Dynamic Analysis of Malware Using Run-Time Opcodes

Domhnall Carlin, Philip O'Kane, and Sakir Sezer

Abstract The continuing fight against intentionally malicious software has, to date, favoured the proliferators of malware. Signature detection methods are growingly impotent against rapidly evolving obfuscation techniques. Research has recently focussed on the low-level opcode analysis of disassembled executable programs, both statically and dynamically. While able to detect malware, static analysis often still cannot unravel obfuscated code; dynamic approaches allow investigators to reveal the run-time code. Old and inadequately sampled datasets have limited the extrapolation potential of much of the body of research. This work presents a dynamic opcode analysis approach to malware detection, applying machine learning techniques to the largest dataset of its kind, both in terms of breadth (610–100k features) and depth (48k samples). N-gram analysis of opcode sequences from $n = 1..3$ was applied as a means of enhancing the feature set. Feature selection was then investigated to tackle the feature explosion which resulted in more than 100,000 features in some cases. As the earliest detection of malware is the most favourable, run-length, i.e. the number of recorded opcodes in a trace, was examined to find the optimal capture size. This research found that dynamic opcode analysis can detect malware from benignware with a 99.01% accuracy rate, using a sequence of only 32k opcodes and 50 features. This demonstrates that a dynamic opcode analysis approach can compare with static analysis in terms of speed. Furthermore, it has a very real potential application to the unending fight against malware, which is, by definition, continuously on the back foot.

D. Carlin (✉) • P. O'Kane • S. Sezer
Centre for Secure Information Technologies, Queen's University, Belfast, Northern Ireland, UK
e-mail: dcarlin05@qub.ac.uk; p.okane@qub.ac.uk

I. Palomares et al. (eds.), *Data Analytics and Decision Support for Cybersecurity*,
Data Analytics, DOI 10.1007/978-3-319-59439-2_4

1 Introduction

Since the 1986 release of *Brain*, regarded as the first PC virus, both the incessant proliferation of malware (**mal**icious soft**ware**) and the failure of standard anti-virus (AV) software, have posed significant challenges for all technology users. In the second quarter of 2016, McAfee Labs reported that the level of previously unseen instances of malware had exceeded 40 million for the preceding three months. The total number of *unique* malware samples reached 600 million, a growth of almost one-third on the previous year [4].

Malware was previously seen as the product of miscreant teenagers, whereas it is now largely understood to be a highly effective tool for criminal gangs and alleged national malfeasance. As a tool, malware facilitates a worldwide criminal industry and enables theft and extortion on a global scale. One data breach in a US-based company has been estimated to cost an average $5.85 million, or $102 for each record compromised [1]. GDP estimates which attribute the cost of cybercrime to modern developed countries are as much as 1.6% in Germany, with other technology-heavy nations showing alarming levels (USA—0.64% and China—0.63%) [4].

This chapter is presented in the following way: the remainder of the current section describes malware, techniques for detecting malicious code and subsequent counter-attacks. Section 2 presents related research in the field and the gap in the current body of literature. Section 3 depicts the methodology used in establishing the dataset and Section 4 presents the results of machine learning analyses of the data. The final section discusses the conclusions which can be reached about the results presented, and describes potential future work.

1.1 Taxonomy of Malware

Malware can be categorised according to a variety of features, but typically according to its family type. Bontchev, Skulason and Solomon in [8] established naming conventions for malware, which gave some consideration to the taxonomy of such files. With the rapid advancement of malware, this original naming convention required updating to cope with the new standards in developing malware [15]. The standard name of a unique piece of malware is generally comprised of four parts, as seen in Fig. 1, although the format presented (delimiters etc) varies widely according to the organisation.

The pertinent parts of the name are *Type* and *Family*, as these provide the broad delineations between categories. In the original naming standard [8], noted that it was virtually impossible to adequately formally define a malware family, but that structurally similar malware variants could be grouped as a family e.g. MyDoom, Sasser etc. Malware is categorised according to *Type*, when the behaviour of the malware and resultant effect on the victim system is considered.

$$Type.Platform.Family\text{-}Name.Minor\text{-}Variant : Modifier$$

$$e.g.Trojan.Win32.Crilock.B :$$

= A Windows-based Trojan, and the second variant found from the Crilock family

Fig. 1 Naming convention for malware

Such types include:

- *Backdoor*: installed onto the victim machine, allowing the attacker remote access;
- *Botnet*: as per a *back door*, but the victim machine is one of a cluster of breached machines linked to a command-and-control server;
- *Downloader*: preliminary code which serves to download further malicious code;
- *Information-stealing malware*: code which scrapes information from the victim machine and forwards it to an intended source e.g. banking info;
- *Launcher*: used to launch other malicious programs through non-legitimate methods in order to maintain stealth;
- *Rootkit*: allows access to unauthorised areas of a machine and designed to conceal the existence of other malcode;
- *Scareware*: often imitates an anti-virus (AV) alert, designed to scare an uninformed user into purchasing other software;
- *Spam-sending malware*: causes the victim machine to send spam unknowingly;
- *Virus*: code that can copy itself and infect additional computers, but requires a host file;
- *Worm*: malware which self-propagates via network protocols;
- *Trojan*: malware which is normally dressed up as innocent software and does not spread by itself, but allows access to the victim machine;
- *Ransomware*: malware which encrypts the victim machine's files, extorting money from the victim user for their release;
- *Dropper*: designed to bypass AV scanners and contains other malware which is unleashed from within the original code, after the installation of the original Dropper.

As malware very often overlaps categories, or implements several mechanisms, a clear taxonomy system can be a difficult task [25]. Deferring then to *family* as the discrete nominal attribute can provide an adequate definition of the malware being referenced. For example, the well-publicised and highly-advanced *Stuxnet* cyber-weapon, which was used to attack and disrupt the Iranian nuclear program, spans multiple categories of malware. The initial attack vector was through an infected USB drive, which contains the worm module and a *.lnk* file to the worm itself. Stuxnet exploits several vulnerabilities, one of which allows automatic execution of *.lnk* files on USB drives. The worm module contains the routines for the payload and a third module, a rootkit, hides the malicious activities and codes from the end-user [42]. As this is a multifaceted attack, it could be difficult to adequately describe the

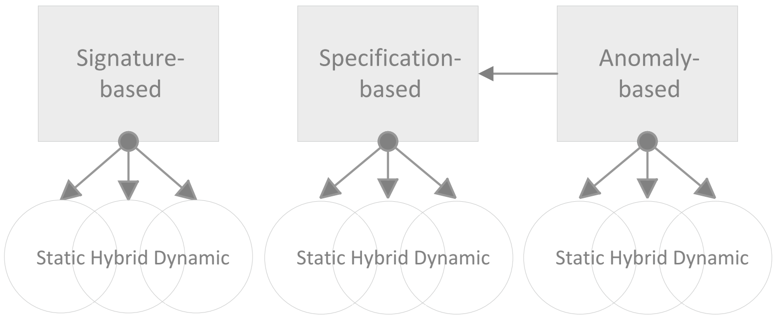


Fig. 2 Malware detection methods

type of the malware using a traditional approach. However, referring to the *family* variable enables a descriptive reference to the whole malware.

1.2 Malware Detection Techniques

The evolution of malware detection techniques is analogous to an arms race against malware writers. When AV vendors deploy new methods to detect malware, the malware writers create new methods to usher their code past detection algorithms. Figure 2 provides a taxonomy of the standard malware detection techniques in use at present.

The main categories of analysis are outlined below:

1.2.1 Signature-Based

Signature-based detection relies on comparing the payload of a program against a repository of pre-learned regular expressions extracted from malware [14]. The extraction, storage and dissemination of these strings requires a large effort and a substantial amount of human time [13]. This detection is generally performed statically and so suffers from the major disadvantages of being ineffective against unknown regular expressions and being unable to search obfuscated code for such strings. This renders this technique increasingly impotent against an onslaught of exponentially growing malware.

1.2.2 Anomaly-Based

Anomaly-based detection systems classify what are considered typical behaviours for a file or system and any deviations from these are considered anomalies [41]. A training phase creates a model of typical behaviour or structure of the file or system. The monitoring phase then detects deviation from the baselines established in training [16]. For example, a poisoned PDF may differ greatly from the typical or expected structure of a clean PDF. Likewise, a one-hour TV show file may be expected to be around 350 MB in size. If the file was a mere 120 KB, it would be classed as suspicious. One of the major benefits of anomaly detection is the potential to detect zero-day threats. As the system is trained on typical traits, it does not need to know all of the atypical traits in advance of detection. Anomaly-based systems are, however, limited by propensity towards false-positive ratings and the scale of the features required to adequately model the system under inspection [16].

1.2.3 Specification-Based

Specification-based detection is a descendant of anomaly detection with a view to addressing the issue of high false-positive rates of the latter. This detection method seeks to create a rule set which approximates the requirements of the system, rather than the implementation of the system [41]. The adequate modelling of a large system is a difficult task and, as such, specification-based detection may have a similar disadvantage as to an anomaly-based system, in that the model does not adequately capture the behaviours of a complex system.

Each of these analysis types can be performed in one of three manners:

1.2.4 Static Analysis

Analysis is considered static when the subject is not executed and can be conducted on the source code or the compiled binary representation [13]. The analysis seeks to determine the function of the program, by using the code and data structures within [40]. MD5 hash representations of the executable can be checked against databases of hashes from previously detected malware. Call graphs can demonstrate the architecture of the software and the possible flows between functions. String analysis seeks out URL s, IP addresses, command line options, Windows Portable Executable files (PE) and passwords [29]. The main advantage of using static analysis techniques is that there is no threat from the malware sample, as it is never executed, thus can be safely analysed in detail. All code paths can potentially be examined, when the whole of the code is visible, whereas with dynamic analysis only the code being executed is examined. Examining all code paths is not always an advantage, as the information revealed may be redundant dead code, inserted to mask the true intentions of the file. Static analysis techniques are increasingly rendered redundant against growingly complex malware, which can obfuscate itself from analysis until the point of execution. As the code is never executed in static analysis, the obfuscation method does not always reveal the true code.

1.2.5 Dynamic Analysis

Dynamic analysis involves the execution of the file under investigation. Techniques mine information at the point of memory entry, during runtime and post-execution, therefore bypassing many of the obfuscation methods which stymie static analysis techniques. The dynamic environment can be: (a) *native*: where no separation between the host and the analysis environment is provided, as analysis takes place in the native host OS. With malware analysis, this obviously creates issues with the effects of the malware on the host. Software is available to provide a snapshot of the clean native environment, which can then be reloaded following each instance of malware execution, such as DeepFreeze. This can be quite a timely process however, as the entire system must be restored to the point of origin [23]; (b) *emulated*: where the host controls the guest environment through software emulating hardware. As this software is driven by the underlying host architecture, emulation can create performance issues; (c) *virtualised*: a virtual machine provides an isolated guest environment controlled by the host. In a perfect environment, the true nature of the subject is revealed and examinable. However, in retaliation, malware writers have created a battery of evasion techniques in order to detect an emulated, simulated or virtualized environment, including anti-debugging, anti-instrumentation and anti-Virtual Machine tactics. One major disadvantage of dynamic approaches is the runtime overhead in having to execute the program for a specified length of time. Static tools such as IDAPro can disassemble an executable in a few seconds, whereas dynamic analysis can take a few hundred times longer.

1.2.6 Hybrid Analysis

Hybrid techniques employ components of both static and dynamic analysis in their detection algorithms. In [34], Roundy and Miller used parsing to analyse suspect code statically pre-execution, building an analysis of the code structure, while dynamic execution was used for obfuscated code. The authors established an algorithm to intertwine dynamic and static techniques in order to instrument even obfuscated code.

1.3 Detection Evasion Techniques

Code obfuscation is the act of making code obscure or difficult to comprehend. Legitimate software houses have traditionally used obfuscation to try to prevent reverse engineering attempts on their products. This is the foundation of approaches such as Digital Rights Management, where emails, music etc are encrypted to prevent undesired use. Malware writers adopted these techniques to try to avoid detection of their code by AV scanners, in a similar fashion. In [11] the four most commonly used obfuscation techniques are listed as;

1. *Junk-insertion*: inserts additional code into a block, without altering its end behaviour. This can be dead code, such as *NOP* padding, irrelevant code, such as trivial mathematical functions and complex code with no purpose;
2. *Code transposition*: reorders the instructions in the code so that the resultant binary differs from the executable or from the predicted order of execution;
3. *Register reassignment*: substitutes the use of one register for another in a specific range;
4. *Instruction substitution*: takes advantage of the concept that there are many ways to do the same task in computer science. Using an alternative instruction to achieve the same end result provides an effective method for altering code appearance.

Packing software, or packers, are used for obfuscation, compression and encryption of a PE file, primarily to evade static detection methods. When loaded into RAM, the original executable is restored from its packed state in ROM, ready to unleash the payload on the victim machine [5].

Polymorphic (*many shapes*) malware is comprised of both the payload and an encryption/decryption engine. The polymorphic engine mutates the static code/-payload, which is decrypted at runtime back to the original code. This can cause problems for AV scanners, as the payload never appears the same in any descendant of the original code. However, the encryption/decryption engine potentially remains constant, and as such may be detected by signature- or pattern-matching scanners [26]. A mutation engine may be employed to obfuscate the decryption routine. This engine randomly generates a new decryption routine on each iteration, meaning the payload is encrypted and the decryptor is altered to appear different to the previously utilized decryptors [41].

Metamorphic (*changes shapes*) malware takes obfuscation a step further than polymorphism. With metamorphic malware, the entire code is rewritten on each iteration, meaning no two samples are exactly alike. As with polymorphic malware, the code is semantically the same, despite being different in construction or bytecode, with as few as six bytes of similarity between iterations [21]. Metamorphic malware can also use inert code excerpts from benignware to not only cause a totally different representation, but to make the file more like benignware, and thus evade detection [20].

1.4 Summary

The ongoing fight against malware seeks to find strategies to detect, prevent and mitigate malicious code before it can harm targeted systems. Specification-based and Anomaly-based detection methods model whole systems or networks for a baseline, and report changes (anomalous network traffic, registry changes etc), but are prone to high levels of false detections. Signature-detection is the most commonly used method among commercial AV applications [43]. With the ever-apparent evolution of malware, such approaches have been shown to be increasingly ineffective in the detection of malicious code [31]. This is particularly apparent in Zero Day exploits, where the time-lag between the malware's release and the generation of a signature leaves a significant window for the malcode to cause substantial damage. As these methods rely on a pre-learned signature, and as the ability of modern malware to obfuscate such signatures grows, signature-based AV approaches are by definition on a continuous back foot. Therefore, a new approach is required to be able to more accurately and efficiently detect malware which is unseen to the detection algorithm.

2 Related Research

Obfuscation has frustrated attempts at effective and efficient malware detection, both from a commercial application viewpoint and in a research context. Previous research has sought to find a successful malware detection algorithm by applying machine learning and data mining approaches to the problem. Schultz et al. [38] were the first researchers to present machine learning as a tool to investigate malicious binaries. Three classifiers (RIPPER, Naïve Bayes and Multi-Naïve Bayes) were implemented and compared to a signature-based algorithm, i.e. an AV scanner, using *program header*, *strings* and *byte sequences* as features. All of the machine learning approaches out-performed the AV scanner, with two of the classifiers doubling the accuracy of the commercial approach. Kolter and Maloof [20] successfully used a binary present/not present attribute for n-gram analysis of the hexadecimal representations of malware PE files. Among other classifiers assessed, Boosted Decision Trees gave an area under the receiver operating characteristic (AU-ROC) curve of 99.6%. In [27], features were used based on a host system to detect previously unseen worms. Data were collected on 323 features per second, focussing on the configuration, background activity and user activities of the host. Five worms were compared against a baseline behavioural set up, and with as few as 20 features, the average detection accuracy was >90%, with >99% for specific individual worms.

2.1 Opcode Analysis

Recent investigations into malware detection approaches have examined the behaviour of malware at run time, i.e. *what* the code is doing rather than *how* it is doing it, and how these behvaviours differ from benign code. *Opcodes* (**op**erational **codes**) are human-readable machine language instructions, issued directly to the CPU. Analysis at such a low-level provides the opportunity to detect malware, circumventing obfuscation at run time [31].

In [36], Santos et al. analysed n-gram combinations of opcodes in statically-generated datasets. An accuracy and an f-measure (the weighted average of precision over recall) of over 85% were observed using a ROC-SVM classifier, although n was limited to 2.

A graph-analysis approach was taken by [35] and applied to PE files and metamorphic malware, in an extension of [6]. Their model was able to detect metamorphic malware from benign applications, and metamorphic malware families from each other.

The authors in [31] analysed opcodes from a dynamically-yielded dataset using supervised machine learning. Run-traces of benign and malicious executables were taken by executing the applications using a virtual machine. This controlled environment allowed isolation of the guest OS, decoupling the effects of the malware from the host machine. A debugging program (Ollydbg), in conjunction with a masking tool (StrongOD), was used to capture the run-time trace of each program under investigation. This provided a raw trace file with each opcode and its operand, and further details such as a memory register address. A bespoke parser was used to extract the opcodes from this trace file and count their occurrences. The density of each opcode was calculated based on the frequency of the opcode within that sample. Different run-lengths were addressed by normalizing the opcode frequency using the total count within the sample. A pre-filter was used to select the features likely to provide the maximum information to the SVM, while minimizing the overall size problem created by use of n-gram analysis to investigate all combinations of opcodes. Principal components analysis (PCA) was used to rank each opcode by its importance to the SVM classification task. PCA compresses the data size while maintaining the variance in the data, allowing a subset of principal components to be generated. The researchers found that 99.5% of the variance in the data could be attributed to the top 8 opcodes, thus reducing the data from the original 150 opcodes, which was further validated by use of an SVM. However, the dataset used was only comprised of 300 benign and 350 malware PEs. Although previous research had fewer samples (e.g. [7]), low sample quantity and coverage have been recurring issues throughout the literature.

In [38], Schultz et al. used 3265 malware and 1001 benignware executables in the first study in this branch of malware literature. Santos et al. [37] considered 2000 files, evenly divided between classes. Both [28] and [39] used the same dataset, comprised of 7688 malicious files and 22,735 benign, listed by the researchers as the largest dataset in the literature at that time. However, there are notable limitations in the data collection methodology from these studies. The researchers stated that the disassembler used could only analyse 74% of their samples. Malware showed a greater attrition rate (26%) than benignware (10%), as the main reason for failure was the compression or packing of a file (i.e. obfuscation).

The dataset used by [36] contained 2000 files, due to unspecified technical limitations. The malware was randomly sampled from the 17,000-strong corpus of the VxHeaven website [2], though the static analysis used did not include packed files. The implication is that more sophisticated obfuscated malware was not used in the investigation, a considerable limitation of the study. Furthermore, >50% of the sample set was composed of three malware families: *hack-tool*, *back door* and *email worm*, which may have induced artificial within-class imbalances.

6721 malware samples were used in [17], harvested from VxHeaven and categorized into three types (*back door*, *worm*, and *trojan*), comprised of 26 malware families and >100 variations per family. No controls were used in the sampling across categories, with 497 worm, 3048 backdoor, and 3176 trojan variants, nor was there a methodological statement as to how these categories were sampled.

2.2 Rationale for the Current Research

Previous research in the field has typically used samples taken from the VxHeaven website, which are old and outdated, having last been updated in 2010. There are clear and notable limitations in the methodologies employed, with the majority having datasets which were not adequately sized, structured or sampled. Datasets which were generated statically failed to adequately address any serious form of obfuscation, with some just initially excluding any packed sample. While dynamic analysis does have disadvantages, it does offer a snapshot of the malware behaviour at runtime, regardless of obfuscation, and has been recommended by researchers who had previously focussed on static analysis [36]:

> Indeed, broadly used static detection methods can deal with packed malware only by using the signatures of the packers. As such, dynamic analysis seems like a more promising solution to this problem ([18]) [36, p.226]

3 Methodology

3.1 Source Data

Research in this field has typically used online repositories such as VxHeaven [2] or VirusShare [33]. The corpus of the former is dated, with the last updates being in 2010. VirusShare, however, contains many more samples, is continuously updated, and offers useful metadata. The site receives malware from researchers, security teams and the public, and makes it available for research purposes on an invite-only basis. The three most-recent folders of malware, at the time of writing, were downloaded, representing approximately 195,000 instances of malware, of varying formats. The Malicia dataset harvested from drive-by-downloads in the wild in [30] was also obtained and incorporated into the total malware corpus. Benign software was harvested from Windows machines, representing standard executables which would be encountered in a benchmark environment.

3.2 Database Creation

As the files in the malware corpus were listed by MD5 hash, with no file format information, a bespoke system was built to generate a database of attribute metadata for the samples. Consideration was given to using standard **nix* commands such as *file* to yield the file formats. However, this did not provide all information necessary, and so a custom solution was engineered. Figure 3 illustrates the system employed.

A full-feature API key was granted by VirusTotal for this pre-processing stage. This allows a hash to be checked against the VirusTotal back-end database, which includes file definitions and the results of 54 AV scanners on that file. The generated report was parsed for the required features (Type, Format, First Seen, 54 AV scans etc) and written to a local database for storage. A final step in the attribution system allows the querying of the local database for specific file types and for files to be grouped accordingly. For the present research, we were interested in Windows PE files and our initial dataset yielded approximately 90,000 samples. Other file types (Android, PDF, JS etc) were categorized and stored for future research.

On creation of the local attribute dataset, there was a noticeable variation in whether each AV scanner determined that the specific file was malicious or not and also the designated malware type and family. This presented a challenge to our methodological stance. If a file is detected by very few of the 54 AV scanners, it could be viewed as a potential false positive (i.e. detected as malicious, though truly benign), or indeed that it is an instance of malware which can successfully evade being detected by most of the AV scanners. False positive detections can occur when a scanner mistakenly detects a file as malicious. This is exacerbated by the use of fuzzy hashing, where hash representations of a file are seen as similar to a malicious file, and the fact that scanning engines and signature databases are often shared by more than one vendor. The false positive issue can have serious consequences, such as rendering an OS unusable. It has become problematic enough that VirusTotal has attempted to build a whitelist of applications from their trusted sources, so that false positives can be detected and mitigated [24]. As such, we

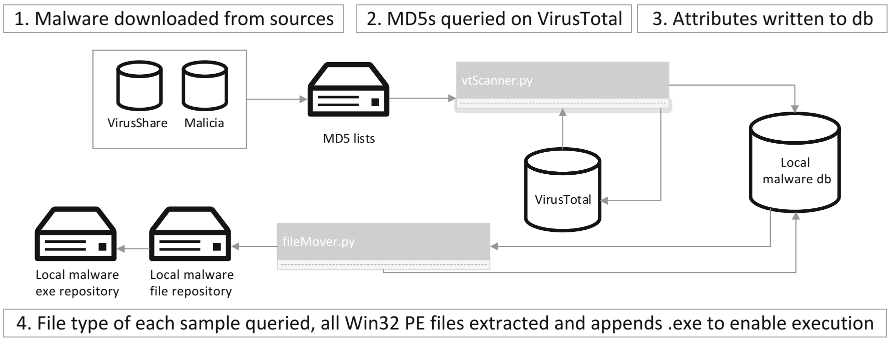

1. Malware downloaded from sources
2. MD5s queried on VirusTotal
3. Attributes written to db
VirusShare
Malicia
MD5 lists
vtScanner.py
VirusTotal
Local malware db
fileMover.py
Local malware exe repository
Local malware file repository
4. File type of each sample queried, all Win32 PE files extracted and appends .exe to enable execution

Fig. 3 Processing malware samples to store their descriptive attributes

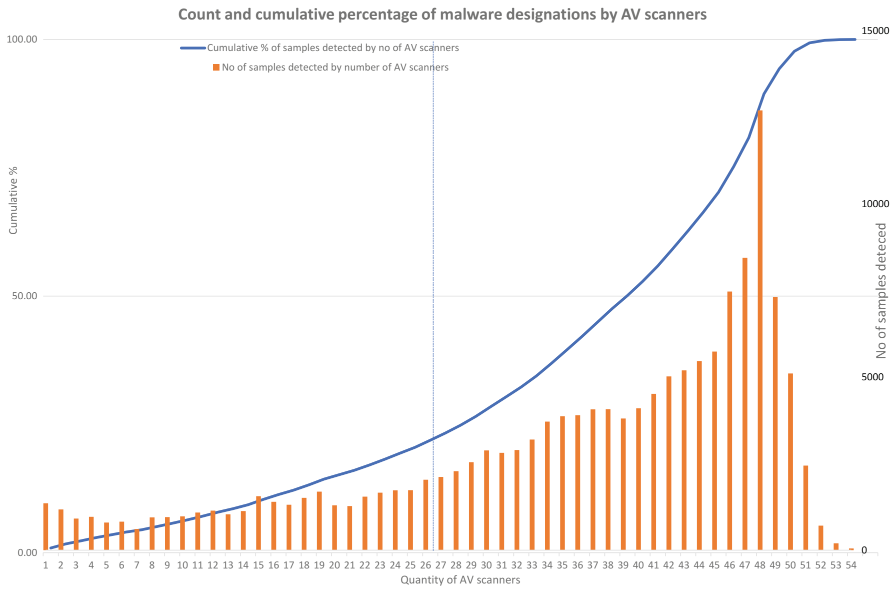


Fig. 4 The distribution of number of samples which were detected by the number of AV scanners (horizontal axis), with the cumulative percentage)

used a majority-rule algorithm and focussed on instances which were assessed as malware by 50% or more of the AV scanners. This can be justified by the fact that we wished to establish a fundamental malware dataset, with established malware being given priority over questionable samples. Indeed, the fact that there is disagreement between AV scanners over samples highlights one of the key issues in the field.

Figure 4 depicts the distribution of samples which had a specified number of AV scanners deem the sample as malicious. For example, 964 samples were judged to be malicious by 4 AV scanners from the VirusTotal report (right vertical axis), whereas 12,704 were detected by 48 scanners. Cumulatively, this represented 3% (left vertical axis) of the malware being detected by 4 or fewer scanners, and 89% by 48 or fewer. In Fig. 4, the central bisecting line represents the threshold after which malware was chosen for the experimental dataset, i.e. 26 or more of the AV scanners recognised the sample as malicious. This threshold demonstrates that 18% of the dataset was not included in the experimental dataset, by choosing to operate with a majority-rules decision. In other words, 82% of the malicious samples were retained while having the certainty of a rigorous inclusion mechanism. As the purpose of these experiments was to build a solid model of the dynamic behaviour of malware, the importance of widely agreed-upon samples for training purposes was a priority.

3.3 Automated Run-trace Collection

A process for extracting opcode run-traces from executed malware was established in [31, 32]. This generates the required traces, but is manually set up and operated, and so is unfeasible for generating a large dataset. As such, automated scalable processes were given consideration.

The work in [23] shows practical automated functions for use with VirtualBox virtual machines, though the methods used are now deprecated. A virtualization test-bed which resides outside of the host OS was developed by [12]. The authors claimed the tool was undetectable when tested by 25,118 malware samples, though no breakdown of the dataset is provided. However, the software was last updated in October 2015, is only compatible with Windows XP and requires a deprecated version of the Xen hypervisor, and so was not implemented.

The ubiquitous Cuckoo sandbox environment was considered also. While a valuable tool in malware and forensic analysis, we sought a system which provided the exact output for the method established in [31, 32]. Furthermore, we wish to develop an opcode-focussed malware analysis environment, and so chose to engineer a bespoke automated execution and tracing system.

VirtualBox was implemented as the virtualization platform. While other virtualization software exists, such as VirtualPC, Xen and VMWare, the API for VirtualBox is feature-rich and particularly suited to the present research. VirtualBox comprises multiple layers on top of the core hypervisor, which operates at the kernel level and permits autonomy for each VM from the host and other VMs. The API allows a full-feature implementation of the VirtualBox stack, both through the GUI and programmatically, and as such is well-documented and extensively tested [3].

A baseline image was created of Windows 10 64-bit, 2 GB RAM, VT-x acceleration, 2 cores of an Intel i5 5300 CPU, and the VBoxGuestAdditions v4.3.30 add-on installed. Windows 10 was selected as the latest guest OS due to the modernity and market share. VBoxGuestAdditions allows extra features to be enabled inside the VM, including folder-sharing and application execution from the host, both of which are key features of the implementation. Full internet access was granted to the guest and traffic was partially captured for separate research, and monitored as a further check for the liveness of the malware. Security features were minimised within the guest OS to maximise the effects of the executed malware.

OllyDbg v2 was preinstalled in the VM snapshot to trace the runtime behaviour of each executed file. This is an open-source assembler-level debugger, which can directly load both PE and DLL files and then debug them. The presence of a debugger can be detected by both malicious and benign files, and such instances may deploy masking techniques. As per [31], we used StrongOD v0.4.8.892 to cloak the presence of the debugger.

The guest OS was crafted to resemble a standard OS, with document and web histories, Java, Flash, .Net, non-empty recycling bin etc. While anti-anti-virtualization strategies were investigated, and implemented where appropriate, not all could be achieved while maintaining essential functionality, such as remote

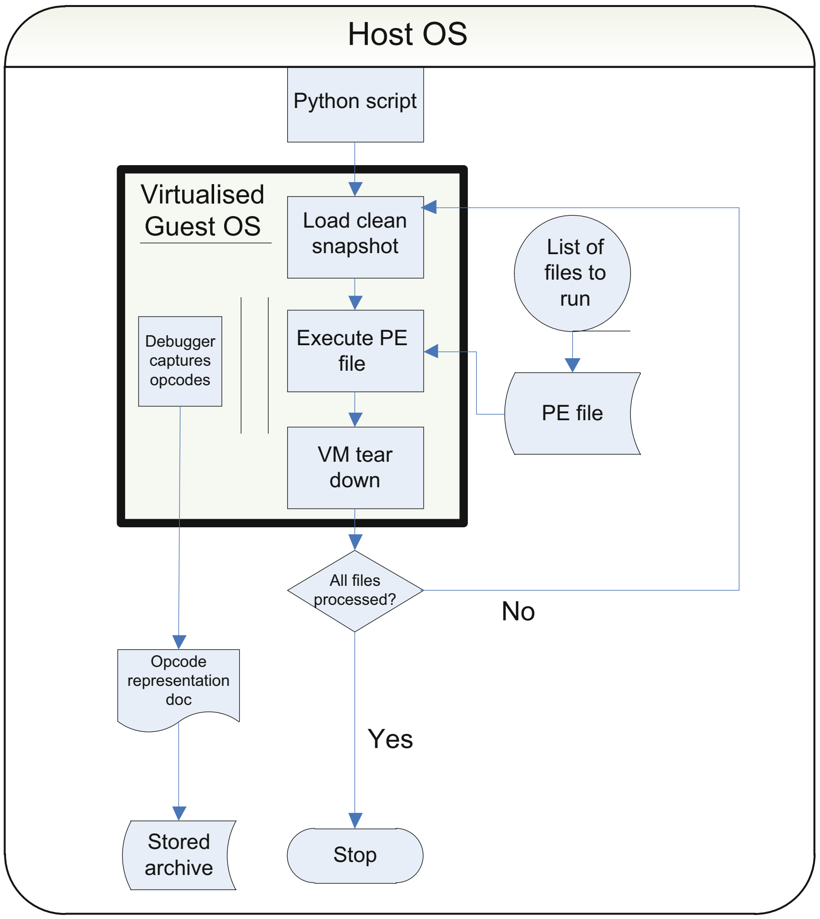
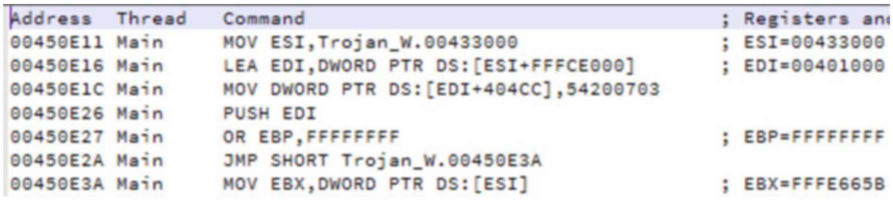


Fig. 5 Automated data collection system model

execution. However, while operating at the opcode level, we get the opportunity to monitor anti-virtualization behaviour and can turn this into a feature for detection. Furthermore, this research seeks to investigate the execution of malware as a user would experience it, which would include cloud-based virtualized machine instances. Figure 5 illustrates the workflow of the automated implementation used for the present research. On starting, a user-specified shared folder is parsed for all files which are listed for execution. The VM is spun-up using the baseline snapshot, mitigating any potential hang-overs from previously executed malware. OllyDbg is loaded with the masking plug-in and the next file from the execution list as a parameter. The debugger loads the PE into memory and pauses prior to execution. The debugger is set to live overwrite an existing empty file with the run-trace of opcode instructions which have been sent to the CPU, set at a trace-into level so that every step is captured. This is in contrast to static analysis, which only yields



```
Address   Thread    Command                               ; Registers an
00450E11  Main      MOV ESI,Trojan_W.00433000             ; ESI=00433000
00450E16  Main      LEA EDI,DWORD PTR DS:[ESI+FFFCE000]   ; EDI=00401000
00450E1C  Main      MOV DWORD PTR DS:[EDI+404CC],54200703
00450E26  Main      PUSH EDI
00450E27  Main      OR EBP,FFFFFFFF                       ; EBP=FFFFFFFF
00450E2A  Main      JMP SHORT Trojan_W.00450E3A
00450E3A  Main      MOV EBX,DWORD PTR DS:[ESI]            ; EBX=FFFE665B
```

Fig. 6 Sample lines from a runtime trace file

potentially executed code and code-paths. Figure 6 shows an example run-trace file which has been yielded. After the set run-length, which was set to 9 min, elapsed, the VM is torn down and the process restarts with the next file to be executed until the list is exhausted. A 9 min execution time was selected, with 1 min for start up and teardown, as we are focussed on detecting malware as accurately as possible, but within as short a runtime as possible. While a longer runtime may potentially yield richer trace files and more effectively trace malware with obfuscating sleep functions, our research concentrates on the initial execution stage for detection processes.

The bottleneck in dynamic execution is run-time, so any increase in capture rate will be determined by parallelism while run-time is held constant. Two key benefits of the implementation presented are the scalability and automated execution. The initial execution phase was distributed across 14 physical machines, allowing the dataset to be processed in parallel. This allowed the creation of such a large dataset, in the context of the literature, within a practical period of time. Indeed, this was only capped at the number of physical machines available. It is entirely feasible to have a cloud- or server- based implementation of this tracing system, with a virtually unlimited number of nodes.

3.4 *Opcode Statistics Collection*

Prior to analysis, the corpus of trace files require parsing to extract the opcodes in sequence. A bespoke parser was employed to scan through each run-trace file and keep a running count of the occurrence of each of the 610 opcodes listed in the Intel x86/x64 architecture [22]. This list is more comprehensive than that of [31, 32], as previous research has indicated that more rarely occurring opcodes can provide better discrimination between malware and benignware than more frequently occurring opcodes [7, 17].

As the run-traces vary in length (i.e. number of lines), a consistent run-length is needed for the present research in order to investigate the effects of run-length on malware detection. A file slicer was used on the run-trace dataset to truncate all files to the maximum required in order to control run-length. As partly per [31, 32],

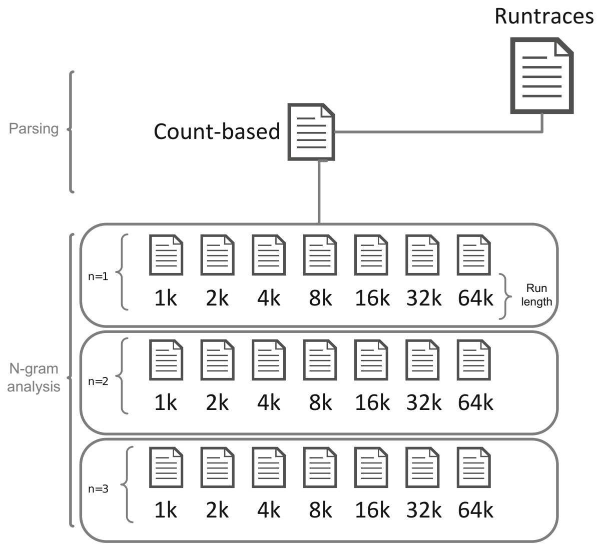


Fig. 7 Derived datasets

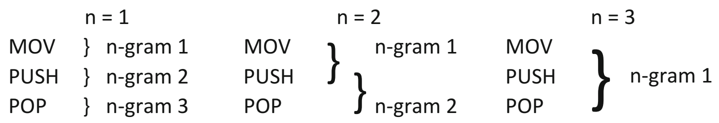


Fig. 8 Sample n-gram combinations

run-lengths were capped at 1, 2, 4, 8, 16, 32 and 64 thousand opcodes in sequence, along with the non-truncated set enumerated in Fig. 7.

The count for each trace file is appended to the overall dataset and stored as a CSV file for analysis.

As the behaviour of the malware at runtime is represented textually, it can be seen as a natural-language problem, with a defined dictionary. N-grams are used in text processing for machine learning, which can equally be applied to the problem at hand. Previous research has shown that n-gram analysis can be used to discriminate malicious and benign applications, though datasets have been severely limited. As such, this paper focuses on n-gram analysis to investigate classification of malware on a substantial dataset.

A second separate phase of the opcode counting process then parses the run-traces for bi-grams and trigrams from the reduced list ($n = 2..3$). The system removes all opcodes which did not occur, as with the increase in n, the possible combinations rise exponentially. As the opcodes which did not occur at $n = 1$ cannot be in combinations of $n = 2$ etc, this can mitigate the feature explosion without losing any data. This is depicted in Fig. 8.

Table 1 Composition of dataset

Malware category	*Samples*
Adware	592
Backdoor	1031
BrowserModifier	721
Malicia	8242
Null	5491
PWS	1964
Ransom	76
Rogue	498
SoftwareBundler	630
Trojan	6018
TrojanDownloader	3825
TrojanDropper	1619
TrojanSpy	405
VirTool	2206
Virus	6341
Worm	7251
Benign	1065
	47,975

From initial download through to execution, the number of samples was depleted by approximately 20%. A further 10 % of files failed to provide a run-trace for a variety of reasons, such as missing dlls or lock-outs (Table 1).

The quantity of samples and features in any large dataset can provide methodological challenges for analyses, which was pertinent in the present study. For example, with the $n = 3$ *64k run-length* dataset, there were 47,975 instances, with 100,318 features, or 4.82 *billion* data points. When contained within a CSV format, the file size approaches 9.4 GB. All analyses were conducted on a server-class machine with a 12-core Intel Xeon CPU and 96 GB of RAM. Despite this computational power, datasets of such magnitude are problematic, and so a sparse-representation file format was chosen (i.e. zeros were explicitly removed from the dataset, implied by their omission). This reduced the dataset file sizes by as much as 90%, indicating a large quantity of zero entries, and so feature reduction was investigated, as discussed further below.

3.5 Application of Machine Learning

All malware types were merged into a single malware category and compared to the benign class. Due to the size imbalance, a hybrid over/under/sub- sampling approach was taken to balance the classes. The minority (benign) class was synthetically oversampled with the Synthetic Minority Oversampling Technique (SMOTE) [10]. This approach uses a reverse-KNN algorithm to generate new instances with values

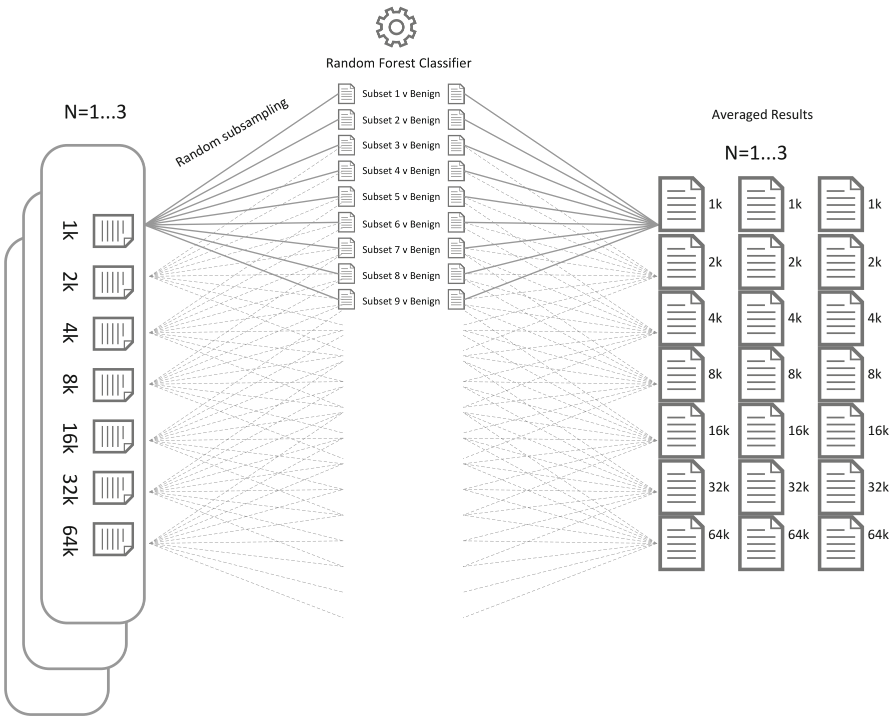


Fig. 9 Sampling approach

based on the values of 5 nearest neighbours, generating a 400% increase in benign sample size in our dataset. The majority class was then sub-sampled randomly into ninefolds, with a matching algorithm ensuring datasets were of equal size. Figure 9 illustrates this approach. This ensures that the training model was built with data which were balanced between the two classes, to try to mitigate negative impacts on model performance which imbalanced data can have.

The Random Forest (RF) [9] classification algorithm was used, as implemented in WEKA 3.9, to classify the data in all experiments. RF is an ensemble learner, combining the decisions of multiple small learners (decision trees). RF differs from traditional tree-based learning algorithms, as a random number of features are employed at each node in the tree to choose the parameter, which increases noise immunity and reduces over-fitting. RF functions well with datasets containing large numbers of instances and features, particularly with wide datasets (i.e. more features than instances). It parallelizes well [9], and is also highly recommended with imbalanced data [19]. In pilot experiments, RF provided high accuracy, despite class imbalances and short run-lengths, even with the large number of features being used. Tenfolds cross-validation was employed on the classification of each subset, and the average of the results over the nine subsets was taken to produce the overall results. Parameter selection for *number of trees* and *number of features* was conducted initially by a coarse grid search, followed by a fine grid search. For both parameters higher is better, especially over large datasets, though classification will plateau and a greater computational overhead penalty will be enacted.

4 Results

Standard machine learning outputs were yielded in the evaluation of the classifier for each dataset:

1. Accuracy, i.e. the percentage of correctly assigned classes (0% indicating a poor classifier, 100% showing perfect accuracy)

$$ACC = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TotalPopulation}}$$

2. F_1 score is the harmonic mean of precision and sensitivity (1 is the ideal).

$$F_1 = \frac{2\mathrm{TP}}{2\mathrm{TP} + \mathrm{FP} + \mathrm{FN}}$$

3. Area Under Receiver Operating Characteristic curve, i.e. when the True Positive % rate is plotted against the False Positive % rate, the area underneath this curve (1 is the ideal).
4. Area Under Precision Recall curve (1 is the ideal) where

$$\frac{Precision = \dfrac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}}{Recall = \dfrac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}}$$

4.1 *Effects of Run-Length and n-gram Size on Detection Rates*

The evaluations of each classification, prior to any feature reduction, are depicted in Fig. 10.

The overall accuracy across all 21 analyses had a mean of 95.87%, with a range of 93.01%–99.05%. The accuracy level declined in 3 of the 7 categories, between $n = 2$ and $n = 3$, though increased in 3 and remained stable in one. This indicates that there is marginal gain (0.06% on average) in increasing n from 2 to 3. The $n = 1$ cohort did, however, show an overall higher level of accuracy in all but one run-length (16k) and a greater mean by 1.29% vs $n = 2$ and 1.23% vs $n = 3$. F-scores ranged from 0.93 to 0.991, with the 3 n-gram means again showing greater

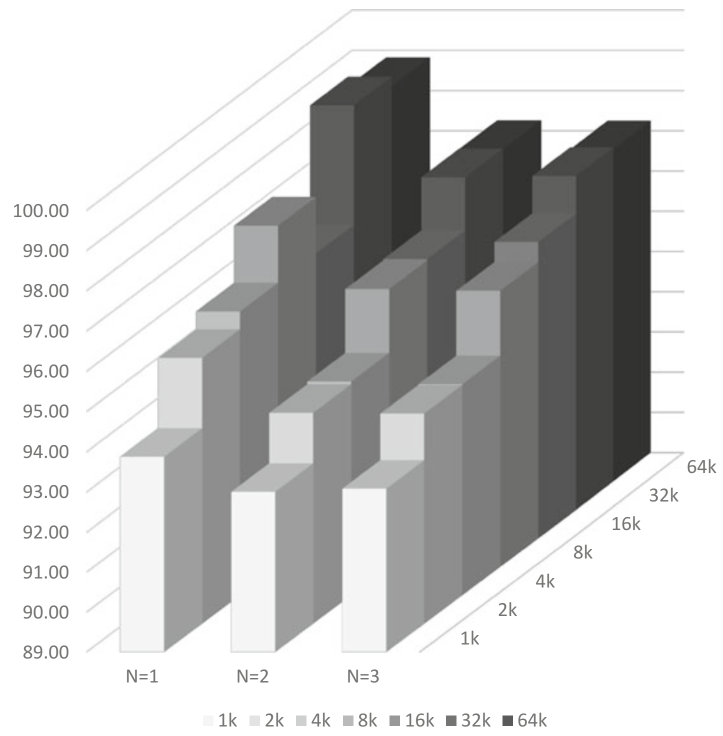


Fig. 10 Accuracies for each run-length across FS approaches, for $n = 1$

performance by $n = 1$, with marginal difference between $n = 2$ and $n = 3$. The AU-ROC showed high values, >0.979, with $n = 1$ from 0.99 to 0.999.

All measures increased as run-length increased, up to 32k. At 64k, all measures were either static or showed a marginal decline in performance, though most notably in accuracy (from 99.05% to 98.82%). Across all measures, the overall best performing strand was $n = 1$ at 32k run-length with an accuracy of 99.05%, f-score of 0.991 and AU-ROC of 0.999.

4.2 Feature Selection and Extraction

With the large quantity of features, attempts were made to reduce the number of attributes, while maintaining the detection capability of the model. Furthermore, it would not be computationally possible to employ feature extraction algorithms within a feasible time period, with such a large feature set. As such, a feature selection strategy was employed in the first instance.

Table 2 Quantity of features remaining per run-length for each FS category

	No FS			GR $\geq$0.01			Top 20			Top 50		
	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$
1k	610	1023	66,443	111	912	10,103	20	20	20	50	50	50
2k	610	6075	70,777	111	2448	12,270	20	20	20	50	50	50
4k	610	6358	75,907	115	2597	14,220	20	20	20	50	50	50
8k	610	6677	80,876	116	2734	15,804	20	20	20	50	50	50
16k	610	6956	85,693	135	2909	17,493	20	20	20	50	50	50
32k	610	7215	91,403	119	2947	19,533	20	20	20	50	50	50
64k	610	8135	100,316	139	3532	22,796	20	20	20	50	50	50

The Gain Ratio algorithm was employed to reduce the number of features in the datasets. Gain Ratio attempts to address the biases which can be introduced by Information Gain, when considering attributes with a large range of distinct values, by considering the number of branches that would result due to a split. In the ratio, the numerator represents the information yielded about the label and the denominator represents the information learned about the feature. The *ranker* search method was used to traverse the feature set. With respect to the current problem, *ranker* is advantageous as it is linear and does not take more steps than the number of features. With the breadth of the present datasets, this provides a saving of computational overhead.

Table 2 shows the quantity of features in each cohort and run-length at each step. The number of features rises in $n = 2$ and $n = 3$, and in each increase in run-length, as the potential number of observed opcodes rises.

Four levels of feature selection using Gain Ratio were investigated: no feature selection, Gain Ratio merit score >=0.01 (i.e. non-zero), top 20 features, and top 50 features. A further category was originally investigated (merit score >=10) however, this resulted in a model equivalent to random choice (i.e. all measures scored 50%). The same RF classifier was then presented with the reduced datasets for classification.

Removing features with a merit score of <0.01 had a negligible impact on the accuracy rating (0.013–0.018%) compared to the full feature set, but removed up to 84% of features.

Selecting the top *n* features was subsequently investigated, as the removal of non-zero features is unique to each dataset but *top n* would provide uniformity. The $n = 1$ dataset proved more resilient across top 20 and top 50 features than the $n = 2$ and $n = 3$ cohorts. This may be due to the difference in quantity of features from no reduction to top n, e.g. $n = 3$ 64k was reduced from 100,316 to 20 features in this approach. Figure 11 shows accuracy for $n = 1$.

With any malware detection model using machine learning, consideration should be given to the false positive rates (FP) (i.e. detected incorrectly as a specific class). This is particularly relevant with imbalanced data. Table 3 lists the false positive ratings for both the benign and malicious classes, per run-length and n-gram size.

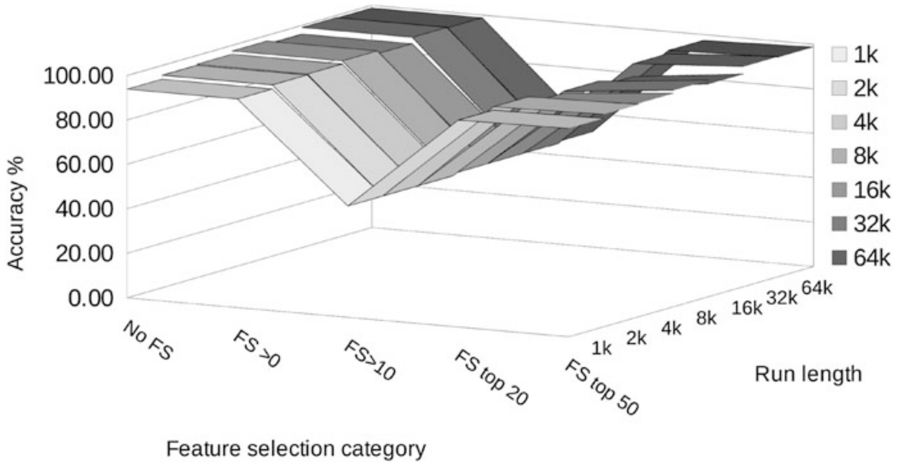


Fig. 11 Accuracies for each run-length across FS approaches, for $n = 1$

Table 3 False positive rates for each run-length for $n = 1..3$ using all features

	$n = 1$			$n = 2$			$n = 3$		
	Benign	Malicious	Mean	Benign	Malicious	Mean	Benign	Malicious	Mean
1k	0.042	0.081	0.061	0.029	0.111	0.070	0.031	0.107	0.069
2k	0.040	0.048	0.044	0.036	0.079	0.057	0.040	0.075	0.058
4k	0.046	0.032	0.039	0.046	0.068	0.057	0.044	0.071	0.057
8k	0.022	0.028	0.025	0.023	0.059	0.041	0.024	0.059	0.041
16k	0.215	0.027	0.121	0.011	0.044	0.027	0.020	0.052	0.036
32k	0.008	0.011	0.009	0.008	0.047	0.027	0.010	0.045	0.027
64k	0.013	0.011	0.012	0.022	0.060	0.041	0.011	0.044	0.028
Mean	0.055	0.034	0.044	0.025	0.067	0.046	0.026	0.065	0.045

The ‘Benign’ columns quantify the rate of the model labeling the sample incorrectly as benign, and vice versa for the ‘Malicious’ columns. Overall FP rates are low, though a range of values is evident, consistent with the other machine learning measures. Again, the $n = 1$ data performs slightly better than the other n-gram sizes (mean=0.044 vs means of 0.046 and 0.045 respectively). The 32k dataset shows the lowest false positive rates (benign 0.008 and malicious 0.011, giving a mean of <1%). This indicates the strength of the model for accurate detection, while controlling for false positives. Furthermore, the sampling approach taken to balance the training data multiple times appears to have reduced the risk of increased false positives with unbalanced datasets (Fig. 12).

The further metrics for the model are presented in Table 4. As run-length increases, model performance increases for the *No FS* and *merit>0* cohorts, with the exception of the anomalous 16k set. The $n = 2$ and $n = 3$ datasets decline in performance as run-length increases when a set number of features are used (*Top 20*

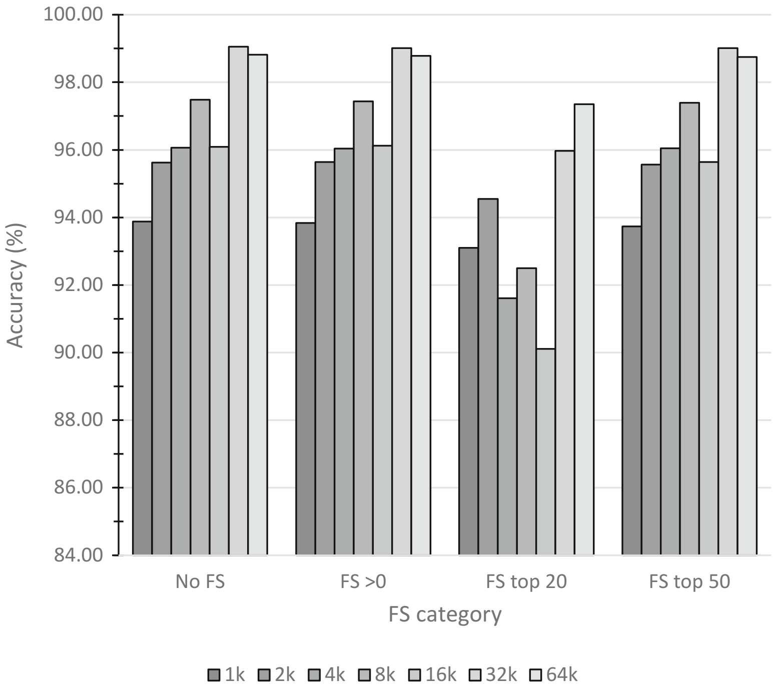


Fig. 12 Mean accuracies for each run-length across FS approaches for all n-gram lengths

and *Top 50*). In particular the AU-ROC scores show a decline as run-length increases for $n = 2$ using 20 features. An intuitive explanation would be due to the ratio of original to selected feature numbers. However, of the three n-gram sizes, $n = 2$ had the lowest average percentage loss of features after the GR>0 feature selection ($n = 1$: 80.19%, $n = 2$: 51.81%, $n = 3$: 80.67%). It had approximately 10 times fewer features than $n = 3$ in the original cohort, and so the reduction to a fixed 20 or 50 should have had more of an impact if feature ratio was the important factor. As AU-ROC measures discrimination, it would appear that the information for correctly discriminating between benign and malicious code is distributed differently when bi-grams are inspected.

5 Conclusions

With accuracy of 99.05% across such a large dataset, and using tenfold cross-validation, there is a clear indication that dynamic opcode analysis can be used to detect malicious software in a practical application. While the time taken by dynamic analysis can be a disadvantage, the work presented here shows that a trace with as few as 1000 opcodes can correctly discriminate malware from benign with

Table 4 Model metrics per run-length, n-gram length and FS method

No feature selection

	Accuracy (% correct)			F-score			ROC			PRC		
	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$
1k	93.88	93.01	93.09	0.939	0.93	0.931	0.99	0.983	0.979	0.99	0.983	0.979
2k	95.62	94.26	94.24	0.956	0.943	0.942	0.994	0.986	0.984	0.994	0.986	0.983
4k	96.07	94.32	94.26	0.961	0.943	0.943	0.994	0.987	0.984	0.993	0.986	0.984
8k	97.48	95.91	95.87	0.975	0.959	0.964	0.996	0.99	0.988	0.995	0.99	0.986
16k	96.09	95.94	96.37	0.961	0.959	0.964	0.991	0.991	0.988	0.99	0.991	0.986
32k	99.05	97.26	97.30	0.991	0.973	0.973	0.999	0.994	0.991	0.998	0.993	0.99
64k	98.82	97.25	97.25	0.988	0.973	0.972	0.998	0.993	0.99	0.998	0.993	0.989
Mean	96.72	95.42	95.48	0.97	0.95	0.96	0.99	0.99	0.99	0.99	0.99	0.99

After Feature Selection using GainRatio with all features showing merit >0

	Accuracy (% correct)			F-score			ROC			PRC		
	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$
1k	93.84	92.97	93.09	0.938	0.93	0.931	0.99	0.983	0.979	0.99	0.982	0.979
2k	95.64	94.26	94.24	0.956	0.943	0.942	0.994	0.986	0.984	0.994	0.986	0.983
4k	96.04	94.27	94.24	0.96	0.943	0.942	0.994	0.987	0.984	0.993	0.986	0.983
8k	97.44	95.88	95.81	0.974	0.959	0.958	0.996	0.99	0.987	0.995	0.99	0.986
16k	96.13	95.95	96.32	0.961	0.96	0.963	0.991	0.991	0.988	0.991	0.991	0.987
32k	99.01	97.29	97.30	0.99	0.973	0.973	0.999	0.993	0.991	0.999	0.993	0.99
64k	98.78	97.23	97.26	0.988	0.972	0.973	0.998	0.993	0.99	0.998	0.993	0.99
Mean	96.70	95.41	95.47	0.97	0.95	0.95	0.99	0.99	0.99	0.99	0.99	0.99

After Feature Selection using GainRatio with top 20 features

	Accuracy (% correct)			F-score			ROC			PRC		
	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$
1k	93.10	82.59	83.95	0.931	0.821	0.837	0.987	0.841	0.859	0.986	0.827	0.845
2k	94.54	83.27	82.28	0.945	0.829	0.818	0.989	0.843	0.828	0.988	0.832	0.815
4k	91.61	83.72	83.21	0.916	0.833	0.828	0.971	0.844	0.845	0.97	0.833	0.833
8k	92.50	81.54	81.90	0.925	0.809	0.813	0.972	0.814	0.82	0.971	0.797	0.807
16k	90.11	81.60	82.28	0.901	0.811	0.817	0.953	0.816	0.824	0.949	0.798	0.812
32k	95.97	62.92	82.73	0.96	0.57	0.822	0.987	0.627	0.827	0.986	0.629	0.812
64k	97.35	62.84	82.33	0.974	0.569	0.818	0.99	0.624	0.823	0.989	0.627	0.807
Mean	93.60	76.93	82.67	0.94	0.75	0.82	0.98	0.77	0.83	0.98	0.76	0.82

After Feature Selection using GainRatio with top 50 features

	Accuracy (% correct)			F-score			ROC			PRC		
	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$
1k	93.74	85.23	84.57	0.937	0.85	0.844	0.99	0.889	0.872	0.989	0.885	0.862
2k	95.57	86.03	82.80	0.956	0.858	0.824	0.994	0.884	0.837	0.993	0.878	0.825
4k	96.05	84.93	83.96	0.96	0.847	0.836	0.993	0.882	0.86	0.992	0.879	0.849
8k	97.39	84.65	82.47	0.974	0.844	0.82	0.995	0.869	0.827	0.995	0.862	0.815
16k	95.64	84.10	82.73	0.956	0.838	0.822	0.987	0.869	0.829	0.987	0.861	0.817
32k	99.01	85.98	82.79	0.99	0.857	0.823	0.999	0.89	0.828	0.998	0.881	0.813
64k	98.75	79.87	82.65	0.988	0.791	0.822	0.998	0.805	0.826	0.998	0.796	0.809
Mean	96.59	84.40	83.14	0.97	0.84	0.83	0.99	0.87	0.84	0.99	0.86	0.83

93.8% success. Increasing the run-trace to the first 32,000 opcodes increases the accuracy to over 99%. In terms of execution time, this run-length would not be dissimilar to static tools such as IDA Pro. Further reducing the feature set to the top 50 attributes and using a unigram representation upholds the classification accuracy >99%, while reducing the parsing, processing, and classification overheads. In terms of n-gram analysis, there is minimal gain when increasing between $n = 2$ and $n = 3$. Furthermore, n>1 offers no increase in detection accuracy when runtime is controlled for. Considering the computational overhead and feature explosion with increasing levels of n, focus should be maintained on a unigram investigation.

The results presented concur with [32], who found 32k run-length to provide the highest accuracy. However, the accuracy of the model peaked at 86.31%, with 13 features being included. Furthermore, the present study employs a dataset approximately 80 times the size of [32]. In the context of previous similar research, the accuracy of the present model is superior while the dataset is significantly larger and more representative of malware.

Acknowledgements The authors wish to acknowledge the extra support of VirusTotal.com by providing enhanced capabilities on their system.

References

1. 2014 cost of data breach study. Tech. rep., Ponemon Inst, IBM (2014). URL http://public.dhe.ibm.com/common/ssi/ecm/se/en/sel03027usen/SEL03027USEN.PDF
2. Vxheaven (2014). URL http://vxheaven.org/vl.php
3. Oracle vm virtualbox r programming guide and reference. Tech. rep., Oracle Corp. (2015b). URL http://download.virtualbox.org/virtualbox/SDKRef.pdf
4. Mcafee labs threats report sept 2016. Tech. rep., McAfee Labs (2016). URL http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2015.pdf
5. Alazab, M., Venkatraman, S., Watters, P., Alazab, M., Alazab, A.: Cybercrime: The case of obfuscated malware, pp. 204–211. Global Security, Safety and Sustainability & e-Democracy. Springer (2012)
6. Anderson, B., Quist, D., Neil, J., Storlie, C., Lane, T.: Graph-based malware detection using dynamic analysis. Journal in Computer Virology **7**(4), 247–258 (2011)
7. Bilar, D.: Opcodes as predictor for malware. International Journal of Electronic Security and Digital Forensics **1**(2), 156–168 (2007)
8. Bontchev V., S.F., Solomon, A.: Caro virus naming convention (1991)
9. Breiman, L.: Random forests. Machine Learning **45**(1), 5–32 (2001)
10. Chawla, N., Japkowicz, N., Kolcz, A.: Special issue on class imbalances. SIGKDD Explorations **6**(1), 1–6 (2004)
11. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: Proceedings of the 12th Usenix Security Symposium, pp. 169–185. USENIX Association (2006)
12. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM conference on Computer and communications security, pp. 51–62. ACM (2008)
13. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Computing Surveys **44**(2), 6–6:42 (2012). DOI

10.1145/2089125.2089126. URL http://search.ebscohost.com/login.aspx?direct=true&db=buh&AN=77698357&site=eds-live&scope=site
14. Ellis, D.R., Aiken, J.G., Attwood, K.S., Tenaglia, S.D.: A behavioral approach to worm detection. In: Proceedings of the 2004 ACM workshop on Rapid malcode, pp. 43–53. ACM (2004)
15. FitzGerald, N.: A virus by any other name: Towards the revised caro naming convention. Proc.AVAR pp. 141–166 (2002)
16. Idika, N., Mathur, A.P.: A survey of malware detection techniques. Purdue University **48** (2007)
17. Kang, B., Han, K.S., Kang, B., Im, E.G.: Malware categorization using dynamic mnemonic frequency analysis with redundancy filtering. Digital Investigation **11**(4), 323–335 (2014). DOI http://dx.doi.org.queens.ezp1.qub.ac.uk/10.1016/j.diin.2014.06.003
18. Kang, M.G., Poosankam, P., Yin, H.: Renovo: A hidden code extractor for packed executables. In: Proceedings of the 2007 ACM workshop on Recurring malcode, pp. 46–53. ACM (2007)
19. Khoshgoftaar, T.M., Golawala, M., Hulse, J.V.: An empirical study of learning from imbalanced data using random forest. In: 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007), vol. 2, pp. 310–317. IEEE (2007)
20. Kolter, J.Z., Maloof, M.A.: Learning to detect malicious executables in the wild. In: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 470–478. ACM (2004)
21. Leder, F., Steinbock, B., Martini, P.: Classification and detection of metamorphic malware using value set analysis. In: Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on, pp. 39–46. IEEE (2009)
22. Lejska, K.: X86 opcode and instruction reference. URL http://ref.x86asm.net/
23. Ligh, M., Adair, S., Hartstein, B., Richard, M.: Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code. Wiley Publishing (2010)
24. Martinez, E.: A first shot at false positives (2015). http://blog.virustotal.com/2015/02/a-first-shot-at-false-positives.html
25. McGraw, G., Morrisett, G.: Attacking malicious code: A report to the infosec research council. IEEE Software (5), 33–41 (2000)
26. Mehra, V., Jain, V., Uppal, D.: Dacomm: Detection and classification of metamorphic malware. In: Communication Systems and Network Technologies (CSNT), 2015 Fifth International Conference on, pp. 668–673. IEEE (2015)
27. Moskovitch, R., Elovici, Y., Rokach, L.: Detection of unknown computer worms based on behavioral classification of the host. Computational Statistics & Data Analysis **52**(9), 4544–4566 (2008)
28. Moskovitch, R., Feher, C., Tzachar, N., Berger, E., Gitelman, M., Dolev, S., Elovici, Y.: Unknown malcode detection using OPCODE representation, pp. 204–215. Intelligence and Security Informatics. Springer (2008)
29. Namanya, A.P., Pagna-Disso, J., Awan, I.: Evaluation of automated static analysis tools for malware detection in portable executable files. In: 31st UK Performance Engineering Workshop 17 September 2015, p. 81 (2015)
30. Nappa, A., Rafique, M.Z., Caballero, J.: Driving in the cloud: An analysis of drive-by download operations and abuse reporting, pp. 1–20. Detection of Intrusions and Malware, and Vulnerability Assessment. Springer (2013)
31. O'kane, P., Sezer, S., McLaughlin, K., Im, E.G.: Svm training phase reduction using dataset feature filtering for malware detection. IEEE transactions on information forensics and security **8**(3-4), 500–509 (2013)
32. O'kane, P., Sezer, S., McLaughlin, K., Im, E.G.: Malware detection: program run length against detection rate. IET software **8**(1), 42–51 (2014)
33. Roberts, J.M.: VirusShare.com (2014)
34. Roundy, K.A., Miller, B.P.: Hybrid analysis and control of malware. In: Recent Advances in Intrusion Detection, pp. 317–338. Springer (2010)

35. Runwal, N., Low, R.M., Stamp, M.: Opcode graph similarity and metamorphic detection. Journal in Computer Virology **8**(1-2), 37–52 (2012)
36. Santos, I., Brezo, F., Sanz, B., Laorden, C., Bringas, P.G.: Using opcode sequences in single-class learning to detect unknown malware. IET information security **5**(4), 220–227 (2011)
37. Santos, I., Sanz, B., Laorden, C., Brezo, F., Bringas, P.G.: Opcode-sequence-based semi-supervised unknown malware detection, pp. 50–57. Computational Intelligence in Security for Information Systems. Springer (2011)
38. Schultz, M.G., Eskin, E., Zadok, E., Stolfo, S.J.: Data mining methods for detection of new malicious executables. In: Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on, pp. 38–49. IEEE (2001)
39. Shabtai, A., Moskovitch, R., Feher, C., Dolev, S., Elovici, Y.: Detecting unknown malicious code by applying classification techniques on opcode patterns. Security Informatics **1**(1), 1–22 (2012)
40. Sikorski, M., Honig, A.: Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. No Starch Press (2012)
41. Thunga, S.P., Neelisetti, R.K.: Identifying metamorphic virus using n-grams and hidden markov model. In: Advances in Computing, Communications and Informatics (ICACCI), 2015 International Conference on, pp. 2016–2022. IEEE (2015)
42. Veluz, D.: Stuxnet malware targets scada systems (2010). https://www.trendmicro.com/vinfo/us/threat-encyclopedia/web-attack/54/stuxnet-malware-targets-scada-systems
43. Vemparala, S., Troia, F.D., Corrado, V.A., Austin, T.H., Stamp, M.: Malware detection using dynamic birthmarks. In: IWSPA 2016 - Proceedings of the 2016 ACM International Workshop on Security and Privacy Analytics, co-located with CODASPY 2016, pp. 41–46 (2016). DOI 10.1145/2875475.2875476