranteed.

6.1.2 Document the analysis tool used

Within dynamic malware analysis, the tool being used to capture the data can have a profound effect on the classification results. Chapter 3 illustrated this point by showing the differences in classification results just from using different hooking strategies. However, hooking strategies aside, tools can come with a variety of additional features that can be enabled. Using Cuckoo as an example, in addition to gathering system calls, it simulates user activity which includes clicking the mouse and pressing buttons on the

keyboard. In addition, it can be made to simulate network activity amongst many other things. Therefore, it's not enough to simply state the name of the tool being monitored, but, any parameter values that were modified must be listed to aid reproduce-ability. This is particularly important since much of the additional functionality mentioned can significantly affect classification results. For example, malware frequently looks for user activity before actually running. This includes mouse movements and recently created files [179]. Furthermore, if a relatively new and undocumented tool is being used, it's essential that its functionality and, in particular, call capturing methodology are carefully documented. Another important aspect that needs to be highlighted when describing the tool being used is the level at which it gathers calls. Tools can gather calls at a local process level (where they only monitor the process being investigated) or a global system level (where they monitor the entire system). Within the local level, however, tools could also monitor child processes created by the process under investigation. In addition, they may be concerned with monitoring a process that the process under investigation injected itself into. All of these can have a significant effect on the calls captured and therefore must be documented. The reason for such a strong emphasis on accurately documenting the methodology employed by the tool being used is due to the fact that its effect on the results is significant (refer to table 3.2 and 3.9). This transparency will also allow other researchers to faithfully reproduce the results.

6.1.3 Document the features used

In addition to analysing and discussing the influential features, chapter 4 showed the importance of carefully considering the categories of calls being monitored (refer to section 4.4.8). For the research in this thesis, the UI calls were ignored. Part of the reason for this is the obvious risk from overfitting that can occur. Since UI activity is non-essential, it is trivial to alter it to avoid detection. Section 4.4.8 analysed the performance of Cuckoo with and without the addition of UI calls and while it did

not change much for differentiating ransomware from benignware, when classifiers were trained on data with UI calls, the ability to detect emulated ransomware dropped considerably. Therefore, it is important to carefully select calls to monitor with the help of expert knowledge in the domain.

6.1.4 Document the hyper-parameters

Chapter 4 briefly showed the effects of hyper-parameters on classification results (section 4.4.3). Though brief, section 4.4.3 found that the values of the hyper-parameters could have a significant effect on the results, particularly when it came to detecting the emulated ransomware. The values of the hyper-parameters can also reveal whether the classifier is more likely to overfit the data. Therefore, the hyper-parameters values must be clearly documented to aid reproducibility.

6.1.5 Monitor at multiple levels of abstraction

The best classification results in chapter 3 and chapter 4 came from using system calls gathered at both a user- and kernel-level (sections 3.3.3 and 4.4.6). Conversely, in chapter 5 (sections 5.2.1.4 and 5.2.2.4) it was found that combining the data from both levels caused the results to drop significantly. However, while combining the data from different views can be dangerous, consulting the data from each independently can be very beneficial. Chapter 5 also found that classifiers were able to detect different types of emulated ransomware depending on what data was used, since each focused on different elements of the malware's behaviour. Therefore using an additional layer of security is likely to be beneficial. In addition to monitoring at user- and kernel-level, monitoring at a global as well as local level can be beneficial. Monitoring at a local level allows the analysis tool to observe the minutiae of the sample's behaviour, however, this could miss malicious activity that malware forces benignware to perform on its behalf. Therefore, monitoring at a global level can remedy this. Monitoring at

a global level also allows a tool to observe changes in the general behaviour of the system. In more general terms, when monitoring malware, it's important not to rely on only one data source since there is less chance of malware evading or compromising multiple data capturing sources.

6.1.6 Evaluate available hooking methodologies

The results from chapters 3, 4 and 5 all unequivocally show that the choice of hooking methodology affects the classification results obtained. Therefore, as alluded to in section 6.1.2, the hooking methodology used must be carefully considered. Aside from the general differences in data captured, a number of considerations must be made when selecting an appropriate hooking methodology. At the kernel-level, the three main distinctions between each hooking methodology is performance, scope and difficulty. Performance is very important at a kernel-level and poor performance can be exploited by malware to detect the presence of an analysis tool since it will affect the performance of the whole system [65]. Scope refers to the categories of system calls that are available to be intercepted by the hooking methodology chosen. Most of the hooking methodologies have the potential to intercept all system calls, but not all. Filter drivers for example, tend to observe events from specific categories (as mentioned in section 2.3.6). In addition, as some hooking methodologies have a greater impact on performance, the actual quantity of calls available to them is limited. Finally, another important consideration is the difficulty of implementing the hook. With some hooking methodologies (such as VMI), the implementation is complicated by the size of the semantic gap that needs to be bridged to create a working solution. This is not trivial as the location of the hook will determine whether the analysis tool has to decipher and interpret register values or simple data structures.

In terms of user-level hooks, there is little difference between the inline and IAT hooks, therefore, both are interchangeable. DBI, on the other hand, allows for an analyst to obtain detailed information regarding a sample's behaviour but unless that level of

information is required, it is not worth the performance impact that it brings.

As can be seen, there are a number of factors that must be considered when selecting a hooking methodology. Each of these factors can also have a small but noticeable affect on the data gathered and therefore must be carefully evaluated within the context of the tool they will be deployed in.

6.1.7 Conclusion

Much of the principles detailed in this section relate to transparency. The effects that each component of the dynamic malware analysis pipeline can have is often underestimated or not anticipated. Therefore, by providing more information on the matter, it makes it possible for both the authors and others to accurately verify the robustness of the results produced from the research.

6.2 Limitations & Future work

Despite the extensive analysis conducted in this thesis, it is not without its limitations. There are still many aspects within the dynamic malware analysis process that could not be studied. However, these understudied areas provide plenty of opportunities for future work.

The main limitation in this work relates to the platform chosen. While the work conducted in this thesis can be generalised to other platforms, it would have been better to have conducted the experiments in this thesis on each version of Windows that came after XP as well. The reason for choosing XP over the latest version of Windows is that, as explained previously, malware still targets older versions of Windows. Unsupported versions tend to be the low hanging fruit in any organisation. This was evidenced by the recent cyber-attack by WannaCry targeting NHS systems still using

Windows XP [64, 81]. Therefore, ideally, it would be more beneficial to use each version of Windows from XP onwards. This would show if the results differ on different versions of Windows. Unfortunately, due to the constraints of time, this could not be achieved. With each new version of Windows, the kernel is updated with additional calls to support new functionality. Therefore a modified kernel driver would be needed for each version. Furthermore, with the introduction of 64 bit versions of Windows came PatchGuard [229]. PatchGuard provides Kernel Patch Protection for Windows by triggering a blue screen of death as soon as it sees that structures within the kernel (such as the SSDT) have been modified. Though there are ways to get around PatchGuard [231, 228], and methods provided by Microsoft to intercept kernel activity, it would require a considerable development effort to create kernel drivers for both 32 and 64 bit versions of Windows.

Throughout the research in this thesis, the focus was on system calls made. For simplicity's sake, and due to the constraints of time, the arguments of each system call were not considered. In addition, gathering the arguments from every system call in the SSDT would have had a considerable effect on the system's performance. Cuckoo already provides this information, and the kernel driver could quite easily be modified to supply this information (since it is already embedded in each system call). To work around the performance issue, a reduced feature set using the results in chapter 3 in section 3.3.4 could be used to guide the process. This is a potential future work.

The localised kernel results in chapter 3 - section 3.3.2 only included calls made by the process under investigation and any child processes it created. However, malware is known to inject its code into a benign process and execute it from there. The localised kernel driver would not see any calls executed in this manner. This severely limits the amount of data that can be captured as evidenced in a report by Pao Alto Networks in 2013 that found injection techniques were being used in 13.5% of the samples they analysed [181]. Therefore, a potential future work could be to compare the localised kernel driver's results with a version of the kernel driver that is also gathering calls from

malware samples using process injection. The prevalence of process injection also brings into question the suitability of user-level hooks in dynamic malware analysis tools. Currently the only method by which user-level tools can combat injection is if the designers of the tools are aware of the injection method beforehand. However, process injection methods are on the rise. In 2019, Safebreach labs documented 26 techniques by which process injection can be achieved in Windows [144]. Therefore, it would also be useful to study which hooking methodology suffers more from malware employing unseen process injection techniques.

This thesis tested whether classifiers trained on malware and benignware could correctly classify malware that didn't look like the typical malware sample and found them lacking. However, it would also be useful to determine whether classifiers trained on malware from the previous decade, for example, are able to detect malware in the current decade. This is important since one of the claimed benefits of machine learning in malware analysis is that solutions will not need to be constantly updated since the rules learned will be generic enough to cover a wide range of malicious behaviours. Therefore, it would be beneficial to determine how the accuracy of the classifiers changes over time and how long the classifiers will be relevant.

Besides some of the basic defences against anti-analysis techniques provided by Cuckoo, no additional techniques were applied to defeat anti-analysis techniques within malware. Though there have been a number of solutions proposed in the literature to counter anti-analysis techniques [38, 143, 155], it was important in this research to follow the most commonly used technique in dynamic malware analysis. This is because, the aim of the thesis was to test flaws with the most commonly used technique which is why Cuckoo was chosen. In the future, it would be interesting to compare how the behaviours of malware learned by classifiers differ based on the environment that was used (environment with anti-anti-analysis vs environment without anti-anti-analysis).

The malware datasets used in this thesis consist solely of Portable Executable files. This is due to the fact that PE files are easier to run and determine the compatibility

requirements for. However, it would be interesting to determine whether malware distributed through other means (for e.g. Word documents, PDF files) produce similar results.

The dataset used in chapter 3 consisted of a random collection of malware. However, the results may have been different had an equal amount of malware been gathered for each category. The difficulty with doing this is that currently there is not a significant amount of agreement amongst Anti-Virus vendors regarding the definitions of categories or the category that the various malware samples belong to. The benefit of having an equal spread of malware from each category is that it would show how user and kernel level differ within each category. For example, rootkits tend to operate in kernel mode to avoid user mode detectors, therefore, it would have been interesting to determine whether rootkits are more likely to be detected by the kernel driver.

The research in this thesis compared one user-level hooking methodology against one kernel-level hooking methodology. However, as chapter 2 showed, there are many hooking methodologies within both categories. Therefore, it is not completely clear that all kernel hooking methodologies would outperform all user level hooking methodologies. As a result, the next potential avenue that could be explored is how each hooking methodology differs in the information it gathers and whether there is one particular method that stands above the others.

The research performed in chapter 4 and chapter 5 focused solely on ransomware, however, the study can certainly be extended to other types of malware (some of which may be more or less evasive than ransomware). This is important as the results may differ for different malware families. This would be a more holistic test for classifiers trained in the traditional dynamic malware analysis process.

In chapter 4, two of the hyper-parameters were found to have a significant effect on the classifiers' results on the test set. This highlighted the importance of carefully tuning the hyper-parameters. However, there are many hyper-parameters that were not studied that would could also have a significant impact on results. Therefore, in the future, this

would be an interesting avenue to explore.

The research in this thesis aimed to study the differences in different hooking methodologies rather than find the best machine learning method for dynamic malware analysis. Therefore, the feature representation method used was quite simplistic (frequency histograms). As mentioned in chapter 2, there are other feature representation methods whose impact on the results could be compared. In future, it would be interesting to compare the performance of each feature representation method when it comes to detecting emulated malware. It's possible that using a feature representation method at a higher level of abstraction will help classifiers to detect emulated malware and reduce their sensitivity to small changes in system calls.

Chapter 4 and chapter 5 studied the dependence of classifiers on evasive behaviour to detect malware. Only one evasive technique was studied to ensure that the reasons for the results would be easily interpretable. However, many more exist and they may effect classification results differently. Therefore, in future, experiments in chapter 5 will be repeated using different evasion tactics to study their effects on classification accuracy.

Chapter 4 and chapter 5 employed similar techniques to those used in adversarial learning, however, they do not fall directly under the banner of adversarial learning. This will be an important avenue to explore in future particularly with regards to whether kernel- or user-level data produces more vulnerable classifiers.

Chapter 7

Conclusion

In this chapter, the research conducted in this thesis is summarised and the main contributions are highlighted.

This thesis started by assessing the current state of the literature. This involved examining each of the tools used to conduct dynamic malware analysis to determine their novelty. The findings from this were highlighted in chapter 2 - Background. Chapter 2 found that the literature in the field of malware analysis was moving away from static analysis due to its inherent weakness to obfuscation and polymorphism, and moving towards dynamic analysis [171, 75]. Within dynamic analysis the general trend was to gather and use system calls to identify malware since all behaviours are ultimately expressed in system calls. System calls can be gathered using a number of hooking methodologies. These can be broadly separated into two categories; those that intercept calls at a user-level privilege, and those that intercept calls at a kernel-level privilege. A number of different hooking methodologies seemed to be in use within the literature and authors did not always justify why they chose the hooking methodology that they did. Furthermore, the literature was not clear on the advantages and disadvantages of each hooking methodology. Rather, the focus in dynamic analysis seemed to be on the machine learning method used to classify the data gathered as opposed to the method by which the data was gathered. In fact, there was not much agreement on the method by which system calls should be gathered for optimal results. In addition, there did not seem to be an industry standard tool that was recommended.

Table 2.1 shows the sheer volume of tools in use within the field of dynamic malware analysis and this excludes single-use tools (tools used only in the paper they are proposed in). The lack of consensus regarding the most appropriate hooking methodology is reflected in the overabundance of tools available. Part of the reason for there being so many tools is that as dynamic analysis experienced a surge in popularity, malware authors started adding more evasive behaviours to their samples to escape analysis. Therefore, new tools were proposed to defeat a new anti-analysis technique.

Table 2.1 did, however, show a growing consensus forming around Cuckoo Sandbox [111]. Although table 2.1 did not show the reason behind the consensus around Cuckoo. There are many reasons to prefer Cuckoo to other tools; it's free, open source, expandable, and easy to integrate with other tools. Though these are all valid reasons to choose Cuckoo over another tool, they do not say anything about its ability to gather reliable data regarding the behaviour of malware. The risk with using a tool that has not been carefully analysed and compared to other tools is that it may not be most optimal tool for the task. In particular, it could be missing essential information regarding the behaviour of malware. If this data is never gathered, regardless of how powerful the machine learning classifiers used are, a detection tool can only do so much.

This was the motivation behind chapter 3. Chapter 3 sets out to discover if the ability of classifiers to distinguish malicious from benign differs significantly if the classifiers are given data from different privilege levels in the OS. A collection of malware and benignware totalling 5000 samples was gathered. Each sample was analysed by a tool operating at user-level and a tool operating at kernel-level. The user-level tool used was Cuckoo since this tool had the largest consensus around it. Tools monitoring at a kernel-level were harder to source. Many of the respected tools appropriate for the task had been commercialised. None of the tools available were able to monitor the amount of system calls required. Therefore a kernel driver was written specifically for this thesis (available here: [176]).

The data gathered was then used to train and test commonly used machine learning

classifiers in the field. Of the classifiers, Random Forest produced the best performance distinguishing malicious from benign. It obtained an accuracy of 94.0% using the data from Cuckoo and 95.2% using the data from the kernel driver. This difference in results was found to be statistically significant, thereby showing that data from different privilege levels could not be used interchangeably.

To obtain an understanding into the reason behind the difference in the results from chapter 3, the features contributing the most towards the results for the best performing classifier were analysed separately for the data from Cuckoo and the kernel driver. This revealed that, when trained on data from Cuckoo, the main behavioural feature of malware that was being used by the classifiers to distinguish malicious from benign was the evasive behaviour of malware. In other words, the behavioural properties of malware that were being used by classifiers to identify it was the very behaviour that malware contained to evade detection. This observation was noted to a lesser degree within the kernel-level data. When using the kernel data, classifiers also used the differences in general behaviour to distinguish malware from benignware. Therefore, since the behavioural properties used to distinguish malicious from benign differed depending on the level that the data was gathered from, the data from the kernel driver and Cuckoo was combined. As expected, this caused the classification results to improve with the accuracy of Random Forest rising to 96.0%.

Given the behavioural properties of malware being used by the classifiers to distinguish it in chapter 3, chapter 4 sought to determine whether this presented a security risk. The question that chapter 4 attempted to answer is that if classifiers are identifying malware largely through their evasive behaviour, can they detect samples that only present malicious behaviour? To answer this question, classifiers were trained in the traditional dynamic malware analysis process as performed in chapter 3. Then, the trained classifiers were tested against emulated malware that contained varying degrees of evasive behaviour. The category of malware focused on in this chapter was ransomware. Ransomware was chosen due to its recent surge in popularity. In addition, the

malicious symptoms of ransomware are quite straightforward to emulate (since it is simply the encryption of files) and extensively documented [101, 140, 177].

The emulated malware being used as a test set was written in Java. It essentially encrypts a file and then waits for a specified period of time before encrypting another file. The file encryption represented the malicious behaviour while the wait represented the evasive behaviour (since evasive behaviour largely aims to stall the execution of malicious behaviour). To determine how vulnerable classifiers were to such samples, the rate of encryption of each emulated ransomware sample was varied. In total 1500 emulated ransomware samples were created.

The results from these experiments showed that while the classifiers had no trouble distinguishing real ransomware from benignware, they were not able to detect the emulated ransomware with the same confidence. Using the Cuckoo data, the highest accuracy detecting real ransomware came from Gradient Boost at 97.3%. On the other hand, the highest accuracy detecting the 1500 emulated ransomware samples using the Cuckoo data came from Decision Tree at 76.7%. Similarly, using the Kernel data, the highest accuracy detecting real ransomware (also from Gradient Boost) was 98.2% and the highest accuracy detecting the emulated ransomware (also from Decision Tree) was 67.0%. The accuracy values alone indicated that data gathered by Cuckoo was better suited to detecting the emulated ransomware than data gathered by the kernel driver. Analysing the influential features revealed some interference from the JVM that might have assisted with the classification results, particularly given the features ranked highly by classifiers using the Cuckoo data.

However, the real goal was not to simply compare accuracy values with regards to the emulated ransomware. The goal was to determine the file encryption rate at which classifiers can no longer detect the emulated ransomware. The results from this indicated that when using data from Cuckoo, Decision Tree was unable to detect emulated ransomware that frequently encrypted files with little time delay. In other words, as the evasiveness of the emulated ransomware sample increased, so did the detection ac-

curacy. When using the data from the kernel driver, the opposite was true. Decision Tree was able to detect emulated ransomware with high file encryption rates, but as the activity of the emulated sample dropped, it got lost amongst every other process being monitored by the kernel driver. As with chapter 3, combining the Cuckoo and kernel data produced the best results when detecting the emulated ransomware, obtaining an accuracy of 88.5%.

As the emulated ransomware used in chapter 4 was written in Java, a high-level language, there is not complete control over each system call made. Given that unintended activity from the emulated ransomware was observed in chapter 4 due to the JVM, chapter 5 verifies the integrity of the emulated ransomware results. In chapter 5, the emulated ransomware was recreated in C due to the level of control it provides. Two variants of the emulated ransomware were recreated in C, one used the delay function frequently seen in malware, NtDelayExecution. The other implemented the delay using the standard C time function. Apart from that, the experiments carried out were identical to the experiments in chapter 4.

The results in chapter 5 showed that when the C time function was used to implement the delay, the emulated ransomware was not detected by classifiers using the Cuckoo data at all. However, it was detected when the kernel data was used with an accuracy of 83.1% by AdaBoost. The only time it was incorrectly classified was when the average file activity of the emulated ransomware closely resembled that of real ransomware. When using NtDelayExecution, AdaBoost performed much better with the Cuckoo data, obtaining an accuracy of 74.4%. The only time AdaBoost was unable to detect the emulated ransomware was when it had a high rate of encryption. Using the kernel data, AdaBoost was unable to detect the emulated ransomware using NtDelayExecution. This seemed to be due to a reduction in average file activity caused by internal differences between NtDelayExecution and the C time function. Finally, unlike previous chapters, combining the user- and kernel-level data did not improve results in this case but caused them to drop.

The results from chapter 5 further confirm that classifiers trained on user-level data use evasive behaviour to identify malware. In addition, the huge swing in results after changing a single system call shows the lack of robustness in classifiers trained on system calls. Finally, while some of the conclusions concur with those made in chapter 4, the results from chapter 4 suggested that the classifiers were not as vulnerable as they are. In addition, many of the results were not as convincing and interpretable in chapter 4 due to noise from the JVM. Therefore, using Java to emulate malware for the purposes of testing detectors monitoring at a system call level is not appropriate.

Chapter 6 combined the findings from chapter 3, chapter 4 and chapter 5 into general principles to abide by when performing dynamic malware analysis. For example, with data collection, the recommendation is to gather data at multiple levels of abstraction that are consulted independently. This adds an extra layer of security and provides much better protection against malware with novel anti-analysis techniques. The remaining principles outlined aim to ensure that newly proposed detectors are robust to trivial attacks. They also ensure that the manner in which new solutions function is transparent to the rest of the research community, so that they can be peer-reviewed. This will also allow for a fair comparison to existing solutions.

To conclude, this thesis looked at some of the assumptions within the field of dynamic malware analysis to determine whether they needed rethinking. In particular, whether the data deserves as much attention as the method. The findings indicate that the data collection method requires much more study if the detectors produced are to be robust. This thesis has shown that a detector cannot be evaluated on its classification results alone. In a field as critical as security, the inner workings of the detector must also be evaluated before it can be ratified.

Bibliography

[1] APIMon - Home. https://apimon.codeplex.com/. visited on 2017-07-26.

[2] Cascade Description | F-Secure Labs. https://www.f-secure.com/v-descs/cascade.shtml. visited on 2019-08-30.

[3] Cryptlab Description | F-Secure Labs. https://www.f-secure.com/v-descs/mte.shtml. visited on 2019-08-30.

[4] Deviare API Hook | Nektra - Fast Custom Software Development Company. (visited on 2017-09-30).

[5] EasyHook. https://easyhook.github.io/. visited on 2017-07-26.

[6] FileHippo.com - Download Free Software. https://filehippo.com/. visited on 2019-06-07.

[7] IntellectualHeaven StraceNT - Strace For Windows. http://intellectualheaven.com/default.asp?BH=StraceNT. visited on 2017-07-26.

[8] Malpimp : Advanced API Tracing Tool. http://securityxploded.com/malpimp.php. visited on 2017-07-26.

[9] Microsoft Windows - 'nt!NtNotifyChangeDirectoryFile' Kernel Pool Memory Disclosure. https://www.exploit-db.com/exploits/42219/. visited on 2017-07-26.

[10] NtTrace. http://www.howzatt.demon.co.uk/NtTrace/. visited on 2017-07-26.

[11] Number of internet users worldwide 2005-2018 | Statista. https://www.statista.com/statistics/273018/number-of-internet-users-worldwide/. (visited on 2019-04-26).

[12] Oracle VM VirtualBox. https://www.virtualbox.org/. visited on 2017-11-28.

[13] PS-MPC Description | F-Secure Labs. https://www.f-secure.com/v-descs/ps-mpc.shtml. visited on 2019-08-30.

[14] SourceForge. https://sourceforge.net/. (visited on 2017-08-02).

[15] The Honeynet Project. http://old.honeynet.org/index.html. visited on 2017-07-26.

[16] Trojan:W32/Ransom. https://www.f-secure.com/v-descs/trojan_w32_ransom.shtml. visited on 2019-09-25.

[17] Vcl.716 description | f-secure labs. https://www.f-secure.com/v-descs/vcl.shtml. visited on 2019-08-30.

[18] VirusShare.com. https://virusshare.com/. visited on 2017-11-28.

[19] WinAPIOverride : Free Advanced API Monitor, spy or override API or exe internal functions. http://jacquelin.potier.free.fr/winapioverride32/index.php. visited on 2017-10-23.

[20] Static Analysis of Executables to Detect Malicious Patterns. *Critical Care Medicine*, 33(1):31–38, 2005.

[21] The molecular virology of Lexotan32: Metamorphism Illustrated, 2007.

[22] Data Execution Prevention. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc738483(v=ws.10), 2009.

[23] Anti-Debug NtQueryObject | ntquery. https://web.archive.org/web/20180118061815/https://ntquery.wordpress.com/2014/03/30/anti-debug-ntqueryobject/, 2014.

[24] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste. Malware Dynamic Analysis Evasion Techniques: A Survey. *ArXiv e-prints*, November 2018.

[25] Faraz Ahmed, Haider Hameed, M. Zubair Shafiq, and Muddassar Farooq. Using spatio-temporal information in api calls with machine learning algorithms for malware detection. In *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence*, AISec '09, pages 55–62, New York, NY, USA, 2009. ACM.

[26] M. A. M. Ali and M. A. Maarof. Dynamic innate immune system model for malware detection. In *2013 International Conference on IT Convergence and Security (ICITCS)*, pages 1–4, Dec 2013.

[27] APIMonitor.com. API Monitor — Spy and Display Win32 API Calls Made by Applications. http://www.apimonitor.com/. visited on 2017-07-28.

[28] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[29] Aubrey Allegretti. Cost of WannaCry cyber attack to the NHS revealed, 2018.

[30] AVTEST. The AV-TEST Security Report 2017/2018: The latest Analysis of the IT Threat Scenario. Technical report, 2018.

[31] AVTEST. Malware. Technical report, 2019.

[32] John Aycock. *Spyware and Adware*, volume 50. Springer Science & Business Media, 2010.

[33] John Aycock, Heather Crawford, and Rennie deGraaf. Spamulator: The internet on a laptop. *SIGCSE Bull.*, 40(3):142–147, June 2008.

[34] Matthew McDonald James Ong John Aycock, Heather Crawford, and Nathan Friess. A lightweight drive-by download simulator.

[35] Ludmila Babenko and Alexey Kirillov. Development of method for malware classification based on statistical methods and an extended set of system calls data. In *Proceedings of the 11th International Conference on Security of Information and Networks*, SIN '18, pages 8:1–8:6, New York, NY, USA, 2018. ACM.

[36] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[37] Yanko Baychev and Leyla Bilge. Spearphishing malware: Do we really know the unknown? In Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–66, Cham, 2018. Springer International Publishing.

[38] Ulrich Bayer. TTAnalyze: A Tool for Analyzing Malware. 2005.

[39] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A view on current malware behaviors. In *LEET*, 2009.

[40] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. Anubis: Analyzing unknown binaries, 2009.

[41] Th Bayes. An essay towards solving a problem in the doctrine of chances. 1763. *MD computing: computers in medical practice*, 8(3):157, 1991.

[42] A. Bazzi and Y. Onozato. Ids for detecting malicious non-executable files using dynamic analysis. In *2013 15th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 1–3, Sept 2013.

[43] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[44] Fabio Bellifemine, Federico Bergenti, Giovanni Caire, and Agostino Poggi. Jade—a java agent development framework. In *Multi-Agent Programming*, pages 125–147. Springer, 2005.

[45] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Márk Félegyházi. The cousins of stuxnet: Duqu, flame, and gauss. *Future Internet*, 4(4):971–1003, Nov 2012.

[46] Konstantin Berlin, David Slater, and Joshua Saxe. Malicious behavior detection using windows audit logs. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, AISec '15, pages 35–44, New York, NY, USA, 2015. ACM.

[47] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, and Filip Železný, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 387–402, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[48] Kristina Blokhin, Josh Saxe, and David Mentis. Malware similarity identification using call graph based system call subsequence features. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*, ICDCSW '13, pages 6–10, Washington, DC, USA, 2013. IEEE Computer Society.

[49] Bill Blunden. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones and Bartlett Publishers, Inc., USA, 2nd edition, 2012.

[50] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat*, 2012.

[51] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001.

[52] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[53] Tom Brosch and Maik Morgenstern. Runtime packers: The hidden problem. *Black Hat USA*, 2006.

[54] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[55] S Buehlmann and C Liebchen. Joebox: a secure sandbox application for windows to analyse the behaviour of malware, 2010.

[56] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. *CoRR*, abs/1309.0238, 2013.

[57] Alexei Bulazel and Bülent Yener. A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, ROOTS, pages 2:1–2:21, New York, NY, USA, 2017. ACM.

[58] Buster. Buster Sandbox Analyzer. `http://bsa.isoftware.nl/`. visited on 2017-07-26.

[59] Raymond Canzanese, Moshe Kam, and Spiros Mancoridis. Inoculation against malware infection using kernel-level software sensors. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 101–110, New York, NY, USA, 2011. ACM.

[60] Y. Cao, J. Liu, Q. Miao, and W. Li. Osiris: A malware behavior capturing system implemented at virtual machine monitor layer. In *2012 Eighth International Conference on Computational Intelligence and Security*, pages 534–538, Nov 2012.

[61] Lorenzo Cavallaro. Malicious software and its underground economy: Two sides to every story, 2013.

[62] Victor Chebyshev, Fedor Sinitsyn, Denis Parinov, Alexander Liskin, and Oleg Kupreev. IT threat evolution Q1 2018. Statistics. Technical report, Kaspersky Lab, 2018.

[63] Ping Chen, Lieven Desmet, and Christophe Huygens. A study on advanced persistent threats. In Bart De Decker and André Zúquete, editors, *Communications and Multimedia Security*, pages 63–72, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[64] Q. Chen and R. A. Bridges. Automated behavioral analysis of malware: A case study of wannacry ransomware. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 454–460, Dec 2017.

[65] Xu Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 177–186, June 2008.

[66] Zhi-Guo Chen, Ho-Seok Kang, Shang-Nan Yin, and Sung-Ryul Kim. Automatic ransomware detection and analysis based on dynamic api calls flow graph.

In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, RACS '17, pages 196–201, New York, NY, USA, 2017. ACM.

[67] Julia Yu-Chin Cheng, Tzung-Shian Tsai, and Chu-Sing Yang. An information retrieval approach for malware classification based on windows api calls. In *2013 International Conference on Machine Learning and Cybernetics*, volume 04, pages 1678–1683, July 2013.

[68] Eric Chien. Techniques of adware and spyware. In *the Proceedings of the Fifteenth Virus Bulletin Conference, Dublin Ireland*, volume 47, 2005.

[69] In Kyeom Cho and Eul Gyu Im. Extracting representative api patterns of malware families using multiple sequence alignments. In *Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems*, RACS, pages 308–313, New York, NY, USA, 2015. ACM.

[70] In Kyeom Cho, Tae Guen Kim, Yu Jin Shim, Minsoo Ryu, and Eul Gyu Im. Malware analysis and classification using sequence alignments. *Intelligent Automation & Soft Computing*, 22(3):371–377, 2016.

[71] J. Choi, Y. Han, S. Cho, H. Yoo, J. Woo, M. Park, Y. Song, and L. Chung. A static birthmark for ms windows applications using import address table. In *2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 129–134, July 2013.

[72] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S P'05)*, pages 32–46, May 2005.

[73] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. Shieldfs: a self-healing, ransomware-aware filesystem. In *Proceedings of the*

32nd Annual Conference on Computer Security Applications, pages 336–347. ACM, 2016.

[74] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, Sep 1995.

[75] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H. Austin, and Mark Stamp. A Comparison of Static, Dynamic, and Hybrid Analysis for Malware Detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, feb 2017.

[76] Marie Delacre, Daniël Lakens, and Christophe Leys. Why psychologists should by default use welch's t-test instead of student's t-test. *International Review of Social Psychology*, 30(1), 2017.

[77] Arshi Dhammi and Maninder Singh. Behavior analysis of malware using machine learning. In *Contemporary Computing (IC3), 2015 Eighth International Conference on*, pages 481–486. IEEE, 2015.

[78] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[79] A. Dolgikh, T. Nykodym, V. Skormin, J. Antonakos, and M. Baimukhamedov. Colored petri nets as the enabling technology in intrusion detection systems. In *2011 — MILCOM 2011 Military Communications Conference*, pages 1297–1301, Nov 2011.

[80] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, March 2008.

[81] Jesse M. Ehrenfeld. Wannacry, cybersecurity and health information technology: A time to act. *Journal of Medical Systems*, 41(7):104, May 2017.

[82] Don Marshall Nathan Brazan Eliot Graff, Aaron Hill. Driver verifier. `https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/driver-verifier`, 2017. visited on 2019-05-14.

[83] Rosenthal Engineering. Rosenthal Virus Simulator, 1991.

[84] T Faistenhammer, M Klöck, K Klotz, T Krüger, P Reinisch, and J Wagner. Virlab 2.1. *Geltendorf, Germany kklotz. de/html/virlab. html*, 1993.

[85] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5(6):29, 2011.

[86] C. Fan, H. Hsiao, C. Chou, and Y. Tseng. Malware detection systems based on api log data mining. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 3, pages 255–260, July 2015.

[87] Parvez Faruki, Vijay Laxmi, M. S. Gaur, and P. Vinod. Behavioural detection with api call-grams to identify malicious pe files. In *Proceedings of the First International Conference on Security of Internet of Things*, SecurIT '12, pages 85–91, New York, NY, USA, 2012. ACM.

[88] Peter Ferrie. The ultimate anti-debugging reference (may 2011), 2015.

[89] I. Firdausi, C. lim, A. Erwin, and A. S. Nugroho. Analysis of machine learning techniques used in behavior-based malware detection. In *2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies*, pages 201–203, Dec 2010.

[90] J. B. Fraley and M. Figueroa. Polymorphic malware detection using topological feature extraction with data mining. In *SoutheastCon 2016*, pages 1–7, March 2016.

[91] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, August 1997.

[92] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

[93] A. Fujino, J. Murakami, and T. Mori. Discovering similar malware samples using api call topics. In *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*, pages 140–147, Jan 2015.

[94] Y. Fukushima, A. Sakai, Y. Hori, and K. Sakurai. A behavior based malware detection scheme for avoiding false positive. In *2010 6th IEEE Workshop on Secure Network Protocols*, pages 79–84, Oct 2010.

[95] Hisham Shehata Galal, Yousef Bassyouni Mahdy, and Mohammed Ali Atiea. Behavior-based features model for malware detection. *Journal of Computer Virology and Hacking Techniques*, 12(2):59–67, May 2016.

[96] E. Gandotra, D. Bansal, and S. Sofat. Zero-day malware detection. In *2016 Sixth International Symposium on Embedded Computing and System Design (ISED)*, pages 171–175, Dec 2016.

[97] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Integrated framework for classification of malwares. In *Proceedings of the 7th International Conference on Security of Information and Networks*, SIN '14, pages 417:417–417:422, New York, NY, USA, 2014. ACM.

[98] Y. Gao, Z. Lu, and Y. Luo. Survey on malware anti-analysis. In *Fifth International Conference on Intelligent Control and Information Processing*, pages 270–275, Aug 2014.

[99] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, volume 3, pages 191–206, 2003.

[100] Maria Garnaeva, Fedor Sinitsyn, Yury Namestnikov, Denis Makrushin, and Alexander Liskin. Overall statistics for 2016. 2016.

[101] Alexandre Gazet. Comparative analysis of various ransomware virii. *Journal in Computer Virology*, 6(1):77–90, Feb 2010.

[102] Jan Goebel, Thorsten Holz, and Carsten Willems. Measurement and analysis of autonomous spreading malware in a university environment. In *Proceedings of the 4th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '07, pages 109–128, Berlin, Heidelberg, 2007. Springer-Verlag.

[103] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. *ArXiv e-prints*, December 2014.

[104] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[105] Sarah Gordon. Are good virus simulators still a bad idea? *Network Security*, 1996(9):7 – 13, 1996.

[106] Sarah Gordon and David Chess. Where there's smoke, there's mirrors: the truth about trojan horses on the internet. In *Virus Bulletin International Conference Proceedings*, 1998.

[107] James Gosling, David Colin Holmes, and Ken Arnold. The java programming language, 2005.

[108] Mariano Graziano, Davide Canali, Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. Needles in a haystack: Mining information from public dynamic analysis sandboxes for malware intelligence. In *24th USENIX Security Symposium*

(USENIX Security 15), pages 1057–1072, Washington, D.C., 2015. USENIX Association.

[109] André R. A. Grégio, Dario S. Fernandes Filho, Vitor M. Afonso, Rafael D. C. Santos, Mario Jino, and Paulo L. de Geus. Behavioral analysis of malicious code through network traffic and system call monitoring. volume 8059, pages 8059 – 8059 – 10, 2011.

[110] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security – ESORICS 2017*, pages 62–79, Cham, 2017. Springer International Publishing.

[111] Claudio Guarnieri, Allessandro Tanasi, Jurriaan Bremer, and Mark Schloesser. The cuckoo sandbox, 2012.

[112] Atli Guḥmundsson. 32-Bit Virus Threats on 64-Bit Windows. Technical report, Symantec.

[113] Sanchit Gupta, Harshit Sharma, and Sarvjeet Kaur. Malware characterization using windows api call sequences. In Claude Carlet, M. Anwar Hasan, and Vishal Saraswat, editors, *Security, Privacy, and Applied Cryptography Engineering*, pages 271–280, Cham, 2016. Springer International Publishing.

[114] O. Hachinyan. Detection of malicious software on based on multiple equations of api-calls sequences. In *2017 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 415–418, Feb 2017.

[115] S. S. Hansen, T. M. T. Larsen, M. Stevanovic, and J. M. Pedersen. An approach for detection and family classification of malware based on behavioral analysis. In *2016 International Conference on Computing, Networking and Communications (ICNC)*, pages 1–5, Feb 2016.

[116] SA Hex-Rays. Ida pro disassembler, 2008.

[117] Joe Hirst. Virus simulation suite. *British Computer Virus Research Centre, Brighton, United Kingdom*, 1990.

[118] Greg Hoglund and Jamie Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[119] Thorsten Holz, Moritz Steiner, Frederic Dahl, Ernst Biersack, Felix C Freiling, et al. Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm. *LEET*, 8(1):1–9, 2008.

[120] Sajad Homayoun, Ali Dehghantanha, Marzieh Ahmadzadeh, Sattar Hashemi, and Raouf Khayami. Know abnormal, find evil: Frequent pattern mining for ransomware threat hunting and intelligence. *IEEE Transactions on Emerging Topics in Computing*, 2017.

[121] S. Hsiao and F. Yu. Malware family characterization with recurrent neural network and ghsom using system calls. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 226–229, Dec 2018.

[122] W. Hu and Y. Tan. Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN. *ArXiv e-prints*, February 2017.

[123] Thomas Hungenberg and Matthias Eckert. Inetsim: Internet services simulation suite. *Internet*, 2014.

[124] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *3rd usenix windows nt symposium*, 1999.

[125] G. Hăjmăşan, A. Mondoc, and O. Creţ. Dynamic behavior evaluation for malware detection. In *2017 5th International Symposium on Digital Forensic and Security (ISDFS)*, pages 1–6, April 2017.

[126] M. Ijaz, M. H. Durad, and M. Ismail. Static and dynamic malware analysis using machine learning. In *2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pages 687–691, Jan 2019.

[127] Daisuke INOUE, Katsunari YOSHIOKA, Masashi ETO, Yuji HOSHIZAWA, and Koji NAKAO. Automated malware analysis system and its sandbox for revealing malware's internal and external activities. *IEICE Transactions on Information and Systems*, E92.D(5):945–954, 2009.

[128] Grégoire Jacob, Hervé Debar, and Eric Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 4(3):251–266, Aug 2008.

[129] S. Jamalpur, Y. S. Navya, P. Raja, G. Tagore, and G. R. K. Rao. Dynamic malware analysis using cuckoo sandbox. In *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*, pages 1056–1060, April 2018.

[130] Jae-wook Jang, Jiyoung Woo, Aziz Mohaisen, Jaesung Yun, and Huy Kang Kim. Mal-netminer: Malware classification approach based on social network analysis of system call graph. *CoRR*, abs/1606.01971, 2016.

[131] Sangmoon Jung and Yoojae Won. Ransomware detection method based on context-aware entropy analysis. *Soft Computing*, 22(20):6731–6740, Oct 2018.

[132] Arzu Gorgulu Kakisim, Mert Nar, Necmettin Carkaci, and Ibrahim Sogukpinar. Analysis and evaluation of dynamic feature-based malware detection methods. In Jean-Louis Lanet and Cristian Toma, editors, *Innovative Security Solutions for Information Technology and Communications*, pages 247–258, Cham, 2019. Springer International Publishing.

[133] BooJoong Kang, Suleiman Yerima, Kieran McLaughlin, and Sakir Sezer. Pagerank in malware categorization. In *Proceedings of the 2015 Conference on Re-*

search in Adaptive and Convergent Systems, RACS, pages 291–295, New York, NY, USA, 2015. ACM.

[134] M. Karresand. Separating trojan horses, viruses, and worms - a proposed taxonomy of software weapons. In *IEEE Systems, Man and Cybernetics SocietyInformation Assurance Workshop, 2003.*, pages 127–134, June 2003.

[135] Martin Karresand. A proposed taxonomy of software weapons, 2002.

[136] T. Kasama, K. Yoshioka, D. Inoue, and T. Matsumoto. Malware detection method by catching their random behavior in multiple executions. In *2012 IEEE/IPSJ 12th International Symposium on Applications and the Internet*, pages 262–266, July 2012.

[137] Masahiko Kato, Takumi Matsunami, Akira Kanaoka, Hiroshi Koide, and Eiji Okamoto. *Tracing Advanced Persistent Threats in Networked Systems*, pages 179–187. Springer International Publishing, Cham, 2013.

[138] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. UNVEIL: A large-scale, automated approach to detecting ransomware. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 757–772, Austin, TX, August 2016. USENIX Association.

[139] Amin Kharraz and Engin Kirda. Redemption: Real-time protection against ransomware at end-hosts. In Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 98–119, Cham, 2017. Springer International Publishing.

[140] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. Cutting the gordian knot: A look under the hood of ransomware attacks. In Magnus Almgren, Vincenzo Gulisano, and Federico Maggi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24, Cham, 2015. Springer International Publishing.

[141] Danny Kim, Amir Majlesi-Kupaei, Julien Roy, Kapil Anand, Khaled ElWazeer, Daniel Buettner, and Rajeev Barua. Dynodet: Detecting dynamic obfuscation in malware. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 97–118, Cham, 2017. Springer International Publishing.

[142] Dhilung Kirat and Giovanni Vigna. Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 769–780, New York, NY, USA, 2015. ACM.

[143] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: Efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 403–412, New York, NY, USA, 2011. ACM.

[144] Amit Klein and Itzik Kotler. Windows Process Injection in 2019. Technical report, Safebreach Labs, 2019.

[145] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 351–366, Berkeley, CA, USA, 2009. USENIX Association.

[146] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. The power of procrastination: Detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 285–296, New York, NY, USA, 2011. ACM.

[147] Bojan Kolosnjaji, Apostolis Zarras, Tamas Lengyel, George Webster, and Claudia Eckert. Adaptive semantics-aware malware classification. In *Proceedings of the 13th International Conference on Detection of Intrusions and*

Malware, and Vulnerability Assessment - Volume 9721, DIMVA 2016, pages 419–439, Berlin, Heidelberg, 2016. Springer-Verlag.

[148] Joxean Koret and Elias Bachaalany. *The Antivirus Hacker's Handbook*. John Wiley & Sons, 2015.

[149] Christopher Kruegel. Full system emulation: Achieving successful automated dynamic analysis of evasive malware. In *Proc. BlackHat USA Security Conference*, pages 1–7, 2014.

[150] Iltaek Kwon and Eul Gyu Im. Extracting the representative api call patterns of malware families using recurrent neural network. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, RACS '17, pages 202–207, New York, NY, USA, 2017. ACM.

[151] F. Leder, B. Steinbock, and P. Martini. Classification and detection of metamorphic malware using value set analysis. In *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 39–46, Oct 2009.

[152] Taejin Lee, Bomin Choi, Youngsang Shin, and Jin Kwak. Automatic malware mutant detection and group classification based on the n-gram and clustering coefficient. *The Journal of Supercomputing*, 74(8):3489–3503, Aug 2018.

[153] Taejin Lee and Jin Kwak. Effective and reliable malware group classification for a massive malware environment. *International Journal of Distributed Sensor Networks*, 12(5):4601847, 2016.

[154] John Leitch. Iat hooking revisited, 2011.

[155] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 386–395, New York, NY, USA, 2014. ACM.

[156] Rafał Leszczyna, Igor Nai Fovino, and Marcelo Masera. Simulating malware with malsim. *Journal in Computer Virology*, 6(1):65–75, Feb 2010.

[157] H. J. Li, C. Tien, C. Tien, C. Lin, H. Lee, and A. B. Jeng. Aos: An optimized sandbox method used in behavior-based malware detection. In *2011 International Conference on Machine Learning and Cybernetics*, volume 1, pages 404–409, July 2011.

[158] Zhuowei Li, XiaoFeng Wang, Zhenkai Liang, and M. K. Reiter. Agis: Towards automatic generation of infection signatures. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 237–246, June 2008.

[159] Michael Ligh, Steven Adair, Blake Hartstein, and Matthew Richard. *Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code*. Wiley Publishing, 2010.

[160] C. Lim and K. Ramli. Mal-one: A unified framework for fast and efficient malware detection. In *2014 2nd International Conference on Technology, Informatics, Management, Engineering Environment*, pages 1–6, Aug 2014.

[161] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of malicious code: Insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 349–358, New York, NY, USA, 2012. ACM.

[162] J. Liu, Y. Wang, and Y. Wang. The similarity analysis of malicious software. In *2016 IEEE First International Conference on Data Science in Cyberspace (DSC)*, pages 161–168, June 2016.

[163] S. Liu, H. Huang, and Y. Chen. A system call analysis method with mapreduce for malware detection. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 631–637, Dec 2011.

[164] Mischel Internet Security Ltd. Trojan Simulator. `http://www.testmypcsecurity.com/securitytests/trojan_simulator.html`. visited on 2019-09-11.

[165] Robert Luh, Helge Janicke, and Sebastian Schrittwieser. Aidis: Detecting and classifying anomalous behavior in ubiquitous kernel processes. *Computers & Security*, 84:120 – 147, 2019.

[166] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[167] Weiqin Ma, Pu Duan, Sanmin Liu, Guofei Gu, and Jyh-Charn Liu. Shadow attacks: automatically evading system-call-behavior based malware detection. *Journal in Computer Virology*, 8(1):1–13, May 2012.

[168] Melvin Earl Maron. Automatic indexing: an experimental inquiry. *Journal of the ACM (JACM)*, 8(3):404–417, 1961.

[169] Qiguang Miao, Jiachen Liu, Ying Cao, and Jianfeng Song. Malware detection using bilayer behavior abstraction and improved one-class support vector machines. *International Journal of Information Security*, 15(4):361–379, August 2016.

[170] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The Spread of the Sapphire/Slammer Worm. Technical report, CAIDA, ICSI, Silicon Defense, UC Berkeley EECS and UC San Diego CSE, Jan 2003.

[171] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430, Dec 2007.

[172] Vinod P. Nair, Harshit Jain, Yashwant K. Golecha, Manoj Singh Gaur, and Vijay Laxmi. Medusa: Metamorphic malware dynamic analysis usingsignature from api. In *Proceedings of the 3rd International Conference on Security of Information and Networks*, SIN '10, pages 263–269, New York, NY, USA, 2010. ACM.

[173] S. Naval, V. Laxmi, M. Rajarajan, M. S. Gaur, and M. Conti. Employing program semantics for malware detection. *IEEE Transactions on Information Forensics and Security*, 10(12):2591–2604, Dec 2015.

[174] Gary Nebbett. *Windows NT/2000 Native API Reference*. New Riders Publishing, Thousand Oaks, CA, USA, 2000.

[175] Matthew Nunes. Dynamic Malware Analysis kernel and user-level calls. `https://doi.org/10.5281/zenodo.1203289`, March 2018.

[176] Matthew Nunes. Matthewnunes/kernelssdtdriver: Kernel driver (with localisation). Feb 2018. `http://dx.doi.org/10.5281/zenodo.1169136`.

[177] Gavin O'Gorman and Geoff McDonald. *Ransomware: A growing menace*. Symantec Corporation, 2012.

[178] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Comput. Surv.*, 52(5):88:1–88:48, September 2019.

[179] Yoshihiro Oyama. Trends of anti-analysis operations of malwares observed in api call logs. *Journal of Computer Virology and Hacking Techniques*, 14(1):69–85, Feb 2018.

[180] Aurélien Palisse, Antoine Durand, Hélène Le Bouder, Colas Le Guernic, and Jean-Louis Lanet. Data aware defense (dad): Towards a generic and practical ransomware countermeasure. In Helger Lipmaa, Aikaterini Mitrokotsa, and

Raimundas Matulevičius, editors, *Secure IT Systems*, pages 192–208, Cham, 2017. Springer International Publishing.

[181] Palo Alto Networks. THE MODERN MALWARE REVIEW: Analysis of New and Evasive Malware in Live Enterprise Networks. Technical report, 2013.

[182] N. Papernot, P. McDaniel, and I. Goodfellow. Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples. *ArXiv e-prints*, May 2016.

[183] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 372–387, March 2016.

[184] Masarah Paquet-Clouston, Bernhard Haslhofer, and Benoit Dupont. Ransomware payments in the bitcoin ecosystem. *CoRR*, abs/1804.04080, 2018.

[185] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, pages 45:1–45:4, New York, NY, USA, 2010. ACM.

[186] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas. Malware classification with recurrent networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1916–1920, April 2015.

[187] Chinmaya Kumar Patanaik, Ferdous A. Barbhuiya, and Sukumar Nandi. Obfuscated malware detection using api call dependency. In *Proceedings of the First International Conference on Security of Internet of Things*, SecurIT '12, pages 185–193, New York, NY, USA, 2012. ACM.

[188] Gábor Pék and Levente Buttyán. Towards the automated detection of unknown malware on live systems. In *2014 IEEE International Conference on Communications (ICC)*, pages 847–852, June 2014.

[189] A. Pektaş, T. Acarman, Y. Falcone, and J. Fernandez. Runtime-behavior based malware classification using online machine learning. In *2015 World Congress on Internet Security (WorldCIS)*, pages 166–171, Oct 2015.

[190] Stephen Crane Philipp Reinecke. Amsel – The Abstract Malware Symptom Emulation Library. `https://github.com/AmselProject/amsel`. visited on 2019-09-11.

[191] Stephen Crane Philipp Reinecke. Malware modelling, July 2015. Patent no. WO2017020929A1.

[192] Stephen Crane Philipp Reinecke. Malware modelling, July 2015. Patent no. WO2017020928A1.

[193] Stephen Crane Philipp Reinecke. Malware modelling, July 2015. Patent no. WO2017020926A1.

[194] Geoffrey Phipps. Comparing observed bug and productivity rates for java and c++. *Software: Practice and Experience*, 29(4):345–358, 1999.

[195] Matt Pietrek. Inside windows-an in-depth look into the win32 portable executable file format. *MSDN magazine*, 17(2), 2002.

[196] R. S. Pirscoveanu, S. S. Hansen, T. M. T. Larsen, M. Stevanovic, J. M. Pedersen, and A. Czech. Analysis of malware behavior: Type classification using machine learning. In *2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, pages 1–7, June 2015.

[197] Michal Piskozub, Riccardo Spolaor, and Ivan Martinovic. Malalert: Detecting malware in large-scale network traffic using statistical features. *SIGMETRICS Perform. Eval. Rev.*, 46(3):151–154, January 2019.

[198] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. Measuring and defeating anti-instrumentation-equipped malware. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 73–96, Cham, 2017. Springer International Publishing.

[199] Y. Qiao, Y. Yang, L. Ji, and J. He. Analyzing malware by abstracting the frequent itemsets in api call sequences. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 265–270, July 2013.

[200] Yong Qiao, Jie He, Yuexiang Yang, Lin Ji, and Chuan Tang. A lightweight design of malware behavior representation. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 1607–1612. IEEE, 2013.

[201] Yong Qiao, Yuexiang Yang, Jie He, Chuan Tang, and Zhixue Liu. Cbm: Free, automatic malware analysis framework using api call sequences. In Fuchun Sun, Tianrui Li, and Hongbo Li, editors, *Knowledge Engineering and Management*, pages 225–236, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[202] Marco Ramilli, Matt Bishop, and Shining Sun. Multiprocess malware. In *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, MALWARE '11, pages 8–13, Washington, DC, USA, 2011. IEEE Computer Society.

[203] M. Rhode, L. Tuson, P. Burnap, and K. Jones. Lab to soc: Robust features for dynamic malware detection. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks – Industry Track*, pages 13–16, June 2019.

[204] Ronny Richardson and Max M North. Ransomware: Evolution, mitigation and prevention. *International Management Review*, 13(1):10, 2017.

[205] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 108–125, Berlin, Heidelberg, 2008. Springer-Verlag.

[206] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, December 2011.

[207] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Generic black-box end-to-end attack against state of the art api call based malware classifiers. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 490–510, Cham, 2018. Springer International Publishing.

[208] E. M. Rudd, A. Rozsa, M. Günther, and T. E. Boult. A survey of stealth malware attacks, mitigation measures, and steps toward autonomous open world solutions. *IEEE Communications Surveys Tutorials*, 19(2):1145–1172, Secondquarter 2017.

[209] Mark E. Russinovich. Process Monitor — Windows Sysinternals | Microsoft Docs. `https://docs.microsoft.com/en-gb/sysinternals/downloads/procmon`. visited on 2017-07-27.

[210] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals - Parts 1 and 2*. Microsoft Press, Redmond, WA, USA, 6th edition, 2012.

[211] Joanna Rutkowska and Alexander Tereshkin. Isgameover() anyone? *Black Hat, USA*, 2007.

[212] Graeme D. Ruxton. The unequal variance t-test is an underused alternative to student's t-test and the mann–whitney u test. *Behavioral Ecology*, 17(4):688–690, 2006.

[213] Moustafa Saleh, Tao Li, and Shouhuai Xu. Multi-context features for detecting malicious programs. *Journal of Computer Virology and Hacking Techniques*, 14(2):181–193, May 2018.

[214] Z. Salehi, M. Ghiasi, and A. Sami. A miner for malware detection based on api function calls and their arguments. In *The 16th CSI International Symposium on Artificial Intelligence and Signal Processing (AISP 2012)*, pages 563–568, May 2012.

[215] Zahra Salehi, Ashkan Sami, and Mahboobe Ghiasi. Using feature generation from api calls for malware detection. *Computer Fraud & Security*, 2014(9):9–18, 2014.

[216] Kevin Savage, Peter Coogan, and Hon Lau. The evolution of ransomware. *Symantec, Mountain View*, 2015.

[217] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin RB Butler. Cryptolock (and drop it): stopping ransomware attacks on user data. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pages 303–312. IEEE, 2016.

[218] Mike Schiffman. A brief history of malware obfuscation: Part 2 of 2. *Cisco Blog*, 2010.

[219] Michael D Schroeder and Jerome H Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3):157–170, 1972.

[220] S. Z. Mohd Shaid and M. A. Maarof. In memory detection of windows api call hooking technique. In *2015 International Conference on Computer, Communications, and Control Technology (I4CT)*, pages 294–298, April 2015.

[221] A. Shalaginov and K. Franke. Automated intelligent multinomial classification of malware species using dynamic behavioural analysis. In *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, pages 70–77, Dec 2016.

[222] Arushi Sharma, Ekta Gandotra, Divya Bansal, and Deepak Gupta. Malware capability assessment using fuzzy logic. *Cybernetics and Systems*, 50(4):323–338, 2019.

[223] Saiyed Kashif Shaukat and Vinay J Ribeiro. Ransomwall: A layered defense system against cryptographic ransomware attacks using machine learning. In *Communication Systems & Networks (COMSNETS), 2018 10th International Conference on*, pages 356–363. IEEE, 2018.

[224] Alisa Shevchenko. Virus Bulletin :: Advancing malware techniques 2008. Technical report, Virus Bulletin, 2009.

[225] Hao Shi, Jelena Mirkovic, and Abdulla Alwabel. Handling anti-virtual machine techniques in malicious software. *ACM Trans. Priv. Secur.*, 21(1):2:1–2:31, December 2017.

[226] S. L. Shiva Darshan and C. D. Jaidhar. Empirical study on features recommended by lsvc in classifying unknown windows malware. In Jagdish Chand Bansal, Kedar Nath Das, Atulya Nagar, Kusum Deep, and Akshay Kumar Ojha, editors, *Soft Computing for Problem Solving*, pages 577–590, Singapore, 2019. Springer Singapore.

[227] Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

[228] Skywing Skape. Bypassing patchguard on windows x64, 2005.

[229] Skywing Skape. A Brief Analysis of PatchGuard Version 3. *Uninformed*, 8(5), 2007.

[230] Skywing Skape. Dynamic binary instrumentation. *Uninformed.org*, 7, 2007.

[231] Skywing Skape. Subverting PatchGuard. *Uninformed*, 6(1):23, 2007.

[232] M. Smith, J. Ingram, C. Lamb, T. Draelos, J. Doak, J. Aimone, and C. James. Dynamic analysis of executables to detect and characterize malware. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 16–22, Dec 2018.

[233] A. Snihurov, O. Shulhin, and V. Balashov. Experimental studies of ransomware for developing cybersecurity measures. In *2018 International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S T)*, pages 691–695, Oct 2018.

[234] Norman Solutions. Norman sandbox whitepaper, 2003.

[235] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

[236] Abhinav Srivastava, Andrea Lanzi, Jonathon Giffin, and Davide Balzarotti. Operating system interface obfuscation and the revealing of hidden operations. In Thorsten Holz and Herbert Bos, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 214–233, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[237] Bo Sun, Akinori Fujino, and Tatsuya Mori. Poster: Toward automating the generation of malware analysis reports using the sandbox logs. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1814–1816, New York, NY, USA, 2016. ACM.

[238] M. Sun, M. Lin, M. Chang, C. Laih, and H. Lin. Malware virtualization-resistant behavior detection. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 912–917, Dec 2011.

[239] Symantec. Internet Security Threat Report. 21.

[240] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *ArXiv e-prints*, December 2013.

[241] Peter Szor. *The Art of Computer Virus Research and Defense: ART COMP VIRUS RES DEFENSE _p1*. Pearson Education, 2005.

[242] Yuki Takeuchi, Kazuya Sakai, and Satoshi Fukumoto. Detecting ransomware using support vector machines. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, ICPP '18, pages 1:1–1:6, New York, NY, USA, 2018. ACM.

[243] K. Thebeyanthan, M. Achsuthan, S. Ashok, P. Vaikunthan, A. N. Senaratne, and K. Y. Abeywardena. E-secure: An automated behavior based malware detection system for corporate e-mail traffic. In Kohei Arai, Supriya Kapoor, and Rahul Bhatia, editors, *Intelligent Computing*, pages 1056–1071, Cham, 2019. Springer International Publishing.

[244] R. Tian, R. Islam, L. Batten, and S. Versteeg. Differentiating malware from cleanware using behavioural analysis. In *2010 5th International Conference on Malicious and Unwanted Software*, pages 23–30, Oct 2010.

[245] Shun Tobiyama, Yukiko Yamaguchi, Hajime Shimada, Tomonori Ikuse, and Takeshi Yagi. Malware detection with deep neural network using process behavior. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, volume 2, pages 577–582. IEEE, 2016.

[246] Virus Total. Virustotal-free online virus, malware and url scanner. *Online: https://www. virustotal. com/en*, 2012.

[247] T. Tungjitviboonkun and V. Suttichaya. Complexity reduction on api call sequence alignment using unique api word sequence. In *2017 21st International Computer Science and Engineering Conference (ICSEC)*, pages 1–5, Nov 2017.

[248] Tzu-Yen Wang, Shi-Jinn Horng, Ming-Yang Su, Chin-Hsiung Wu, Peng-Chu Wang, and Wei-Zen Su. A surveillance spyware detection system based on data mining methods. In *2006 IEEE International Conference on Evolutionary Computation*, pages 3236–3241, July 2006.

[249] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. Rambo: Run-time packer analysis with multiple branch observation. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 186–206, Cham, 2016. Springer International Publishing.

[250] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005.

[251] D. Uppal, R. Sinha, V. Mehra, and V. Jain. Malware detection and classification based on extraction of api sequences. In *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 2337–2342, Sept 2014.

[252] Amit Vasudevan and Ramesh Yerraballi. Stealth breakpoints. In *Proceedings of the 21st Annual Computer Security Applications Conference*, ACSAC '05, pages 381–392, Washington, DC, USA, 2005. IEEE Computer Society.

[253] Amit Vasudevan and Ramesh Yerraballi. Spike: Engineering malware analysis tools using unobtrusive binary-instrumentation. In *Proceedings of the 29th Australasian Computer Science Conference - Volume 48*, ACSC '06, pages 311–320, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[254] Swapna Vemparala, Fabio Di Troia, Visaggio Aaron Corrado, Thomas H. Austin, and Mark Stamo. Malware detection using dynamic birthmarks. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, IWSPA '16, pages 41–46, New York, NY, USA, 2016. ACM.

[255] Bill Venners. *The java virtual machine*. McGraw-Hill, New York, 1998.

[256] Peter Viscarola and W. Anthony Mason. *Windows NT Device Driver Development*. New Riders Publishing, Thousand Oaks, CA, USA, 1st edition, 1998.

[257] Xun Wang, Wei Yu, A. Champion, Xinwen Fu, and Dong Xuan. Detecting worms via mining dynamic program execution. In *2007 Third International Conference on Security and Privacy in Communications Networks and the Workshops - SecureComm 2007*, pages 412–421, Sept 2007.

[258] B. L. Welch. The generalization of 'student's' problem when several different population variances are involved. *Biometrika*, 34(1/2):28–35, 1947.

[259] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security Privacy*, 5(2):32–39, March 2007.

[260] Tobias Wüchner, Martín Ochoa, and Alexander Pretschner. Robust and effective malware detection through quantitative data flow graph metrics. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9148*, DIMVA 2015, pages 98–118, Berlin, Heidelberg, 2015. Springer-Verlag.

[261] T. Wüchner, M. Ochoa, E. Lovat, and A. Pretschner. Generating behavior-based malware detection models with genetic programming. In *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, pages 506–511, Dec 2016.

[262] Y. Xu, H. Koide, D. V. Vargas, and K. Sakurai. Tracing mirai malware in networked system. In *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 534–538, Nov 2018.

[263] Zhaoyan Xu, Lingfeng Chen, Guofei Gu, and Christopher Kruegel. Peerpress: Utilizing enemies' p2p strength against them. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 581–592, New York, NY, USA, 2012. ACM.

[264] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2e: Combining hardware virtualization and softwareemulation for transparent and extensible malware analysis. *SIGPLAN Not.*, 47(7):227–238, March 2012.

[265] Yuan Yang, Zhongmin Cai, Weixuan Mao, and Zhihai Yang. Identifying intrusion infections via probabilistic inference on bayesian network. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9148*, DIMVA 2015, pages 307–326, Berlin, Heidelberg, 2015. Springer-Verlag.

[266] Yanfang Ye, Tao Li, Donald Adjeroh, and S. Sitharama Iyengar. A survey on malware detection using data mining techniques. *ACM Comput. Surv.*, 50(3):41:1–41:40, June 2017.

[267] Heng Yin and Dawn Song. Temu: Binary code analysis via whole-system layered annotative execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley, Jan 2010.

[268] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[269] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pages 297–300, Nov 2010.

[270] X. Yuan, P. He, Q. Zhu, and X. Li. Adversarial Examples: Attacks and Defenses for Deep Learning. *ArXiv e-prints*, December 2017.

[271] F. Zhang and Y. Ma. Using irp with a novel artificial immune algorithm for windows malicious executables detection. In *2016 International Conference on Progress in Informatics and Computing (PIC)*, pages 610–616, Dec 2016.

[272] Hanqi Zhang, Xi Xiao, Francesco Mercaldo, Shiguang Ni, Fabio Martinelli, and Arun Kumar Sangaiah. Classification of ransomware families with machine learning based on n-gram of opcodes. *Future Generation Computer Systems*, 90:211 – 221, 2019.

[273] J. Zhang, C. Gao, L. Gong, Z. Gu, D. Man, W. Yang, and X. Du. Malware detection based on dynamic multi-feature using ensemble learning at hypervisor. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, Dec 2018.

[274] Y. Zhang, C. Rong, Q. Huang, Y. Wu, Z. Yang, and J. Jiang. Based on multi-features and clustering ensemble method for automatic malware categorization. In *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 73–82, Aug 2017.

[275] J. Zhao, S. Zhang, B. Liu, and B. Cui. Malware detection using machine learning based on the combination of dynamic and static features. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6, July 2018.

GNU Free Documentation License

Version 1.2, November 2002

0. Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. Applicability and Definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of

this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools

are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. Copying in Quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the

back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. Modifications

you may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties — for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. Aggregation with Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. Future Revisions of this License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © `YEAR YOUR NAME`. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free

> Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with…Texts." line with this:

> with the Invariant Sections being `LIST THEIR TITLES`, with the Front-Cover Texts being `LIST`, and with the Back-Cover Texts being `LIST`.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.