Journal of Information Security and Applications 67 (2022) 103202

Contents lists available at ScienceDirect

Journal of Information Security and Applications

journal homepage: www.elsevier.com/locate/jisa

Bane or Boon: Measuring the effect of evasive malware on system call classifiers

Matthew Nunes *, Pete Burnap, Philipp Reinecke, Kaelon Lloyd

Cardiff University School of Computer Science, Queens Building, 5 The Parade, Cardiff, CF24 3AA, South Glamorgan, United Kingdom

ARTICLE INFO

Dataset link: https://doi.org/10.17035/d.2022.0176862059

Keywords:
Dynamic malware analysis
Machine learning
Evasive malware
Ransomware
System call analysis

ABSTRACT

Malware refers to software that is designed to achieve a malicious purpose usually to benefit its creator. To accomplish this, malware hides its true purpose from its target and malware analysts until it has established a foothold on the victim's machine. Malware analysts, therefore, have to find increasingly sophisticated methods to detect malware prompting malware authors to increase the number of evasive techniques employed by their malware. Dynamic malware analysis has been framed as a potential solution as it runs malware in its preferred environment to ensure that it observes its true behaviour. However, it is usually a restricted form of the preferred environment and malware may only be run for two minutes or less. This means that if malware does not demonstrate its malicious intent within that time frame and environment, the behaviour observed and subsequently learned may not be the behaviour that needs to be prevented. There is a risk that classifiers trained using the standard dynamic malware analysis process will only recognise malware by its evasive behaviour rather than a mix of behaviours. In this paper, we study the extent to which classifiers are dependent on evasive behaviour when identifying malware. We achieve this by training them on real ransomware and benignware and then testing their ability to detect carefully crafted simulated ransomware. The simulated ransomware gives us the freedom to create samples with different levels of evasive and malicious behaviour. The simulated samples, like the real samples, are run in a sandboxed environment where data is collected at a user- and Kernel-level. The results of our experiments indicated that, in general, the classifiers were more likely to label the simulated samples as malicious once the amount of evasive behaviour present in a sample went beyond a threshold. Generally, this threshold was crossed when the simulated ransomware waited 2 s or more between each file it encrypted. Additionally, the classifiers trained on the user-level data were not as robust against small changes in system calls made. Whereas, when trained on system calls gathered at a Kernel, system-wide level, the classifiers' results were less variable. Finally, in attempting to simulate malware for our experiments, we discovered that the field of malware simulation is relatively unstudied despite its potential and therefore provide recommendations for simulating malware for system-call analysis.

1. Introduction

Malware analysis refers to the study of malicious software (malware) to understand its behaviour and identifying traits. There are two techniques that can be used to analyse malware — static analysis and dynamic analysis. In static analysis, the binary (malicious file) in question is studied without ever running it. Therefore much of the analysis involves examining the binary's code in order to understand its behaviour. Dynamic analysis (also known as behavioural analysis) refers to actually running the binary in its preferred environment and observing it for any signs of malicious behaviour. Dynamic analysis is favoured over static analysis since it is not as easily hindered by evasion tactics such as obfuscation [1,2] and polymorphism [3]. Studies have also shown that Dynamic analysis is more effective for detecting malware [4,5]. While it is also possible to combine features from static and dynamic analysis to perform "Hybrid Analysis", it tends to achieve a similar performance to dynamic analysis [4].

A popular method for detecting malware in dynamic analysis is to collect all the calls made by the binary to the Operating System (OS) (sometimes referred to as system calls or API calls) [6]. This is because any program that wants to do anything noteworthy on the system needs to interact with the OS. Therefore, by capturing the system calls, it is possible to have a detailed understanding of the behaviour of the binary.

Once system calls have been gathered, the patterns within them can 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

* Corresponding author.
E-mail address: nunesma@cardiff.ac.uk (M. Nunes).

https://doi.org/10.1016/j.jisa.2022.103202

Available online 18 May 2022

infeasible. To obtain complete coverage over all malware samples produced, an expert would have to analyse each new malware sample within 1.6 s [7]. Therefore, rather than manually extracting patterns to identify malware from system calls, the process can be automated using Machine Learning (ML). ML refers to the process by which a machine automatically learns how to perform a task (such as distinguishing malware from benignware). Classification is the ML process in which an 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 could be API calls made and the output from the classifier is the label, 'malicious' or 'benign'. The trained classifier can then be tested against system calls gathered from running completely unseen samples. This technique has produced promising results [6,8].

However, dynamic analysis is not without weaknesses. Malware uses the fundamental properties of dynamic malware analysis to mislead analysts. When malware is run, it will first seek to identify the environment it is running in before exhibiting malicious behaviour. If malware detects that it is being analysed, it will not display its true malicious behaviour and instead show benign behaviour. One method by which malware detects it is being analysed is by looking for evidence that it is running in a virtualised or emulated environment [9–11]. Virtualised or emulated environments are preferred when conducting dynamic malware analysis since, among other things, it is trivial to restore a virtual machine to a previously saved clean state after running malware. Furthermore, these environments provide an analyst with a number of options when it comes to instrumenting a machine to glean information regarding a sample's behaviour. Unfortunately, these environments cannot perfectly recreate a real environment. Malware can look for these inconsistencies to determine the type of environment it is being executed in. If malware suspects that it is running in a virtual environment, it might assume it is being analysed and choose not to run. In response, analysis environments are sometimes instrumented to trick malware into thinking it is running in a real environment forcing malware authors to find increasingly sophisticated methods to evade analysis.

Such evasive behaviours have become so prevalent in malware that Chen et al. [11] proposed a protection mechanism against malware that added artefacts to normal environments to make them appear to be analysis environments. This discouraged malware from executing in these environments and thereby essentially protected them. A study of 4 million malware samples found that 72% of the samples contained techniques to detect that they were being run in a VM [12]. Previous work [13,14] also observed the prevalence of anti-vm/anti-debug properties in malware and has been the inspiration for this work. Yet, the majority of the existing literature on the development of ML to detect malware uses data collected from virtualised or emulated environments. This poses the question of how sensitive ML malware classifiers are to evasive malware? If the majority of samples are actually refraining from demonstrating malicious behaviour, then how much of their behaviour are the ML models actually being trained with? In practical terms, are these models actually learning to detect malicious behaviour — or are they actually learning how to detect evasive behaviour? For real world applications, this presents the risk that when malware is running on "real" systems rather than VMs, they will exhibit different behaviours that the ML model will not be able to pick up.

The concern over ML models being evaded is already growing in the literature 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 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.

Adversarial attacks have also been used in the context of malware classification to make previously detectable samples go undetected [15–17]. However, this paper takes a different approach to adversarial attacks. Rather than attempting to produce adversarial samples, we focus on questioning the limitations of ML models built in constrained testing environments. In this case, the evasion is not deliberately crafted to confuse an ML model — rather it exists as a function of malware detecting it is in a virtualised or emulated environment, and not exhibiting the same behaviour as it would on a real system.

In this paper we hypothesise that ML classifiers trained using data from dynamic malware analysis conducted within virtualised environments (as in much of the literature) are largely recognising malware by its evasive behaviour, rather than malicious behaviour. While there is nothing inherently wrong with classifiers using evasive traits to identify malware, an over-emphasis on these traits could be problematic and our aim is to understand the amount of evasive behaviour a sample needs to possess to be classified as malicious. To test our hypothesis we train a selection of state of the art classifiers in this field on data collected from running 2500 ransomware samples and 2500 benign samples in a sandboxed Windows environment. We collect data at both a user-level and kernel-level to determine if one is better suited to creating more robust classifiers. We then test the trained classifiers against ransomware containing adjustable levels of evasive properties. To obtain ransomware samples with changeable levels of evasive behaviour, we create simulated ransomware samples using a number of tools and test the trained classifiers against data collected from running them. The simulated ransomware simulates the malicious behaviour of ransomware, that is, file encryption as well as an evasive technique. The evasive behaviour chosen is idleness. This behaviour is chosen since it is a platform independent evasive technique that is highly effective and easy to implement [18]. It also means that the rate at which encryption occurs can easily be altered by increasing or decreasing the sleep between each successive encryption which allows us to determine the level of evasiveness at which ransomware is detected or undetected. We use four sets of 1500 simulated ransomware samples, one set is written in Java and the other three are written in C. Each of the sets all accomplish the same tasks (encrypting and sleeping), they only differ in how this is implemented in the source. This allows to verify the results but also determine if the manner in which the malware implements an evasive or malicious technique affects the classifier's ability to recognise it. This allows us to determine the level of evasiveness at which ransomware is detected or undetected. This will not only help us to understand the effect of evasive malware on ML classifiers produced from dynamic malware analysis, but also shine further light on the extent of the problem with evasive malware. In summary, the contributions made in this paper are the following:

1. We determine the extent to which evasive features within malware contribute to a classifier's ability to recognise malware analysed within dynamic malware analysis.
2. We evaluate how the detection capability of classifiers trained on Kernel-level (privileged) data compare with classifiers trained on user-level data with regards to the level of evasion present in malware and the manner in which it is implemented.
3. We analyse the results produced by the classifiers to understand the actual system calls that influenced decisions, and explain what these calls are doing.
4. We evaluate the effectiveness of high-level languages such as Java when it comes to simulating malware to use within dynamic malware analysis and show the potential benefits from employing simulated malware when evaluating classifiers.

2. Related work

2.1. Evading detection

Our contributions do not fall into a single category within the literature, therefore we have reviewed the work in each related category. The two general categories that our work falls into relate to

methods designed to evade detection, and the simulation of malware. With regards to evading detection within dynamic malware analysis, there are two categories that the various techniques fall into. The first category consists of those techniques that attempt to evade the data capturing component of the malware analysis tool to render the data captured regarding the malware's behaviour inaccurate. The second category consists of the methods that focus on deceiving the machine learning classifier otherwise known as adversarial learning. On the other hand, malware simulation is an underdeveloped field consisting of very few working solutions.

2.1.1. Evading data capture

Methods attempting to evade the data capturing component focus on identifying hallmarks of the dynamic malware analysis process. These include looking for properties specifically present or absent from analysis environments as compared to real environments [9–11,19–21]. For example, the Storm Worm is able to detect that it is running in VMware by querying the I/O communication port for a magic number not found on real systems [11,22]. Malware can also look outside the system for indicators that it is in an analysis environment. The Wannacry ransomware checks for a valid internet connection before executing [23]. This works because it is quite common for an analysis environment to contain a simulated to no internet connection to prevent malware from spreading unintentionally. This is just a small proportion of the number of techniques available to malware to evade detection. There are many more techniques used to evade the data capturing component that are extensively documented in the literature [10–12,18,24–26]. As a result there are also many techniques that are used to detect evasive behaviours [27–31] and even counter them [32,33].

In addition to these methods, a few methods have been proposed in the literature that take advantage of the way calls are gathered during dynamic malware analysis. Ramilli et al. [34] 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, but together, these processes could cooperate to achieve the malicious outcome. They analysed the divided malicious sample using 43 different anti-viruses and seven dynamic analysis tools (including Anubis [35], JoeBox [36], and Norman Sandbox [37]) and found that it evaded detection in every case. Ma et al. [38] automate the theory of the technique employed by Ramilli et al. [34] 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. The tool then added the required communication code between the samples created. The resulting malware produced by their tool was tested on CWSandbox [39] and Norman Sandbox [37] and succeeded in evading analysis. Srivastava et al. [40] 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.

2.1.2. Evading classifiers

The second category of evasive techniques is focused on evading the machine learning classifiers. This is commonly referred to as adversarial attacks. Adversarial attacks first emerged in the field of image recognition, where the alterations made to images were so slight that there was no human-observable difference between the original and altered images. Despite that, state of the art classifiers incorrectly classified them with extremely high confidence [41]. To perform an adversarial attack, the decision boundary of the classifier being attacked must first be determined. Once that is known, the attacker can tactically determine the minimum number of features of the input sample to alter so that it falls outside the decision boundary despite no change in the sample's behaviour. Adversarial attacks can be classed as 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 [42,43]. 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 [42]. These attacks have been quite successful as adversarial samples have been found to be transferable between classifiers trained to make the same decision [44]. The success of adversarial attacks in general is attributed to the linear behaviour of some classifiers in high dimensions [45].

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, for example, it would immediately end execution when called thereby significantly altering the malicious sample's behaviour. Despite these limitations, there is still a significant amount of literature within the field of adversarial learning.

Biggio et al. [15] perform 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'. 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.

Grosse et al. [16] attack classifiers trained on a well-known dataset, the DREBIN dataset [46]. After training a neural network to obtain the current state-of-the-art performance on the dataset, adversarial samples are crafted by adding features, that, when modified, produce the most change in the classifier's output. These are identified using the method employed by Papernot et al. [47] 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.

Hu and Tan [17] propose MalGAN, an adversarial neural network, which takes malware samples and produces adversarial samples that can evade classifiers. They differ in that they perform a black box attack, assuming that access to the machine learning classifier's internals are not available. They tested MalGAN on a number of classifiers and manage to alter malware samples such that the accuracy of many of the classifiers fell from within 90% to 0%.

It is clear that an attacker has plenty of options to choose from when adding evasive techniques to their malware. However, now that it is commonplace for malware to contain evasive behaviour, it is not clear what effect this is having on the data collection within dynamic malware analysis. In order to get a better understanding of that, we assess classifiers' ability to detect malware with varying levels of evasiveness. To obtain such malware, we had to use malware simulators.

2.2. Simulating malicious behaviour

When choosing a malware simulator, there are very few modern solutions available. The Rosenthal Virus Simulator [48] was the first solution proposed. It is capable of producing harmless programs that contain virus signatures. Trojan Simulator [49] goes slightly further, simulating a property of malware that ensures it is run every time the machine is powered on. However, as with previous solutions, it simulates no malicious symptoms. More recently, MalSim [50] was proposed. MalSim, written using the Java Agent DEvelopment Framework (JADE) [51], 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. The solution chosen in this research is Amsel [52], an open-source, Java-based malware simulator. Amsel is capable of doing actual harm to the system and is easily extensible allowing users to 'plug-in' additional behaviours that are not provided that they want to simulate. This makes it a good fit for our needs.

The literature surrounding malware simulators shows that Java [53] is a common choice as a programming language [50,52,54,55]. Java is a general purpose programming language that is platform independent. Every Java application runs inside the Java Virtual Machine (JVM) (with a new instance created for each application) [56]. Java and the JVM allow the programmer to focus 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 [57], there are questions around its suitability to simulate 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 (simulated 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 using Java to create a malware simulator.

2.3. Conclusion

The number of options available to malware when it comes to evading detection are clearly innumerable. It is also clear that there are high percentages of malware using evasive techniques [11,12,18]. Dynamic analysis of such malware is largely conducted using data obtained from Cuckoo Sandbox [13]. Therefore, we intend to determine how the current state of malware and the dynamic malware analysis process is affecting the way classifiers recognise malware. We intend to achieve this by training classifiers on real ransomware and benignware and then observe how their performance changes when detecting simulated ransomware with different levels of evasiveness. If classifiers use solely evasive traits to recognise ransomware then the simulated ransomware showing largely malicious activity (or activity in general) will be labelled as benign and vice versa. In the next section we detail our experimental method.

3. Method

3.1. Malware selection

The class of malware that we focus on for this paper is ransomware. Ransomware is a class of malware that prevents a user from accessing a core component on their machine and demands a payment, or ransom, for the release of that component. Our focus in this paper is on crypto ransomware. Crypto ransomware encrypts the files typically in a user's home directory and demands a payment from the user in exchange for the decryption key. Ransomware is still remarkably popular as evidenced in the recent report from Sophos which found that over a third of the organisations they surveyed were hit by a ransomware attack in 2020 [58]. We chose ransomware as it is relatively straightforward to simulate its main malicious symptom since it is consistent and visible. Coupled with this, ransomware provides the benefit of containing a continuous stealth-efficiency trade-off which is easily parametrised by altering the rate of encryption. This is particularly important for this study.

3.2. Experimental setup

To begin with, we trained classifiers on real ransomware and benignware as is commonly the case when creating a tool to detect malware [59,60]. To do this, we collected 2500 ransomware samples from VirusShare [61] and 2500 benign files from SourceForge [62] and FileHippo [63]. The ransomware we used is from a dedicated dataset provided by VirusShare. Most of the samples were written between (and including) 2008 and 2016. The collection of the benign files is described in more detail in [13]. Each sample (benign and malicious) was then run for two minutes as recommended by Willems et al. [39] and Küchler et al. [64] in a virtual machine with Windows XP SP3 installed. We chose to use Windows XP due to the relative ease with which it can be instrumented thanks to the large amount of documentation (official or otherwise) available. We do not feel this affects the validity of our results or their applicability to newer, 64-bit versions of Windows since currently, all 64 bit systems are backwards compatible with 32 bit binaries [65]. Additionally, the most commonly prevailing malware samples in the wild are 32 bit [66]. To increase the probability of each sample running, real files (such as documents, presentations, images, and videos etc.) were placed in the user's home directory to make the environment seem more realistic. For each sample, the system calls it made whilst running were extracted at both user-level and Kernel-level. To gather system calls at user-level, we used Cuckoo Sandbox [67]. To gather calls at Kernel-level we used our own custom-built Kernel driver that hooks the System Service Descriptor Table (SSDT). The implementation details of our driver are provided in [13].

After gathering the system calls made by a sample, they were represented as a frequency vector that showed how many times each system call was called as is commonly the case in malware literature [68–72], and ransomware literature [73,74]. These vectors were then used to train and test the classifiers. We used a number of popular classifiers to observe how the performance varied with each classifier. The classifiers we used were: AdaBoost, Decision Tree, Gradient Boosting, Linear SVM, Nearest Neighbours, and Random Forest. The reason for choosing AdaBoost, Gradient Boosting, and Random Forest was that ensemble methods obtained impressive performance in the literature on ransomware [75–77]. Similarly, Linear SVMs and Decision Trees are also widely used [75–79]. Nearest Neighbours was chosen for its simplicity as a baseline.

Initially, we performed 10-fold-cross-validation on the data collected from the ransomware and benignware samples to determine how effectively the classifiers are able to detect real ransomware. The measures we use to assess a classifier's performance are: AUC, Accuracy, F-measure, and Precision. These are well known measures, however, to understand them in this context, it is important to define

a few basic terms. We interpret True Positives (TP) as ransomware samples that are correctly labelled by the classifier as ransomware. False Positives (FP) are benign samples that are incorrectly labelled as ransomware. True Negatives (TN) are benign samples that are correctly classified as benign. False Negatives (FN) are ransomware samples that are incorrectly classified as benign. Regarding the actual measures used, AUC relates to ROC curves. ROC curves plot True Positive Rate (TPR) against False Positive Rate (FPR). Accuracy refers to the correct predictions divided by all the predictions. Precision is defined by the formula $\frac{TP}{TP + FP}$ and Recall is defined by the following formula $\frac{TP}{TP + FN}$.

3.3. Malware simulator experiments

After assessing the classifiers' performance in detecting real ransomware, we test the trained classifiers' ability to detect simulated ransomware. In our experiments, the simulated ransomware is treated similarly to an unseen test set. This means that rather then using 10-fold cross-validation to obtain results, we train the classifiers on all the real ransomware and benignware and then make the classifiers predict the class of the simulated ransomware. Unlike traditional test sets, our test set only contains simulated ransomware and no benignware. It is important to note that this is not the equivalent of creating a test set comprised solely of ransomware since there are some important differences between a real dataset and simulated dataset. Our goal when constructing the simulated dataset is to determine where classifiers draw the line between malicious and benign. Therefore the samples in the simulated dataset vary with regards to the amount of malicious symptoms they display. While some samples may be overtly malicious, others behave more ambiguously. Therefore, the simulated set is not equivalent to a test set simply containing ransomware but much more complex and better viewed as a different type of dataset. When assessing the classifiers' performance on the simulated set, the only summary statistic we present is the percentage of simulated samples that are classified as malicious, which we usually refer to as 'Accuracy' for brevity. However, we are more interested in how different levels of evasion affect a classifier's result and therefore devote more space to analysing the results for each simulated ransomware sample. This provides a much clearer (and more nuanced) picture of each classifier's performance than the summary statistic. We use two categories of simulated ransomware, the first type is produced by an existing tool written in Java. The second type we implemented ourselves in C.

3.3.1. Java-based malware simulator experiments

Amsel [52] 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 simulate. Potential symptoms range from 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 the use of Amsel can be found at [80–82].

The functionality required of Amsel for our experiments is relatively simplistic to reduce the possibility of the classifiers being biased by additional behaviours. The only symptom we have used is the file encryption symptom since that is the main malicious symptom of ransomware that solutions want 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 an XOR operation which works by XOR-ing the byte stream of each file with a secret key. There are several parameters within the symptom's settings that can be altered; however, for our experiments, only one parameter is altered between each run, the interarrival time. This parameter determines how long the simulated malware sample should wait between encrypting two successive files in the directory specified. The interarrival time was gradually incremented from 1.0×10^{-6} to just below 60 s. The size of the increment was 0.01 initially. After reaching 1 s, the size of the increment was increased to 2 s. 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 it is the one parameter that can navigate the trade-off between malicious and evasive behaviour.

When the time between encrypting each file, a.k.a the interarrival time, is set to a lower value, the simulated ransomware is encrypting more frequently and thereby exhibiting its malicious symptom much more frequently. Whereas when the interarrival time is at higher values, the simulated malware is idle for longer 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 s), 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 achieves the goal of a wide variety of evasive behaviours, which is to stall or halt execution. Another reason we chose idleness as our evasive behaviour is that it is quite pervasive in malware and platform independent [18]. It is also difficult to detect and compatible with all dynamic analysis techniques [18]. Examples of malware within the real world employing evasive techniques of this nature includes Duqu [83], Kelihos [84], Trojan DelfInj [85], Rombertik [86] and Carbanak [87]. In practical terms, one example of a method by which malware can achieve idleness is through using the system call NtDelayExecution to delay executing its payload [22,88] as seen in Trojan Nap [89].

To ensure that the results obtained are not due to chance, each unique simulated sample (unique with regards to its interarrival time) is replicated ten times. This provides more than enough samples per unique time value to ensure that the results are consistent rather than accidental. It is also not so high that the experiments become infeasible due to the constraints of time. Finally, as with the traditional dataset, each simulated 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 histograms. Since the goal of these experiments is to evaluate how well classifiers trained on real ransomware and benignware can detect malicious symptoms, the simulated 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 simulated ransomware. Since some of the classifiers used have a random element (such as Decision Trees), each classifier is trained and tested 1000 times on the same dataset and the mean accuracy is reported. The overall experimental process described so far is summarised in Fig. A.13.

3.3.2. C-based malware simulator experiments

We repeat the experiments performed using Amsel, but re-implement the functionality of Amsel in C to determine the integrity of the results. The only difference here is that the simulated 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. It allows us to specifically choose the system calls made.

Using C we create three different types of simulated ransomware samples. The first two types are exact replicas of the simulated ransomware written in Java. They differ from one another in that they use different methods to implement the delay between each file encryption. The third type of simulated ransomware differs from the simulated ransomware written in Java in that it uses a much more robust encryption methodology.

Much of the implementation for the first two types of simulated ransomware is quite straightforward, as the programs simply encrypt files in a specified directory and wait a specified amount of time between encrypting each file. However, to test the robustness of the trained classifiers, and avoid a bias, the delay between encrypting each file was implemented in two different ways. The first set of simulated ransomware used the `time` function provided by C. This function returns the number of seconds since January 1, 1970. The source code for this is shown in listing 1.

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

Listing 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. As a result, this is considered to be a 'busy wait'.

The second set of simulated 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 as shown in listing 2.

```
1 Sleep(millisecondsToWait);
```

Listing 2: Delay implemented using a recommended Windows method

The use of NtDelayExecution by malware is already well documented [3,90]. However, GetSystemTimeAsFileTime has not been encountered as much [12], although it is the standard method in C for achieving the functionality required. 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 simulated ransomware.

The third set of simulated ransomware written in C uses much more complex encryption provided by the functions in the Windows API as opposed to XOR encryption. This is to determine the significance attached to the encryption method by the classifiers in identifying ransomware. The source code for this simulated ransomware is much more complex, however, it is very similar to that shown in [91]. The encryption algorithm used is the same as that used by CryptoLocker since it is the most common ransomware family in our dataset and the world [92]. The exact algorithms used are RSA and AES 256.

The experiments carried out in this section are identical to those carried out in the previous section. As with Amsel, 1500 simulated ransomware samples using the C time function were generated and 1500 simulated ransomware samples using the Windows Sleep function were generated with the same spread of time delays as Amsel. Likewise, 1500 simulated ransomware samples using more complex encryption were also generated. The classifiers were trained on the real ransomware and benignware and then separately tested on each group of simulated ransomware. These experiments were conducted for the Cuckoo and Kernel data. The breakdown of the dataset used within the paper is shown in Table 1.

Table 1
Complete dataset breakdown.

Dataset type	Dataset name	Number of samples
Real	Benignware	2500
	Ransomware	2500
Simulated	Amsel (Java-based)	1500
	C-time Ransomware	1500
	Windows Sleep Ransomware	1500
	Windows Encryption Ransomware	1500

3.4. Feature ranking

To determine the features contributing the most to the results, we ranked them by importance 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 in [93] to assign a value to each feature. The same is true for Random Forest, AdaBoost and Gradient Boost since they are composed of a multitude of Decision Trees. The only difference being that as they are composed of multiple Decision Trees, the Gini importance is averaged over each tree. 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, we do not include it in this measure. To understand how it values features, we use an independent feature ranking approach. In this method, all the data from a single feature is fed to the classifier and its ability to differentiate ransomware from benignware using only that feature is recorded. This is done separately for every feature after which the features are ranked according to their classification scores. Both feature ranking methods are discussed in much more detail in Nunes et al. [13].

3.5. Misclassified samples

An important aspect of the experimental process is to determine the rate of encryption at which simulated ransomware goes undetected and to ascertain if that value differs depending on whether Cuckoo or the Kernel data is used. We calculated this using the prediction results that were obtained from the classifying the simulated ransomware. Since the classifiers were tested against all simulated ransomware samples 1000 times, the mean prediction value for each simulated 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, 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). Once the incorrectly classified samples have been found, they can be linked to their interarrival time (i.e. time between encrypting each file). This will allow us to understand the amount of evasive behaviour a sample must possess to be labelled as malicious.

4. Results

In the following subsections, we first analyse the ability of classifiers to differentiate real ransomware from benignware. After training the classifiers on real ransomware and benignware, we also test them against the various types of simulated ransomware. We perform these experiments both for the data from Cuckoo and the Kernel driver to determine whether that affects the results.

Table 2
Classification results of real ransomware and benignware using **Cuckoo data**. The last column contains the accuracy from classifying the data obtained when running Amsel after training the classifiers on all the real ransomware and benignware.

Machine learning algorithm	Ransomware					Amsel
	AUC	Accuracy (%)	Precision	Recall	F-measure	Accuracy (%)
AdaBoost	0.992	96.4	0.963	0.968	0.965	72.1
Decision Tree	0.952	95.2	0.951	0.958	0.954	56.3
Gradient Boost	**0.996**	**97.3**	0.969	**0.980**	0.974	**100**
Linear SVM	0.850	75.3	0.753	0.826	0.788	2.80
Nearest Neighbour	0.972	91.5	0.939	0.896	0.917	1.07
Random Forest	0.994	96.7	**0.977**	0.977	**0.977**	62.9

Table 3
Classification results of real ransomware and benignware using data from the **Kernel Driver**. The last column contains the accuracy from classifying the data obtained when running Amsel after training the classifiers on all the real ransomware and benignware.

Machine learning algorithm	Ransomware					Amsel
	AUC	Accuracy (%)	Precision	Recall	F-measure	Accuracy (%)
AdaBoost	0.992	95.8	0.958	0.963	0.960	29.0
Decision Tree	0.957	95.6	0.957	0.960	0.958	**97.4**
Gradient Boost	**0.997**	**97.6**	**0.978**	0.975	**0.976**	28.8
Linear SVM	0.553	55.5	0.556	**0.997**	0.714	32.7
Nearest Neighbour	0.975	91.6	0.923	0.915	0.919	4.08
Random Forest	0.994	96.6	0.978	0.972	0.975	58.1

4.1. Java ransomware simulator

Tables 2 and 3 show the results from performing 10-fold cross-validation using the real ransomware and benignware samples. The last column in both tables shows the percentage of Amsel samples (Java-based ransomware simulator) that were classified as malicious by the classifiers when trained on all the real ransomware and benignware. For the sake of brevity, this has been referred to as 'Accuracy' in the table.

The results in Tables 2 and 3 indicate that Gradient Boost is the best performing classifier for distinguishing real ransomware from benignware. It obtains an accuracy of 97.6% and 97.3% for the Kernel Cuckoo data. Similarly to the results in previous work [13], the difference in results is quite small but still significant. Based on that data alone, Gradient Boost would ordinarily be the recommended classifier to use for this scenario alongside data from the Kernel Driver. However, we are also assessing how dependent the classifiers' results are on evasive features through the use of simulated ransomware.

The percentage of Amsel samples classified as malicious paints a different picture (shown in the last column of Tables 2 and 3). When using data from Cuckoo, Gradient Boost classified all the samples as malicious, whereas, when using the Kernel data, Gradient Boost only classified 28.8% of the simulated ransomware as malicious. When using the Kernel data, Decision Tree classified the most Amsel samples as malicious (97.4%). However, classification results this high are not necessarily desirable since it means that the classifiers are also likely to label benignware as malicious as they are also labelling the simulated ransomware with no evasive features as malicious.

The classification accuracy obtained by the remaining classifiers when labelling the simulated ransomware is considerably lower. Importantly here, the data from Amsel is being treated as a test set and so the results cannot simply be reversed if below 50%. Regardless, it is not the overall accuracy that we are using to assess the classifiers' performance in detecting the simulated ransomware, but the manner in which the accuracy changes when the evasiveness within the simulated samples is increased. The method by which this is achieved is described in Section 3.5.

The performance of each classifier per simulated ransomware sample is illustrated in Fig. 1 as a histogram. The histogram has been drawn in a form similar to a bar chart to make it easy to compare the proportion of samples per time value that were classified as malicious or benign. Fig. 1 is separated into two subfigures to show the classification results using the Kernel data (Fig. 1(a)) and the Cuckoo data (Fig. 1(b)). These subfigures consist of two plots per classifier. The first plot shows the results for samples with interarrival times of 2 s or less and the second plot shows the results for samples with interarrival times above 2 s.

The results of Fig. 1(a) show that for three of six classifiers (AdaBoost, Gradient Boost and Linear SVM), when using the Kernel data, once the time between each encryption (interarrival time) is approximately beyond 10 s, the simulated ransomware is consistently classified as malicious. However, before that, it is labelled as benign. The opposite occurs with Gradient Boost as once the interarrival time goes beyond two seconds, the classifier largely classifies the simulated ransomware as benign. The majority of the classifiers in Fig. 1(a) classify the simulated ransomware as benign when the interarrival time is 2 s or less. Decision Tree classifies the most samples as malicious with only a small fraction classified as benign. Whereas Nearest Neighbours largely classifies the samples as benign. The results from the Amsel data collected at a Kernel level suggest the majority of the classifiers trained on this data see ransomware as containing largely evasive behaviour.

Fig. 1(b) shows how the classifiers label Amsel when using the data from Cuckoo. There is not as clear a pattern as compared to when the Kernel data was used suggesting that the correlation between interarrival time and classification accuracy is not as strong for this set of simulated ransomware. Decision Tree and Random Forest tend to label the simulated ransomware samples as benign as the interarrival time or evasive behaviour increases. However, both those classifiers along with AdaBoost do not seem to be particularly robust since for many of the interarrival time values, the classification result of the same sample alternates between benign and malicious. On the other hand, Linear SVM and Nearest Neighbours label all samples as benign while Gradient Boost labels all as malicious (neither of which are particularly desirable).

To gain a better understanding of the reasons behind the results, the features of the simulated ransomware that were used to relate it to malware, we analyse the top ten features of the classifiers that obtained the highest accuracy on the Amsel samples.

4.1.1. Feature ranking results

Cuckoo data

Fig. 2 shows the top ten features (from left to right) for the Cuckoo data as determined by Gradient Boost. It also shows the relative frequency with which those features were called by the ransomware, benignware, and simulated ransomware.

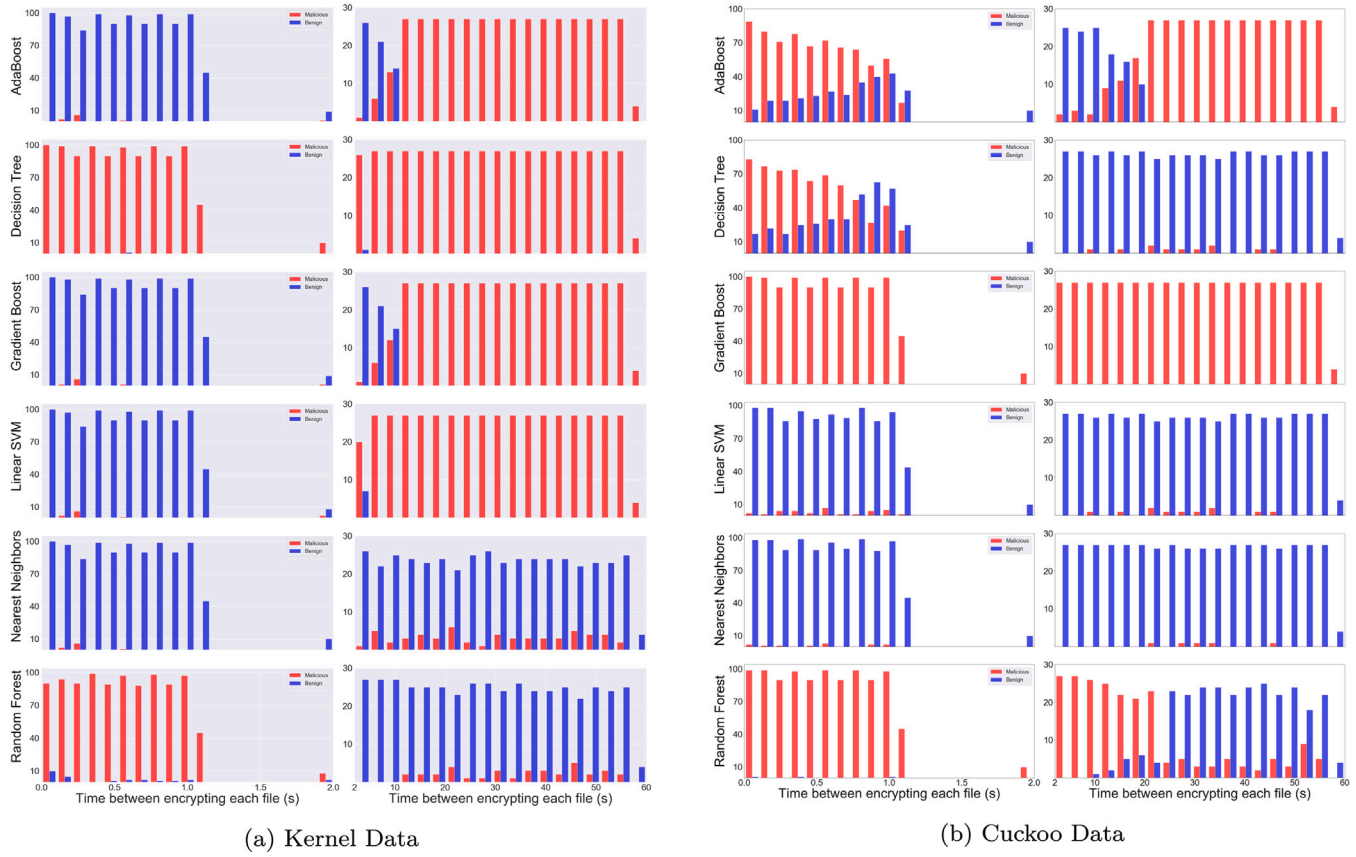


Fig. 1. Classification results per Amsel sample represented as a histogram. On the *x-axis* the time between each encryption for each sample is plotted. The *y-axis* shows the proportion of samples classified as malicious (red) and benign (blue) for those time values. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

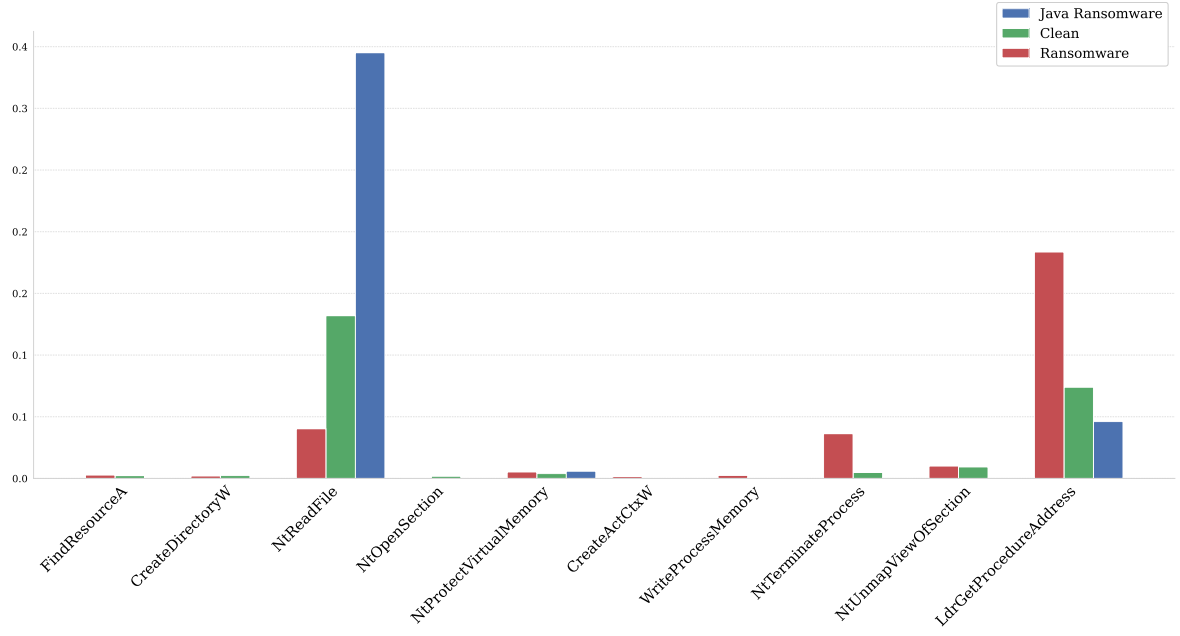


Fig. 2. Normalised frequency (*y-axis*) of the most influential features (*x-axis*) of Gradient Boost using the Cuckoo data.

The most prominent call (with regards to frequency) in Fig. 2 is NtReadFile, which, as expected, is used considerably by the simulated ransomware. Although, it is not called as frequently by ransomware as it is by benignware on average. Therefore, the frequency with which its called by benignware is closer to that of the simulated ransomware. The next most prominent feature is LdrGetProcedureAddress. This feature is commonly used by malware (particularly obfuscated malware) as it allows it to import methods at runtime [22], thereby evading static analysis. LdrGetProcedureAddress is also called by simulated ransomware quite frequently despite not actually being used in its

source code. This is due to the Java Virtual Machine (JVM) which performs dynamic loading at runtime.

The top ten features for which the relative calling frequency of simulated ransomware is closer to that of ransomware are NtProtectVirtualMemory, CreateDirectoryW, NtOpenSection and CreateActCtxW. NtProtectVirtualMemory can be used by malware to detect a debugger by creating what is known as guard pages, that, when accessed raise an EXCEPTION_GUARD_PAGE. If a debugger is present, the exception is intercepted by a debugger [94]. In the case of the simulated ransomware, its presence is due to the JVM since it was not explicitly called in its source code. The same is true for CreateDirectoryW as its presence in the simulated ransomware is due to the JVM creating temporary folders. The feature NtOpenSection is used to access 'sections', which are shared memory regions. This is used by ransomware and to a lesser extent by the simulated ransomware. Malware is known to use NtOpenSection in conjunction with NtUnmapViewOfSection (also in the top ten) to inject its code into a legitimate process and execute it from there to avoid detection. However, its presence in the simulated ransomware is also an artefact from how the JVM manages memory.

Some of the remaining features further highlight the tendency of classifiers favouring evasive features when differentiating ransomware from benignware. WriteProcessMemory, for example, can be used to write malicious code into an external process' memory space [22]. NtTerminateProcess can be used to terminate other processes running on the system. This can be used by malware to stop any antivirus solutions from running [95].

Therefore, the top ten features within the Cuckoo data ranked by Gradient Boost seem to suggest that the evasive properties of malware are favoured by classifiers. Though the simulated ransomware was written to simulate the malicious properties and one evasive property of ransomware, this did not translate in the features used due to interference from the JVM. Therefore, the main reason the simulated ransomware was detected so confidently by Gradient Boost was due to the use of evasive features by the JVM. This also further explains the reason for the lack of a visible relationship between interarrival time and classification accuracy.

Kernel data

The classifier that obtained the highest accuracy relating the simulated ransomware to real ransomware using the Kernel data was Decision Tree. Therefore, its top ten features and their relative frequencies are displayed in Fig. 3. Due to the large differences in frequencies of features, the y-axis is represented as a logarithmic scale to aid with visualisation.

Two prominent features in Fig. 3 are NtReadFile and NtWriteFile. Unlike the Cuckoo data, the frequency with which they are called by the simulated ransomware is closer to that of real ransomware, suggesting that they assist with the classification of simulated ransomware. On the other hand, NtYieldExecution is called considerably more by simulated ransomware than benignware and ransomware. NtYieldExecution serves to stop the execution of the current thread and start the execution of another. The high frequency of this call is a side-effect of the simulated ransomware "sleeping" for a certain amount of time as NtYieldExecution is used to stop executing the current thread and start executing a new one. In addition, its frequency is further elevated by the JVM which uses it to juggle the many tasks it performs (such as garbage collection). The call, NtDelayExecution, is commonly used by malware to prevent executing its malicious payload until a specified time period [90]. As can be seen, it is used marginally more by malware than benignware. It is also used to a lesser extent by the simulated ransomware. As with the real ransomware, the simulated ransomware uses it to implement the sleep.

Of the less prominent features, NtOpenMutant is particularly interesting. 'Mutants' is the name given to 'mutexes' in Kernel mode. Mutexes are used to control access to shared resources. Malware authors tend to use them to ensure that only one instance of their malware is running on the machine at a time (to avoid re-infecting the machine). This could explain the relatively high frequency with which it is called by ransomware as opposed to benignware.

The other feature of note here is NtOpenThread. Again, this is called significantly more by malware as opposed to benignware. Malware frequently uses threads to run the code it has injected into another process [22,30].

As the Kernel data captures calls made from all processes running on the system, it is not as easy to interpret the reasons for the differences in the frequencies of the calls made. However, it can be concluded that a mix of file-handling and evasive features played a part in assisting with the detection of the simulated ransomware (as well as real ransomware). Unlike the Cuckoo data, the classifier trained on Kernel data is not looking largely at evasive behaviours to detect ransomware, but a greater variety of behaviours. This accounts for the clearer link between frequency of encryption and classification accuracy when observing classifiers trained on data from the Kernel.

Discussion

The classification results per simulated ransomware sample seem to indicate that, in general, the Kernel data is causing classifiers to focus more on evasive behaviour rather than malicious behaviour. The classifiers trained on Cuckoo data do not seem to show as clear a connection between the classification accuracy and the frequency of encryption which is most likely because they are overwhelmingly using evasive behavioural traits to detect ransomware. More evasive traits than were intentionally included in the simulated ransomware. Therefore the frequency of encryption and even encryption itself does not matter as much as the variety of evasive features present in a ransomware sample. Analysis of the top ten features shows that the simulated ransomware is using more evasive features than it has been programmed to which Gradient Boost utilises to detect the simulated ransomware when using the data from Cuckoo. The additional calls within the simulated ransomware are largely due to the calls made by the JVM during the execution of Amsel.

To understand how much influence the JVM has had over the results, and to get a better picture on the importance classifiers place on evasive behavioural traits when detecting malware, we rerun the tests with simulated ransomware written in C. This will give us complete control over the calls made and will also allow us to determine the effectiveness of utilising malware simulators written in Java for system call based analysis.

4.2. C-based malware simulator

4.2.1. Standard C time method

As discussed in Section 3.3.2, there are three methods by which we implemented simulated ransomware in C, the first two implementations are identical to Amsel in functionality, and the final method employs more complex encryption. The first two methods use XOR encryption but differ in the manner through which the evasiveness is implemented as there are two common methods to do this within C. One uses the Windows-specific Sleep function. The other method (and the method discussed in this section) uses the standard method in C to implement the sleep. Table 4 shows the accuracy obtained by the classifiers when trying to classify simulated ransomware using the standard C time function.

The results in Table 4 are vastly different from those obtained using simulated ransomware written in Java. The classifier that identified the most simulated ransomware samples as malicious using the Kernel data is Gradient Boost, with an accuracy of 89.9%. Of the remaining classifiers, Decision Tree and Random Forest still obtain an accuracy above 50% with 74.9% and 73.6%. Whereas, using the Cuckoo data, the classifier that classified the most simulated samples as malicious is Nearest Neighbours, obtaining an accuracy of 42.6%. The remaining classifiers experience sharp drops in accuracy with both Decision Tree

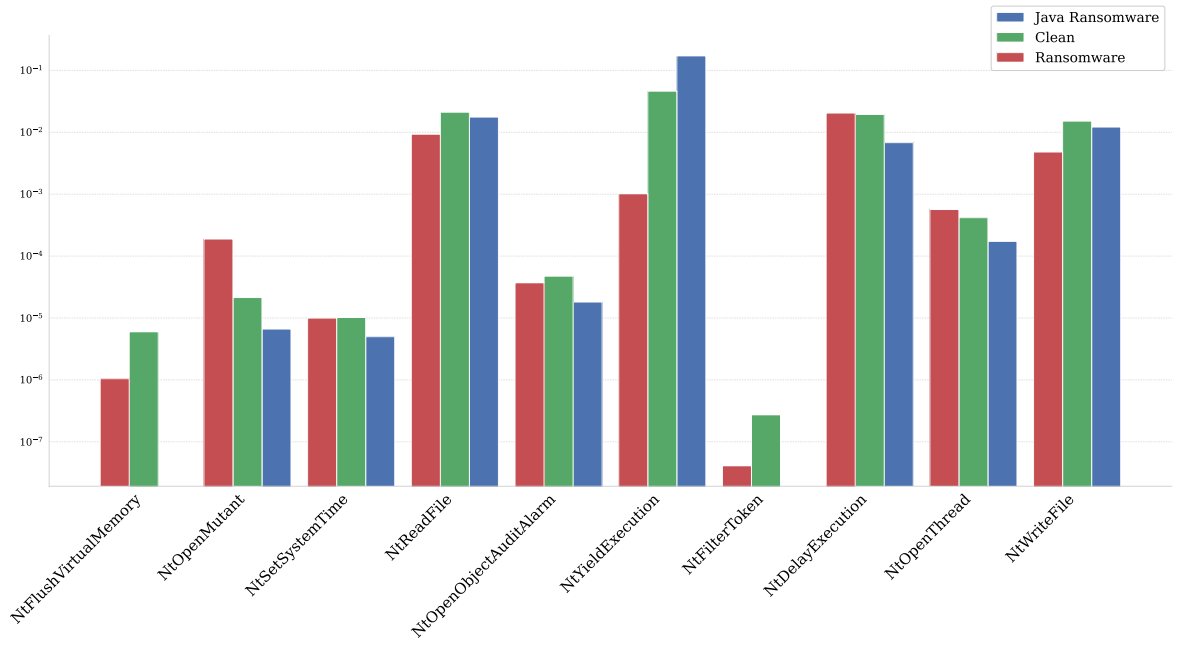


Fig. 3. Normalised frequency on a logarithmic scale (*y-axis*) of the most influential features (*x-axis*) of Decision Tree using the Kernel data.

Table 4
Classification accuracy of simulated ransomware with C time function using data from Cuckoo and the Kernel driver.

Machine learning algorithm	Kernel driver accuracy (%)	Cuckoo accuracy (%)
AdaBoost	40.4	3.09
Decision Tree	74.9	0.0
Gradient Boost	**89.9**	0.0
Linear SVM	33.0	2.99
Nearest Neighbour	27.9	**42.6**
Random Forest	73.6	12.7

and Gradient Boost (the best performing classifier against the Java simulated ransomware) classifying all simulated sample as benign. As before, more important than the overall performance is the level of evasion at which the simulated ransomware is considered benign. This is illustrated in Fig. 4.

Fig. 4(a) shows the performance per sample of each classifier using the Kernel data. As with Amsel, Nearest Neighbours and Linear SVM identify the simulated ransomware as malicious once the interarrival time is greater than 10 s. Whereas, with Decision Tree and AdaBoost, there is not as clear a connection between interarrival time and classification accuracy. This also shows a lack of robustness in the two classifiers since in essence the same sample is occasionally classified as malicious and occasionally as benign (showing just how fragile the rules used by the models are). Gradient Boost and Random Forest classify the simulated ransomware as malicious in the majority of instances up until the delay reaches one second at which point it is classified as benign. After the time between each encryption goes beyond two seconds, the simulated ransomware is, once again, classified as malicious in almost all instances with these two classifiers. This suggests that these classifiers have developed quite a specific understanding of ransomware with regards to its ratio of malicious to benign behaviour.

Fig. 4(b) shows the performance per sample of each classifier using the Cuckoo data. As suggested in the results, other than Nearest Neighbours, every classifier has performed poorly. It seems that the use of the C-time function does not fit with many of the classifiers' model of typical ransomware behaviour. With Nearest Neighbours, however, when the frequency of encryption is high (and thus the amount of evasive behaviour low), the simulated ransomware is classified as benign. However, as the time spent sleeping between each encryption exceeds 1 s, Nearest Neighbours classifies the simulated ransomware as malicious. *This suggests that when using the Cuckoo data, ransomware is recognised more from its evasive behaviour than malicious behaviour.* To better understand the results from the Kernel and Cuckoo data, the most influential features for the classifiers with the highest accuracy are analysed.

Feature ranking results

Analysing the top ten features for the best performing classifiers should also help reveal whether anything outside the intended behaviour impacted the results. The classifier with the highest accuracy using the Cuckoo data, Nearest Neighbours, does not have an inbuilt feature ranking mechanism, therefore, we use an independent feature ranking mechanism described in Section 3.4 to determine the most influential features. Essentially, the mechanism works by giving the classifier data from one feature at a time and recording the classification score. The features are then ranked from highest to lowest classification score. The top ten features of nearest neighbours and their relative frequency are shown in Fig. 5.

The benefits of using C over Java to simulate malware are immediately obvious from Fig. 5. The simulated ransomware is not showing any behaviour not programmed into it. In fact, the only features in Fig. 5 that record any behaviour from the simulated ransomware are NtReadFile and NtWriteFile. The frequency with which NtReadFile and NtWriteFile are called by the simulated ransomware is significantly more than that of benignware which exceeds that of ransomware. This shows why simulated ransomware is only detected by classifiers using the Cuckoo data when the file activity shown by simulated ransomware is low since then the file activity resembles the file activity of actual ransomware more closely. The preference given to evasive features by classifiers is also evidenced by the rest of the features in the top ten for Nearest Neighbours. Of these features, LdrGetProcedureAddress, NtTerminateProcess, SetUnhandledExceptionFilter, LdrGetDllHandle, LdrLoadDll and NtUnmapViewOfSection are all known to be used by malware for evasive purposes [11,22,95,96].

Fig. 6 shows the top ten features of the classifier with the highest accuracy relating real ransomware to simulated ransomware (Gradient Boost) on the Kernel data. To aid with visualisation, the *y*-axis is represented as a logarithmic scale.

As with Fig. 5, in Fig. 6, the most prominent features are NtReadFile and NtWriteFile. While some of the other features show some activity

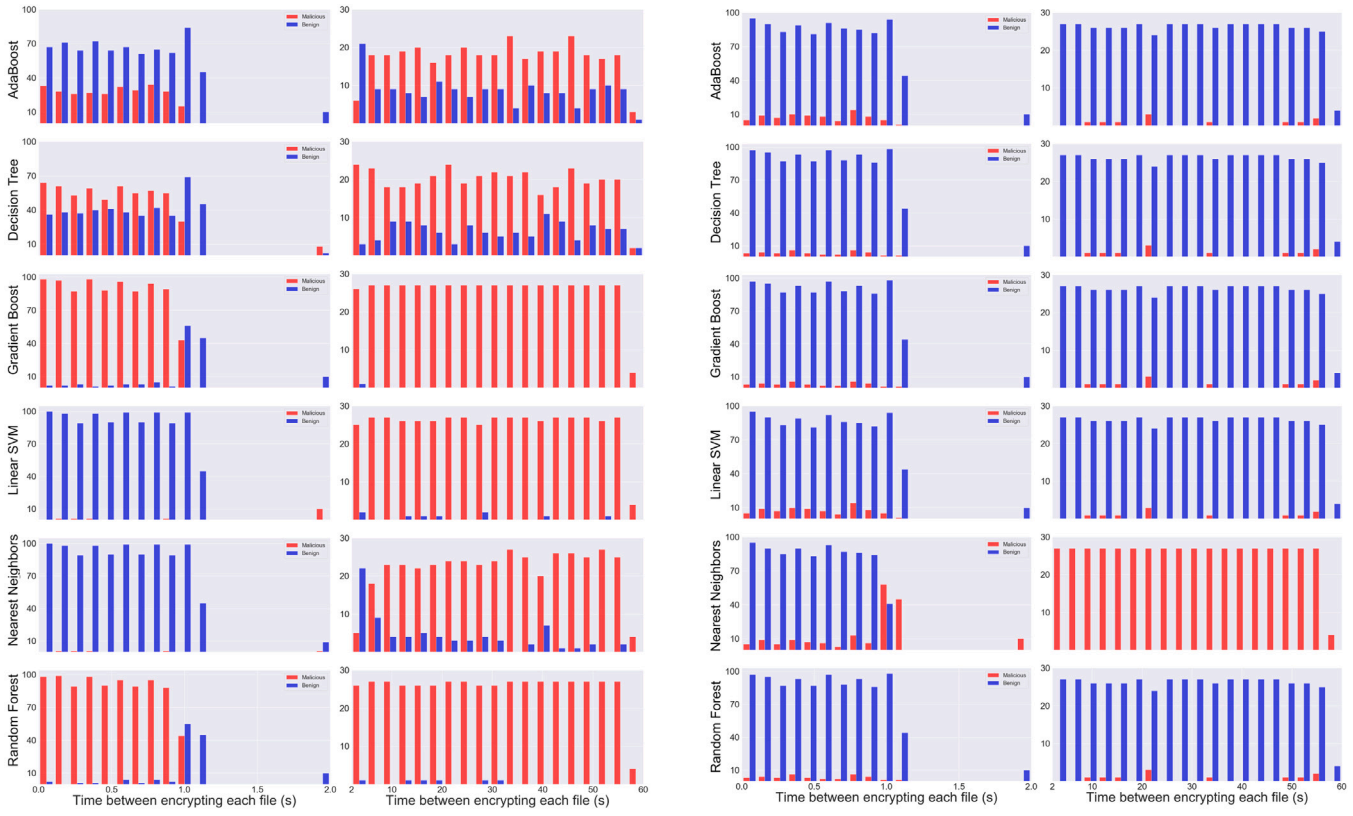


Fig. 4. Classification results per C-time simulated ransomware sample represented as a histogram. On the *x-axis* the time between each encryption for each sample is plotted. The *y-axis* shows the proportion of samples classified as malicious (red) and benign (blue) for those time values. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

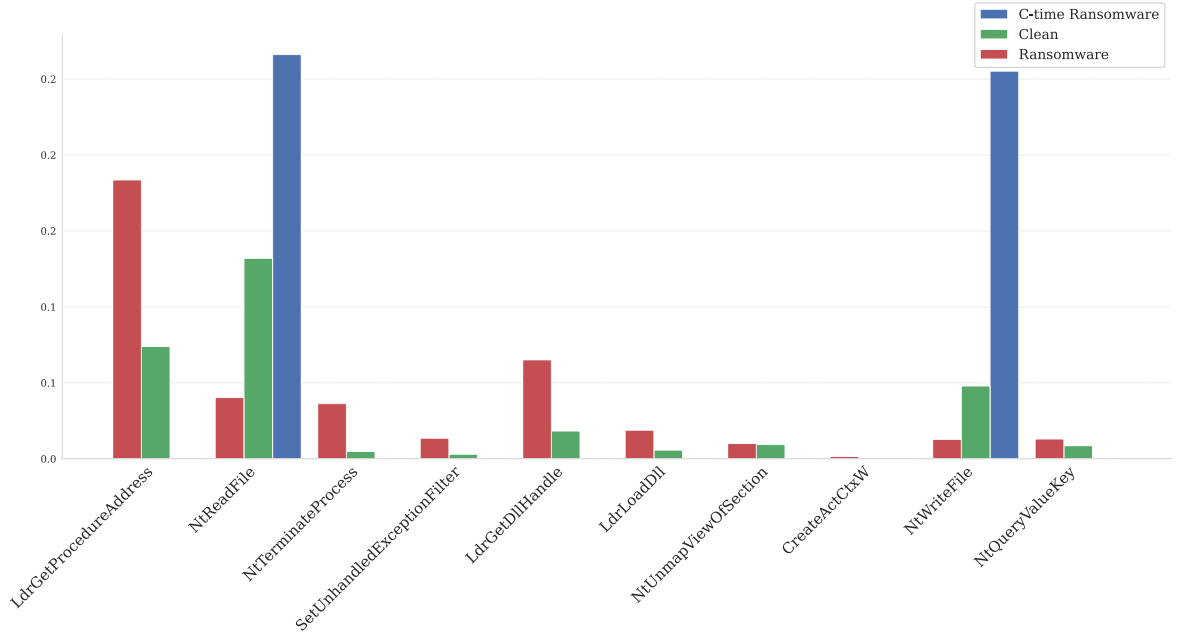


Fig. 5. Normalised frequency (*y-axis*) of the most influential features (*x-axis*) using the Cuckoo Data with Nearest Neighbours.

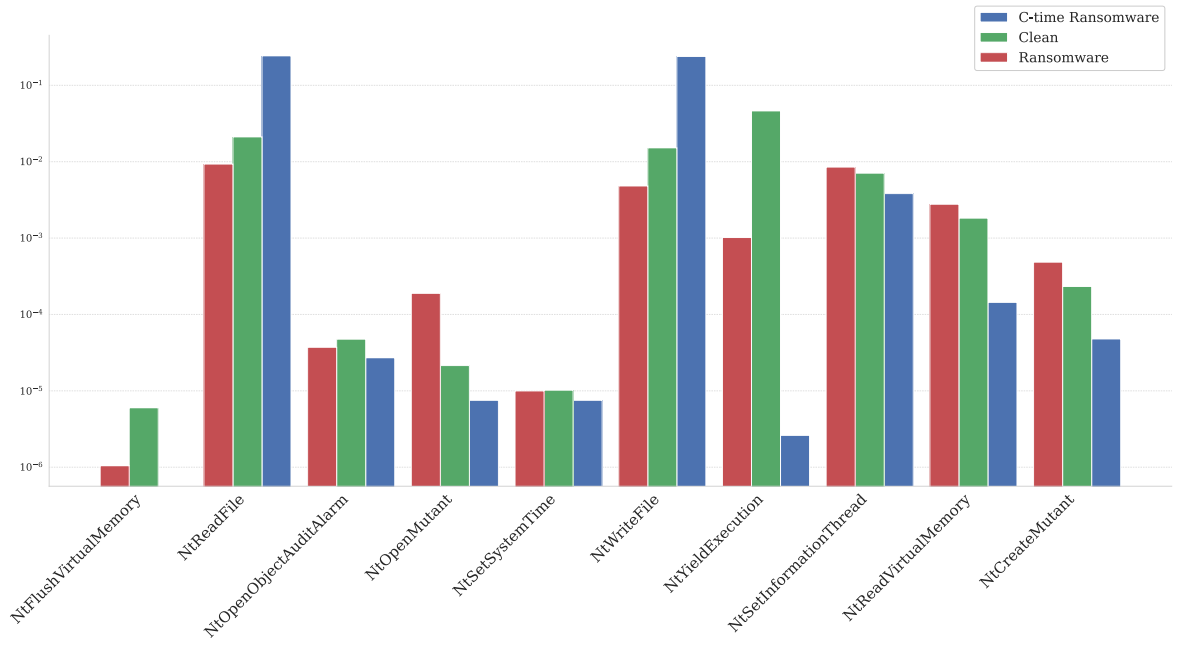


Fig. 6. Normalised frequency (*y-axis*) of the most influential features (*x-axis*) of Gradient Boost using the Kernel Data from the simulated ransomware using C time.

Table 5
Classification accuracy detecting simulated ransomware using Windows Sleep function (NtDelayExecution) using data gathered by Cuckoo and the Kernel driver.

Machine learning algorithm	Kernel driver accuracy (%)	Cuckoo accuracy (%)
AdaBoost	44.2	**32.1**
Decision Tree	**47.8**	0.0
Gradient Boost	36.1	6.54
Linear SVM	34.0	31.4
Nearest Neighbour	18.8	28.5
Random Forest	40.0	31.2

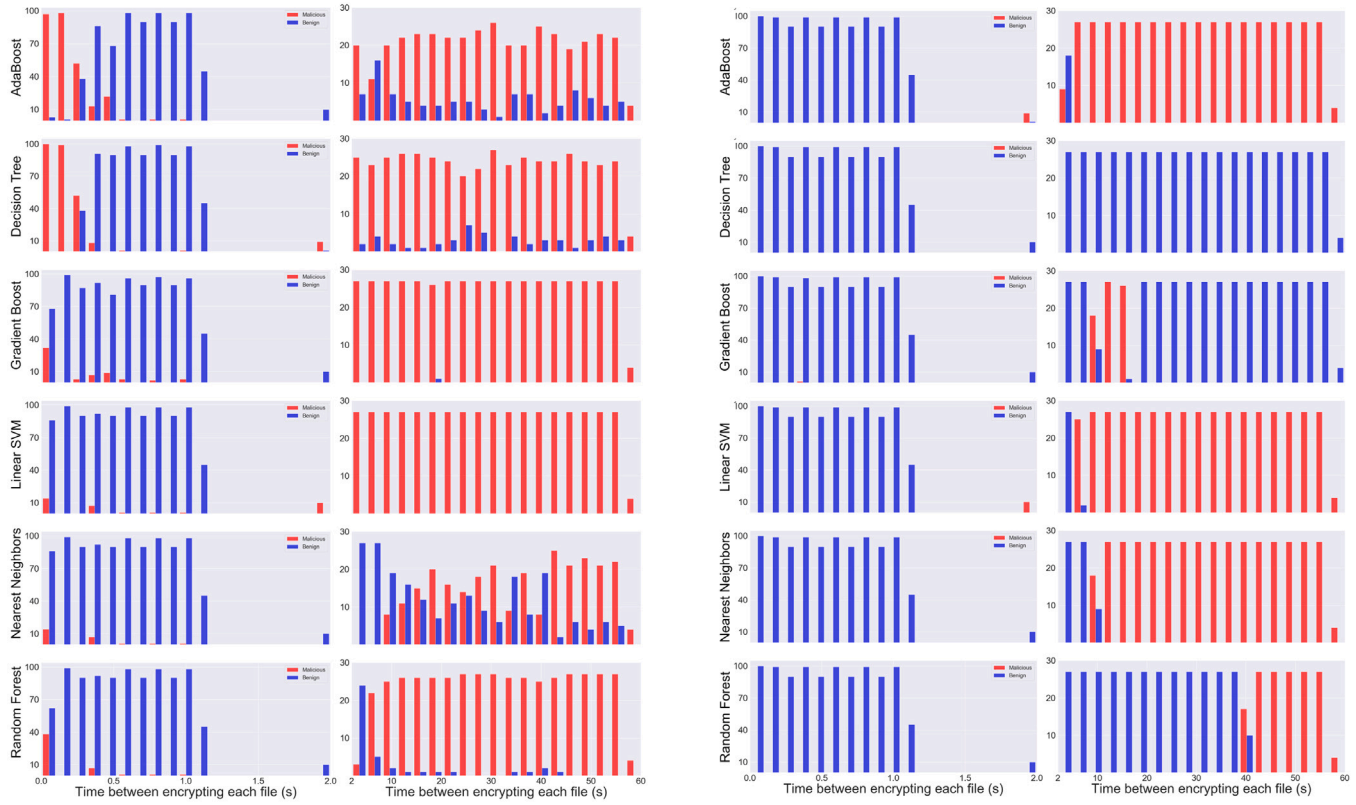
in the simulated ransomware, this is simply because the activity of the entire machine is being recorded by the Kernel driver. As with the Cuckoo data, the file calls are being called more frequently by benignware as opposed to ransomware. Therefore, the implication is that the simulated ransomware is being detected largely through the features both it and regular malware do not use (or do not use as frequently). It is their lack of complexity that they have in common.

4.2.2. Windows sleep method

The results from the previous section suggest that Java is not suitable as a language to simulate malware when it is being identified at a system call level. It also highlighted the importance that classifiers place on evasive features of malware when identifying it. The results also hinted at a limitation of the data gathered at a user-level in that classifiers trained on it have a very narrow understanding of what constitutes evasive behaviour. This is evident from the fact that only one classifier trained on the data from Cuckoo was able to classify the simulated samples as malicious. To further test the robustness of the classifiers we must observe their performance against different implementations of the simulated ransomware. In this section, we adapted the implementation of the simulated ransomware to use the recommended method from Windows (Sleep function) to delay execution. This is of particular significance since there is a possibility that malware writers favour Windows specific functions since it is generally the targeted OS. Beside this, the behaviour of the simulated ransomware has not been altered. Table 5 shows the accuracy obtained by each classifier classifying the simulated ransomware using data from Cuckoo and the Kernel driver.

Table 5 shows that altering one system call can have a remarkable effect on the classification ability. This is particularly noticeable in the results using the Cuckoo data. When classifying the simulated ransomware using C-time, all but one classifier trained on data from Cuckoo had a classification accuracy less than 15%. Whereas now, as seen in Table 5, only two classifiers (Decision Tree and Gradient Boost) obtain an extremely low classification accuracy. The classifier with the highest accuracy using the Cuckoo data, is AdaBoost (32.1%). For the Kernel data, Decision Tree obtains the highest accuracy (47.8%) relating simulated ransomware to real ransomware. The performance of the remaining classifiers using Kernel data does not differ as strongly from the best performing classifier as it does with the Cuckoo data suggesting that the Kernel data encourages more robust and consistent classifiers. We further analyse the results by looking at the actual samples that were classified as malicious and benign. This is shown in Fig. 7.

Fig. 7(a) shows the performance of each classifier per sample for the Kernel data from running the simulated ransomware using the Windows Sleep method. Four of the classifiers consistently label the samples with an interarrival time less than 2 s as benign (Gradient Boost, Linear SVM, Nearest Neighbours and Random Forest). Five classifiers generally label simulated ransomware samples with an interarrival time greater than 10 s as malicious (the previous four and AdaBoost). This suggests that there needs to be a certain level of evasiveness before a sample is recognised as malicious by the classifiers. However, once that threshold is reached, it would seem that classifiers continue to label samples as malicious regardless of how far above the threshold the quantity of evasive behaviour has reached. Nearest Neighbours differs from the others in that it does not show a clear relationship between interarrival time and classification accuracy once the interarrival time goes beyond 10 s suggesting a lack of robustness in the classifier. AdaBoost and Decision Tree generally label samples with interarrival times below 2 s as benign and those above as malicious. However, for samples with really low interarrivals (between 1.0×10^{-6} and 0.25 s), both classifiers label the samples as malicious suggesting they have developed a more precise model of what constitutes ransomware. In general, the classifiers seem to be identifying malware more from their evasive traits than their malicious traits particularly since, for most of the classifiers there needs to be a minimum of 2 s between each encryption. Additionally, there is a much clearer



(a) Kernel Data (b) Cuckoo Data

Fig. 7. Classification results per simulated ransomware sample using Windows Sleep represented as a histogram. On the *x-axis* the time between each encryption for each sample is plotted. The *y-axis* shows the proportion of samples classified as malicious (red) and benign (blue) for those time values. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

relationship between interarrival time and classification accuracy for more of the classifiers as compared to the C-time simulated ransomware suggesting that this strain of simulated ransomware more accurately reflects the real ransomware seen by the classifiers.

Fig. 7(b) shows how the classifiers labelled the simulated ransomware with the Windows Sleep function using data gathered by Cuckoo. Unlike the results with the simulated ransomware using C-time, the results in this instance are more balanced, with at least three classifiers obtaining an acceptable performance. The results show a preference towards simulated ransomware with a considerable amount of evasive behaviour as all the classifiers labelled the simulated ransomware as benign when the interarrival time was below 10 s. The clear relationship between interarrival time and classification accuracy for every classifier in this stage suggests that the simulated ransomware using the Windows Sleep function has more in common with real ransomware (with regards to evasive technique) as understood by classifiers trained on data from Cuckoo. The main difference between each classifier in Fig. 7(b) is the threshold after which the simulated ransomware is labelled as malicious. Decision Tree is the only classifier that classifies all the simulated ransomware samples as benign. This is particularly worrying given that its accuracy against real ransomware was 95.6%. The only other anomaly is Gradient Boost which only classifies simulated ransomware samples with interarrival times between 9 and 17 s as malicious. This would imply that it has obtained a very specific understanding of what constitutes ransomware and potentially over-fitted on the real ransomware.

Feature ranking results

Fig. 8 shows the top ten features of AdaBoost using the Cuckoo data and the relative frequencies with which they were called by ransomware, benignware and simulated ransomware.

In Fig. 8, the only feature that is used by the simulated ransomware is NtDelayExecution. This is the feature that is eventually called by the Windows Sleep function used in the simulated ransomware. In fact, there are no file-handling related calls in the top ten for AdaBoost. This explains why AdaBoost classified the simulated ransomware as benign when the time between each encryption was short as the frequency with which NtDelayExecution was called is closer to that of benignware than ransomware. This also explains why AdaBoost was unable to detect the simulated ransomware using the C time functions since it was looking for this specific evasive call in malware.

Fig. 9 shows the top ten features for Decision Tree using the data from the Kernel driver for simulated ransomware using the Windows Sleep function. To aid with visualisation, the *y*-axis is plotted on a logarithmic scale.

There are three features in the top ten that are frequently called by simulated ransomware, NtReadFile, NtDelayExecution, and NtWriteFile. In prioritising the evasive call being used in the simulated ransomware, Decision Tree identifies the simulated ransomware as malicious provided its behaviour is either largely malicious or largely evasive. However, in between the two extremes of sleep times, the frequency with which all three features are called by the simulated ransomware is closer to that of benignware than malware. Unlike AdaBoost when trained on the Cuckoo data (Fig. 8), Decision Tree, when trained on Kernel data, is using more than one dimension to detect simulated ransomware since its focus is not just the evasive trait, but the features relating to file-handling.

4.2.3. Windows encryption method

The results from the previous experiments show just how much the results can change from just altering a single system call to implement the same behaviour in malware. In addition, it further highlights the

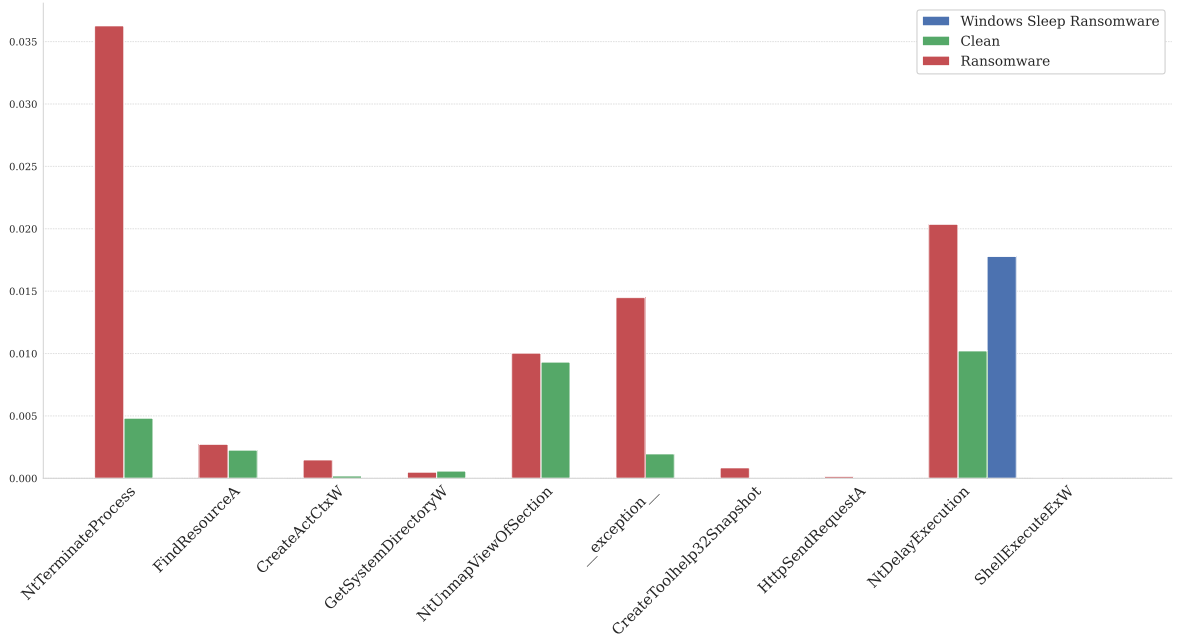


Fig. 8. Normalised average frequency (*y-axis*) of the most influential features (*x-axis*) according to AdaBoost when trained on Cuckoo data.

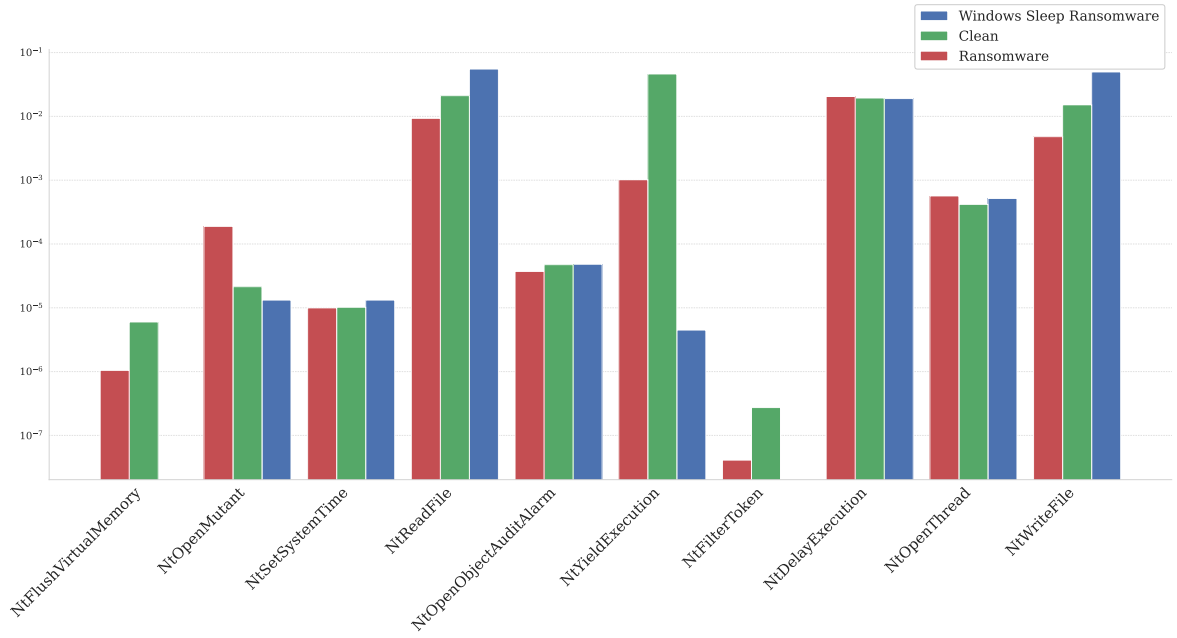


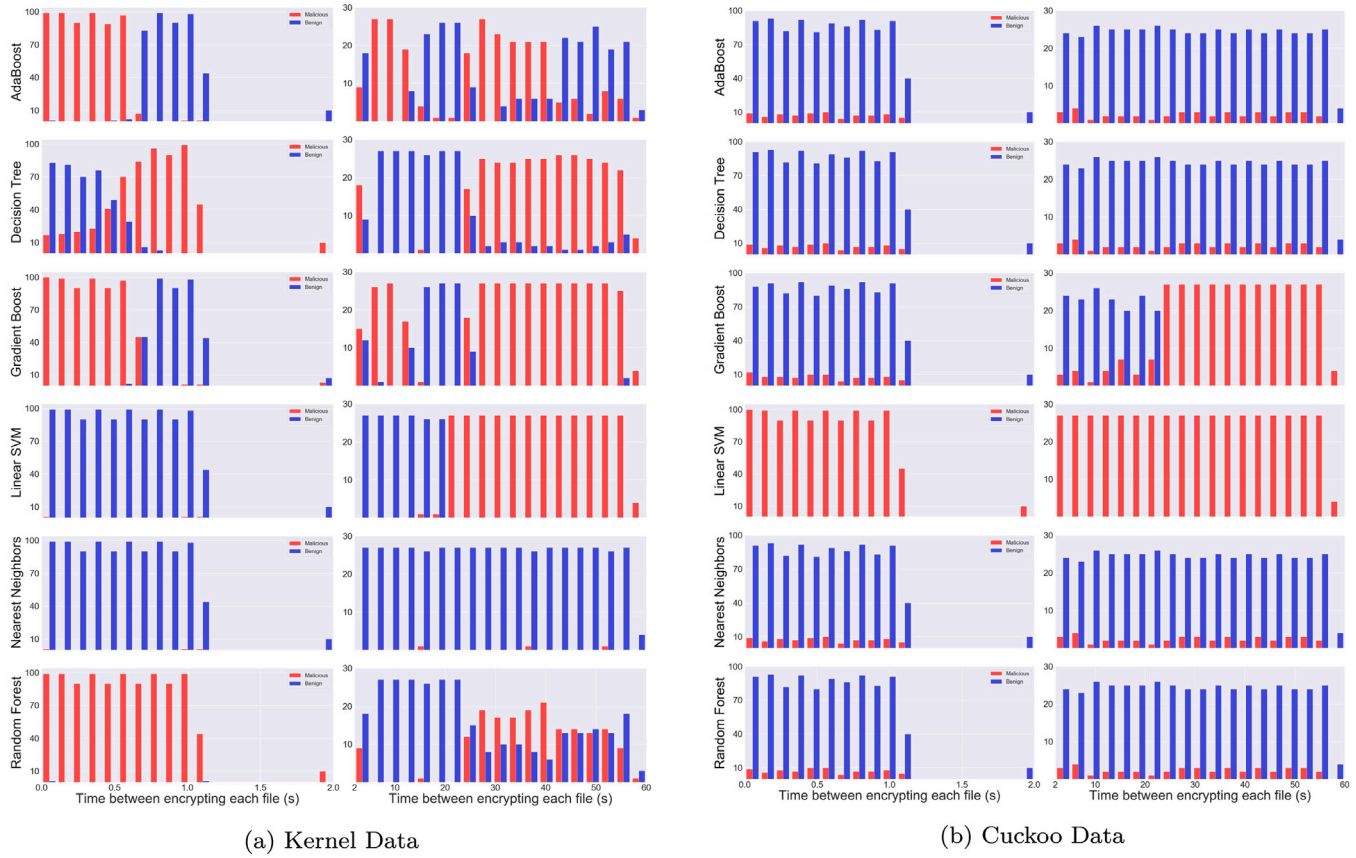
Fig. 9. Normalised average frequency (*y-axis*) of the most influential features (*x-axis*) according to Decision Tree when trained on Kernel data.

emphasis the classifiers place on evasive features when identifying malware. Our last set of experiments seek to determine the extent to which the encryption method used affects the results. In our previous experiments, the encryption method was written from scratch since it was just using an XOR function. This time, however, we use the encryption functions supplied by Windows to encrypt files with AES-256 and use the Windows Sleep function to implement the evasive behaviour so that all of the main components of the ransomware use the Windows API. The results for this are shown in Table 6.

The results in Table 6 show quite a dramatic change. Uniquely, the classifier with the highest accuracy using Cuckoo data is Linear SVM obtaining an accuracy of 100%. In addition, it is the only classifier that has a higher accuracy when using Cuckoo data as opposed to Kernel data to detect the simulated ransomware. All the other classifiers obtain a higher accuracy using the Kernel data further cementing the notion that Kernel data creates more robust ad consistent classifiers. The classifier with the highest accuracy using the Kernel data is Gradient Boost

Table 6
Classification accuracy detecting simulated ransomware using Windows Encryption routines using data gathered by Cuckoo and the Kernel driver.

Machine learning algorithm	Kernel driver accuracy (%)	Cuckoo accuracy (%)
AdaBoost	55.3	32.1
Decision Tree	43.6	26.4
Gradient Boost	**73.5**	25.9
Linear SVM	22.2	**100**
Nearest Neighbour	0.4	0.0
Random Forest	67.9	31.0



(a) Kernel Data (b) Cuckoo Data

Fig. 10. Classification results per simulated sample using Windows encryption functions represented as a histogram. On the *x-axis* the time between each encryption for each sample is plotted. The *y-axis* shows the proportion of samples classified as malicious (red) and benign (blue) for those time values. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

(73.5%). Compared to the results from using XOR encryption and Windows Sleep (Table 5) three classifiers (Decision Tree, Gradient Boost and Linear SVM) show an increase in accuracy here using the Cuckoo data, suggesting a greater inclination to classify samples as malicious if they are using encryption methods provided by Windows. Interestingly, with AdaBoost (the classifier that had the highest accuracy with the Windows Sleep method) the accuracy has not changed at all showing how little emphasis it places on the actual encryption method used. With the Kernel data, three classifiers (AdaBoost, Gradient Boost and Random Forest) showed a significant increase in accuracy as compared to when XOR encryption was used (as seen in Table 5). This suggests that the choice of encryption method does have an effect on how the classifier sees a sample. However, it is likely to be an indirect effect since there are not any explicit encryption calls in the Kernel, therefore it is likely that it is through the file calls (or rate of those calls) made by the simulated ransomware that they were identified. To better assess the results, we visualise the results for each simulated ransomware sample. This is shown in Fig. 10.

Fig. 10(a) shows the performance of the classifiers per sample of the simulated ransomware with complex encryption using the data from the Kernel driver. AdaBoost, Decision Tree and Gradient Boost developed a more specific understanding of what constitutes malware. Rather than there being one time value after which samples are classified as malicious, the classifiers fluctuate between malicious and benign classifications as the interarrival time is changed. However, this is not the case for Linear SVM which classifies simulated ransomware samples that sleep for 20 s or longer as malicious. In fact, the results of Linear SVM have been quite consistent across all sets of simulated ransomware using the Kernel data. Unusually, Random Forest classifies all samples with an interarrival time of 2 s or less as malicious and samples with interarrival times between 2 and 25 as benign. However, after that point, the classifier's results show a lack of robustness since the same sample is classified differently in different runs. This also suggests that the focus shifts to other features beyond this time value. Finally, Nearest Neighbours stands out as the only classifier to label all samples as benign. Generally, there is less agreement amongst the classifiers as to what is considered malicious as compared to when the other simulated ransomware samples were used. However, for the most part, the classifiers produced are fairly robust.

Fig. 10(b) shows how the classifiers using data from Cuckoo label each simulated ransomware sample using complex encryption functions. Aside from Linear SVM and Gradient Boost, the classifiers largely label the samples as benign. The accuracy values reported in Table 6 for the Cuckoo data hid the fact that the samples classified as malicious by all but two classifiers are not consistent but scattered amongst different time values. This shows just how fragile the majority of the classifiers created using the Cuckoo data are. This suggests that if the encryption functions within the Windows API are used to implement the encryption mechanism, the classifiers trained on Cuckoo data are more likely to label the sample as benign. Gradient Boost differs from the other classifiers in that samples with interarrival times above 20 s are classified as malicious. This is an improvement on its performance when only XOR encryption functions are used suggesting that Gradient Boost is using encryption functions within the Windows API to detect ransomware. In contrast, Linear SVM labels all the simulated ransomware samples as malicious. However, a 100% accuracy is not desirable since it is likely that benignware will also be classified as malicious. To make more sense of the results, the top ten features of the classifiers that were able to relate the most simulated samples to ransomware are dissected.

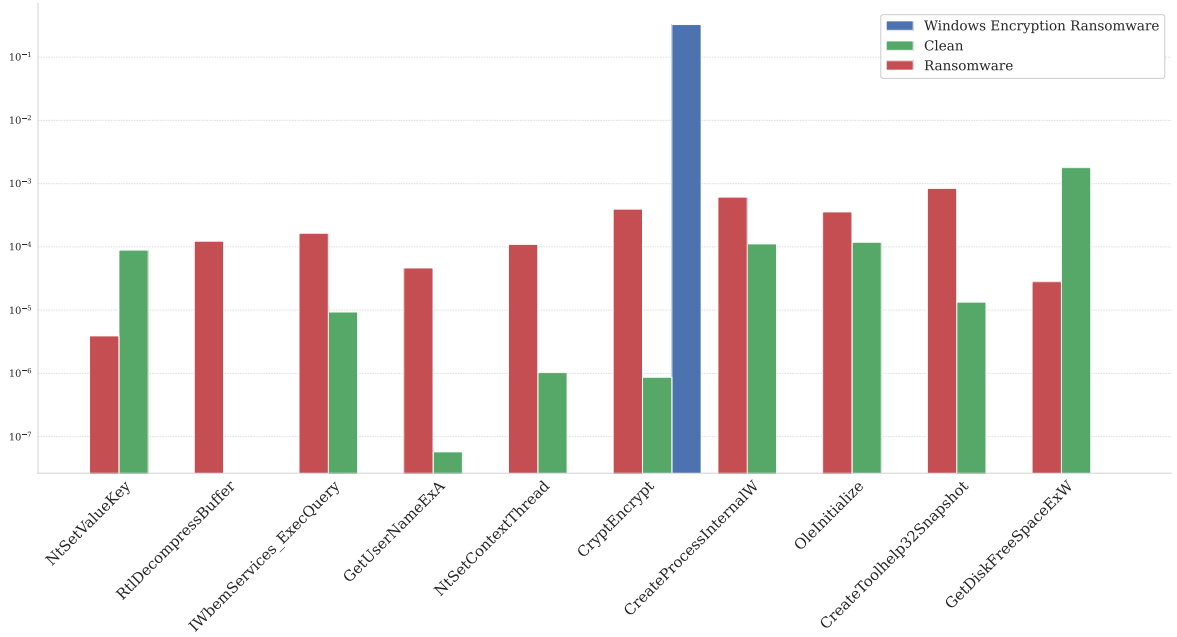


Fig. 11. Normalised average frequency on a logarithmic scale (*y-axis*) of the most influential features (*x-axis*) according to Linear SVM when trained on Cuckoo data.

Feature ranking results

Fig. 11 shows the top ten features of Linear SVM using the data from Cuckoo. For this graph, the *y*-axis is logarithmic due to the tremendous differences in frequencies.

One call in particular stands out in Fig. 11, the call CryptEncrypt. As its name suggests, this call is used to encrypt data. This explains why Linear SVM classifies all the simulated ransomware as malicious since it is using a feature related to encryption provided by Windows to differentiate ransomware from benignware. In fact, CryptEncrypt is the only call within the top ten that is used by the simulated ransomware. This also explains why Linear SVM did not perform as well with the previous versions of simulated ransomware since it was looking for ransomware specifically using the functions provided by Windows to encrypt files. Of the remaining features in the top ten some relate to evasion. For example, some of the calls are frequently used to detect virtualization (GetDiskFreeSpaceExW, CreateToolhelp32Snapshot [88] and GetUserNameExA [10]).

Fig. 12 shows the top ten features of Gradient Boost using the Kernel data from running the simulated ransomware using Windows encryption functions. As before, to aid with visualisation, the *y*-axis is a logarithmic scale.

Fig. 12 shows that Gradient Boost places importance on file-handling features. Unlike the Cuckoo data, however, there are no explicit features pertaining to encryption at the Kernel level on this iteration of Windows. Therefore the classifiers trained on Kernel data are not as significantly affected by the encryption methodology used. This has the advantage of preventing overfitting. From the top ten features, it is clear that there is not just one or two features contributing to the detection of the simulated ransomware. This is a good sign since it means the classifier is not placing too much importance on a single property. What can also be seen is that the simulated ransomware is detected partially by the features that it and real ransomware do not use.

4.3. Discussion

The experiments in this paper have shown that classifiers trained using the traditional dynamic malware analysis process do tend to identify malware through its evasive properties. This is a common observation throughout most of our experiments. As the time spent sleeping was increased and the amount of malicious activity of simulated ransomware was reduced, the classifiers started classifying the simulated ransomware as malicious. There were exceptions to this, however, but they did not weaken our conclusion. For example, the classification results from the simulated ransomware written in Java using data gathered by Cuckoo did not suggest much of a link between evasive behaviour and classification accuracy, particularly when considering the best performing classifier's results (100% accuracy). However, analysis of the top ten features showed that, in actuality, many of the calls made by the JVM were interpreted by the classifier as evasive features. This abundance of evasive behaviours led the classifier to label all the simulated ransomware samples as malicious, highlighting just how easily these systems can be deceived. The other notable exception to our conclusion was Linear SVM's results on the simulated ransomware using Windows encryption functions. When using data from Cuckoo, Linear SVM managed to obtain a 100% accuracy against the simulated ransomware using Windows encryption functions. Once again, study of the top ten features revealed that this was down to the overemphasis Linear SVM was placing on one encryption call in the Windows API. As a result anything using that call was immediately labelled as malicious. However, generally the evidence pointed to the fact that classifiers recognised malware by both evasive and malicious traits. Given that classifiers generally required a sample to wait for 2 s or more between each encryption before they classified a sample as malicious, it can be argued that a considerable portion of a sample's behaviour needs to be evasive before it is recognised as malicious.

Another conclusion that can be drawn from our experiments is that, in its raw form, the Kernel data is better suited to creating robust classifiers than the Cuckoo data. There were a number of observations that led to this conclusion. To begin with, the difference in results between the best performing classifier and the remaining classifiers was not as considerable for the classifiers when using Kernel data. This was true for all sets of simulated ransomware except for the simulated ransomware written in Java. However, those results were tampered by the interference from the JVM and even so, they showed the Kernel data and Cuckoo data to be equally matched with three classifiers performing better on Kernel data and three performing better on data from Cuckoo. For the rest of the simulated ransomware, the classifiers trained on Kernel data did not show as considerable changes in results (even between different sets of simulated ransomware) as those using

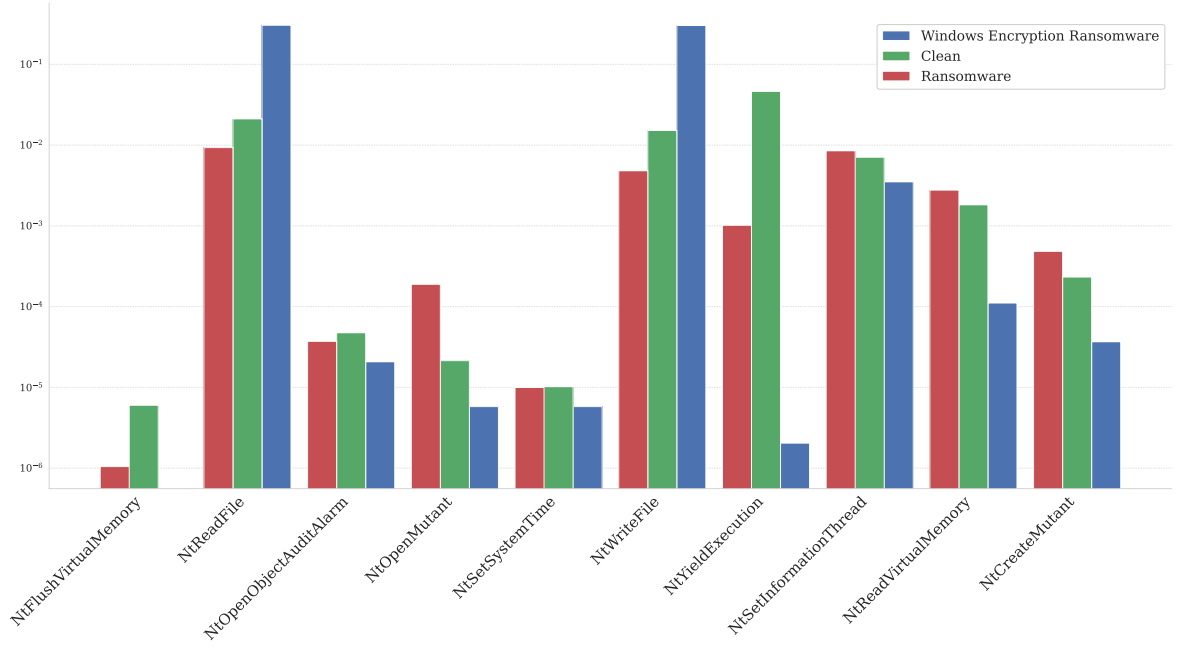


Fig. 12. Normalised average frequency (*y-axis*) of the most influential features (*x-axis*) according to Gradient Boost when trained on Kernel data.

data from Cuckoo. An obvious reason for this is that the Kernel driver operates at a higher level of abstraction and therefore the data fed to a classifier is likely to be less specific and more critical. Less specific because many user-level calls feed into a single Kernel call meaning that a classifier trained on data from the Kernel is less likely to focus on fine-grained details. More critical because a call that makes its way to the kernel is likely to be crucial to the proper functioning of the program or OS. We saw an example of the dangers from the Cuckoo data when comparing the change in results between the standard C-time simulated ransomware and the Windows Sleep simulated ransomware. With the standard C-time simulated ransomware only one classifier obtained an accuracy greater than 15%, whereas with the Windows Sleep simulated ransomware three classifiers achieved that using the Cuckoo data.

Another sign of the lack of robustness in the classifiers trained on data from Cuckoo is that for three out of four of the simulated ransomware types, the best performing classifier on the training data, Gradient Boost, was not the best performing classifier for the simulated ransomware. The only case where Gradient Boost was the best performing classifier for the simulated ransomware was when using the simulated ransomware written in Java. This was largely due to the interference the JVM had on the system calls made meaning that the simulated ransomware was showing more evasiveness then it was designed to. The Kernel data tells a slightly different story. The best performing classifiers for the simulated ransomware fluctuated between Decision Tree and Gradient Boost, both of which performed well against the training data. The two classifiers differed only on three features within their top ten. However, the crucial feature was NtDelayExecution. While Decision Tree included it in its top ten, Gradient Boost did not. As a result, Decision Tree performed better when detecting simulated ransomware written in Java and the simulated ransomware that used Windows Sleep as both of these made extensive use of that call. This slight disagreement between the classifiers suggests that the classifiers need to be fine-tuned somewhat.

The main problem with using system calls at user-level as features is that they result in classifiers learning to recognise specific calls as opposed to behaviours of malware (as was seen in Linear SVM). One possible remedy is to group calls into categories that are then used as features, or alternatively use features such as CPU and memory usage as recommended by [97] to ensure that classifiers learn to identify the effect that malware has on a system rather then specific calls it makes.

With regards to the Kernel data, it was the classifiers with the highest accuracy on the simulated ransomware that showed interesting behaviour consistently for all the simulated ransomware written in C. Rather then there being a specific interarrival time value beyond which samples were classified as malicious or benign, the samples classified as benign were sandwiched between malicious classifications. For example, the C-time simulated ransomware was classified as benign when the time between each encryption was 1 s and greater but less than 2 s. Outside of that range, Gradient Boost classified all simulated samples as malicious. This was similar for Decision Tree with the Windows Sleep simulated ransomware and Gradient Boost for the simulated ransomware using Windows encryption functions. Given that the ransomware was detected confidently at both extremes (high rate of encryption and low rate of encryption), it would suggest that these classifiers are looking for behaviour differing significantly from the norm to identify ransomware.

Finally, our research also found that there is scope for additional study into malware simulators, as carefully researched and crafted simulators could provide a useful means by which to further evaluate classifiers and determine how they are recognising malware. While Java is not a suitable language for simulating malware if it is being studied at a fine-grained, system-call level, it can still be useful if malware is being studied at a higher level of abstraction.

5. Future work

Our research is far from complete, the main scope for additional work is with regards to the feature representation technique. We employed a frequency histogram as the feature representation method due to its widespread use, however, there are many other methods by which system calls can be represented. For example there is the n-gram approach whereby each unique sequence of system calls of length ‘n’ are bundled together to create features. Additionally, some approaches also encode arguments and return values of system calls to pass to the classifier. This research represents a starting point into future work in which we would like to test more feature representation methods and determine how they affect the types of behaviours that classifiers use to identify malware.

Also in relation to the data collected, one of the aspects of our research was to compare whether the manner in which the data was

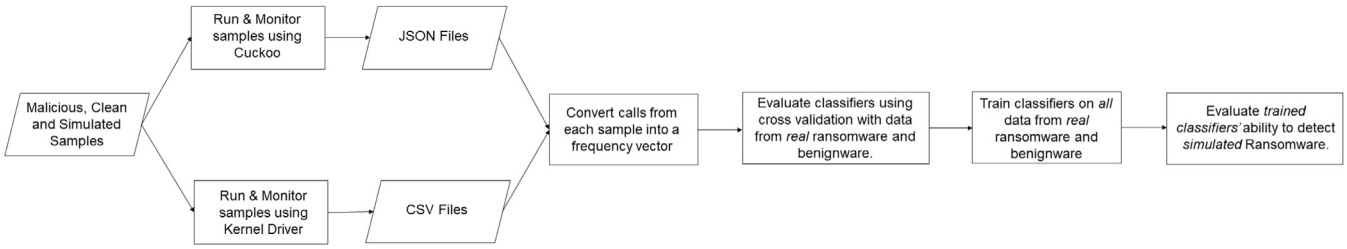


Fig. A.13. System diagram.

collected affected the classifiers' ability to detect ransomware. We compared data collected using one user-level method and one kernel-level method. However, as documented in our previous paper [13], there are a number of methods by which system-call data can be collected from a machine that may yield different results. Furthermore, using alternative kernel-level methods may allow us to repeat the experiments on different versions of Windows. Therefore, future work will explore the effectiveness of different data gathering techniques with regards to producing robust classifiers.

Finally, we looked at one form of malware using one type of evasive behaviour, however, there are many other families of malware (such as worms, Trojans, rootkits etc.) as well as evasive behaviours [94]. There is also malware that does not fit into a neat category and employs a variety of behaviours (for example, a Trojan that also has the capability to act like ransomware). Therefore we would like to further our study into more complex malware and behaviours to determine if our results here are consistent across the spectrum. For us to conduct a study of that scale, more sophisticated malware simulators need to be developed. Therefore, we hope that this paper will encourage the field to create such simulators with us so that we can conduct further research.

6. Conclusion

The aim of this paper was to assess whether classifiers trained within the dynamic malware analysis process were being biased by the volume of evasive behaviour present in modern-day malware. This was tested by training state-of-the-art classifiers on real ransomware and benignware and then assessing their ability to detect simulated ransomware with varying levels of evasive behaviour. Four types of simulated ransomware were used for our experiments, one type was written in Java, while the other three were written in C. Of the three written in C, two differed by the function they used to sleep while the last one differed from the rest by using Windows' own encryption functions instead of the XOR encryption functions that were used by the other sets of simulated ransomware.

The experiments and detailed analysis revealed that the classifiers are generally more likely to classify ransomware as malicious if a significant proportion of its behaviour consists of evasive techniques. This was true for both the data from Cuckoo and the Kernel driver. The best performing classifiers using the Kernel data, however, classified simulated ransomware as malicious only when its behaviour differed significantly from the norm and therefore tended to classify the simulated ransomware as benign when it contained a small amount of evasive behaviour. We also found that the Cuckoo data discouraged the creation of robust classifiers due to how fine-grained it is. Unlike the Kernel data, the classifiers trained on Cuckoo data tended to recognise malware through specific calls rather then behaviours. We therefore discourage the use of user-level data (such as that gleaned from Cuckoo) without further processing (such as the grouping of calls into categories).

There are still many things we were unable to study, such as how different feature selection methods affect the behaviours learned by classifiers and how the results change with each family of malware. Our intention is that this work is a starting point into further analysis. However that will only be possible if more sophisticated and powerful malware simulators are created.

CRediT authorship contribution statement

Matthew Nunes: Conceptualisation, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualisation. **Pete Burnap:** Conceptualisation, Methodology, Investigation, Resources, Writing – original draft, Writing – review & editing, Supervision, Project administration. **Philipp Reinecke:** Conceptualisation, Methodology, Software, Investigation, Writing – review & editing, Supervision. **Kaelon Lloyd:** Methodology, Software, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Dataset

Information on the data underpinning the results presented in this article, including how to access them, can be found in the Cardiff University data repository at https://doi.org/10.17035/d.2022.0176862059.

Acknowledgements

We would also like to thank VirusShare for providing us with samples and information regarding malware.

Funding

This work has been supported by the Engineering and Physical Sciences Research Council [project no. 1657416].

Appendix. System diagram

See Fig. A.13.

References

[1] Cohen F. Computer viruses: Theory and experiments. Comput Secur 1987;6(1):22–35. http://dx.doi.org/10.1016/0167-4048(87)90122-2, URL: http://www.sciencedirect.com/science/article/pii/0167404887901222.

[2] Lin D, Stamp M. Hunting for undetectable metamorphic viruses. J Comput Virol 2011;7(3):201–14.

[3] Szor P. The art of computer virus research and defense. Addison-Wesley Professional; 2005.

[4] Moser A, Kruegel C, Kirda E. Limits of static analysis for malware detection. In: Twenty-third annual computer security applications conference (ACSAC 2007). 2007, p. 421–30. http://dx.doi.org/10.1109/ACSAC.2007.21.

[5] Damodaran A, Di Troia F, Visaggio CA, Austin TH, Stamp M. A comparison of static, dynamic, and hybrid analysis for malware detection. J Comput Virol Hack Tech 2017;13(1):1–12.

[6] Ucci D, Aniello L, Baldoni R. Survey of machine learning techniques for malware analysis. Comput Secur 2019;81:123–47. http://dx.doi.org/10.1016/j.cose.2018.11.001, URL: http://www.sciencedirect.com/science/article/pii/S0167404818303808.

[7] Schiffman M. A brief history of malware obfuscation: Part 2 of 2. Cisco Blog 2010. URL: https://github.com/deepmipt/demo-cisco/blob/master/security/a_brief_history_of_malware_obfuscation_part_2_of_2.html.

[8] Ye Y, Li T, Adjeroh D, Iyengar SS. A survey on malware detection using data mining techniques. ACM Comput Surv 2017;50(3). http://dx.doi.org/10.1145/3073559.

[9] Holz T, Raynal F. Detecting honeypots and other suspicious environments. In: Proceedings from the sixth annual IEEE SMC information assurance workshop; 2005, p. 29–36.

[10] Bulazel A, Yener B. 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, New York, NY, USA: ACM; 2017, p. 2:1–21. http://dx.doi.org/10.1145/3150376.3150378, URL: http://doi.acm.org/10.1145/3150376.3150378.

[11] Chen X, Andersen J, Mao ZM, Bailey M, Nazario J. 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). 2008, p. 177–86. http://dx.doi.org/10.1109/DSN.2008.4630086.

[12] Branco RR, Barbosa GN, Neto PD. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. Black Hat 2012. URL: https://raw.githubusercontent.com/rrbranco/blackhat2012/master/blackhat2012-paper.pdf.

[13] Nunes M, Burnap P, Rana O, Reinecke P, Lloyd K. Getting to the root of the problem: A detailed comparison of kernel and user level data for dynamic malware analysis. J Inf Secur Appl 2019;48:102365. http://dx.doi.org/10.1016/j.jisa.2019.102365, URL: http://www.sciencedirect.com/science/article/pii/S2214212619300109.

[14] Nunes M. Comparing the utility of user-level and kernel-level data for dynamic malware analysis (Ph.D. thesis), Cardiff University; 2019.

[15] Biggio B, Corona I, Maiorca D, Nelson B, Šrndić N, Laskov P, Giacinto G, Roli F. Evasion attacks against machine learning at test time. In: Blockeel H, Kersting K, Nijssen S, Železný F, editors. Machine learning and knowledge discovery in databases. Berlin, Heidelberg: Springer Berlin Heidelberg; 2013, p. 387–402.

[16] Grosse K, Papernot N, Manoharan P, Backes M, McDaniel P. Adversarial examples for malware detection. In: Foley SN, Gollmann D, Snekkenes E, editors. Computer security – ESORICS 2017. Cham: Springer International Publishing; 2017, p. 62–79.

[17] Hu W, Tan Y. Generating adversarial malware examples for black-box attacks based on GAN. 2017, arXiv e-prints arXiv:1702.05983.

[18] Afianian A, Niksefat S, Sadeghiyan B, Baptiste D. Malware dynamic analysis evasion techniques: A survey. ACM Comput Surv 2019;52(6):1–28.

[19] Martignoni L, Paleari R, Fresi Roglia G, Bruschi D. Testing system virtual machines. In: Proceedings of the 19th international symposium on software testing and analysis. ISSTA '10, New York, NY, USA: ACM; 2010, p. 171–82. http://dx.doi.org/10.1145/1831708.1831730, URL: http://doi.acm.org/10.1145/1831708.1831730.

[20] Martignoni L, Paleari R, Roglia GF, Bruschi D. Testing CPU emulators. In: Proceedings of the eighteenth international symposium on software testing and analysis. ISSTA '09, New York, NY, USA: ACM; 2009, p. 261–72. http://dx.doi.org/10.1145/1572272.1572303, URL: http://doi.acm.org/10.1145/1572272.1572303.

[21] Pék G, Bencsáth B, Buttyán L. nEther: In-guest detection of out-of-the-guest malware analyzers. In: Proceedings of the fourth European workshop on system security. EUROSEC '11, New York, NY, USA: ACM; 2011, p. 3:1–6. http://dx.doi.org/10.1145/1972551.1972554, URL: http://doi.acm.org/10.1145/1972551.1972554.

[22] Sikorski M, Honig A. Practical malware analysis: The hands-on guide to dissecting malicious software. No Starch Press; 2012.

[23] Hsiao S, Kao D. The static analysis of WannaCry ransomware. In: 2018 20th international conference on advanced communication technology (ICACT). 2018, p. 153–8.

[24] Ferrie P. Attacks on more virtual machine emulators. Symantec Technol Exch 2007;55.

[25] Ferrie P. Anti-unpacker tricks–part one. Virus Bull 2008;4.

[26] Galloro N, Polino M, Carminati M, Continella A, Zanero S. A systematical and longitudinal study of evasive behaviors in windows malware. Comput Secur 2022;113:102550. http://dx.doi.org/10.1016/j.cose.2021.102550, URL: https://www.sciencedirect.com/science/article/pii/S0167404821003746.

[27] Lindorfer M, Kolbitsch C, Milani Comparetti P. Detecting environment-sensitive malware. In: Sommer R, Balzarotti D, Maier G, editors. Recent advances in intrusion detection. Berlin, Heidelberg: Springer Berlin Heidelberg; 2011, p. 338–57.

[28] Sun M, Lin M, Chang M, Laih C, Lin H. Malware virtualization-resistant behavior detection. In: 2011 IEEE 17th international conference on parallel and distributed systems. 2011, p. 912–7. http://dx.doi.org/10.1109/ICPADS.2011.78.

[29] Kirat D, Vigna G, Kruegel C. Barecloud: Bare-metal analysis-based evasive malware detection. In: Proceedings of the 23rd USENIX conference on security symposium. SEC'14, Berkeley, CA, USA: USENIX Association; 2014, p. 287–301, URL: http://dl.acm.org/citation.cfm?id=2671225.2671244.

[30] Rudd EM, Rozsa A, Günther M, Boult TE. A survey of stealth malware attacks, mitigation measures, and steps toward autonomous open world solutions. IEEE Commun Surv Tutor 2017;19(2):1145–72. http://dx.doi.org/10.1109/COMST.2016.2636078.

[31] Keragala D. Detecting malware and sandbox evasion techniques. SANS Inst InfoSec Read Room 2016;16. URL: https://www.sans.org/white-papers/36667/.

[32] Kang MG, Yin H, Hanna S, McCamant S, Song D. Emulating emulation-resistant malware. In: Proceedings of the 1st ACM workshop on virtual machine security. VMSec '09, New York, NY, USA: ACM; 2009, p. 11–22. http://dx.doi.org/10.1145/1655148.1655151, URL: http://doi.acm.org/10.1145/1655148.1655151.

[33] Shi H, Mirkovic J, Alwabel A. Handling anti-virtual machine techniques in malicious software. ACM Trans Priv Secur 2017;21(1):2:1–31. http://dx.doi.org/10.1145/3139292, URL: http://doi.acm.org/10.1145/3139292.

[34] Ramilli M, Bishop M, Sun S. Multiprocess malware. In: 2011 6th international conference on malicious and unwanted software. 2011, p. 8–13. http://dx.doi.org/10.1109/MALWARE.2011.6112320.

[35] Bayer U, Kruegel C, Kirda E. Anubis: Analyzing unknown binaries. 2009, URL: https://www.virusbulletin.com/conference/vb2009/abstracts/anubis-analyzing-unknown-binaries-automatic-way/.

[36] Buehlmann S, Liebchen C. Joebox: a secure sandbox application for windows to analyse the behaviour of malware. 2010, URL: https://www.joesecurity.org/.

[37] Norman sandbox whitepaper. Technical report, Norman Solutions; 2003, URL: https://ivanlef0u.fr/repo/madchat/vxdevl/papers/avers/03_sandbox.pdf.

[38] Ma W, Duan P, Liu S, Gu G, Liu J-C. Shadow attacks: automatically evading system-call-behavior based malware detection. J Comput Virol 2012;8(1):1–13. http://dx.doi.org/10.1007/s11416-011-0157-5.

[39] Willems C, Holz T, Freiling F. Toward automated dynamic malware analysis using cwsandbox. IEEE Secur Priv 2007;5(2):32–9. http://dx.doi.org/10.1109/MSP.2007.45.

[40] Srivastava A, Lanzi A, Giffin J, Balzarotti D. Operating system interface obfuscation and the revealing of hidden operations. In: Holz T, Bos H, editors. Detection of intrusions and malware, and vulnerability assessment. Berlin, Heidelberg: Springer Berlin Heidelberg; 2011, p. 214–33.

[41] Szegedy C, Zaremba W, Sutskever I, Bruna J, Erhan D, Goodfellow I, Fergus R. Intriguing properties of neural networks. 2013, arXiv e-prints arXiv:1312.6199.

[42] Rosenberg I, Shabtai A, Rokach L, Elovici Y. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In: Bailey M, Holz T, Stamatogiannakis M, Ioannidis S, editors. Research in attacks, intrusions, and defenses. Cham: Springer International Publishing; 2018, p. 490–510.

[43] Yuan X, He P, Zhu Q, Li X. Adversarial examples: Attacks and defenses for deep learning. 2017, arXiv e-prints arXiv:1712.07107.

[44] Papernot N, McDaniel P, Goodfellow I. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. 2016, arXiv e-prints arXiv:1605.07277.

[45] Goodfellow IJ, Shlens J, Szegedy C. Explaining and harnessing adversarial examples. 2014, arXiv e-prints arXiv:1412.6572.

[46] Arp D, Spreitzenbarth M, Hubner M, Gascon H, Rieck K, Siemens C. DREBIN: Effective and explainable detection of android malware in your pocket. In: Ndss, Vol. 14. 2014, p. 23–6.

[47] Papernot N, McDaniel P, Jha S, Fredrikson M, Celik ZB, Swami A. The limitations of deep learning in adversarial settings. In: 2016 IEEE European symposium on security and privacy (EuroS P). 2016, p. 372–87. http://dx.doi.org/10.1109/EuroSP.2016.36.

[48] Rosenthal virus simulator. Rosenthal Engineering; 1991, URL: https://vxug.fakedoma.in/archive/VxHeaven/vx.php@id=sr00.html.

[49] Trojan simulator. Mischel Internet Security Ltd; 2019, http://www.testmypcsecurity.com/securitytests/trojan_simulator.html, visited on 2019-09-11.

[50] Leszczyna R, Nai Fovino I, Masera M. Simulating malware with malsim. J Comput Virol 2010;6(1):65–75. http://dx.doi.org/10.1007/s11416-008-0088-y.

[51] Bellifemine F, Bergenti F, Caire G, Poggi A. JADE—a Java agent development framework. In: Multi-agent programming. Springer; 2005, p. 125–47.

[52] Philipp Reinecke SC. Amsel – the abstract malware symptom emulation library. 2019, https://github.com/AmselProject/amsel, visited on 2019-09-11.

[53] Arnold K, Gosling J, Holmes D. The Java programming language. Addison Wesley Professional; 2005.

[54] Kato M, Matsunami T, Kanaoka A, Koide H, Okamoto E. Tracing advanced persistent threats in networked systems. In: Al-Shaer E, Ou X, Xie G, editors. Automated security management. Cham: Springer International Publishing; 2013, p. 179–87. http://dx.doi.org/10.1007/978-3-319-01433-3_11.

[55] Xu Y, Koide H, Vargas DV, Sakurai K. Tracing MIRAI malware in networked system. In: 2018 sixth international symposium on computing and networking workshops (CANDARW). 2018, p. 534–8. http://dx.doi.org/10.1109/CANDARW.2018.00104.

[56] Venners B. The Java virtual machine. In: Java and the Java virtual machine: Definition, verification, validation. McGraw-Hill; 1998.

[57] Phipps G. Comparing observed bug and productivity rates for Java and C++. Softw - Pract Exp 1999;29(4):345–58.

[58] The state of ransomware 2021. Technical report, Sophos; 2021, URL: https://secure2.sophos.com/en-us/medialibrary/pdfs/whitepaper/sophos-state-of-ransomware-2021-wp.pdf.

[59] Sgandurra D, Muñoz-González L, Mohsen R, Lupu EC. Automated dynamic analysis of ransomware: Benefits, limitations and use for detection. 2016, arXiv preprint arXiv:1609.03020.

[60] Shijo P, Salim A. Integrated static and dynamic analysis for malware detection. Procedia Comput Sci 2015;46:804–11. http://dx.doi.org/10.1016/j.procs.2015.02.149, URL: http://www.sciencedirect.com/science/article/pii/S1877050915002136. Proceedings of the International Conference on Information and Communication Technologies, ICICT 2014, 3-5 December 2014 at Bolgatty Palace & Island Resort, Kochi, India.

[61] Virusshare.com. 2017, https://virusshare.com/, visited on 2017-11-28.

[62] Sourceforge. 2017, https://sourceforge.net/, (visited on 2017-08-02).

[63] Filehippo. 2017, https://filehippo.com/, (visited on 2017-08-02).

[64] Küchler A, Mantovani A, Han Y, Bilge L, Balzarotti D. Does every second count? Time-based evolution of malware behavior in sandboxes. 2021.

[65] Guḥmundsson A. 32-bit virus threats on 64-bit windows. Technical report, Symantec; 2020, URL: https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/32-bit-virus-threats-64-bit-windows-02-en.pdf.

[66] Chebyshev V, Sinitsyn F, Parinov D, Liskin A, Kupreev O. IT threat evolution Q1 2018. Statistics. Technical report, Kaspersky Lab; 2018, URL: https://securelist.com/it-threat-evolution-q1-2018-statistics/85541/.

[67] Guarnieri C, Tanasi A, Bremer J, Schloesser M. The cuckoo sandbox. 2012, URL https://www.cuckoosandbox.org.

[68] Malik S, Khatter K. System call analysis of android malware families. Indian J Sci Technol 2016;9(21).

[69] Burguera I, Zurutuza U, Nadjm-Tehrani S. Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM workshop on security and privacy in smartphones and mobile devices; 2011, p. 15–26.

[70] Asmitha KA, Vinod P. A machine learning approach for linux malware detection. In: 2014 international conference on issues and challenges in intelligent computing techniques (ICICT). 2014, p. 825–30. http://dx.doi.org/10.1109/ICICICT.2014.6781387.

[71] Afonso VM, de Amorim MF, Grégio ARA, Junquera GB, de Geus PL. Identifying android malware using dynamically obtained features. J Comput Virol Hack Tech 2015;11(1):9–17.

[72] Natani P, Vidyarthi D. Malware detection using API function frequency with ensemble based classifier. In: International symposium on security in computing and communication. Springer; 2013, p. 378–88.

[73] Hampton N, Baig Z, Zeadally S. Ransomware behavioural analysis on windows platforms. J Inf Secur Appl 2018;40:44–51. http://dx.doi.org/10.1016/j.jisa.2018.02.008, URL: https://www.sciencedirect.com/science/article/pii/S2214212617306506.

[74] Vinayakumar R, Soman KP, Senthil Velan KK, Ganorkar S. Evaluating shallow and deep networks for ransomware detection and classification. In: 2017 international conference on advances in computing, communications and informatics (ICACCI). 2017, p. 259–65. http://dx.doi.org/10.1109/ICACCI.2017.8125850.

[75] Homayoun S, Dehghantanha A, Ahmadzadeh M, Hashemi S, Khayami R. Know abnormal, find evil: Frequent pattern mining for ransomware threat hunting and intelligence. IEEE Trans Emerg Top Comput 2017.

[76] Shaukat SK, Ribeiro VJ. RansomWall: A layered defense system against cryptographic ransomware attacks using machine learning. In: Communication systems & networks (COMSNETS), 2018 10th international conference on. IEEE; 2018, p. 356–63.

[77] Zhang H, Xiao X, Mercaldo F, Ni S, Martinelli F, Sangaiah AK. Classification of ransomware families with machine learning based on N-gram of opcodes. Future Gener Comput Syst 2019;90:211–21. http://dx.doi.org/10.1016/j.future.2018.07.052, URL: http://www.sciencedirect.com/science/article/pii/S0167739X18307325.

[78] Hasan MM, Rahman MM. Ranshunt: A support vector machines based ransomware analysis framework with integrated feature set. In: Computer and information technology (ICCIT), 2017 20th international conference of. IEEE; 2017, p. 1–7.

[79] Daku H, Zavarsky P, Malik Y. Behavioral-based classification and identification of ransomware variants using machine learning. In: 2018 17th IEEE international conference on trust, security and privacy in computing and communications/ 12th IEEE international conference on big data science and engineering (TrustCom/BigDataSE). 2018, p. 1560–4. http://dx.doi.org/10.1109/TrustCom/BigDataSE.2018.00224.

[80] Philipp Reinecke SC. Malware modelling. 2015, Patent no. WO2017020929A1.

[81] Philipp Reinecke SC. Malware modelling. 2015, Patent no. WO2017020928A1.

[82] Philipp Reinecke SC. Malware modelling. 2015, Patent no. WO2017020926A1.

[83] Bencsáth B, Pék G, Buttyán L, Felegyhazi M. The cousins of stuxnet: Duqu, flame, and gauss. Future Internet 2012;4(4):971–1003.

[84] MalwareTech. Kelihos analysis - part 1 - MalwareTech. 2015, URL: https://www.malwaretech.com/2015/12/kelihos-analysis-part-1.html.

[85] Trojan:Win32/DelfInj!MTB threat description. 2018, URL: https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Trojan:Win32/DelfInj!MTB&ThreatID=-2147236481.

[86] Baker ACB, Chiu A. Threat spotlight: Rombertik, gazing past the smoke, mirrors, and trapdoors. 2018, Retrieved November.

[87] SOPHOS Threat Center. Mal/Carbanak-A. 2017, URL: https://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/Mal~Carbanak-A.aspx.

[88] Kleymenov A, Thabet A. Mastering malware analysis: The complete malware analyst's guide to combating malicious software, APT, cybercrime, and IoT attacks. Packt Publishing Ltd; 2019.

[89] Singh A, Bu Z. Hot knives through butter: Evading file-based sandboxes. Threat Res Blog 2013.

[90] Oyama Y. Trends of anti-analysis operations of malwares observed in API call logs. J Comput Virol Hack Tech 2018;14(1):69–85.

[91] Example C program: Encrypting a file - win32 apps | microsoft docs. 2018, https://docs.microsoft.com/en-us/windows/win32/seccrypto/example-c-program-encrypting-a-file.

[92] Jarvis K. Cryptolocker ransomware. Technical report, Secureworks; 2013, URL: https://www.secureworks.com/research/cryptolocker-ransomware.

[93] Breiman L, Friedman J, Stone CJ, Olshen RA. Classification and regression trees. CRC Press; 1984.

[94] Ferrie P. The ultimate anti-debugging reference. Tech. rep, 2011.

[95] Ligh M, Adair S, Hartstein B, Richard M. Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code. Wiley Publishing; 2010.

[96] Malin CH, Casey E, Aquilina JM. Malware forensics: Investigating and analyzing malicious code. Syngress; 2008.

[97] Rhode M, Tuson L, Burnap P, Jones K. LAB to SOC: Robust features for dynamic malware detection. In: 2019 49th annual IEEE/IFIP international conference on dependable systems and networks – industry track. 2019, p. 13–6.