A Survey of Machine Learning Methods and Challenges for Windows Malware Classification

Edward Raff
Booz Allen Hamilton
`raff_edward@bah.com`
University of Maryland, Baltimore County
`raff.edward@umbc.edu`

Charles Nicholas
University of Maryland, Baltimore County
`nicholas@umbc.edu`

Abstract

Malware classification is a difficult problem, to which machine learning methods have been applied for decades. Yet progress has often been slow, in part due to a number of unique difficulties with the task that occur through all stages of the developing a machine learning system: data collection, labeling, feature creation and selection, model selection, and evaluation. In this survey we will review a number of the current methods and challenges related to malware classification, including data collection, feature extraction, and model construction, and evaluation. Our discussion will include thoughts on the constraints that must be considered for machine learning based solutions in this domain, and yet to be tackled problems for which machine learning could also provide a solution. This survey aims to be useful both to cybersecurity practitioners who wish to learn more about how machine learning can be applied to the malware problem, and to give data scientists the necessary background into the challenges in this uniquely complicated space.

1 Introduction

The impact of malicious software, or "malware", such as viruses and worms, is a long standing and growing problem. As society becomes increasingly dependent on computer systems, this impact only increases. Single incidents regularly cost companies tens of millions of dollars in damages (Musil, 2016; Riley and Pagliery, 2015; Frizell, 2015). In 2014, for example, the total economic cost associated with malware distributed through pirated software, a subset of all malware, was estimated to be nearly $500 billion (Gantz et al., 2014). Overall malware has proven to be effective for its authors, as indicated by the exponential growth of new malware (Spafford, 2014; AV-TEST, 2016a; F-Secure, 2016). This growth in malware only increases the need for better tools to stop malware and aid analysts and security professionals.

One specific area for improvement is *malware classification*. The task of malware classification has been long studied, and generally refers to one of two related tasks: 1) detecting new malware (i.e., distinguishing between benign and malicious applications) and 2) differentiating between two or more known malware types or families. The former of these we will refer to as *malware detection*, and it is intrinsically useful in stopping the spread of malware. Anti-virus (AV) products currently perform this function using a predominantly signature-based approach. Signatures are intrinsically specific to the malware they detect, and can be labor-intensive for an analyst to create. This makes signatures unlikely to scale as malware becomes more prevalent, an issue publicly recognized by AV vendors (Yadron, 2014; Hypponen, 2012).

The second class of malware classification we will refer to as *malware family classification*. Analysts and security professionals use this process to sort through new binaries and process an ever growing amount of data. In this case we assume or know that the binary is malicious, and wish to designate

it as a member of a specific family. For example, Conficker is a specific malware family that was prevalent from 2008 through 2009, and evolved through five major revisions (Porras, 2009). Modern obfuscation techniques employed by malware authors means that a single variant of a piece of malware may have several instantiations in the wild that do not have the same MD5 or SHA hash, but do have substantially identical functionality. Thus there is a need to automatically determine if newly observed binaries in fact belong to a previously known family. Such a finding aids in attribution, and reduces the number of files that analysts must look at.

The field of machine learning would seem the first and most likely avenue to provide relief to the malware classification problem, and indeed ML has been applied to this problem for decades (Kephart et al., 1995). While the problem of malware detection is an issue that spans across many file formats (Tabish et al., 2009) and operating systems, we will focus on the case of Microsoft Portable Executable (PE) binaries for 32 and 64-bit versions of the Windows OS, as was first considered by Schultz et al. (2001). In many cases the same issues and methods have been applied to other malware domains, such as malicious PDF files, or malware for the Linux or Android platforms. We focus on Windows executables because it is one of the longest standing platforms, and thus has the longest history of continuous research. Machine learning is an important contributor to a solution due to its focus on generalization, meaning that the models learned should work effectively on new malware specimens, which have not previously been seen. The importance of having methods that generalize is only growing, as a recent study has found that 94% of applications on a sample of 100 million machines are unique to the machine (Li et al., 2017). This means a system deployed widely will have to deal with hundreds of millions of unique files that would have never been encountered before.

The combination of machine learning and malware draws from many branches of Computer Science: From high level algorithms, statistics, and information theory to build classification models, to low level details on assembly instruction set architectures, software design, and networking needed to understand how malware works and propagates. Challenges from across this spectrum interact to make malware classification one of the more difficult tasks to which one could apply machine learning. Given that researchers often (if not always) have limited resources and time, it is not surprising that this area has received less attention and struggled to make progress, even as machine learning becomes an increasingly mainstream tool. In particular we note that a wide array of potential machine learning methods have not yet been explored for malware classification. Thus in this survey we make an effort to address techniques we feel should be investigated, especially if they are particularly appropriate or readily available in free software projects.

Our primary goal is to provide a common base of understanding for both malware researchers and machine learning practitioners who are interested in this combined space. What techniques have been looked at, what are their strengths or challenges, and what areas are in need of continued work? We present the information about what has been done and why effective application is a challenge from the bottom up, starting with the difficulties of getting representative data in section 2. This is a critical issue that has been overlooked in many prior works, as all subsequent stages depend on that data being representative of the population at large. Next we will discuss the features that have been extracted from these binaries in section 3, focusing on both static and dynamic features and their respective strengths and weaknesses. The next three sections will be on the types of machine learning approaches that have been applied to these features. More "standard" machine learning methods used on simple feature vectors are discussed in section 4. In section 5 we will discuss the machine learning models used that understand the input representation as being a type of sequence, which maps better to the true nature of the data. The most expressive representational form, that of a graph, is discussed in section 6. While this form is one that often best encodes the nature of the features used, it is also the least frequently used. Once we have chosen our data, features, and model, the next step is evaluation in section 7. Finally, we discuss some future research directions in section 8 and our conclusions in section 9. In each section of this survey we will attempt to provide a broad overview of what is currently done and the challenges that are impeding progress or need to be considered.

It is unfortunate that for many of the issues we will discuss, there is little quantitative information about how widespread or significant their impact is. The complexity of malware ranges from simple applications relying on user error and unpatched vulnerabilities to work, to viruses like Stuxnet - written (it is said) by nation state actors, which attempt to evade detection and may have selective intent that goes unnoticed or unused on most systems (Kushner, 2013). This broad spectrum of sophistication means that different issues and countermeasures may have a more or less noticeable impact on a learning system depending on the current prevalence of such measures, the malware we

would like to classify, and the systems on which a solution would be deployed. This can change over time and it is not currently feasible to tackle all of these issues at once. For these reasons we refrain from declaring any method the "state of the art" for malware classification, and instead focus on the pros and cons of the various approaches, as well as the underlying issues that cause this slippery situation. In particular, we will focus on any theoretical shortcomings that would prevent a system from working in practice, such as, for example, any machine learning processes which an adversary could circumvent with minimal effort if they wished to do so.

2   Data Collection Challenges

As with many applications, the first task to building a machine learning model is to obtain data that accurately represents the distribution of binaries that will be observed. It is indeed well known that obtaining more and better labeled data is one of the most effective ways to improve the accuracy of a machine learning system (Domingos, 2012; Halevy et al., 2009). However, by its very nature the potential scope of what a binary can do is unbounded. There is no way for us to randomly sample from the binaries that may exist in the world, and we have no way to measure how much of the "space" of binaries we have covered with any given dataset. Beyond the unbounded scope, the malware domain poses a number of unique challenges to data collection. This makes it almost impossible to perform canonical best practices, such as having multiple labelers per file and judging inter-labeler agreement (Geiger et al., 2020).

When obtaining data, it is often the case that malware is the easiest to get. Not only are there websites dedicated to collecting malware sent in by volunteers (Roberts, 2011; Quist, 2009), but it is not unusual for a researcher to obtain their own malware specimens through the use of honeypots (Baecher et al., 2006). A honeypot is a system connected to the Internet that intentionally tries to get infected by malware, often by leaving open security holes and foregoing standard protections. At the same time, both of these sources of malware can have data quality issues. Honeypots will have data biased toward what that system is capable of collecting, as malware may require interaction from the honeypot through specific applications in order to successfully infect a machine (Zhuge et al., 2007). That is, a malware sample's infection vector may rely on a specific version of Firefox or Chrome to be running, and it may not be possible to account for all possible application interactions. Malware may also attempt to detect that a potential target is in fact a honeypot, and avoid infection to defer its detection (Krawetz, 2004). The issues that bias what malware is collected by honeypots are also likely to impact the quality of larger malware repositories, as users may run honeypots and submit their catches to these larger collections. Malware repositories will also have a self-selection bias from those who are willing to share their malware and take the time to do so.

Benign data, or "goodware", has proven to be even more challenging to physically obtain than malware. This is in part because malware actively attempts to infect new hosts, whereas benign applications do not generally spread prolifically. As far as we are aware, no work has been done to quantify the diversity or collection of benign samples, or how to best obtain representative benign data. Most works take the easiest avenue of data collection, which is to simply collect the binaries found on an installation of Microsoft Windows. This tactic can lead to extreme over-fitting, where models literally learn to find the string "Microsoft Windows" to make a determination (Seymour, 2016; Raff et al., 2016). The population of binaries from Windows share too much of a common base to be useful for training more general models Instead, the model learns to classify everything that does not come from Microsoft as malware (Seymour, 2016). This bias is strong enough that even using only a subset of the information will still lead to over-fitting (Raff et al., 2017). This issue is particularly wide spread, and occurs in almost all cited papers in this survey. The significant exception to this are papers produced by corporate entities that have private data they use to develop anti-virus software. When this goodware bias issue is combined with the fact that there is no standard data-set for the task of malware detection, it is almost impossible to compare the results from different papers when different datasets are used. In addition, prior work using benign samples from clean Microsoft installations may significantly over-estimate the accuracy of their methods.

Only recently has effort been made to address this lack of a standard dataset for malware detection. Anderson and Roth (2018) released the EMBER dataset, which contains features extracted from 1.1 million benign and malicious binaries. EMBER is the first standardized corpus that has been released for malware detection. Their work has taken important steps toward reproducible science and a shared benchmark, but more work is still needed. By the author's own admission, the method

of its labeling makes it an "easy" corpus. If users want to create new features from the raw binaries, they have to obtain the binaries themselves independently — as the authors are unable to share the raw files due to copyright concerns. Information regarding malware family is also not present in the original version. A 2018 version of the Ember corpus (released in 2019, so that labels would be of higher confidence) has attempted to rectify a number of these issues by using more challenging data and malware family information.

Once data has been obtained, labeling the data must follow (when labels do not come "for free" as they do with honeypots). The issue of labeling malware into families, or determining if an unknown binary is or is not malware, is labor intensive and requires significant domain knowledge and training. This is in contrast to many current machine learning domains, like image classification, where labeling can often be done by individuals with no special expertise and with minimal time. For example, an expert analyst can often take around 10 hours to characterize a malicious binary (Mohaisen and Alrawi, 2013). This observation of the expense to understand what a file does is not unique, with a recent survey reporting hours to weeks of time, with participants ranging from 3-21 years experience (Votipka et al., 2019).This effort makes manually labeling large corpora impractical. For an entirely expert-labeled corpus for malware family classification, the largest public corpus we are aware of was developed by Upchurch and Zhou (2015). They grouped 85 malware samples by functional similarity into a total of 8 groups.

For benign vs malicious labeling, many have attempted to circumvent this issue through the use of anti-virus (AV) products. One popular strategy is to upload the binaries to websites such as *VirusTotal*, which will run several dozen AV products against each binary, and return individual results. If more than 30% of the AVs claim a file is malicious, it is assumed malicious. If none of the AVs say it is malware, it is assumed benign. Any specimen that tests between these two levels (at least one but less than 30% of the products say it's malicious) is then discarded from the experiment (Berlin et al., 2015; Saxe and Berlin, 2015). We note there is nothing particularly special about choosing 30% as the threshold, and many works have used different rules. Others have used ≥ 4 AV hits as a splitting point between malicious and benign (Incer et al., 2018), or left the issue unspecified (Kolosnjaji et al., 2016).

A recent study by Zhu et al. (2020) of different thresholding strategies found that using a threshold of ≤ 15 as the decision point between benign and malicious is a reasonable compromise to varying factors. This includes the fact that 1) AV decisions fluctuate over time, stabilizing after several months. 2) The false positive rate of AV engines in not trivial for novel files, 3) the false positive rate on packed benign files can be significantly higher, and 3) many AV engines have correlated answers and some appear to alter their own decisions based on the results of other AV engines over time. We note that these results regarding the use of VirusTotal labels are only for a benign vs malicious determination, and an equally thorough study of family labeling using VirusTotal has not yet been presented.

While this selection is easy to perform, the labels will be intrinsically biased to what the AV products already recognize. More importantly, binaries marked by only a few AV products as malicious are likely to be the most important and challenging examples. This middle ground will consist of either benign programs which look malicious for some reason (false positives), or malicious binaries that are not easy to detect (false negatives). Removing such examples will artificially inflate the measured accuracy, as only the easiest samples are kept. Removing such difficult to label points will also prevent the model from observing the border regions of the space between benign and malicious. The aforementioned EMBER dataset uses this style of labeling, and hence the "easy" designation (Anderson and Roth, 2018). This AV-bias issue also hampers effective model evaluation, as we are skewing the data and thus the evaluation to an easier distribution of benign and malicious samples. This causes an artificially large accuracy by any of the metrics we will discuss later in section 7.

Only recently have some of these AV biases been categorized and described. Botacin et al. (2020) has shown that the detection rate of AV products may vary by country (i.e., is this malware global, or country specific, in its proliferation), executable type (e.g., COM files vs. DLL), and family type (e.g., ransomware vs trojans). These biases will naturally be embedded into any model and evaluation built from labels that are AV produced. Further, using older files to try and maximize confidence is not a guaranteed workaround, since AV engines will have label regressions over time, where they stop detecting sufficiently old files as malicious (Botacin et al., 2020).

We also note that the subscription service to VirusTotal allows for downloading the original files based on their hash values. This is how users can get the raw version of the EMBER dataset, or create their own preliminary datasets. However, the subscription to VirusTotal is not cheap (even with academic discounts), and may be beyond the budget of smaller research groups or those just getting into this space. As such it represents an unfortunate barrier to entry, especially since VirusTotal is widely adopted within the industry.

When the desired labels are for malware families, the use of AV outputs becomes even more problematic. The family labels provided by AVs are not standardized and different AV products will often disagree on labels or type (Bailey et al., 2007a). While more advanced methods exist than simple thresholding (e.g., 3/5 of AVs say the label is "Conficker") for determining benignity (Kantchelian et al., 2015) and malware family (Sebastián et al., 2016), the use of many AV products remains the only scalable method to obtain labels. High quality family labels require manual analysis, which as noted before, requires days-to-weeks of effort. Worse still, malware authors have historically copied/stolen code from one-another, which can make determining a specific family (and the related problem of attribution) even more difficult (Calleja et al., 2019).

Beyond the issue of collecting data, there is also the fact that binaries exhibit *concept drift*, meaning the population as a whole changes over time. This is true of both benign and malicious binaries, as changes will percolate through the population as the Windows API changes, code generation changes with newer compilers, libraries fall in and out of favor, and other factors. It then becomes important to investigate the performance of a classification system as change occurs (Masud et al., 2011), which is not widely explored. The distribution of malware in particular drifts at a faster rate, as malware authors attempt to modify their code to avoid detection. For example, Rajab et al. (2011) performed an extensive study of web based malware on Google's Safe Browsing infrastructure. Over four years they saw an increase in malware that relied on social engineering, a short lifespan for the use of most exploits documented by Common Vulnerabilities and Exposures (CVEs), and an increase in attempts at "IP cloaking" to obscure their source. The fast evolution of malware is a result of an *adversarial* scenario, and only further complicates the development of a long term solution (Kantchelian et al., 2013; Singh et al., 2012).

3   Features for Binaries

Feature extraction is the prerequisite step to applying any machine learning method. In the domain of malware analysis for PE binaries, the approaches are generally divided into one of two groups: static or dynamic. Dynamic features are extracted from running a binary and extracting the features of interest from its execution. The fact that a program's execution may alter from run to run and in different environments is why the features are called dynamic. Given the potentially malicious intent of any arbitrary executable, dynamic analysis is often done through a virtual machine (VM). Conversely, static features are extracted from the binary itself without any attempt to execute it. A summary of the most commonly used features, and the representations that are regularly used, is given in Table 1.

3.1   Dynamic Analysis Features

There are a number of common feature types to extract from dynamic analysis. For example, an early type of dynamic analysis was to modify the linker in the operating system to wrap each function call to the OS or other libraries with a special prologue and epilogue (Willems et al., 2007). In doing so the functions called could be tracked in the order of their occurrence and one could obtain a sequence of *API or function calls*. Such trackings of API calls can be used in many ways, and is often interpreted as a sequential ordering or as a directed graph (Elhadi et al., 2014; Fredrikson et al., 2010). Special tracking can be added for common tasks, such as registry edits, files created or deleted, mutex operations, and TCP/IP calls (Rieck et al., 2008). These are all common tasks or operations that malware might perform, so recording extra information (such as method arguments) can be beneficial to analysis. Ultimately, there are many ways to combine the API functions called and the operations performed, with many works using one of or both options, and tracking different subsets of actions. These approaches are often called "behavior based", and make up a large portion of the dynamic features used. Directly related to tracking of API calls is tracking *system calls*. For our purposes, we define a system call as a service provided by the Windows kernel, and (usually)

Table 1. Summary of the features commonly used for malware analysis. Feature Source columns indicate whether the feature type is *commonly* obtained via dynamic or static analysis. Feature Representation indicates the which ways of interpreting the original features are used. The fixed-length column does not consider cases where a approach is used that converts sequences and graphs to fixed length representations while retaining significant information of the sequential nature.

Feature Type	Feature Source		Feature Representation		
	Static	Dynamic	Fixed-Length	Sequence	Graph
Bytes	✓		✓	✓	
Header Values	✓		✓		
Entropy	✓			✓	
Assembly	✓	✓	✓	✓	✓
API/Function Calls	✓	✓	✓	✓	✓
System Calls		✓	✓	✓	✓
Network Traffic		✓		✓	✓
Performance Counters		✓	✓	✓	
System Changes		✓	✓	✓	
Contextual	✓	✓			✓

accessed via an entry point in `Ntdll.dll`. (Russinovich et al., 2012a,b) There are several hundred of these functions, and they are often called by the APIs Microsoft provides, but less often by user code. In fact, use of functions in Ntdll.dll by user code is regarded as a malware indicator (Sikorski and Honig, 2012). One advantage of tracking system calls, rather than all calls to the Windows API, is that the set of system calls tends to remain stable from one version of Windows to another, for the sake of compatibility.

The same technology that allows for API call traces can also be used to track changes to the state of the system. Such *system changes* may include the registry edits and files created, as well as processes that started or ended, and other various configurable settings within the OS (Bailey et al., 2007b; Kirat et al., 2014). System changes may also be obtained from system logs (Berlin et al., 2015), which can be used as a convenient feature source with minimal overhead (since the system was going to collect such logs anyway) or for retroactively detecting malware and determining the time of infection.

Though not as popular, more granular information can be extracted as well. It is possible to record the sequence of assembly instructions as they run (Dai et al., 2009). Though this approach in particular can require additional feature selection and processing, as the amount of data can grow quickly and the length of program execution may be unbounded. Another option is to track the results of various performance and hardware counters that are present in modern CPUs as well as process related counters tracked by the OS (Tang et al., 2014). These could include the number of memory pages being allocated or swapped, voluntary and forced context switches, cache hits and misses, and other various fields. The intuition being that the performance behavior of malware will be distinct from benign applications due to the different nature of their operation.

Another less frequently used approach is to monitor the network traffic and content that a binary may produce (Stakhanova et al., 2011; Nari and Ghorbani, 2013; Wehner, 2007; Perdisci et al., 2010; Mohaisen and Alrawi, 2013). Many different malware applications make use of command-and-control servers (the existence or location of which may be obfuscated) to direct the actions of infected hosts, making it a potentially informative behavior. Use of the local network is also one of the most common ways for malware to self proliferate. While the population of malware that does not use the Internet or any local network may be small, it may also be one of the more interesting and important ones to classify correctly.

The methods discussed in this section make up the majority of features that are extracted via dynamic analysis. While the set of options may seem simple, the systems to capture them represent their own significant engineering efforts. Many such systems have been developed over time, and we refer the reader to (Egele et al., 2008) for a survey of the numerous systems for dynamic analysis and their relative pros and cons. The focus of this work will remain not on the method of collection, but what is collected and the challenges that are faced in doing so.

3.2 Dynamic Analysis Challenges

At first glance, a preference for dynamic over static features may seem obvious. The actual behavior of an application would intuitively be a strong indicator of the intent of a binary, and an effective way to group applications and measure their similarity. However, this perforce requires allowing the binary to execute — which opens a proverbial can of worms that must be considered.

For safety, malware must generally be run inside a Virtual Machine where its effects can be contained and reverted. But the malware authors are aware of this, and can attempt to detect that the malware is being run in a controlled environment and then alter the malware's behavior in response. It is even possible for malware authors to detect which specific emulator they are being run in, be it standard Virtual Machine emulation software (e.g., VirtualBox) or custom environments used by AVs (Blackthorne et al., 2016). For evasive malware, that means the apparent behavior inside of a safe Virtual Machine may differ in a substantial way from when the same program is running on real hardware (Kirat et al., 2014). This makes features built from running binaries inside a VM less reliable.

Unfortunately there exist a number of potential ways for a malicious author to detect a virtual environment, and there is no simple way to prevent such detection. One particular avenue is through CPU and timing attacks (Kang et al., 2009) that are applicable to hypervisor virtualization (Popek and Goldberg, 1974). For a bare-metal hypervisor that allows most instructions to run at native speed, it is necessary to intercept and emulate certain instruction calls (such as changing from ring 3 to ring 0) in order to keep the VM contained. Such instructions will incur a significant performance penalty due to the extra overhead to intercept and emulate them. While this is normally acceptable, as such cases are the minority of instructions, the performance discrepancy may be used by the binary to determine that it is running under emulation, and thus alter its behavior. Similarly, if the whole system is being emulated equally "slowly", malware could request information about the CPU, network card, and other hardware to determine if the time to execute is abnormally slow for the given hardware or inferred hardware age. Even beyond just timing attacks, the numerous possible discrepancies between real and emulated hardware have lead many to consider the task of creating a virtual-machine undetectable by malware effectively impossible (Garfinkel et al., 2007).

One avenue of research to circumvent this problem is to force binaries to follow some path of execution (Peng et al., 2014; Brumley et al., 2007). Such approaches successfully avoid the issue of allowing malware to determine its own behavior, at the cost of not necessarily knowing which execution path to take to observe desired behavior. That is to say, we do not know which execution path and sequence will exhibit the malicious behavior we wish to detect. Even if we ignore looping constructs and backwards branches, if a binary has b conditional branches (e.g. If-Else statements) in it, there may be up to 2^b different possible execution paths to take. Some heuristics to select execution paths must be applied, and this may be difficult given the unusual behavior of malicious binaries. For example, one may heuristically switch execution paths if one path causes illegal behavior or results in an interrupt from the OS. However, such behavior may be intentional, in causing side effects or triggering a bug that the malware intends to exploit.

Even given the pros and cons between execution in a virtual environment and forced execution, both approaches share a common issue in application. Behavior of malware may depend on the user environment in a non-trivial way. A trivial case would be malware behavior dependent on a bug specific to an OS version, such as Windows XP over Windows Vista. It has been found that malware may depend on specific applications being installed and running at the same time as the malware, and the interactions between programs in regular use (Rossow et al., 2012). Such scenarios are not general or easily covered in experimental testing, and can cause a large discrepancy between the lab and deployments to real users. Such cases may easily cause a machine learning model to stop generalizing, or miss certain subsets of malware in practice.

Another environmental factor in dynamic analysis is Internet traffic and connectivity. Allowing unconstrained Internet access to running malware is risky at best, and opens ethical concerns in allowing malware under examination to infect and attack other machines. Yet disconnecting Internet access entirely may dramatically alter the behavior of malware, not including the possibility of malware updating itself or downloading new functionality. Maximizing the containment of malware while allowing Internet access can require extensive design and engineering effort (Kreibich et al., 2011). A further complication exists in experiment reproducibility, as the servers malware connects to may change or go offline over short periods of time. When these servers do not respond, or even if

they do, the malware's behavior may change or cease altogether. This makes dynamic analysis of older malware difficult, as these servers are unlikely to return(Rafique and Caballero, 2013).

The issue of reproducibility and infection can be partially addressed by network emulation, in which the host environment running the malware intercepts and alters network traffic, and potentially provides fake responses, in order to let the malware run as if it had Internet connectivity while keeping it isolated (Graziano et al., 2012). These issues are significant impediments in using network traffic as a reliable feature, and only further complicate dynamic analysis. A new approach to help make dynamic feature extraction more reproducible is to design special VM recording techniques, which save all of the non-deterministic events so that a VM can be replayed at a later point in time (Severi et al., 2018). While powerful and storage efficient, if the malware successfully detects the VM at first run and alters its behavior (or fails to run properly for other reasons), the replay will always reflect this failure.

3.3 Static Analysis Features

By its very nature, static analysis greatly reduces the scope of features options to consider for classification. One common choice is to use the raw-bytes themselves as features (Raff et al., 2016; Kolter and Maloof, 2006; Stolfo et al., 2007). A subset of the raw-byte approach is simply to search for and extract what appear to be ASCII strings (Islam et al., 2010). This approach assumes the least amount of knowledge and is widely applied to other file types because of its ease of application. Another approach is to instead compute a windowed entropy over the raw bytes, mapping each file to a entropy sequence (Han et al., 2015; Baysa et al., 2013; Sorokin, 2011). Regardless of how processed, these approaches have an attractive simplicity at the cost of ignoring relevant properties. For example, while the raw bytes may be processed as one long linear sequence, the locality within a binary is non-linear. Different portions will relate to others through pointers in the storage format as well as various local and long jumps in the assembly. It is also common to build histograms from this information to reduce it to a fixed length format (Saxe and Berlin, 2015).

Using more domain knowledge, it is also popular to parse the PE-Header (Mic, 2013) for relevant information, extracting the fields and imports and encoding them as numeric and categorical features (Shafiq et al., 2009; Raff et al., 2017). Being able to process the PE-Header is also important for finding and disassembling the binary code, which is one of the more popular feature types to use (Santos et al., 2010; Moskovitch et al., 2008) in static analysis. As mentioned in subsection 3.1, assembly sequences can be used in dynamic analysis as well. The difference then becomes what assembly sequences appear in the file and overall structure, versus the sequence of instructions actually run (Damodaran et al., 2015). In each case one may observe sequences not seen by the other. The dynamic version may not run all of the code present, and the static version may not find obfuscated instructions.

The extraction of PE-Header disassembly from static analysis are more readily available, and provided by many open-source projects. For the PE-Header, there are projects like PortEx (Hahn, 2014), pefile[1], and this functionality is even built-into the Go language runtime[2]. For disassembly, relevant projects include Capstone[3], Xori[4], Distorm[5], BeaEngine[6], and others. Many different disassemblers have become available in part because disassembling a binary is non-trivial, especially when malware may attempt to create obscure and obfuscated byte code that attempts to thwart disassembly. Each of the many options available have different pros and cons in terms of run-time, accuracy, supported architectures, and other issues.

Once a binary is successfully disassembled (which requires the PE Header), it is also possible to resolve API function calls from the assembly using the Import Address Table (Ferrand and Filiol, 2016). The IAT stores the functions the library wishes to load as well as the virtual address at which the function will be stored. Then any `jump` or `call` function's arguments can be converted to the canonical target function. This allows us to not only use the imported functions and APIs as features

---

[1]`https://github.com/erocarrera/pefile`

[2]`https://golang.org/pkg/debug/pe/`

[3]`http://www.capstone-engine.org/`

[4]`https://github.com/endgameinc/xori`

[5]`https://github.com/gdabah/distorm`

[6]`https://github.com/BeaEngine/beaengine`

in a fixed-length feature vector (function present / absent), but also as a sequence or graph of API call order.

Finally, the most knowledge-intensive and time-consuming option is to consult malware analysts on what information to look for, and attempt to automatically extract said information (Dube et al., 2012). Such approaches may obtain a distinct advantage from expert opinion, but will require additional work to update due to concept drift as malware authors adjust their code to avoid detection.

3.4 Static Analysis Challenges

While the feature extraction process is often simpler for static analysis, it exhibits its own set of problems that must be dealt with. Notably the contents and intent of a binary are often obfuscated, with the first line of obfuscation being the use of packing (Wei et al., 2008). Packing wraps the original binary content inside a new version of the binary, often storing the original version with some form of compression or encryption. Upon execution, the packed version of the binary extracts the original version and then executes it. Packing may be applied recursively multiple times, with different types of packers each time, to maximize the effort needed to extract its contents.

This technique has been widely used in malware, and among well-meaning software developers. Packing is often employed as an attempt to thwart reverse engineering by a competitor, avoid or delay the prevalence of "cracked" versions of commercial software, or just to reduce the file size for transfer (Guo et al., 2008). There are attempts to encourage the authors of packing software to cooperate with AV vendors to add information that would reduce the magnitude of this problem (Singh and Lakhotia, 2011), incorporating "tags" that would make it easier to determine if a packed binary is safe and where it came from. Currently it remains that it is not sufficient to simply detect packing and infer maliciousness. The development of automated unpacking tools is an active area of research (Martignoni et al., 2007; Royal et al., 2006), however it generally requires some level of emulation of the binary. This brings back many of the issues discussed in subsection 3.2 with performing dynamic analysis. Though there has been some work in static unpacking (Coogan et al., 2009), the dynamic approach to this problem has been the preferred method in most works.

Packing is often considered as a "catch all" that thwarts all static analysis, and always increases the entropy of the original file. Recent work by (Aghakhani et al., 2020) has challenged many of these "known" assumptions about packing. In a large study they have shown it is possible for machine learning based models to make benign vs malicious distinctions even when contents are packed, provided the packers are not too novel and the training distribution of packers is properly accounted for. They also showed that many packers lower the entropy of resulting file. A result in particular that is counter-intuitive from this work is the utility of byte n-grams as features. This feature type will be discussed more in section 3, but many assumed packing would invalidate such byte based processing from being useful.

There exists other types of obfuscations as well within the binary code itself, including a host of possible obfuscations done by packers (Roundy and Miller, 2013). Some simpler forms of obfuscation may include the use of extra instructions that don't change the result of a program, executing instructions generated at run-time (separate from unpacking), and unnecessary or unreachable code (Christodorescu et al., 2005, 2007).

There also exist other sophisticated obfuscation techniques that are widely used, but from which information can be extracted with some effort. Polymorphic malware alters itself each time it propagates, creating numerous different versions that are all functionally equivalent while obfuscating the entry point or decryptor of a binary (Newsome et al., 2005). Metamorphic malware goes further, potentially altering all of the binary code as it propagates (Konstantinou, 2008). Some malware even implements its own virtual machine for a custom instruction set in which the malicious code is written (Sharif et al., 2009). Analysis can get particularly difficult when multiple forms of obfuscation are used, since none of them are mutually exclusive. There have been attempts to develop fully generic deobfuscation systems that do not rely on knowing in advance which obfuscation technique is being used, but such attempts have not yet been fully successful (Yadegari et al., 2015). Granted that a competent malware analyst can reverse engineer many if not most obfuscated files, with the right tools and sufficient time, (Schrittwieser et al., 2016), such efforts are expensive and do not scale.

It has been shown that deobfuscating malware improves the recall of signature based approaches (Christodorescu et al., 2007). The presence of obfuscation may be a malware indicator in its own

right, and such a feature could be useful in building a machine learning model. Hence, it is not clear that deobfuscation should be attempted in each and every case, and arguments could be made either way. This question deserves further study.

3.5 Contextual Features

A third type of features are what we will call *contextual* features. These are features that are not properties of the malicious binary itself, but come from the context of how the malware may exist or be distributed. The use of contextual features is less common in research, but has been reported to be highly successful in practice. Such systems are generally graph-based in their approach. For example Chau et al. (2011) used information about the "reputation" of the machines at which an executable file was found to make a determination about maliciousness, without looking at the content of the file itself. Others have followed this same strategy, and attempt to more precisely define the relations between files to improve results (Tamersoy et al., 2014), and to merge both relations and file dependent features (Ye et al., 2011).

Beyond measuring reputation of machines, the reputation of the domain name or IP address from which a file was downloaded can also be used to classify the downloaded binary as malicious if the source address has low reputation. This, as well as counter-measures, were discussed by Rajab et al. (2011). Others have created more elaborate graphs based on how and from where the file was downloaded, including the benign applications (e.g., Internet Explorer) that are also involved in the process (Kwon et al., 2015).

In a similar theme, Karampatziakis et al. (2012) looked at making classifications for files that are found in the same container (e.g., a zip or rar file). This approach is based on the hypothesis that if any file found in a container is malicious, all are more likely to be malicious. A similar approach has recently been proposed to leverage the file name of the malware itself, rather than its contents, to predict maliciousness (Nguyen et al., 2019; Kyadige and Rudd, 2019). While not sufficient on its own, it may be useful in conjunction with other features (Kyadige and Rudd, 2019) or in investigative/prioritization situations where the whole file may not be available (e.g., file path was stored in a log, but the file itself has since been deleted) (Nguyen et al., 2019).

3.6 Contextual Challenges

The contextual information we have discussed can include a mix of both static (files rest on the same system) and dynamic sources (reputation of IP addresses and download path). As such it is not a third type of information, but the nature of the contextual information being outside of the executable makes it intrinsically different from the others we have discussed.

The biggest impediment to using a contextual approach so far appears to be access to the contextual information itself. All of the works we have discussed make use of data owned by private companies and is measured in the millions of files, and can not be made generally available to all researchers. This makes the reproducibility and comparison issues discussed in section 2 especially pertinent.

Similar to the issues discussed in subsection 3.2, the contextual information is sensitive to time. Unless recorded for posterity, it will not be possible to perform a historical study of how contextual features would have performed. This applies to both the static and dynamic sources of contextual information.

4 Machine Learning Methods

Most machine learning methods for classification work on fixed length feature vectors, i.e., $x \in \mathbb{R}^D$ where D is the number of features used. This is a natural representation of information for many domains, however it is a general mismatch for the case of malware classification. With the exception of features from the PE header and some expert knowledge features, almost every feature choice discussed in section 3 is sequential in nature. This leaves us with two choices, both less than ideal: make some simplifications to the problem so that we obtain fixed-length feature vectors, or restrict ourselves to the more limited set of models that support classification of sequences. Below we discuss the primary method by which fixed length features are constructed, and the many algorithms that have have been used for both fixed-length vector and sequence-based classification. Other methods

that more directly tackle the true nature of our feature choices will be discussed in section 5 and section 6.

A natural question to ask is which of the learning approaches and feature combinations work best. Unfortunately this question cannot be easily answered due to the data issues discussed in section 2. For the case of malware detection, many results are likely overestimated, and the lack of any common benchmark dataset further hinders any attempts to compare and contrast results. When distinguishing between malware families the VX-Heaven corpus provides a shared benchmark, but it is a sub-optimal barometer. Not only is the corpus outdated to the point that it does not reflect modern malware, but no particular effort is made to balance the number of samples from each malware family. That is to say, both the types of malware and individual malware families are not evenly distributed. This makes interpretation of results more difficult, especially as many works sub-sample the corpus in different ways to remove families or malware types with fewer samples.

Given these difficulties, for each learning algorithm we discuss the feature types and scenarios where they seem to perform well, and the situations where they perform worse or we believe their utility has been over-estimated. In addition we will give background to relevant extensions and advancements in the machine learning literature that could be relevant to future progress in the field of malware classification, but have not yet been explored.

4.1   N-Grams

The first item we discuss is not a learning algorithm, but a method of constructing features. N-Grams are a "bread and butter" tool for creating feature vectors from sequence information, though capture very little sequential information (and hence are included in this section). Despite this, n-grams have been widely used in malware classification, starting with the work of (Abou-Assaleh et al., 2004) that connected the methods being used with those in the domain of Natural Language Processing (NLP). Since then, n-grams have been one of the most popular feature processing methods for malware classification, and have been used for processing bytes, assembly, and API calls (Dahl et al., 2013) into bag-of-words type models. To give a more concrete example of this process, the byte sequence *0xDEADBEEF* would have the 2-grams *DEAD*, *ADBE*, and *BEEF*. At training time all possible 2-grams would be counted, and each 2-gram found would map to an index in a high-dimensional feature vector. The feature vector for *0xDEADBEEF* would have 3 non-zero values, the specific values determined by some feature weighting scheme such as TF-IDF or Okapi (Robertson and Walker, 1994), though a binary present/absent value is popular as well.

There exists a particular desire to use larger values of n for malware classification due to the limited semantic meaning contained within only 6 or so bytes or instructions. To give this context, a 6-byte-gram is not large enough to capture a whole instruction 2.4% of the time (Ibrahim et al., 2010). This is due to the variable length encoding of the x86 instruction set, and a valid x86 instruction can be up to 15 bytes in length. Similarly, an assembly 6-gram is often not sufficient to cover the behavior of a larger function. A simple function can compile to dozens of instructions, let alone more complicated functions which may easily be hundreds to thousands of instructions in length.

While large values of n are desirable, they are also computationally demanding. As n increases, the number of possible n-grams grows exponentially. Counting and tracking these itself is expensive, and feature selection is required before deploying. As such, the use of $n > 6$ has historically been rare. Some work has been done to speedup the collection of n-grams by approximately selecting the top-k most frequent n-grams as an initial feature selection process (Raff and Nicholas, 2018). This is based on the observation that n-grams tend to follow a power-law distribution, and that useful predictive features tend to have a minimum frequency (Luhn, 1958). Later work developed this into a probabilistic algorithm for selecting the top-k n-grams in a faster manner with fixed memory cost, testing values of $n = 8192$ (Raff et al., 2019a). This study found that $n \geq 64$ was surprisingly useful, and had the benefit that a malware analyst could reverse engineering the meaning of a large n-gram to better understand what the model had learned. Their work showed that predictive performance was maximized around $n = 8$, and that n-gram features had a surprisingly long shelf life, still being effective in detecting benign/malicious software up to 3-years newer than the training data.

4.1.1 N-Gram Coalescing

To help mitigate the computational issues with n-grams while retaining as much information as possible, approaches analogous to word stemming have been applied for both byte and assembly n-grams. In NLP stemming attempts to coalesce words with similar semantic meaning into one base form (e.g., "running", "ran", and "runner", all get mapped to "run"). This coalescing may lose important nuance, but can also benefit in a reduction to a more powerful subset of features.

For x86 assembly, the likelihood of seeing an exact match for most instructions and their operands values together is quite low. This results in extra features and may fail to match instructions that are essentially the same. The simplest workaround to this problem is to map each line of assembly to just the instruction being executed (Dolev and Tzachar, 2008; Moskovitch et al., 2008), e.g. `mov eax, 4` is mapped to just `mov`. Shabtai et al. (2012) argued in favor of this approach, noting that the main "engine" or component of malware could be easily re-located between different versions of a file. This would change the relative offsets, and thus the operands — causing the same code to no longer match. By removing the operands completely this issue is resolved, at the cost of specificity. It is then up to the learning method, empowered by appropriately sized assembly n-grams, to learn to detect these patterns.

Another alternative was proposed by Masud et al. (2008), which balances the extremes of removing the operands of the assembly and keeping them in their entirety. They noted that an instruction will have some number of parameters and each parameter could be coalesced into a location type, either memory, register, or constant corresponding to where the value used came from: either an access to memory, directly from a register, or the immediate value from the call to an instruction. For example, the instruction `mov eax, 4` would be coalesced to `mov.register.constant` and `mov [eax], 4` to `mov.memory.constant`. We note that in this form it does not matter that a register was used in the first parameter, it is that the operand value came from a memory access that determined the type.

Reducing the representation space via coalescing is intuitive and attractive, but it can also obfuscate important information depending on the task. The instruction name itself, such as `cmp` is in fact already performing some level of coalescing. This is because while the assembler accepts one "`cmp`" instruction, this instruction will be converted to one of nine different opcodes when assembled. Zak et al. (2017) found that "disambiguating" the specific opcode an instruction was compiled down to improved the predictive performance of assembly n-grams using both of the aforementioned forms of operand coalescing. This however was only for static analysis, and results may differ when instructions are extracted in a dynamic manner.

For bytes and assembly, n-perms have been proposed as an alternative scheme (Karim et al., 2005; Walenstein et al., 2007), particularly for clustering malware. An n-perm represents an n-gram and all possible permutations of an n-gram. An equivalent re-phrasing is: to map every n-gram to a canonical n-perm based on the contents of the n-gram, ignoring their order (e.g., *ACB* and *BCA* would both map to *ABC*). This conflation dramatically reduces the number of features created as n increases, allowing the consideration of larger values of n. This same notion has been re-developed for assembly as well (Dai et al., 2009), as a way to circumvent metamorphic malware which will re-order instructions and add superfluous instructions as a form of obfuscation.

4.2 Linear Models

One of the simplest and most effective classes of machine learning methods are linear models, the general objective function for which is given in (1). In it we have N data-points, a label y_i for each data-point, and a weight vector w that defines the solution. The value that w takes is dependent on the loss function ℓ, the regularizing function $R(w)$ and its weight λ. The basic goal is to assign every feature D_j an importance weight w_j, that is positive or negative depending on which class it is an indicator of. Given a positive and negative class (malicious and benign), we obtain a classification decision by examining the sign of the dot product $\text{sign}(w^T x)$, which is between the weight vector w and a data point x. Despite their simplicity, linear models can be highly effective, Especially when

dealing with high dimensional data sets, where more sophisticated non-linear models may have only minimal improvement in accuracy (Chang et al., 2010; Yuan et al., 2012).

$$\sum_{i=1}^{N} \ell(w^T x_i, y_i) + \lambda R(w) \tag{1}$$

For the loss function ℓ the two most common choices are the Logistic loss (2a) and the Hinge loss (2b). The Logistic loss corresponds to performing Logistic Regression, and the Hinge loss corresponds to using a Support Vector Machine (SVM) (Cortes and Vapnik, 1995). As presented below the value y indicates the true label for a data-point, and the value s would be the raw score for the data-point — the dot product between the weight vector w and the feature vector x.

$$\ell(s, y) = \begin{cases} \log(1 + \exp(-y \cdot s)) & \text{Logistic} \tag{2a} \\ \max(0, 1 - y \cdot s) & \text{Hinge} \tag{2b} \end{cases}$$

When training a linear model, the choice of ℓ does not have a significant impact on accuracy or training time. The choice of the regularizing function $R(w)$, and the amount of regularization we apply, λ, have a much more impact on model performance. For $R(w)$ the L_2 norm $(R(w) = \frac{1}{2}\|w\|_2^2)$[7] is the most common, and a search over penalties λ is done to find the value that best prevents overfitting to the training data. By increasing the value of λ we increase the penalty for model complexity, and encourage w to approach $\vec{0}$.

The other common choice of regularizer, the L_1 norm $(R(w) = \|w\|_1)$, is also a potentially useful choice, especially when dealing with high dimensional data that can result from the use of n-grams. This is often called Lasso regularization (Tibshirani, 1994) and will result in exact zeros occurring in the weight vector w, meaning it performs its own feature selection as part of training. When a hard zero is assigned as a coefficient, the associated feature has no possible impact on the model, and can be removed.

Lasso regularization also comes with theoretical and practical robustness to extraneous and noisy features (Ng, 2004), where a model trained with L_1 regularization will perform better than one trained with L_2 regularization as more and more unrelated features are added. This makes it an excellent fit for n-gram based feature vectors, which can quickly reach $D > 1$ million and has been successfully applied to byte n-grams to improve accuracy and interpretability (Raff et al., 2016). The L_1 norm does have some weaknesses: it's a biased estimator and can reduce accuracy under certain situations. But L_1 can be combined with the L_2 norm to form what is known as Elastic-Net regularization $(R(w) = \frac{1}{2}\|w\|_1 + \frac{1}{4}\|w\|_2^2)$ (Zou and Hastie, 2005). The Elastic-Net often provides the best of both worlds, resulting in models with higher accuracy and retaining the sparsity benefits of Lasso.

The simplicity of linear models provides the practical benefit of many tools being publicly available and able to scale to large datasets. The popular LIBLINEAR library supports both L_1 and L_2 regularized models for both the Hinge and Logistic loss functions (Fan et al., 2008). LIBLINEAR uses exact solvers specially designed for each combination of loss ℓ and regularizer $R(w)$. Similarly the library Vowpal Wabbit (Langford et al., 2007) implements the same algorithms using online methods, meaning it trains one datapoint at a time and can stream the data from disk. This allows Vowpal Wabbit to train faster and scale to terabyte size corpora. While the online training approach may sometimes result in lower accuracy than the approaches used in LIBLINEAR, the difference is usually minimal (if there is a noticeable difference at all). Linear models are also attractive because they are fast to apply, making them realistic for the real-time goals of AV systems.

4.3 Kernel Methods

Kernel methods are an extension of the linear methods discussed in subsection 4.2. Most commonly used with Support Vector Machines (SVMs) using a kernel trick K, the objective function is given in (3). We note that for SVMs the regularization penalty is usually expressed with the term C, where

---

[7]The $\frac{1}{2}$ term is included because it makes the math slightly more convenient when deriving the update. Otherwise it is of no significance, and is sometimes rolled into the value of λ

larger values indicate less regularization. These forms are equivalent where $C = 1/(2\lambda N)$. The kernel-trick is used to project the original data-set into a different space. A linear solution is found in this new space, which may be non-linear in the original space.

$$\sum_{i=1}^{N} \max(0, 1 - y_i K(w, x_i)) + \lambda \|w\|_2^2 \tag{3}$$

A valid kernel K represents the inner product in this new feature space, but does not require us to explicitly form it[8]. This allows us to obtain classifiers that are non-linear in the original feature space (and thus potentially achieve a higher accuracy). We can always pick the linear kernel (4a), which results in a linear model. Practically, two of the more common choices are the polynomial (4b) and Radial Basis Function (RBF) (4c) kernels. The polynomial kernel is particularly helpful to illustrate the intuition behind the kernel-trick, as we can easily compute $(\alpha + \beta)^{10}$ with two operations, an addition and an exponentiation. This is computing the inner product in the polynomial space explicitly, but avoids actually expanding the polynomial. If were were to explicitly form the feature space first by expanding the polynomial, we would end up performing 36 exponentiations, 10 additions, and 38 multiplications.

$$K(a, b) = \begin{cases} a^T b & \text{Linear} & \text{(4a)} \\ (a^T b + c)^p & \text{Polynomial} & \text{(4b)} \\ \exp\left(-\gamma \|a - b\|^2\right) & \text{RBF} & \text{(4c)} \end{cases}$$

The price for this flexibility is generally computational, as solving the kernelized version can take $O(N^3)$ time and $O(N^2)$ memory. On top of that, a parameter search must be done for the values (such as γ) used in the kernel. This is in addition to the regularization penalty C. Most malware data-sets being used are on the order of 40,000 samples or less, which is still in the range of available tools like LIBSVM (Chang and Lin, 2011). More advanced techniques that do not attempt to obtain the exact solution also exist, allowing the use of kernel methods to larger data-sets (Engel et al., 2004; Hsieh et al., 2014).

One of the challenges with the malware domain is the multiplicity of feature options and potential representations. For most machine learning techniques it is necessary to reduce these down to a single feature vector of fixed length for each data-point. This often results in an over-simplification for the malware domain. The use of more sophisticated kernels to alleviate this problem is a yet unexplored possibility. For example, one challenge is that the PE format specifies many different section types, the most common being sections for imports, exports, binary code, and data. However any of these section types may occur in a binary with any multiplicity[9] (e.g., one could have five different executable sections). The standard approach, if differentiating between sections, is to operate as if all instances of a section type were a part of one section. Instead, one could use a kernel that matches sets of feature vectors (Grauman and Darrell, 2005; Bo and Sminchisescu, 2009), allowing the model to learn from these directly.

Kernels can also be defined directly over strings (Lodhi et al., 2002; Leslie et al., 2002), which could be useful for comparing the function names defined within an executable or in handling unusual data content, such as URLs that can be found within malware (Raff et al., 2016). To handle the function graphs that may be generated from dynamic analysis, kernels over graphs may also be defined (Vishwanathan et al., 2010; Neuhaus and Bunke, 2007) and has seem some small amount of use for malware classification (Anderson et al., 2011). Furthermore, the composition of multiple kernels via additions and multiplications also forms a new and valid kernel. This would provide a direct method to incorporate multiple modalities of information into one classifier. For example, we could combine a kernel over graphs on API call sequences, a linear kernel for assembly n-gram features, and a kernel over strings found in the file into one larger kernel. However these options with kernels are largely unexplored for the malware classification problem.

_____

[8]The kernel trick is usually more formally explained as a Reproducing kernel Hilbert space (RKHS). We avoid it to reduce the mathematical background needed for this review

[9]up-to a hard limit on the number of sections specified by the PE format

4.4 Decision Trees

Methods based on Decision Trees have been popular in machine learning, and a number of variants and extensions to them exist. The two most popular base decision tree algorithms are C4.5 (Quinlan, 1993) and CART (Breiman et al., 1984). A number of desirable properties have resulted in their widespread use among many domains, making them some of the most widely used algorithms in general (Wu et al., 2007). In particular, decision trees are able to handle both categorical and numeric features simultaneously, are invariant to shifts or re-scaling of numeric features, can handle missing values at training and test time, are fast to apply at test time, and often obtain competitive accuracies while requiring minimal parameter tuning.

All of these properties can be highly relevant to malware classification, where a mix of numerical and categorical features may be common, and there are often real-time requirements for deployment on user machines. Missing values can be a common issue as well, as obfuscations performed by the malware may prevent successful extraction of a given feature. For these reasons many researchers have used decision tree based methods for malware classification (Perdisci et al., 2008; Dube et al., 2012; Anderson and Roth, 2018). They are easy to apply, provided as a standard tool in most machine learning libraries across several languages (Pedregosa et al., 2011; Hall et al., 2009; Gashler, 2011; Meng et al., 2016; Raff, 2017; Bifet et al., 2010) and with many stand-alone tools dedicated to more powerful extensions (Chen and Guestrin, 2016; Wright and Ziegler, 2015; Ke et al., 2017).

Kolter and Maloof (2006) used boosted decision trees in their seminal byte n-gramming paper. Boosting is one of many ensemble methods that work to improve the accuracy of decision trees by intelligently creating a collection of multiple trees, where each tree specializes to a different subset of the data. While they chose the AdaBoost algorithm because it performed best on their data, were able to utilize the interpretability of decision trees to gain insights to their model. An example of how one would be able to read a decision tree is given in Figure 1.
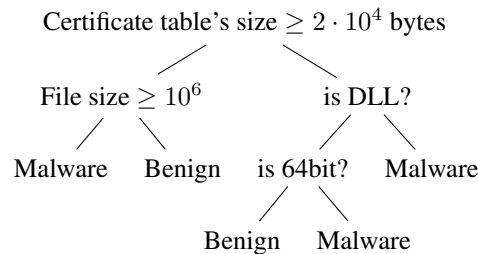
Certificate table's size $\geq 2 \cdot 10^4$ bytes

File size $\geq 10^6$

is DLL?

Malware   Benign   is 64bit?   Malware

Benign   Malware

Figure 1. A hypothetical decision tree.

Raff et al. (2017) used the Random Forests (Breiman, 2001) and Extra Random Trees (Geurts et al., 2006) ensembles to naturally handle the many disparate value types found within the PE-Header. Values from the header can be binary variables, multi-label variables, and numeric values with varying sizes and scales. For example, some values in the header will give the byte offset to another part of the binary, which could be anywhere from a few hundred to millions of bytes away. Most algorithms would have difficulty learning from this value range, and it can be difficult to normalize effectively. also exploited the tree based approaches to obtain ranked feature importance scores (Breiman, 2003; Louppe et al., 2013), another method by which one can glean information about what a decision tree has learned.

Some have worked on developing enhanced ensembling methods for decision trees to improve malware classification accuracy (Menahem et al., 2009). Even when there is no reason to use a tree based approach in particular, many malware classification works still include them as one of many models to try (Elovici et al., 2007; Alshahwan et al., 2015; Masud et al., 2008; Menahem et al., 2009; Moskovitch et al., 2009a; Anderson and Roth, 2018).

The widespread success of decision trees has made them a valuable tool inside and outside the domain of malware classification. This has lead to a large literature of decision tree techniques to tackle various problems, many of which may be applicable to malware classification. For example, the popular AdaBoost algorithm often overfits in the presence of significant labeling errors. An extension known as Modest AdaBoost is more robust to this issue, and may lead to improvements (Vezhnevets and Vezhnevets, 2005) in generalization. Another possibility is to use decision trees to deal with

concept drift. While malware datasets with date-first-seen labels are not publicly available[10], there already exists a literature of decision tree methods designed to work with changing data streams (Hulten et al., 2001). This also relates to how the accuracy of a malware classification system should be evaluated, which we will discuss further in section 7.

4.5 Neural Networks

Neural Networks have seen a recent resurgence in the machine learning community. Though older literature often referred to the technique for classification as Multi-Layer Perceptrons, newer work has placed an emphasis on the depth of trained networks and is often referred to as Deep Learning. We will provide a brief overview of neural networks, and refer the reader to Goodfellow et al. (2016) for a more thorough introduction to modern architectures, activations, and training methods.

Neural networks get their name from their original inspiration in mimicking the connections of neurons in the brain (though the interpretation is often taken too literally). A neuron is connected to multiple real valued inputs by a set of synaptic weights, corresponding to real-valued multipliers. An activation function $f(x)$ is used to produce the output of the neuron, where the input is the weighted sum of every input connected to that neuron. Generally the initial features fed into a network are called the input layer, and a set of neurons that produce our desired outputs form the output layer, from which a loss is derived (such as cross-entropy, or mean squared error). In-between is some number of hidden layers, where we specify the number of neurons, connectivity, and activations for each layer. The classic approach is to connect every neuron in one layer to every neuron in the preceding layer to form a fully connected network. A diagram of this arrangement is presented in Figure 2.
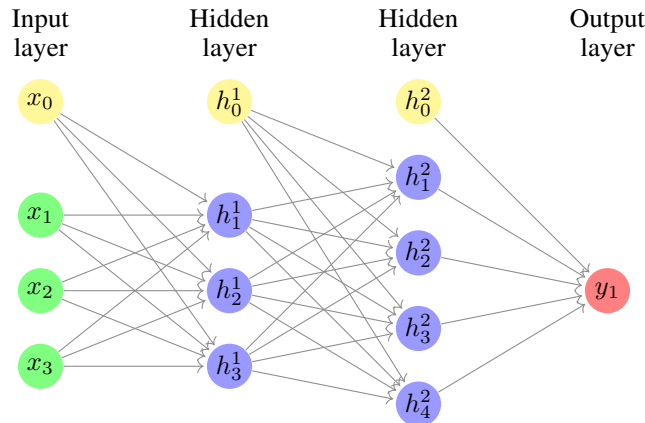

Figure 2. Diagram of a simple 1-layer neural network. Green nodes are input features. Yellow nodes are for the bias variable. Blue nodes are hidden layers. Red nodes are the output layer.

The weights for such a network are learned through an algorithm known as back-propagation (Rumelhart et al., 1986), which is performing gradient decent on the function created by the neuron graph. The view of neural networks as a large function graph has become increasingly popular, and allows for fast development using Automatic Differentiation. The user specifies the functions used by the network, and the software automatically computes the gradients with respect to any weight in the network. This has helped to fuel the resurging neural network literature and is a feature supported by many deep learning frameworks (Tokui et al., 2015; Chollet, 2015; Abadi et al., 2016).

The fundamental strategy enabled by such an approach is that the user should avoid feature engineering, and instead alter the network architecture to the needs of the problem. This works as neural networks have been found to learn their own complex feature hierarchies and representations from raw data alone (Ranzato et al., 2012; Goodfellow et al., 2014). This ability has allowed neural networks to become the state of the art in both speech processing (Graves et al., 2006) and image classification (Krizhevsky et al., 2012), significantly outperforming prior approaches.

---

[10]Such information can be obtained from the paid-for API of VirusTotal

While many works in malware classification have made use of neural networks (Firdausi et al., 2010; Perdisci et al., 2008; Liangboonprakong and Sornil, 2013; Hardy et al., 2016), they are often based on dated approaches to neural network construction. Advances in optimization algorithms (gradient descent), activation functions, architecture design, and regularization have dramatically changed the "best practices" of neural networks while also improving their performance.

One of the first effective applications of a modern neural network style was by Saxe and Berlin (2015), who used a private corpus to obtain reasonable accuracies. Their work performed the feature engineering manually by processing a combination of entropy, string, and PE header features. While their model performed well, it was not compared with any other machine learning algorithms, making it difficult to determine the benefits of neural networks in this particular application. The work of Saxe and Berlin (2015) is also notable for its model evaluation, which we will discuss further in section 7.

Raff et al. (2017) also looked at using a neural network, but instead presented it with raw byte information and did no feature engineering. Their work provided initial evidence that neural networks can reproduce the same feature learning on raw bytes, but was limited to a relatively small (and fixed size) subset of the PE header. As Raff et al. (2017) noted, the behavior and locality within malware is markedly different from signal and image processing tasks. Malware lacks the rigid scope and especially locality properties these other fields enjoy. As an example, it is easy to see how in an image the correlation between a pixel and its neighbors is relatively consistent throughout any image. But for a binary jumps and function calls can directly connect disparate regions, causing correlations between potentially arbitrary locations. No work has yet been done on determine what kinds of architectures can learn best from this type of locality complexity.

Another interesting application of modern neural network design is by Huang and Stokes (2016). Their system looked at System-call like features extracted via dynamic analysis, reducing the feature set by applying domain knowledge about the relationship function calls. Their architecture was modified to perform both malware detection (benign or malicious) and family classification (with 100 malware families) simultaneously. The jointly trained model resulted in a relative improvement of 26% over a model trained to do only malware detection on the same data. This shows the potential for intelligent architecture design choices to provide real gains in accuracy. This is also a method to enable more data use for training, as it is easier to get more malware data labeled with malware family labels than it is to get more benign data. The portion of the network trained to predict malware family can then be trained with this additional data, without biasing the malware detection portion of the network due to class imbalance.

5 Machine Learning Over Sequences

Many of the feature types we have discussed, such as assembly instructions, can be more accurately described as a sequence of events or values. Using n-grams to convert them to fixed length feature vectors allows us to use the wider breadth of models discussed in section 4, at the cost of ignoring most of the sequential nature intrinsic to the data. In this section we will review a number of techniques that are designed specifically for processing sequences, some of which will work directly on the sequence level, while others may attempt to create fixed-length representations more intelligently. In the latter case, the primary difference compared to the n-gram approaches discussed in subsection 4.1 is that n-grams only capture a very small amount of local sequence information. Approaches in this section will more generally capture larger amounts of the sequential structure in the data.

Some of the methods we will talk about face unique challenges regarding sequence length. For example, assembly sequences from binaries can be hundreds of thousands of steps in length or more, which significantly outpaces the longest sequences in other domains. While byte and assembly sequences are obviously the longest, potentially millions of steps long, higher level events and features extracted via dynamic analysis can easy reach hundreds of thousands of steps in length (Pascanu et al., 2015). These sequences are far longer than what machine learning is normally applied to, meaning the tools to tackle this problems are often lacking. For example, the longest sequence length we are aware of for neural networks outside of malware is in audio generation. Here the WaveNet architecture was applied to a sequence length of 16,000 steps (Oord et al., 2016). This was an order of magnitude larger than what others had achieved, yet is still an order of magnitude smaller than the sequences in the malware space.

5.1 Hidden Markov Models

Given the sequential nature of our feature choices, such as byte sequences, instructions, and API calls, the Hidden Markov Model (HMM) has become a popular choice (Damodaran et al., 2015; Shafiq et al., 2008; Wong and Stamp, 2006; Konstantinou, 2008), as it explictly models sequences and can handle sequences of variable lengths. Given a sequence of observations $\mathcal{O} = O_1, O_2, \ldots, O_T$ that are discrete ($O_t \in \{1, \ldots, K_O\}$), HMMs make three simplifying assumptions:

1. A sequence of hidden (or unobserved) variables $\mathcal{X} = X_1, X_2, \ldots, X_T$, where each state is discrete ($X_t \in \{1, \ldots, K_X\}$) governs the observed values $\mathcal{O}$.
2. Each observation O_t is generated solely by state X_t, and $P(O_t|X_t)$ is parameterized by a $K_X \times K_O$ emission matrix B.
3. Each hidden state X_t is generated solely by state X_{t-1}, and $P(X_t|X_{t-1})$ is parameterized by a $K_X \times K_X$ emission matrix A. The first hidden state is a special case, and is specified by an initial probability matrix π.
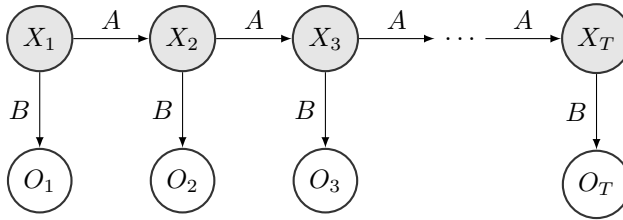

Figure 3. First-Order Hidden Markov Model, hidden states are gray and observed states are white.

Thus the matrices A, B, and π fully specify a HMM, an example of which is given in Figure 3. A and B are emission matrices, where the row index r corresponds to the source token being r, and the column c indicate the probability of token c preceding token r. Because these are generative models, to apply HMMs for malware classification a separate HMM must be fit to each class from the sequences corresponding to that class. At test time we get a new sequence $\tilde{\mathcal{O}}$, and compute the probability $P(\tilde{\mathcal{O}}|A_i, B_i, \pi_i)$ for each HMM we learned, choosing the class with the highest probability[11]. For a more thorough review of HMMs we refer the reader to (Rabiner, 1989; Ghahramani, 2001).

One can construct m'th-order HMMs to try to capture more history. However the learning process scales at $O(K_O^{m+1})$, which quickly becomes computationally intractable. This makes HMMs a better fit to shorter sequences such as API call sequences, if they are to be used. It may also be the case that the use of HMMs, as generative models, make the learning problem harder than necessary. Given data x and labels y, it models the joint distribution $P(x, y)$, which may be more complicated than the posterior distribution (i.e., benign or malicious) $P(y|x)$. While it is not always the case that discriminative models perform better (Ng and Jordan, 2002), we suspect that it is likely true for malware classification given that just modeling a binary (i.e., $P(x)$) is its own highly complicated problem, and the joint distribution is intrinsically harder still.

Deploying a solution using HMMs over bytes or instructions could also be problematic when we consider that malware evolves in an adversarial manner. It is not difficult for polymorphic code to generate assembly matching a specific low order transition distribution (Song et al., 2010), which would allow a malicious author to generate binaries matching a particular distribution.

5.2 Byte Similarity Measures

A number of methods have been used that seek to measure the similarity between two arbitrary byte sequences. These methods make no assumptions about the contents of formating of the underlying bytes, and thus can be used for any arbitrary byte input. This makes them attractive for malware classification, as they can simply use the raw binary as the target for similarity measures. Classification is then done by doing a nearest neighbor search for the most similar known binaries, and assigning a benign/malicious label based on the k nearest neighbors' labels. We note here three methods that have been used for malware analysis and detection to varying degrees.

---

[11]As presented this assumes each class is equally likely. This is generally not the case

5.2.1 Similarity Digests

One method readily available for such work is the use of *similarity digests* from the domain of digital forensics(Harichandran et al., 2016). These digests are similar to an MD5 checksum, but seek to minimize the change in output hash for changes in the input sequence. These techniques provide a similarity measure between the hash functions, which can then be used as the similarity between the digests. The state-of-the-art in this domain is the sdhash algorithm(Roussev, 2009, 2010). The faster ssdeep is also popular, but is not as accurate as sdhash for most problems(Kornblum, 2006). Ssdeep in particular is unlikely to be useful for malware analysis, as ssdeep is sensitive to content-reordering. Since PE files and binary programs make regular use of pointers and can be almost arbitrarily reordered, this is a significant problem. Although sdhash does not suffer from this same limitation, its use for malware detection has been only moderate.

While these digests have been highly successfully in their original domain, searching for similar files in a known database, they have not performed as well for malware classification. Some have found that their results can be improved by changing how the similarity scores are computed(Li et al., 2015), in order to eliminate undesirable properties. For example, the original scoring method did not guarantee symmetry (i.e., $\text{sim}(a, b) \neq \text{sim}(b, a)$). However, they have been useful in related problems to malware detection, such as finding code-reuse among malware(Upchurch and Zhou, 2016).

5.2.2 Normalized Compression Distance

The Normalized Compression Distance (NCD) (Li et al., 2004) is a distance function based on the ability of a compression algorithm to compress two sequences to their smallest possible size. The NCD distance is defined in (5), where C is a function that returns the compressed length of an input, and ab represents the joining of sequences a and b (normally done by simple concatenation). Given an oracle that can perform perfect compression, the NCD is a valid distance metric. This oracle cannot exist, and so NCD must be approximated using algorithms such as LZMA. Because NCD works based on compression, it can be applied to a wide variety of features and has become quite popular, especially for discovering malware families and relationships. It has been used successfully on raw bytes (Wehner, 2007) and on behavior sequences from dynamic analysis (Bailey et al., 2007a).

$$\text{NCD}(a, b) = \frac{C(ab) - \min(C(a), C(b))}{\max(C(a), C(b))} \tag{5}$$

The flexibility of NCD, in that it can be applied to anything encodable as a sequence of bytes, makes it a powerful tool given the multiple different features we may wish to extract. NCD has also seen considerable use for malware detection due to its accuracy, which improves with the quality of the compression algorithm used. Yet the limits of existing compression algorithms also mean that NCD has difficulty in the face of long sequences (Cebrián et al., 2005), causing the distance metric to break down. Attempts to improve NCD have been made by changing how the concatenation ab is done in practice (Borbely, 2015), but room for improvement still exists. When sequences are short enough that NCD works well, a yet unexplored possibility is to use it with some of the kernels discussed in subsection 4.3. A simple merging would be to replace the Euclidean distance in the RBF kernel with the result of NCD, producing $K(a, b) = \exp\left(-\gamma \cdot \text{NCD}(a, b)^2\right)$.

5.2.3 Lempel-Ziv Jaccard Distance

Inspired by the successes and background of NCD, the Lempel-Ziv Jaccard Distance (LZJD) has been developed for malware classification (Raff and Nicholas, 2017a). It draws from the observation that the compression algorithms that tend to work best in this domain make use of the Lempel-Ziv dictionary encoding scheme(Alshahwan et al., 2015; Borbely, 2015), and that the compressed output itself is never used. Instead LZJD creates the compression dictionary, and then measures the similarity of binaries using the Jaccard distance (6) between the dictionaries.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{6}$$

This alone was shown to be more accurate for nearest-neighbor classification of malware, and can be made nearly three orders of magnitude faster than NCD through the use of min-hashing(Broder, 1997),

thus alleviating the runtime cost of NCD. In addition to being faster, LZJD retains the distance metric properties lost by NCD(Raff and Nicholas, 2017a). The use of the Jaccard similarity / distance also means that it is a valid kernel, and can be directly applied to the methods discussed in subsection 4.3.

Raff and Nicholas (2017b) developed a method of converting LZJD's dictionary into a fixed length feature vector using a technique known as the "hashing trick" (Weinberger et al., 2009; Li et al., 2012). More interesting is their observation that the compression dictionary is sensitive to byte ordering, and single byte changes can cause large changes to the dictionary. They exploited this weakness to develop an over-sampling technique for tackling class imbalance, an issue we will discuss further in subsection 8.3. This was later refined to require less hyper-parameter tuning for easier use (Raff et al., 2019b).

LZJD represents an approach of applying the intuition of NCD to a specific compression algorithm, the Lempel Ziv approach. Another approach along this theme is the Burrows Wheeler Markov Distance (BWMD), which again applies the intuition of NCD to the Burrows Wheeler compression algorithm (Raff et al., 2020). The Burrows Wheeler method is not as effective a compression approach as Lempel Ziv, and is reflected in BWMD not being quite as accurate as LZJD in nearest neighbor search. The benefit from BWMD comes from it producing a euclidean feature vector, rather than a a digest like LZJD does. This makes BWMD compatible with a wider class of ML algorithms, which showed how BWMD could produce better clustering and orders of magnitude faster search by leveraging more appropriate algorithms that require euclidean feature vectors (Raff et al., 2020).

5.3 Convolutional and Recurrent Neural Networks

As we discussed in subsection 4.5, neural networks have become popular algorithms and can be viewed as a graph defining a large and potentially complicated function. This flexibility allows them to be extended to sequence tasks by replicating the same network for every step in the sequence. This is often referred to as "weight sharing", and leads to the idea of the Convolution Neural Network (CNN)(LeCun et al., 1989). The success of CNNs in both image and signal processing has been long recognized (LeCun and Bengio, 1998). CNNs embed a strong prior into the network architecture that exploits the temporal/spatial locality of these problems. The convolution operator essentially learns a neuron with a limited receptive field, and re-uses this neuron in multiple locations. Thus a neuron that learns to detect edges can be reused for each part of the image, since edges can occur just about anywhere in an image. This property is not a perfect fit for malware sequence problems, and it remains to be seen if they will be useful despite the structural mismatch. CNNs may be a better fit at higher levels of abstraction, and have been applied to system call traces (Kolosnjaji et al., 2016). We also note that on their own, convolutions do not completely handle the variable length problem that comes with the malware domain.

One common method of dealing with variable length sequences is to further extend the weight sharing idea, by adding a set of connections from one time step to the next, using the previous timestep's activations as a summary of everything previously seen. This high-level idea gives rise to Recurrent Neural Networks (RNNs), and we refer the reader to Lipton et al. (2015) for a deeper introduction to the history and use of RNNs. We note that the CNN and RNN encode different priors about the nature of sequences, and can be used together in the same large architecture, or be kept disjoint. We will again refer the reader to Goodfellow et al. (2016) for a broader background on neural networks. Below we will give only high-level details pertinent to models used in malware classification literature.

Naive construction of a RNN often leads to difficulties with both vanishing and exploding gradients (Bengio et al., 1994), making the training process difficult. One older solution to this problem is the Echo State Network (ESN) (Jaeger, 2001). The ESN circumvents the RNN learning difficulty by selecting the recurrent weights via a stochastic process, so that no learning of the recurrent portion is needed. This may also be interpreted as a stochastic process by which we convert varying length sequences to fixed length feature vectors, after which any of the techniques discussed in section 4 may be applied. The parameters that control the stochastic process can be adjusted to sample different types of ESNs, and cross validation can be used to select the hyper-parameters that worked best. This simple strategy has worked well for many problems, and can be applied to a number of different types of learning scenarios (Lukoševičius, 2012). The ESN has been used by Pascanu et al. (2015) to process a set of high-level events, including API calls, for malware classification and found the ESNs to have an accuracy rate almost twice as high as an n-gram based approach.
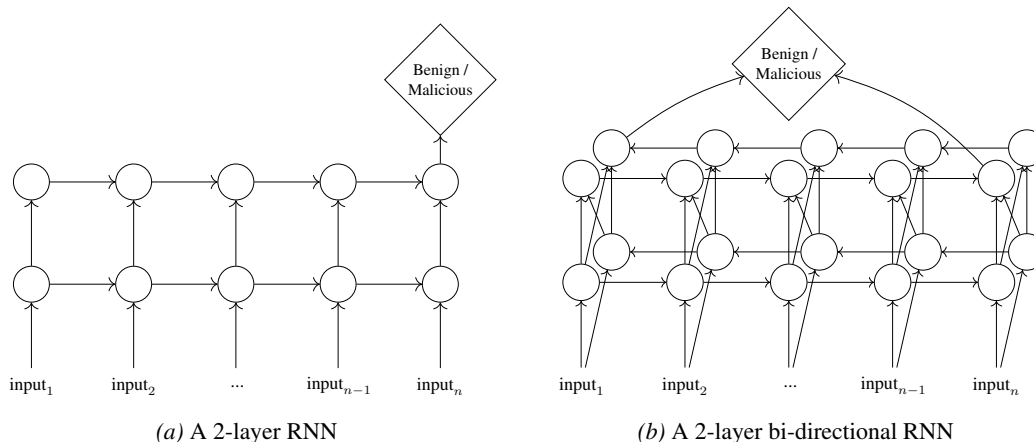
(a) A 2-layer RNN   *(b)* A 2-layer bi-directional RNN

Figure 4. Two simpler RNN architectures for classifying a sequence. In each case the rows of the diagram represent the same neuron being applied on inputs at different time steps. The input includes the bottom up input (previous layer) as well as the previous time step (which is implicitly zero when not present).

In the Deep Learning literature the Long Short Term Memory (LSTM) unit (Hochreiter and Schmidhuber, 1997) has also helped to overcome a number of difficulties in training the recurrent connections themselves, especially in combination with recent advances in gradient-based training and weight initialization. Training works by extending back-propagation "through time" (Werbos, 1988), which amounts to unrolling the sequence of input and output transitions over the course of the sequence (i.e., weight sharing across time). This produces a directed acyclic graph, on which normal back-propagation can be applied. Two examples of this are given in Figure 4, where the neurons in each column all share the same weights. Back-propagation can then be done normally on this unrolled architecture, and the shared weights will be updated based on the average gradient for each time the shared weight was used. This also means that any of the architecture types discussed in subsection 4.5 can be used with RNNs, either before, after, or in-between recurrent stages, and trained jointly with the rest of the network.

Kolosnjaji et al. (2016) have exploited the flexibility of neural networks to combine LSTMs with CNNs for malware classification based on API call sequences. The architecture combination allows the CNNs to learn to recognize small groupings of co-occurring API calls, and the LSTM portion allows the information from multiple co-occurrences to be captured through the whole call trace to inform a decision, and were found to out-perform HMMs by 14 to 42 percentage points. Similar work was done by Tobiyama et al. (2016).

Using just LSTMs, Raff et al. (2017) instead looked at byte sequences and were able to show that LSTMs could learn to process the bytes of the PE header sequentially to arrive at accurate classifications. They also used an *attention mechanism* to show that the LSTM learned to find the same features as a domain knowledge approach learned to use when the features were manually extracted. The purpose of an attention mechanism is to mimic the human ability to focus on only what is important, and ignore or down-weight extraneous information. This has become a tool often used in Machine Translation, and offers a more interpretable component to a neural network.

CNNs to process the raw bytes of a file where first introduced by Raff et al. (2018), which treated the raw bytes of a executable file as a 2 million byte long sequence. Their work found that many of the best practices for building neural networks for image, signal, and natural language processing did not carry over to learning from raw bytes. Notably this required using a shallower and wider network (rather than deep and narrow), and the abandonment of layer normalization techniques like Batch-Norm(Ioffe and Szegedy, 2015). Krčál et al. (2018) expanded this work on their own corpus, but also compared with analyst-derived features used by Avast for their AV product. In doing so they found the approach was competitive with their classical AV, but that combining the features learned by the CNN with those their analysts developed had the best accuracy. This indicates that the CNN approach is learning to detect features that were overlooked by the expert analysts. Otherwise the features would be redundant, and their combination should have no impact.

The ability of CNNs and RNNs to handle variable length sequences makes them an attractive tool for malware classification, however they have yet to receive significant application to that task. Part of this is likely due to the added computational burden they bring. Regular neural network algorithms often require the use of GPUs to train for days at a time. RNNs exacerbate this situation with more computations and a reduction in the parallelism of the problem, making it challenging to scale out training across multiple nodes. It is also difficult to train RNNs for sequences longer than a few thousand time steps, which is easily exceeded by call traces, byte sequences, and entropy measurements. It is likely that intelligent combinations of RNNs with other architectures or advancements in training efficiency will be needed before wide use is seen.

5.4 Haar Wavelet Transform

For numeric sequences, such as the windowed entropy of a file over time, wavelets provide a versatile tool for tackling the problem in a number of ways. Wavelets have long been used to perform machine learning over sequences, and are particularly popular for signal processing tasks (Chaovalit et al., 2011). At a high level, wavelets simply aim to represent an underlying signal $f(t)$ sampled at N_f points using a combination of waves added together. A wave is just a function over time that starts and ends at zero, and goes above and below zero in the middle. The seminal Fast Fourier Transform is one type of continuous wavelet that represents a function as a combination of sine and cosine waves.

For malware classification, the Haar wavelet is becoming increasingly popular and has been used in a number of different ways (Wojnowicz et al., 2016a; Han et al., 2015; Baysa et al., 2013; Sorokin, 2011). Of all possible wavelets, the Haar wavlet is discrete and the simplest possible wavelet. The *Haar wavelet* $\psi(t)$ is defined in (7), and has non-zero vales in the range $[0, 1)$. It is positive in the first half of the range, and then negative in the second.

$$\psi(t) = \begin{cases} 1 & \text{if } t \in [0, \frac{1}{2}) \\ -1 & \text{if } t \in [\frac{1}{2}, 1) \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

$$\psi_{j,k}(t) = 2^{j/2}\psi(2^j t - k) \tag{8}$$

Since an arbitrary function $f(x)$ cannot in general be approximated by adding combinations of (7), we need another function that moves the location of the wavelet. Thus the *Haar function* is defined by (8), and is used in a hierarchy of levels to approximate functions. Both j and k are integers, where j is the level of the hierarchy and $k \in [0, 2^j - 1]$ shifts the location of the wavelet. The smallest levels of the hierarchy ($j = 0$) represent course information over large portions of the sequence, and the largest values ($j \to \infty$) represent the smallest details that apply only to specific time steps t.

$$f(t) = c_0 + \sum_{j=0}^{\log_2 N_f} \sum_{k=0}^{2^j - 1} c_{j,k} \cdot \psi_{j,k}(t) \tag{9}$$

We can now represent any function $f(t)$ using (9). However, in practice this only works for sequences that are a power of 2 in length, so truncation to the smallest power of two or padding the sequence with zeros is often necessary. Computing the Haar wavelet transform then determines the values for the coefficients $c_{j,k} \forall j, k$. The discrete Haar transform takes only $O(N_f)$ time to compute, making it a viable tool for longer sequences (Walker, 2008).

A common use case of the Haar transform is to de-noise signal data, and has been used in that context for malware classification (Shafiq et al., 2009). The intuition being that the higher level signal comes from the coefficients from the earlier levels, so we can remove noise by setting $c_{j,k} = 0 \forall j \geq z$, for some threshold z. Once set to zero, we can then re-construct the original signal which should now have less noise present. While the de-noising application is fairly straightforward and is a standard use case for wavelets, once we have this representation there exist a number of possibilities for comparing sequences.

5.4.1 Haar Distance

Struzik and Siebes (1999) propose comparing time series by defining a distance over the coefficients $c_{j,k}$. First they define a correlation between two Haar transforms as given in (10), where the Kronecker delta function $\delta(j_a, k_a, j_b, k_b)$ returns 1 iff $j_a = j_b$ and $k_a = k_b$, otherwise it returns zero. Using this a normalized distance between two sequences is defined by (11). While this would allow for a nearest neighbor classifier to be used, this specific approach has not yet been used for malware classification.

$$C(f_a, f_b) = \sum_{j_a=0}^{\log_2 N_{f_a}} \sum_{k_a=0}^{2^{j_a}-1} \sum_{j_b=0}^{\log_2 N_{f_b}} \sum_{k_b=0}^{2^{j_b}-1} c_{j_a,k_a} c_{j_b,k_b} \delta(j_a, k_a, j_b, k_b) \tag{10}$$

$$D_{\text{Haar}}(f_a, f_b) = -\log \left| \frac{C(f_a, f_b)}{\sqrt{C(f_a, f_a) \cdot C(f_b, f_b)}} \right| \tag{11}$$

5.4.2 Haar Energy Features

Another approach is to extract a fixed-length representation from the Haar transform, and use those features as inputs into a classifier (Pati and Krishnaprasad, 1993). Wojnowicz et al. (2016a) used this approach on entropy sequences, and found them to be moderately effective independently, but not sufficient for a complete system. For a sequence of length N_f, we can compute $\log_2 N_f$ energy levels according to equation (12). In the case of Wojnowicz et al. (2016a), they decided to use all possible levels and handled different length sequences by building a new classifier for every different power of two.

$$\text{Energy}_j = \sum_{k=1}^{2^{j-1}} (c_{j,k})^2 \tag{12}$$

5.5 Dynamic Programming

A method of comparing numeric sequences that has become increasingly common in the literature is to perform distance computations using dynamic programming methods such as the Levenshtein distance (Levenshtein, 1966; Bellman, 1954). While the use of so-called edit-distances could be applied directly to raw binaries, a run-time complexity of $O(N^2)$, where N is the length of a binary or entropy sequence, is prohibitive. The Haar transform has been used to make such approaches practical. The Haar transform is used to discretize the numeric entropy sequence into bins of varying size (resulting in a shorter sequence), which are then used in the dynamic programming solution (Han et al., 2015; Baysa et al., 2013; Sorokin, 2011; Shanmugam et al., 2013). While similar at a high level, these approaches all have a number of differences and details that do not lend themselves to a compact summary.

Instead we summarize a similar dynamic programming distance for sequences called *Dynamic Time Warping* (DTW) (Berndt and Clifford, 1994). DTW works directly upon numeric time series, and does not need the discretization provided by the Haar Transform (though the wavelets could be used to pre-process the signal before computing DTW). This method has received minimal investigation for malware classification (Naval et al., 2014), but has been used for related tasks such as estimating the prevalence or malware infection (Kang et al., 2016) and detecting botnets (Thakur et al., 2012). Its description is also representative of a simpler common ground between the prior work in this area (Han et al., 2015; Baysa et al., 2013; Sorokin, 2011; Shanmugam et al., 2013).

Given two sequences $f_a(t)$ and $f_b(t)$ of lengths N_{f_a} and N_{f_b} respectively, DTW attempts to find a continuous path of point-wise distance computations that minimizes the total distance. Doing so requires finding a sequence of dilations and contractions of one sequence to fit the other, such that it maximizes the similarity between their shapes. In other words, DTW attempts to measure the distance between two sequences based on overall shape, and ignoring any local contractions or elongations of a sequence. To do so, we define the distance using the recursive equation (13), where $\text{DTW}(f_a, f_b) = \text{DTW}(N_{f_a}, N_{f_b})$. This equation can be solved using dynamic programming, and is

similar to the Levenshtein style distances that have been more prevalent in the malware classification literature.

$$\text{DTW}(i, j) = (f_a(i) - f_b(j))^2 + \begin{cases} 0 & \text{if } i = j = 1 \\ \min \begin{cases} \text{DTW}(i-1, j) \\ \text{DTW}(i-1, j-1) \\ \text{DTW}(i, j-1) \end{cases} & \text{otherwise} \end{cases} \tag{13}$$

Like many other dynamic programming methods, DTW takes $O(N_{f_a} N_{f_b})$ time to compute and has the disadvantage that it is not a true distance metric, as it does not obey the triangle inequality. However, the popularity of DTW in other domains warrants its consideration for malware classification, especially as an existing body of literature addresses many of the known problems and challenges. In particular there are multiple methods of speeding up the DTW computation and retrieval, including the construction of indexes (Keogh and Ratanamahatana, 2005), a fast $O(\max(N_{f_a}, N_{f_b}))$ approximation (Salvador and Chan, 2007), and a definition for a DTW centroid (Petitjean et al., 2014). There even exist methods of learning constraint costs to modify the DTW, which can improve the accuracy of the constrained DTW (Ratanamahatana and Keogh, 2004). This could simplify the manually constructed costs and constraints used in existing malware research (Han et al., 2015; Baysa et al., 2013; Sorokin, 2011).

6 Graph Algorithms

While representing information as a sequence reduces the gap between abstraction and the true nature of the data, this is still a simplification in many instances. For example, while a binary may be one long sequence of bytes, the order in which the bytes are evaluated and accessed may be non-linear. A yet richer representation is as a graph G of vertices V and edges E. Graph analysis and algorithms have been widely studied and used in many domains, and malware classification is no exception. Such techniques have been used most commonly for representing assembly(Alam et al., 2015; Anderson et al., 2011; Hashemi et al., 2016) and API and system calls (Elhadi et al., 2014) collected from dynamic analysis. While these two feature types have already seen use as sequences and with classical machine learning, graphs have also been used for features not well represented in other forms, like mapping the relationship between a malware *per se* and the files it creates or accesses (Karampatziakis et al., 2012).

Similar to section 5, we will review the common graph-based approaches that have been used for malware classification. While the appropriateness of a graph representation has been widely recognized in prior work, little has been done to fully exploit such representations. Most of the prior work in graphs for malware classification use either graph matching, or attempt to construct more meaningful feature vector representations that will allow us to use the wide breadth of machine learning methods in section 4. While there exists a rich diversity in what may be done with graphs, a full survey of graph methods is beyond the scope of this work. Instead we will review the two high-level approaches that have been used in the malware classification literature.

6.1 Graphs to Feature Vectors

Many of the techniques for directly comparing graphs are computationally demanding, often requiring approximations or simplifications to be made. For this reason it is often desirable to convert a graph representation to a fixed length feature vector. Once converted, the classical machine learning methods discussed in section 4 can be used. The simplest approach is to manually craft features about nodes in the graph, and then use those for classification (e.g., as used by Kwon et al. (2015)).

6.1.1 Graph Features and Descriptors

There exist naive ways to create a feature vector from a graph, such as flattening the n by n adjacency matrix into a vector of length n^2 (Eskandari and Hashemi, 2012). But such approaches are often impractical, causing excessively high dimensional spaces (making learning more challenging) and relying on extremely sparse graphs. Another simple approach to creating feature vectors from graphs is to derive informative statistics about the graph itself. Such features could include the number of

vertices or degrees with a given label, the average number of in/outbound edges for a vertex, and various other statistics. This approach was used by Jang et al. (2014) over the system call graph of a binary. This approach has the benefit of being quick to implement and apply, but does not take full advantage of the rich information that can be stored within a graph. It also requires the developer to have some notion about which statistics will be informative to their problem.

These kinds of approaches are often not sufficient in practice, but are easier to apply and generally not compute intensive. Outside of malware classification, machine learning combined with these simple feature types can be used to accelerate other approaches as a kind of low-pass filter, pre-classifying items as likely or unlikely to be related (Lazarescu et al., 2000). This filtering can be used to reduce the compute time of the other approaches we will discuss in this section, which are often more compute intensive.

6.1.2 Graph Embedding

A less labor-intensive means of converting a graph into a feature vector is to perform some type of embedding, where the graph is converted to a fixed-length feature vector through some decomposition (Luo et al., 2003). This is often done by taking the adjacency matrix representation of a graph, and performing an eigenvalue decomposition (or singular value decomposition (SVD)), and using the eigenvalues as the feature vectors. This captures information about the graph as a whole, and does not require manually specifying items of interest.

These embeddings can be used directly for machine learning, or can also be used as a first step toward other methods discussed in this section, such as graph matching in subsection 6.2 (Luo and Hancock, 2001). Hashemi et al. (2016) used this strategy with assembly code, representing an edge in the graph every time one instruction followed another and weighting the edges by the number of times the instruction pairing occurred. Anderson et al. (2011) has also used this strategy for assembly sequences in combination with other feature transforms of a graph. A unique approach by Slawinski and Wortman (2019) defined graphs over the extract abstract syntax trees of functions, and used a PageRank based embedding to better capture information from statically extracted features. Using just the static features in a fixed length feature vector they obtained 89.19% accuracy, which increased to 98.28% when leveraging the graph weighted embedding (Slawinski and Wortman, 2019).

6.2 Graph Matching

The most prevalent method in use for malware classification is the concept of *graph matching*. Given two graphs G_1 and G_2, in the case of this work, the goal is to derive some distance function that will give us a measure of how close G_2 is to G_1. This can be done with all types of graphs, and the methods used may change with the graph size and type.

The general strategy to create this distance measure is to determine the amount of correspondence between the graphs, often by determining how to transform one into the other. There are various strategies to using graph matching as well, with some works even defining their own matching algorithms (Haoran Guo et al., 2010), though more simple heuristics are more common (Kolbitsch et al., 2009). One method is to create templates of what malware looks like, and match to that template (Haoran Guo et al., 2010). Graph matching is also popular for malware family classification, where such systems are often designed for low false positive rates (Park and Reeves, 2011). Graph matching can also be used for normal nearest neighbor type classification against a large database, though this is often avoided due to computational expense (Hu et al., 2009).

6.2.1 Graph Edit Distance

One common approach to graph matching is to create a graph edit distance. Computing the exact graph edit distance is generally NP-complete, and so various different approximations are used in practice. Such approximations are often done via dynamic programming strategies, as discussed in subsection 5.5. These approximations can still be too slow to use for larger corpora, which has also spurred the use of indexing structures and other optimizations to accelerate queries for similar binaries(Hu et al., 2009).

Park et al. (2010) determined distances by finding the maximal common sub-graph between G_1 and G_2, and returned the edit similarity as the cardinality of the sub-graph over the maximum cardinality

of G_1 and G_2. Their graph also used vertex and edge labels depending on the arguments used. Related work has used this common sub-graph approach to build systems with the goal of having minimal false positive rates (Park and Reeves, 2011), producing a kind of system-call "signature" for finding malware families.

Elhadi et al. (2014) used graph matching on labeled graphs derived from both the system call traces and the operating system resources used by each call. The edges and vertices in the graph had different labels depending on the whether the vertices came from each respective group.

6.3 Other Approaches

There are many ways to perform similarity comparisons between graphs, not all of which can be described as a variant of graph matching. Lee et al. (2010) used a metric similar to the Jaccard distance, by having a finite space of possible vertices they were able to take the intersection of the graphs' edges over the union of the edges. Part of the motivation of this was faster compute time so that the system could be practical for real use. Their approach was also unique in that they used call traces from static code analysis, rather than dynamic execution. However, this did necessitate that their approach use unpacking software before code analysis.

Graphs also need not be the method by which similarity is done, but can still be an integral component of the system. Eskandari et al. (2013) used an extensive graph-based approach to process API and system calls from both static and dynamic analysis, and used the graph structure with node classification to infer what a dynamic analysis would have looked like given only the static analysis of the binary. This was to obtain the speed benefits of static analysis (at runtime), while retaining the accuracy benefits of dynamic analysis. Yet their system used a simple fixed length feature vector for the final determination of maliciousness.

7 Evaluation and Metrics

We have now extensively considered the many predictive approaches that have been applied to malware classification. While such techniques are often the most exciting or interesting part of an approach, it is often done without fully considering how such systems should be evaluated. Despite its importance, evaluation metrics and choices are a commonly overlooked part of the machine learning process. Even more so, such choices often do not consider the evaluation of a system as a whole. Most works will simply use overall accuracy or AUC, which are described with some other less frequently used metrics in Table 2.

While maximizing the performance of these metrics can be revealing and informative on its own, it is often done without necessarily justifying why these metrics are the ones to be used, or considering other explicit or implied goals. Each possible metric will be biased towards certain properties, which may impact what model appears to work best even, if it performs worse in practical use (Powers, 2011). Thought should be given to system constraints, the metric of performance, and the specific use case for deployment. The latter of these concerns should in general drive the process as a whole, informing us as to which constraints we can computerize and which scoring methods most closely reflect our needs.

Early work by Marx (2000) developed a thorough set of guidelines and processes for evaluating an AV product, from ease of user by customers to developing and weighting a testing set, and how to perform testing for different types of malware. The crux of their argument is the necessity for evaluation to reflect real world use. It was recognized then that accurately reflecting real world use of malware detectors is anything but trivial, and (we argue) has only gotten more challenging over time.

In this section we will review a number of different scenarios and constraints that may impact how we choose to design and evaluate a malware classification system. Intrinsically related to this is the data quality issues we discussed in section 2. Having high quality data not only means better models but more accurate estimation of model performance. While we have already reviewed why this is not a given, for brevity we will generally assume that the data issue has been resolved in this section.

Table 2. A few metrics that have been used within the malware classification literature. Most works have used only accuracy and AUC.

Metric	Description
Accuracy	The number of data points classified correctly, divided by the total number of data points
Balanced Accuracy	Same as accuracy, except each class has equal weight to the final result. See (Brodersen et al., 2010).
Precision	The number of true positives divided by the true positives and false positives
Recall	The number of true positives divided by the true positives and false negatives
AUC	Takes the integral of the Receiver operating characteristic curve, which is a plot of the true positive rate against the false positive rate. See (Bradley, 1997).
F1 / F-Measure	The harmonic mean between precision and recall.

7.1 Malware Detection

The most obvious goal of malware detection is to act as a replacement for anti-virus products. In this case accuracy and AUC are both acceptable target metrics that are widely used in machine learning. This does not necessarily make them the best metrics when we consider the user of such a system. Numerous works have emphasized the importance of having low false-positives in this scenario (Ferrand and Filiol, 2016; Masud et al., 2008; Alazab et al., 2011; Zhou and Inge, 2008). This is because excessive false positives will be aggravating for the end user, who presumes their desirable goodware applications will continue to work without issue when using an anti-virus. If required to add applications to a white-list too frequently, they will get frustrated and stop using the product. This leaves them exposed to malware, and the accuracy of the system becomes irrelevant. While no consensus has been reached as to an acceptable false positive rate, most work that emphasizes the issue achieve rates between 1% and 0.1% (Yan, 2015; Shafiq et al., 2009). Existing anti-virus products have been tested with fewer than 20 false positives out of over 1 million test files (AV-TEST, 2016b), though the exact details of the test set are not public.

Another use case for malware detection is to rank a backlog of binaries for analysis or investigation. This scenario can occur for malware analysts or incident response personal, when a large number of items need to be considered and it is important to find the malware as soon as possible. The AUC metric is in fact well designed for this use case, as it can be interpreted as measuring the quality of a ranking. In this way the metric accurately aligns to the goal, in that we want to make sure analysts spend their time looking primarily at malware first. If left unordered, the chance of dealing with a benign file before all malware is dealt with increases, which takes up valuable time but does not result in any new insights.

7.2 Malware Family Classification

When performing malware family classification, the normal procedure is to collect some number of malware families C, and divide the dataset into a training and testing set (or use cross validation) to evaluate the accuracy of determining the malware family for new binaries. Because this situation is not commonly needed for desktop deployment, the same run-time constraints are not often emphasized for this task. This evaluation is somewhat flawed with respect to real-life use case scenarios, as inevitably new malware families will be found that are not in the existing set. We argue that there should to be a pseudo $C + 1$'th family for "not in existing families" when evaluating malware family classification. The binaries that will belong to this $C + 1$'th class should also come from multiple malware families, and not be present in the training data. This will better evaluate the quality of a system with respect to situations that will be encountered during real-world usage, but does increase the design burden by requiring both a classification and anomaly detection ability.

The importance of correctly marking a file as "not previously seen" is exemplified in forensic use cases, where a set of known malware is compared against potentially terabytes of data(Roussev, 2010;

Kornblum, 2006; Roussev and Quates, 2012). Similarity preserving hash functions, which have low false positive (and recall) rates, are often used in this scenario.

If we accept the need for a "not previously seen" category, it is also worth considering if benign files should be included in the evaluation. Normally malware family classification is done under the assumption that all files *are* malicious. In practical use, errors will occur — and so it is likely a benign file will occasionally be given. It seems reasonable that the "best" option (given that the benign file was already mislabeled as malicious) is to mark the benign file as "not previously seen". This is an issue we have not yet seen discussed or included in evaluations.

We also note that while a rich diversity of metrics have been used with binary classification problems, such as accuracy, AUC, F_1, Matthews correlation coefficient, etc., most works on malware family classification simply use accuracy. This is not necessarily a good choice, as the distribution of malware families is not uniform. While balanced accuracy is one alternative, it can also pose a problem with very rare malware families. These families will be harder to classify yet have a more significant impact on the balanced accuracy. There is also a question as to how many malware families should be used when evaluating. Others have also proposed that malware should be instead grouped by behavior or effects, given the non-standardized and inconsistent nature of malware family labeling (Bailey et al., 2007a).

7.3 Training and Inference Constraints

The most common type of constraints that are considered are run time requirements. In particular many works have been concerned with real-time inference (Dahl et al., 2013; Alam et al., 2015; Khan et al., 2010), which usually implies a strict or moderate limit on memory use as well. This scenario makes perfect sense from a deployment perspective, for systems that would act in the stead of anti-virus products or network appliances that would inspect live traffic. If such systems were to impose too significant a latency on application start up or network connections, people would stop using the product due to the loss in productivity and aggravation. If this real-time requirement is violated, accuracy becomes irrelevant because the solution is not used. This situation would also discourage us from considering the full breadth of dynamic features, as certain combinations may prove too compute intensive to be practical. While many works report sub-second classification time per datum, there appears to be no current consensus on where the threshold for "real-time" is for malware classification, and most works simply emphasize that their solutions execute quickly.

Another consideration that is less frequently addressed is training time and scalability, particularly as corporate malware collections are on the order of hundreds of millions, if not billions, of samples (Spafford, 2014). In particular it would be ideal for a malware training system to require only one pass over the training data, as data access often becomes the bottleneck at larger scales (Wojnowicz et al., 2016b). Not only does reducing the training time save money (as less resources are needed), but it also allows for tackling the problem of concept-drift through change-detection (Gama et al., 2004; Baena-Garcia et al., 2006; Bo-Heng Chen and Kun-Ta Chuang, 2014). This is a common method of dealing with the difficulties of adapting a model to a changing concept. Instead one uses a change detection algorithm to determine when the concept has drifted significantly enough that accuracy may begin to decrease. At that time one then simply re-trains the model on the newest data, and switches all old models to the most recent and up-to-date version. Though an important consideration is that older data may still be useful to train on, making it necessary to balance between new data and older (yet still informative and representative) data. We refer the reader to (Gama et al., 2014) for a survey of many approaches to change detection. In practice the change detection may not be needed if one instead trains at a fixed interval, say every few months. We are not aware of any work that quantifies this problem on a large dataset over many time frames.

7.4 Specialized Needs

In a conversation about the constraints and metrics that must be met for a malware classification system, it is also important to discuss scenarios with specialized needs. These may be uncommon deployments or scenarios where a system designed for the majority of potential users does not satisfy important requirements. By the nature of such a scenario, we cannot enumerate all possible specialized needs. Instead we present an example scenario that has had some investigation, and how that may impact the measures of success.

A particular variant of the malware detection problem is to detect specific types of malware, such as ones that actively try to avoid detection(Stolfo et al., 2007; Kirat et al., 2014). This can be important for researchers and analysts who wish to track more sophisticated threats, or desire to study a particular type of malware. If such binaries are going to be manually analyzed afterwards, we want to make sure that selected binaries are worth an analyst's time, and so a high precision is required from the system. It would also be appropriate to evaluate the precision at multiple thresholds, reflecting the potential capacity to analyze such binaries on the part of a team that is abnormally busy, under normal workload, or blessed with excess availability. Another way this may be desirable is based on the type of infrastructure that needs protection. If a company's services were deployed in a cloud environment, malware that brings down a single VM may not be a significant issue, as one can easily provision and replace the infected VM. However, malware that exfiltrates data on the VM to a third party may cause the loss of proprietary or confidential information, and thus be a heightened concern. In this case we want a malware evaluation model adjusted to the type of exploits which can cause the most damage to a particular entity and environment.

7.5   Evaluating Over Time

It is also important to evaluate a system based on the longevity of the model's utility. That is to say, we may desire a model that is more robust to concept drift, perhaps at some cost in terms of another metric. This would be important for any system that in some way has limited Internet connectivity, making it expensive (or impossible) to update the model over time. This does not necessarily prevent malware from trying to infect the device when online, or from someone with physical access attempting to install malware on the device. In this case the model needs to be robust to malware over longer periods of time to thwart such efforts until an update of the model can be completed. In our view, Saxe and Berlin (2015) is one of the more important attempts at performing such an evaluation. They used the compile date provided in the PE header to split the dataset into before and after July 31st, 2014. Under this scenario their malware detection rate dropped from 95.2% to 67.7% for a fixed false positive rate of 0.1%.

This dramatic drop shows the importance of considering time in evaluations, which is not a common practice. One issue is that the compile date in the PE header is easily modified, and so malware authors can alter the value seen. File first-seen date may be a usable alternative source for this information, but is necessarily less precise. Performing evaluations split in time like this also requires significant data from a wide breadth of time. This exacerbates the need for good benign data mentioned in section 2. Not addressed in Saxe and Berlin (2015), but also worth considering, is evaluating the performance on the test set as a function of time — which would allow one to more precisely characterize the longevity of generalizing information.

The EMBER dataset follows this evaluation protocol, with date-first-seen information provided by an external source rather than the time stamp of the file (Anderson and Roth, 2018). This avoids the problems caused if the malware lies about its creation date, but has less precision. Valuable information could be obtained by doing a comparative study between these different types of date information, seeing how well they correlate, and how the choice impacts results.

There are still many important questions related to a malware model's performance over time that have not been answered. Most notably, for how long is a model effective? What are the important factors to a model's longevity (i.e., is data or the model type more or less important)? How old can training data be before it becomes uninformative or even detrimental to training? Presuming that such a threshold for training data exists, do benign and malicious data have a different "shelf-life" in terms of utility for training?

7.6   Evaluating Under Adversarial Attack

Machine Learning models are generally susceptible to adversarial attacks. In this context, an adversarial attack against a machine learning model means that an abstract adversary is able to modify an input such that it has not meaningfully changed, yet a machine learning model will run an increased risk of misclassifying the modified example. We refer the reader to Biggio and Roli (2018) for a review of the history of adversarial machine learning and how it is normally performed.

For malware classification, we have a real live adversary (the malware's author) and the use of adversarial machine learning techniques can become a new tool for the malware author to avoid

detection. As such the problem of attack and defending malware classifiers against such attacks is an important area for study, and potentially included as part of the evaluation metrics (Fleshman et al., 2018). The application of these techniques to the malware space is not trivial, however. Normally an input crafted by an adversary will modify its input parameters over a continuous space of values, all of which are valid inputs. For example, in an image the pixel values will change to other adjacent pixel values. However binaries can't be altered in the same way, and changing arbitrary bytes can be destructive to the executables' function.

Anderson et al. (2017) showed one of the first attack methods for arbitrary binaries against arbitrary malware detectors (including regular AV products). They did this by defining a space of possible non-destructive transformations that could be applied to the binary so that its functionality would be maintained. They then trained a reinforcement learning algorithm to learn which transforms it should apply. Importantly, by uploading a sample of files to Virus Total they found that their adversarial modifications, which have no impact on functionality and no awareness of how the AV products worked, was able to evade several of them.

To circumvent the issue of arbitrary byte changes breaking an executable Kolosnjaji et al. (2018) and Kreuk et al. (2018) concurrently developed an attack that avoids this difficulty. They added an unused section to the binary, which is allowed by the PE format, and constrained all of their modifications to this lone section. This allows for inserting arbitrary byte content to trick the detector, but avoids impacting the functionality in any way. While developed as an attack against the byte-based CNNs discussed in subsection 5.3, the approach can be leveraged against other methods as well. Recently Fleshman et al. (2019) proposed an approach that defeats this attack in all cases, but at a cost to accuracy.

These recent works have operated in the threat model that the adversary (malware author) can only add features. While this threat model is reasonable, it is not perfect. This is especially true in the static analysis case, where whole file transformations like packing already exist. We expect future work to attempt new ways to add information to the static case. The dynamic case is less clear, as the malicious behavior needs to eventually run for the malware to be malicious. This is muddied further by malware detecting its environment, as discussed in subsection 3.2. For dynamic analysis, we expect malware authors would further invest in VM evasion techniques, and there always exists the potential for more sophisticated attempts to hide actions taken from the inspecting VM.

While these results are all new, they represent an important area in which we must evaluate each model's accuracy both against unseen binaries, and adversarialy altered ones. This is in addition to the adversarial techniques that were developed to avoid classical AVs (Poulios et al., 2015; Thomas Kittel et al., 2015).

8   Future Research Potential

At this point we have now reviewed the current data, models, and evaluation methods used for malware classification. This includes many challenges, and some potential improvements that exist throughout the whole process. While some areas have received more and less attention than others, we note that there exist relevant techniques in machine learning that have gone almost or are completely un-applied to the problem of malware classification. We briefly review a number of these that we believe could lead to future progress.

8.1   Multiple Views of Malware

Given the wide array of potential feature sources, extraction methods, representations, and models that can be used for malware classification, we could consider each of these combinations a different "view" of the malware problem. Each will be biased toward detecting different types of malware with different representations. This can be a beneficial scenario for ensembling, where multiple models are combined to form a decision. It is commonly found that ensembles work best when the members of the ensemble make uncorrelated errors, which is more likely when each model has a different "view" of what malware looks like. Despite this seemingly good fit, we have found little work in using ensemble methods to combine multiple distinct feature types. Singh et al. (2015) used a SVM to combine the predictions of three models and found it to perform considerably better (though they did not connect that this is a form of Stacking ensembles (Wolpert, 1992)). However, in their experiments

all three models used assembly instructions as the source feature — reducing the potential diversity of the model. Masud et al. (2008) used three different feature types (assembly, byte n-grams, and PE header imports), but instead combined all feature types into one large feature vector. Eskandari et al. (2013) did some work providing a different take on looking at a binary in multiple different ways. They used API call traces features collected by both static and dynamic analysis, where both were used at training and only static analysis was used during testing. They modeled what decisions might have been made in the static version from the dynamic, so that at testing time the faster static-only analysis could be used.

There may be many other ways to exploit the rich diversity in feature and models types that can be applied to malware classification, instead of focusing on finding the single method that performs best. For example, it is possible that different model and feature types work best for differing types of binaries. A Mixture of Experts (Jacobs et al., 1991) ensemble could be built that learns to recognize which models will work best on a given binary, and having that limited subset vote on the decision. We believe this is a mostly unexplored avenue for future improvements in malware classification.

8.1.1 Static and Dynamic Staged Views

While we have discussed some work that uses features collected from both static and dynamic analysis (Islam et al., 2013; Damodaran et al., 2015), we have not seen any work that considers models that are first used in a static pipelines, followed by a dynamic one if the static feature model was unsuccessful. This intentionally keeps the feature spaces disjoint, and creates a cascade reminiscent to those used in early face detection work (Viola and Jones, 2001).

This approach recognizes that dynamic analysis is considerably more expensive than static analysis, both in terms of time to decision and computational resources. It is unrealistic to expect to run everything through dynamic analysis, and so a tiered approach should be used instead. This appears to be an accepted and common practice amongst AV vendors[12], but we are not aware of any published research recognizing the need for such staged approaches.

8.2 Dealing with Labeled Data Expense

Given the time intensive and data gathering issues discussed in section 2, further research is warranted in how to perform malware classification with minimal labeled data. In this vein, it is surprising that no work has yet been done to apply *semi-supervised* learning to malware classification. Semi-Supervised learning involves building a classifier using both labeled and unlabeled data (Zhu, 2008). Semi-Supervised learning also provides a training time workaround for when only a few anti-virus products mark a binary as malicious, casting doubt as to its true labeling. A semi-supervised solution would be to use all the data for which we are sure of the label (no anti-virus fires, or almost all fire) as labeled training data, and the ambiguous cases as unlabeled training data. In this way we avoid poisoning the learning process with bad labels, but do not throw away potentially valuable information about the decision surface. The best type of semi-supervised algorithm to use in this domain is not obvious, and many algorithms make differing assumptions about the nature that unlabeled data and the impact it should have on the decision surface.

If we are to use a small amount of labeled data, it is also important that we label the most informative data possible. *Active Learning* is a technique that can help with this. Active Learning starts with an initial dataset, and a collection of unlabeled data. The model then can select data points for which it would like to request the label. While this framework is often used to help derive algorithms that learn more efficiently and quickly, it is also directly applicable to deciding which data points we should get labels for. The potential impact of having an active learning system was shown by Miller et al. (2016), where a simulation of human labelers found that their system's detection rate could be improved from 72% to 89%, while maintaining a 0.5% false positive rate. However, their approach to active labeling was heuristic, and did not leverage the full potential literature of active labeling methods. There has been little other research in this area for malware classification (Moskovitch et al., 2009b), and so many questions remain. Are the features best for prediction also the best for active learning? What kinds of active learning algorithms work best for this domain? How are these

---

[12]For example, see `https://cloudblogs.microsoft.com/microsoftsecure/2017/12/11/detonating-a-bad-rabbit-windows-defender-antivirus-and-layered-machine-learning-defenses/`

methods impacted by the concept-drift of binaries over time? All of these questions, as far as the authors are aware, have yet to be explored.

8.3   Learning with Class Imbalance

Most machine learning algorithms presume equally balanced classes (He and Garcia, 2009) at training time, and thus also at testing time. Learning from imbalanced classes can naturally cause the algorithm to favor the more numerous class, but also be detrimental in failing to learn how to properly separate the classes. Class imbalance is a common problem in the malware domain (Patri et al., 2017; Cross and Munson, 2011; Li et al., 2017; Yan et al., 2013; Moskovitch et al., 2009a; Yan, 2015), which makes it especially important to consider the evaluation metric used for both malware detection and family classification (as mentioned in subsection 7.1 and subsection 7.2).

One way to tackle such imbalance problems is to over-sample the infrequent classes or under-sample the more populous ones. These approaches are common, but can be out-performed by a more intelligent over-sampling or under-sampling process(Laurikkala, 2001). Indeed there is an existing literature for such methods focusing on both approaches(Lemaître et al., 2017), which have seen almost no application to the malware problem. Exploring their applicability to this domain and how such methods may be adapted for the difficulties of sequence and graph based features is, as far as we are aware, an open problem area.

Oversampling the minority class is an intuitively desirable approach, as it allows us to use the larger amount of of majority class data — and thus more data to train on overall. However, naive oversampling can lead to overfitting (Prati et al., 2009; Kubat and Matwin, 1997). One technique to do this more intelligently is to interpolate new datums from the training distribution, and a popular algorithm SMOTE takes this approach and has many variants as well (Nguyen et al., 2011; Han et al., 2005; Chawla et al., 2002). This also assumes a natural fixed length feature vector, and that interpolated instances are intrinsically meaningful. This may not be the case for all malware features, and may not be readily possible for sequence or graph based approaches to malware classification.

Under-sampling is often done intrinsically for malware detection, where the ratio of malicious to benign samples available is large. While less intuitively desirable than oversampling, smarter approaches for this exist as well. One approach is to train multiple models, with different subsets of the majority class used in each model, allowing for the creation of an ensemble(Liu et al., 2009). Other techniques also attempt to more intelligently select the sub-samples(Kubat and Matwin, 1997).

We are aware of relatively little work that explores the problems of class imbalance in this space. Moskovitch et al. (2009a) investigated training set proportion's impact on differing test-set proportions, but worked under the assumption that malware will make up 10% of seen files in a network stream, and in later work simply set the training ratio equal to this ratio (Moskovitch et al., 2009b). Raff and Nicholas (2017b) looked at developing an algorithm specific oversampling technique, evaluating on an assumption of equal class ratio.

It is generally thought that malware will make the minority of samples in deployment, which is supported by a large study on over 100 million machines which found that the number of benign singletons outnumbers malicious ones at a ratio of 80:1 (Li et al., 2017). However, they also found that this ratio changed over time. We think further study is warranted to determine how applicable this rule of thumb is. Do all institutions share the same ratio of benign to malicious, or would certain institutions or individual users who are targets for malware authors have lower ratios? Similarly, do different types of equipment (e.g., desktop computer, router, or mobile device) see differing ratios of malware? How do these rates change with geographical area? Lastly, we suspect there are a number of niche professions with special needs that will see differing distributions. For example, cyber crime investigators inspecting a recovered hard drive may expect to see a much higher ratio of malware given their targeted goals.

8.4   The Good, the Bad, and the Annoying

For malware detection, we also note the the distinction of malicious versus benign may be overly strong. Some anti-virus products refer to what is known as "potentially unwanted programs"[13] (PUP).

---

[13] As an example, the free anti-virus ClamAV has signatures for such software `https://www.clamav.net/documents/potentially-unwanted-applications-pua`

This third class sits in a gray area between benign and malicious. While these PUP binaries do not intentionally do any harm, they may be annoying or aggravating to the user. They also can present a labeling challenge, where it is not clear on which side of the line between benign vs malicious they should fall. While one could treat this as a three class problem, it would be more accurate to apply ordinal regression to alleviate the issue. Ordinal Regression is classification where the class labels are ranked and the error grows with the distance between the correct label and the predicted label (Gutierrez et al., 2016). These errors need not necessarily be linear. Consider for example our case of benign, PUP, and malicious labels. The errors for marking benign as PUP could be 0.1 points, and PUP as malware results in 0.9 units of error. Then marking something benign as malicious would incur 0.1+0.9=1.0 units of error. This type of ordinal regression could be further extended to take into account the severity of certain types of malware. For example, one could instead classify binaries as "benign, PUP, low risk malware, high risk malware". This would naturally require a distinction between risk levels for malware, which could be user dependent. One reasonable distinction between low and high risk malware could be data loss. A machine attached to a botnet would then be low risk, but ransomware would be high risk. Such alternative labeling schemes are an open area for research and discussion.

8.5 Automatic Malware Remediation

One unit of functionality currently provided by anti-virus products is the removal of found malware. This is a critical function of many anti-virus products, and the time it takes analysts to devise a removal strategy can become part of the lag time between initial malware discovery and updates being sent to user systems. Automating this task, even to only an incremental degree, would help reduce human time spent on this problem and provide relief to users when new malware is found.

We are not aware of any work that has yet explored the ability for a machine learning system to determine how to remove malware from an already infected system. Given a corpus annotated with the various methods by which malware may attempt to gain persistence on a machine, it would seem plausible to detect these mechanisms and develop a "removal policy", a series of steps to perform that would remove the malware. This would fall into the domain of Reinforcement Learning as a method of learning and executing such policies.

Multiple malware infections could complicate this task in interesting ways, increasing the challenge involved. The task is also impacted by the operating system version in used, updates installed and available, and other applications installed on the machine. It is unlikely that initial work will address all of these confounding factors at once, but the situations presents a challenging area for AI that could have important practical consequences if successful. One small advantage to the situation, because it is applied only to found malware, arises since it is realistic to use more compute intensive methods for determining a removal policy. This task is unexplored at the moment, and we are not aware of any corpora with such labels.

8.6 Malware, Natural Language Processing, and Reports

A common task for malware analysts is to generate reports of their findings, which are shared with other security processionals to make them aware of new threats, and how to identify the malware, and other valuable information. These documents may be in any format, and may contain a variety of detail labels. Little work has been done on this data, mostly consisting of performing a variety of NLP tasks on the reports themselves (Lim et al., 2017; Pingle et al., 2019).

In reality, the reports represent one mode of a multi-model data tuple, the textual report of behavior and unique identifies, and the unstructured binary executable(s) that are the subject of the report. A regular occurrence is a new malware family being discovered, and these reports may serve as a source of additional information that could be used to detect novel malware families with limited labeled examples. Other possibilities include work in generating these reports from the malware itself, following inspiration from the automated statistician work (Nguyen and Raff, 2019; Steinruecken et al., 2019; Hwang et al., 2016; Grosse et al., 2012; Lloyd et al., 2014). This could aid in both developing/adapting models more quickly, as well as disseminating information faster. A variety of possibilities exist at this unique intersection of malware detection and NLP that have yet to be explored.

9 Conclusion

We have given an overview of the machine learning methods currently used for the task of malware classification, as well as the features that are used to drive them. At the same time a number of challenges specific to malware have impeded progress and make objective evaluations difficult. These challenges permeate all stages of the machine learning process and touch upon a vast range of computer science sub-domains, requiring many solutions at different levels of the problem. While we have touched upon some avenues of research that have not yet been addressed by the current literature, many of the challenges remaining will require significant engineering effort and community cooperation and action to overcome. This is especially true of the data collection and processing stages, which are vital to all other stages of the machine learning process, due to the number of unique challenges and costs associated with processing binary data.

References

S. Musil, "Home Depot offers $19M to settle customers' hacking lawsuit," 3 2016. [Online]. Available: https://www.cnet.com/news/home-depot-offers-19m-to-settle-customers-hacking-lawsuit/

C. Riley and J. Pagliery, "Target will pay hack victims $10 million," 3 2015. [Online]. Available: http://money.cnn.com/2015/03/19/technology/security/target-data-hack-settlement/

S. Frizell, "Sony Is Spending $15 Million to Deal With the Big Hack," 2 2015. [Online]. Available: http://time.com/3695118/sony-hack-the-interview-costs/

J. F. Gantz, R. Lee, A. Florean, V. Lim, B. Sikdar, L. Madhaven, S. K. S. Lakshmi, and M. Nagappan, "The Link between Pirated Software and Cybersecurity Breaches How Malware in Pirated Software Is Costing the World Billions," IDC, Tech. Rep., 2014.

E. C. Spafford, "Is Anti-virus Really Dead?" *Computers & Security*, vol. 44, p. iv, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404814000820

AV-TEST, "Malware Statistics," 2016. [Online]. Available: https://www.av-test.org/en/statistics/malware/

F-Secure, "DEEPGUARD Proactive on-host protection against new and emerging threats," F-Secure, Tech. Rep., 2016. [Online]. Available: https://www.f-secure.com/documents/996508/1030745/deepguard_whitepaper.pdf

D. Yadron, "Symantec Develops New Attack on Cyberhacking," 5 2014. [Online]. Available: http://www.wsj.com/articles/SB10001424052702303417104579542140235850578

M. Hypponen, "Why Antivirus Companies Like Mine Failed to Catch Flame and Stuxnet," 6 2012. [Online]. Available: https://www.wired.com/2012/06/internet-security-fail/

P. Porras, "Inside Risks: Reflections on Conficker," *Commun. ACM*, vol. 52, no. 10, pp. 23–24, 10 2009. [Online]. Available: http://doi.acm.org/10.1145/1562764.1562777

J. O. Kephart, G. B. Sorkin, W. C. Arnold, D. M. Chess, G. J. Tesauro, and S. R. White, "Biologically Inspired Defenses Against Computer Viruses," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 985–996. [Online]. Available: http://dl.acm.org/citation.cfm?id=1625855.1625983

S. M. Tabish, M. Z. Shafiq, and M. Farooq, "Malware detection using statistical analysis of byte-level file content," in *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics - CSI-KDD '09*, ACM. New York, New York, USA: ACM Press, 2009, pp. 23–31. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1599272.1599278

M. Schultz, E. Eskin, F. Zadok, and S. Stolfo, "Data Mining Methods for Detection of New Malicious Executables," in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. IEEE Comput. Soc, 2001, pp. 38–49. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=924286

B. Li, K. Roundy, C. Gates, and Y. Vorobeychik, "Large-Scale Identification of Malicious Singleton Files," in *7TH ACM Conference on Data and Application Security and Privacy*, 2017.

D. Kushner, "The real story of stuxnet," *IEEE Spectrum*, vol. 50, no. 3, pp. 48–53, 3 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6471059/

P. Domingos, "A Few Useful Things to Know About Machine Learning," *Commun. ACM*, vol. 55, no. 10, pp. 78–87, 10 2012. [Online]. Available: http://doi.acm.org/10.1145/2347736.2347755

A. Halevy, P. Norvig, and F. Pereira, "The unreasonable effectiveness of data," *Intelligent Systems, IEEE*, vol. 24, no. 2, pp. 8–12, 2009.

R. S. Geiger, K. Yu, Y. Yang, M. Dai, J. Qiu, R. Tang, and J. Huang, "Garbage in, Garbage out? Do Machine Learning Application Papers in Social Computing Report Where Human-Labeled Training Data Comes From?" in *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, ser. FAT* '20.   New York, NY, USA: Association for Computing Machinery, 2020, p. 325–336. [Online]. Available: https://doi.org/10.1145/3351095.3372862

J.-M. Roberts, "Virus Share," 2011. [Online]. Available: https://virusshare.com/

D. Quist, "Open Malware," 2009. [Online]. Available: http://openmalware.org/

P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling, "The Nepenthes Platform: An Efficient Approach to Collect Malware," in *Recent Advances in Intrusion Detection: 9th International Symposium, RAID 2006 Hamburg, Germany, September 20-22, 2006 Proceedings*, D. Zamboni and C. Kruegel, Eds.   Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 165–184. [Online]. Available: http://dx.doi.org/10.1007/11856214_9

J. Zhuge, T. Holz, X. Han, C. Song, and W. Zou, "Collecting Autonomous Spreading Malware Using High-Interaction Honeypots," in *Information and Communications Security: 9th International Conference, ICICS 2007, Zhengzhou, China, December 12-15, 2007. Proceedings*, S. Qing, H. Imai, and G. Wang, Eds.   Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 438–451. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-77048-0_34http://link.springer.com/10.1007/978-3-540-77048-0_34

N. Krawetz, "Anti-honeypot technology," *IEEE Security & Privacy Magazine*, vol. 2, no. 1, pp. 76–79, 1 2004. [Online]. Available: http://ieeexplore.ieee.org/document/1264861/

J. Seymour, "How to build a malware classifier [that doesn't suck on real-world data]," in *SecTor*, Toronto, Ontario, 2016. [Online]. Available: https://sector.ca/sessions/how-to-build-a-malware-classifier-that-doesnt-suck-on-real-world-data/

E. Raff, R. Zak, R. Cox, J. Sylvester, P. Yacci, R. Ward, A. Tracy, M. McLean, and C. Nicholas, "An investigation of byte n-gram features for malware classification," *Journal of Computer Virology and Hacking Techniques*, 9 2016. [Online]. Available: http://link.springer.com/10.1007/s11416-016-0283-1

E. Raff, J. Sylvester, and C. Nicholas, "Learning the PE Header, Malware Detection with Minimal Domain Knowledge," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '17.   New York, NY, USA: ACM, 2017, pp. 121–132. [Online]. Available: http://doi.acm.org/10.1145/3128572.3140442

H. S. Anderson and P. Roth, "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models," *ArXiv e-prints*, 2018. [Online]. Available: http://arxiv.org/abs/1804.04637

A. Mohaisen and O. Alrawi, "Unveiling Zeus: Automated Classification of Malware Samples," in *Proceedings of the 22Nd International Conference on World Wide Web*, ser. WWW '13 Companion.   New York, NY, USA: ACM, 2013, pp. 829–832. [Online]. Available: http://doi.acm.org/10.1145/2487788.2488056

D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An Observational Investigation of Reverse Engineers' Process and Mental Models," in *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3290607.3313040

J. Upchurch and X. Zhou, "Variant: a malware similarity testing framework," in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*.   IEEE, 10 2015, pp. 31–39. [Online]. Available: http://ieeexplore.ieee.org/document/7413682/

K. Berlin, D. Slater, and J. Saxe, "Malicious Behavior Detection Using Windows Audit Logs," in *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '15.   New York, NY, USA: ACM, 2015, pp. 35–44. [Online]. Available: http://doi.acm.org/10.1145/2808769.2808773

J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 10 2015, pp. 11–20. [Online]. Available: http://ieeexplore.ieee.org/document/7413680/

I. Incer, M. Theodorides, S. Afroz, and D. Wagner, "Adversarially Robust Malware Detection Using Monotonic Classification," in *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*, ser. IWSPA '18. New York, NY, USA: ACM, 2018, pp. 54–63. [Online]. Available: http://doi.acm.org/10.1145/3180445.3180449

B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep Learning for Classification of Malware System Call Sequences," in *The 29th Australasian Joint Conference on Artificial Intelligence*, 2016.

S. Zhu, J. Shi, L. Yang, B. Qin, Z. Zhang, L. Song, and G. Wang, "Measuring and Modeling the Label Dynamics of Online Anti-Malware Engines," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: {USENIX} Association, 8 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/zhu

M. Botacin, F. Ceschin, P. de Geus, and A. Grégio, "We need to talk about antiviruses: challenges & pitfalls of av evaluations," *Computers & Security*, vol. 95, p. 101859, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404820301310

M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated Classification and Analysis of Internet Malware," *Analysis*, vol. 4637, no. 1, pp. 178–197, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1776449

A. Kantchelian, M. C. Tschantz, S. Afroz, B. Miller, V. Shankar, R. Bachwani, A. D. Joseph, and J. D. Tygar, "Better Malware Ground Truth: Techniques for Weighting Anti-Virus Vendor Labels," in *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '15. New York, NY, USA: ACM, 2015, pp. 45–56. [Online]. Available: http://doi.acm.org/10.1145/2808769.2808780

M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVclass: A Tool for Massive Malware Labeling," in *Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016*, F. Monrose, M. Dacier, G. Blanc, and J. Garcia-Alfaro, Eds. Paris, France: Springer International Publishing, 2016, pp. 230–253. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-45719-2_11

A. Calleja, J. Tapiador, and J. Caballero, "The MalSource Dataset: Quantifying Complexity and Code Reuse in Malware Development," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 12, pp. 3175–3190, 12 2019. [Online]. Available: http://arxiv.org/abs/1811.06888https://ieeexplore.ieee.org/document/8568018/

M. M. Masud, T. M. Al-Khateeb, K. W. Hamlen, J. Gao, L. Khan, J. Han, and B. Thuraisingham, "Cloud-based malware detection for evolving data streams," *ACM Transactions on Management Information Systems*, vol. 2, no. 3, pp. 1–27, 10 2011. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2019618.2019622

M. A. Rajab, L. Ballard, N. Jagpal, P. Mavrommatis, D. Nojiri, N. Provos, and L. Schmidt, "Trends in CircumventingWeb-Malware Detection," Google, Tech. Rep. July, 2011. [Online]. Available: https://security.googleblog.com/2011/08/four-years-of-web-malware.html

A. Kantchelian, S. Afroz, L. Huang, A. C. Islam, B. Miller, M. C. Tschantz, R. Greenstadt, A. D. Joseph, and J. D. Tygar, "Approaches to Adversarial Drift," in *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, ser. AISec '13. New York, NY, USA: ACM, 2013, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/2517312.2517320

A. Singh, A. Walenstein, and A. Lakhotia, "Tracking Concept Drift in Malware Families," in *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, ser. AISec '12. New York, NY, USA: ACM, 2012, pp. 81–92. [Online]. Available: http://doi.acm.org/10.1145/2381896.2381910

C. Willems, T. Holz, and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox," *IEEE Security and Privacy*, vol. 5, no. 2, pp. 32–39, 3 2007. [Online]. Available: http://dx.doi.org/10.1109/MSP.2007.45

A. A. E. Elhadi, M. A. Maarof, B. I. Barry, and H. Hamza, "Enhancing the detection of metamorphic malware using call graphs," *Computers & Security*, vol. 46, pp. 62–78, 10 2014. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0167404814001060

M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 45–60. [Online]. Available: http://ieeexplore.ieee.org/document/5504788/

K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and Classification of Malware Behavior," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 108–125. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70542-0_6

M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals, Part 2: Covering Windows Server 2008 R2 and Windows 7 (Windows Internals)*. Redmond, WA, USA: Microsoft Press, 2012.

——, *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7*, 6th ed. Redmond, WA, USA: Microsoft Press, 2012.

M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, 1st ed. San Francisco, CA, USA: No Starch Press, 2012.

M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated Classification and Analysis of Internet Malware," in *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 178–197. [Online]. Available: http://dl.acm.org/citation.cfm?id=1776434.1776449

D. Kirat, G. Vigna, and C. Kruegel, "BareCloud: Bare-metal Analysis-based Evasive Malware Detection," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 287–301. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kirat

J. Dai, R. Guha, and J. Lee, "Efficient Virus Detection Using Dynamic Instruction Sequences," *Journal of Computers*, vol. 4, no. 5, pp. 405–414, 2009.

A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised Anomaly-Based Malware Detection Using Hardware Features," in *Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, A. Stavrou, H. Bos, and G. Portokalidis, Eds. Cham: Springer International Publishing, 2014, pp. 109–129. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11379-1_6

N. Stakhanova, M. Couture, and A. A. Ghorbani, "Exploring Network-based Malware Classification," in *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, ser. MALWARE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 14–20. [Online]. Available: http://dx.doi.org/10.1109/MALWARE.2011.6112321

S. Nari and A. A. Ghorbani, "Automated malware classification based on network behavior," in *2013 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 1 2013, pp. 642–647. [Online]. Available: http://ieeexplore.ieee.org/document/6504162/

S. Wehner, "Analyzing Worms and Network Traffic Using Compression," *Journal of Computer Security*, vol. 15, no. 3, pp. 303–320, 8 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1370628.1370630

R. Perdisci, W. Lee, and N. Feamster, "Behavioral Clustering of HTTP-based Malware and Signature Generation Using Malicious Network Traces," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, p. 26. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855711.1855737

M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A Survey on Automated Dynamic Malware-analysis Techniques and Tools," *ACM Comput. Surv.*, vol. 44, no. 2, pp. 6:1–6:42, 3 2008. [Online]. Available: http://doi.acm.org/10.1145/2089125.2089126

J. Blackthorne, A. Bulazel, A. Fasano, P. Biernat, and B. Yener, "AVLeak: Fingerprinting Antivirus Emulators Through Black-box Testing," in *Proceedings of the 10th USENIX Conference on Offensive Technologies*, ser. WOOT'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 91–105. [Online]. Available: http://dl.acm.org/citation.cfm?id=3027019.3027028

M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song, "Emulating emulation-resistant malware," in *Proceedings of the 1st ACM workshop on Virtual machine security - VMSec '09*. New York, New York, USA: ACM Press, 2009, p. 11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1655151%5Cnhttp://portal.acm.org/citation.cfm?doid=1655148.1655151http://portal.acm.org/citation.cfm?doid=1655148.1655151

G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, 7 1974. [Online]. Available: http://doi.acm.org/10.1145/361011.361073

T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is Not Transparency: VMM Detection Myths and Realities," in *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, ser. HOTOS'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 6:1–6:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1361397.1361403

F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-Force: Force-Executing Binary Programs for Security Applications," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 829–844. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/peng

D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Ying, "Bitscope: Automatically dissecting malicious binaries," Carnegie Mellon University, Pittsburgh, PA, Tech. Rep., 2007. [Online]. Available: http://bitblaze.cs.berkeley.edu/papers/bitscope_tr_2007.pdf

C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. v. Steen, "Prudent Practices for Designing Malware Experiments: Status Quo and Outlook," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 5 2012, pp. 65–79. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6234405

C. Kreibich, N. Weaver, C. Kanich, W. Cui, and V. Paxson, "GQ: Practical Containment for Measuring Modern Malware Systems," in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, ser. IMC '11. New York, NY, USA: ACM, 2011, pp. 397–412. [Online]. Available: http://doi.acm.org/10.1145/2068816.2068854

M. Z. Rafique and J. Caballero, "FIRMA: Malware Clustering and Network Signature Generation with Mixed Network Behaviors," in *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 8145*, ser. RAID 2013. New York, NY, USA: Springer-Verlag New York, Inc., 2013, pp. 144–163. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41284-4_8

M. Graziano, C. Leita, and D. Balzarotti, "Towards Network Containment in Malware Analysis Systems," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: ACM, 2012, pp. 339–348. [Online]. Available: http://doi.acm.org/10.1145/2420950.2421000

G. Severi, T. Leek, and B. Dolan-Gavitt, "Malrec: Compact Full-Trace Malware Recording for Retrospective Deep Analysis," in *DIMVA 2018, LNCS 10885*, C. Giuffrida, S. Bardin, and G. Blanc, Eds. Cham: Springer International Publishing, 2018, pp. 3–23.

J. Z. Kolter and M. A. Maloof, "Learning to Detect and Classify Malicious Executables in the Wild," *Journal of Machine Learning Research*, vol. 7, pp. 2721–2744, 12 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1248547.1248646

S. J. Stolfo, K. Wang, and W.-J. Li, "Towards Stealthy Malware Detection," in *Malware Detection*, M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, Eds. Boston, MA: Springer US, 2007, pp. 231–249. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-44599-1_11

R. Islam, R. Tian, L. Batten, and S. Versteeg, "Classification of Malware Based on String and Function Feature Selection," in *Cybercrime and Trustworthy Computing Workshop (CTC), 2010 Second*, 7 2010, pp. 9–17.

K. S. Han, J. H. Lim, B. Kang, and E. G. Im, "Malware analysis using visualized images and entropy graphs," *International Journal of Information Security*, vol. 14, no. 1, pp. 1–14, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10207-014-0242-0

D. Baysa, R. M. Low, and M. Stamp, "Structural entropy and metamorphic malware," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 4, pp. 179–192, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11416-013-0185-4

I. Sorokin, "Comparing files using structural entropy," *Journal in Computer Virology*, vol. 7, no. 4, pp. 259–265, 2011. [Online]. Available: http://dx.doi.org/10.1007/s11416-011-0153-9

"Microsoft Portable Executable and Common Object File Format Specification Version 8.3," Microsoft, Tech. Rep., 2013. [Online]. Available: https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx

M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq, "A Framework for Efficient Mining of Structural Information to Detect Zero-Day Malicious Portable Executables," 1Next Generation Intelligent Networks Research Center (nexGIN RC), Tech. Rep., 2009.

I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: Opcode-Sequence-Based Malware Detection," in *Engineering Secure Software and Systems: Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings*, F. Massacci, D. Wallach, and N. Zannone, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 35–43. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11747-3_3

R. Moskovitch, C. Feher, N. Tzachar, E. Berger, M. Gitelman, S. Dolev, and Y. Elovici, "Unknown Malcode Detection Using OPCODE Representation," in *Proceedings of the 1st European Conference on Intelligence and Security Informatics*, ser. EuroISI '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 204–215. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89900-6_21

A. Damodaran, F. D. Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *Journal of Computer Virology and Hacking Techniques*, pp. 1–12, 2015. [Online]. Available: http://dx.doi.org/10.1007/s11416-015-0261-z

K. Hahn, "Robust Static Analysis of Portable Executable Malware," Ph.D. dissertation, HTWK Leipzig, 2014.

O. Ferrand and E. Filiol, "Combinatorial detection of malware by IAT discrimination," *Journal of Computer Virology and Hacking Techniques*, vol. 12, no. 3, pp. 131–136, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11416-015-0257-8

T. Dube, R. Raines, G. Peterson, K. Bauer, M. Grimaila, and S. Rogers, "Malware target recognition via static heuristics," *Computers & Security*, vol. 31, no. 1, pp. 137–147, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404811001131

Y. Wei, Z. Zheng, and N. Ansari, "Revealing packed malware," *IEEE Security and Privacy*, vol. 6, no. 5, pp. 65–69, 2008.

F. Guo, P. Ferrie, and T.-C. Chiueh, "A Study of the Packer Problem and Its Solutions," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, ser. RAID '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 98–115. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87403-4_6

A. Singh and A. Lakhotia, "Game-theoretic design of an information exchange model for detecting packed malware," in *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, 10 2011, pp. 1–7.

L. Martignoni, M. Christodorescu, and S. Jha, "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 12 2007, pp. 431–441. [Online]. Available: http://ieeexplore.ieee.org/document/4413009/

P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 12 2006, pp. 289–300. [Online]. Available: http://ieeexplore.ieee.org/document/4041175/

K. Coogan, S. Debray, T. Kaochar, and G. Townsend, "Automatic Static Unpacking of Malware Binaries," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, ser. WCRE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 167–176. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2009.24

H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel, "When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features," in *Proceedings 2020 Network and Distributed System Security Symposium*. Reston, VA: Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2020/02/24310.pdf

K. A. Roundy and B. P. Miller, "Binary-code Obfuscations in Prevalent Packer Tools," *ACM Comput. Surv.*, vol. 46, no. 1, pp. 4:1–4:32, 7 2013. [Online]. Available: http://doi.acm.org/10.1145/2522968.2522972

M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith, "Malware Normalization," Madison, WI, Tech. Rep., 2005.

M. Christodorescu, S. Jha, J. Kinder, S. Katzenbeisser, and H. Veith, "Software transformations to improve malware detection," *Journal in Computer Virology*, vol. 3, no. 4, pp. 253–265, 2007. [Online]. Available: http://dx.doi.org/10.1007/s11416-007-0059-8

J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically Generating Signatures for Polymorphic Worms," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, ser. SP '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 226–241. [Online]. Available: http://dx.doi.org/10.1109/SP.2005.15

E. Konstantinou, "Metamorphic Virus: Analysis and Detection," Royal Holloway University of London, Tech. Rep., 2008. [Online]. Available: http://digirep.rhul.ac.uk/items/bde3a9fe-51c0-a19a-e04d-b324c0926a4a/1/

M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic Reverse Engineering of Malware Emulators," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 5 2009, pp. 94–109. [Online]. Available: http://ieeexplore.ieee.org/document/5207639/

B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A Generic Approach to Automatic Deobfuscation of Executable Code," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 5 2015, pp. 674–691. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7163054

S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting Software Through Obfuscation: Can It Keep Pace with Progress in Code Analysis?" *ACM Comput. Surv.*, vol. 49, no. 1, pp. 4:1–4:37, 4 2016. [Online]. Available: http://doi.acm.org/10.1145/2886012

D. H. Chau, C. Nachenberg, J. Wilhelm, A. Wright, and C. Faloutsos, "Polonium: Tera-Scale Graph Mining and Inference for Malware Detection," in *Proceedings of the 2011 SIAM International Conference on Data Mining*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 4 2011, pp. 131–142. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9781611972818.12

A. Tamersoy, K. Roundy, and D. H. Chau, "Guilt by Association: Large Scale Malware Detection by Mining File-relation Graphs," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14. New York, NY, USA: ACM, 2014, pp. 1524–1533. [Online]. Available: http://doi.acm.org/10.1145/2623330.2623342

Y. Ye, T. Li, S. Zhu, W. Zhuang, E. Tas, U. Gupta, and M. Abdulhayoglu, "Combining File Content and File Relations for Cloud Based Malware Detection," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '11. New York, NY, USA: ACM, 2011, pp. 222–230. [Online]. Available: http://doi.acm.org/10.1145/2020408.2020448

B. J. Kwon, J. Mondal, J. Jang, L. Bilge, and T. Dumitraş, "The Dropper Effect: Insights into Malware Distribution with Downloader Graph Analytics," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1118–1129. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813724

N. Karampatziakis, J. W. Stokes, A. Thomas, and M. Marinescu, "Using File Relationships in Malware Classification," in *Proceedings of Conference on Detection of Intrusions and Malware & Vulnerability Assessment*. Springer, 7 2012. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=193769

A. T. Nguyen, E. Raff, and A. Sant-Miller, "Would a File by Any Other Name Seem as Malicious?" in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 12 2019, pp. 1322–1331. [Online]. Available: https://ieeexplore.ieee.org/document/9006132/

A. Kyadige and E. M. Rudd, "Learning from Context : Exploiting and Interpreting File Path Information for Better Malware Detection," *ArXiv e-prints*, 2019.

T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, "N-gram-based detection of new malicious code," in *Proceedings of the 28th Annual International Computer Software and Applications*

Conference, 2004. COMPSAC 2004., vol. 2. IEEE, 2004, pp. 41–42. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1342667

G. Dahl, J. Stokes, L. Deng, and D. Yu, "Large-Scale Malware Classification Using Random Projections and Neural Networks," in *Proceedings IEEE Conference on Acoustics, Speech, and Signal Processing*. IEEE SPS, 5 2013. [Online]. Available: https://www.microsoft.com/en-us/research/publication/large-scale-malware-classification-using-random-projections-and-neural-networks/

S. E. Robertson and S. Walker, "Some Simple Effective Approximations to the 2-Poisson Model for Probabilistic Weighted Retrieval," in *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '94. New York, NY, USA: Springer-Verlag New York, Inc., 1994, pp. 232–241. [Online]. Available: http://dl.acm.org/citation.cfm?id=188490.188561

A. H. Ibrahim, M. B. Abdelhalim, H. Hussein, and A. Fahmy, "Analysis of x86 instruction set usage for Windows 7 applications," in *Computer Technology and Development (ICCTD), 2010 2nd International Conference on*, 11 2010, pp. 511–516.

E. Raff and C. Nicholas, "Hash-Grams: Faster N-Gram Features for Classification and Malware Detection," in *Proceedings of the ACM Symposium on Document Engineering 2018*. Halifax, NS, Canada: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3209280.3229085

H. P. Luhn, "The Automatic Creation of Literature Abstracts," *IBM Journal of Research and Development*, vol. 2, no. 2, pp. 159–165, 4 1958. [Online]. Available: http://dx.doi.org/10.1147/rd.22.0159

E. Raff, W. Fleming, R. Zak, H. Anderson, B. Finlayson, C. K. Nicholas, and M. Mclean, "KiloGrams: Very Large N-Grams for Malware Classification," in *Proceedings of KDD 2019 Workshop on Learning and Mining for Cybersecurity (LEMINCS'19)*, 2019. [Online]. Available: https://arxiv.org/abs/1908.00200

S. Dolev and N. Tzachar, "Malware signature builder and detection for executable code," p. 14, 2008.

A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, "Detecting unknown malicious code by applying classification techniques on OpCode patterns," *Security Informatics*, vol. 1, no. 1, pp. 1–22, 2012.

M. M. Masud, L. Khan, and B. Thuraisingham, "A scalable multi-level feature extraction technique to detect malicious executables," *Information Systems Frontiers*, vol. 10, no. 1, pp. 33–45, 3 2008. [Online]. Available: http://link.springer.com/10.1007/s10796-007-9054-3

R. Zak, E. Raff, and C. Nicholas, "What can N-grams learn for malware detection?" in *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 10 2017, pp. 109–118. [Online]. Available: http://ieeexplore.ieee.org/document/8323963/

M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *Journal in Computer Virology*, vol. 1, no. 1, pp. 13–23, 2005. [Online]. Available: http://dx.doi.org/10.1007/s11416-005-0002-9

A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia, "Exploiting similarity between variants to defeat malware," in *Proc. BlackHat DC Conf*, 2007.

Y.-W. Chang, C.-J. Hsieh, K.-W. Chang, M. Ringgaard, and C.-J. Lin, "Training and Testing Low-degree Polynomial Data Mappings via Linear SVM," *The Journal of Machine Learning Research*, vol. 11, pp. 1471–1490, 2010. [Online]. Available: http://jmlr.org/papers/volume11/chang10a/chang10a.pdf

G.-X. Yuan, C.-H. Ho, and C.-J. Lin, "Recent Advances of Large-Scale Linear Classification," *Proceedings of the {IEEE}*, vol. 100, no. 9, pp. 2584–2603, 2012. [Online]. Available: http://dx.doi.org/10.1109/JPROC.2012.2188013

C. Cortes and V. Vapnik, "SUPPORT-VECTOR NETWORKS," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995. [Online]. Available: http://link.springer.com/article/10.1007/BF00994018

R. Tibshirani, "Regression Shrinkage and Selection Via the Lasso," *Journal of the Royal Statistical Society, Series B*, vol. 58, no. 1, pp. 267–288, 1994.

A. Y. Ng, "Feature selection, L1 vs. L2 regularization, and rotational invariance," *Twenty-first international conference on Machine learning - ICML '04*, p. 78, 2004. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1015330.1015435

H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the Royal Statistical Society, Series B*, vol. 67, no. 2, pp. 301–320, 4 2005. [Online]. Available: http://doi.wiley.com/10.1111/j.1467-9868.2005.00503.xhttp://onlinelibrary.wiley.com/doi/10.1111/j.1467-9868.2005.00503.x/full

R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A Library for Large Linear Classification," *The Journal of Machine Learning Research*, vol. 9, pp. 1871–1874, 2008.

J. Langford, L. Li, and A. Strehl, "Vowpal wabbit open source project," Yahoo!, Tech. Rep., 2007. [Online]. Available: https://github.com/JohnLangford/vowpal_wabbit

C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, 4 2011. [Online]. Available: http://dl.acm.org/citation.cfm?doid=1961189.1961199

Y. Engel, S. Mannor, and R. Meir, "The Kernel Recursive Least-Squares Algorithm," *IEEE Transactions on Signal Processing*, vol. 52, no. 8, pp. 2275–2285, 8 2004. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1315946

C.-J. Hsieh, S. Si, and I. Dhillon, "A Divide-and-Conquer Solver for Kernel Support Vector Machines," *JMLR W&CP*, vol. 32, no. 1, p. 566–574, 2014.

K. Grauman and T. Darrell, "The pyramid match kernel: discriminative classification with sets of image features," in *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, no. October. IEEE, 2005, pp. 1458–1465. [Online]. Available: http://ieeexplore.ieee.org/document/1544890/

L. Bo and C. Sminchisescu, "Efficient Match Kernel between Sets of Features for Visual Recognition," in *Advances in Neural Information Processing Systems 22*, Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, Eds. Curran Associates, Inc., 2009, pp. 135–143. [Online]. Available: http://papers.nips.cc/paper/3874-efficient-match-kernel-between-sets-of-features-for-visual-recognition.pdf

H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins, "Text Classification using String Kernels," *Journal of Machine Learning Research*, vol. 2, pp. 419–444, 2002.

C. Leslie, E. Eskin, and W. S. Noble, "The spectrum kernel: a string kernel for SVM protein classification." *Pacific Symposium on Biocomputing*, vol. 575, pp. 564–575, 2002.

S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, "Graph Kernels," *Journal of Machine Learning Research*, vol. 11, pp. 1201–1242, 8 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1859891

M. Neuhaus and H. Bunke, *Bridging the Gap Between Graph Edit Distance and Kernel Machines*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 2007.

B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based malware detection using dynamic analysis," *Journal in Computer Virology*, vol. 7, no. 4, pp. 247–258, 2011. [Online]. Available: http://dx.doi.org/10.1007/s11416-011-0152-x

J. R. Quinlan, *C4.5: Programs for Machine Learning*, ser. Morgan Kaufmann series in {M}achine {L}earning, M. Kaufmann, Ed. Morgan Kaufmann, 1993, vol. 1, no. 3. [Online]. Available: http://portal.acm.org/citation.cfm?id=152181

L. Breiman, J. Friedman, C. J. Stone, and R. Olshen, *Classification and Regression Trees*. CRC press, 1984.

X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg, "Top 10 algorithms in data mining," *Knowledge and Information Systems*, vol. 14, no. 1, pp. 1–37, 2007. [Online]. Available: http://www.springerlink.com/index/10.1007/s10115-007-0114-2

R. Perdisci, A. Lanzi, and W. Lee, "McBoost: Boosting Scalability in Malware Collection and Analysis Using Statistical Classification of Executables," in *2008 Annual Computer Security Applications Conference (ACSAC)*. IEEE, 12 2008, pp. 301–310. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4721567

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python,"

Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011. [Online]. Available: http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html

M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update Mark," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 11 2009.

M. S. Gashler, "Waffles: A Machine Learning Toolkit," *Journal of Machine Learning Research*, vol. MLOSS 12, pp. 2383–2387, 7 2011. [Online]. Available: http://www.jmlr.org/papers/volume12/gashler11a/gashler11a.pdf

X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "MLlib: Machine Learning in Apache Spark," *Journal of Machine Learning Research*, vol. 17, no. 34, pp. 1–7, 2016. [Online]. Available: http://jmlr.org/papers/v17/15-237.html

E. Raff, "JSAT: Java Statistical Analysis Tool, a Library for Machine Learning," *Journal of Machine Learning Research*, vol. 18, no. 23, pp. 1–5, 2017. [Online]. Available: http://jmlr.org/papers/v18/16-131.html

A. Bifet, G. Holmes, B. Pfahringer, and E. Frank, "Fast perceptron decision tree learning from evolving data streams," in *Pacific-Asia conference on Advances in Knowledge Discovery and Data Mining - Volume Part II*, 2010, pp. 299–310. [Online]. Available: http://www.springerlink.com/index/212Q90037V867278.pdf

T. Chen and C. Guestrin, "XGBoost: Reliable Large-scale Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

M. N. Wright and A. Ziegler, "ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R," *arXiv preprint*, 8 2015. [Online]. Available: http://arxiv.org/abs/1508.04409

G. Ke, Q. Meng, T. Wang, W. Chen, W. Ma, and T.-Y. Liu, "A Highly Efficient Gradient Boosting Decision Tree," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 3148–3156. [Online]. Available: http://papers.nips.cc/paper/6907-a-highly-efficient-gradient-boosting-decision-tree.pdf

L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, 3 2006. [Online]. Available: http://www.springerlink.com/index/10.1007/s10994-006-6226-1http://www.springerlink.com/index/Y7882Q0143425083.pdf

L. Breiman, "Manual on setting up, using, and understanding random forests v4.0," *Statistics Department University of California Berkeley, CA, USA*, 2003.

G. Louppe, L. Wehenkel, A. Sutera, and P. Geurts, "Understanding variable importances in forests of randomized trees," in *Advances in Neural Information Processing Systems 26*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds., 2013, pp. 431–439. [Online]. Available: http://media.nips.cc/nipsbooks/nipspapers/paper_files/nips26/281.pdf

E. Menahem, A. Shabtai, L. Rokach, and Y. Elovici, "Improving Malware Detection by Applying Multi-inducer Ensemble," *Comput. Stat. Data Anal.*, vol. 53, no. 4, pp. 1483–1494, 2 2009. [Online]. Available: http://dx.doi.org/10.1016/j.csda.2008.10.015

Y. Elovici, A. Shabtai, R. Moskovitch, G. Tahan, and C. Glezer, "Applying Machine Learning Techniques for Detection of Malicious Code in Network Traffic," in *Proceedings of the 30th Annual German Conference on Advances in Artificial Intelligence*, ser. KI '07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 44–50. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74565-5_5

N. Alshahwan, E. T. Barr, D. Clark, and G. Danezis, "Detecting Malware with Information Complexity," *arXiv*, 2 2015. [Online]. Available: http://arxiv.org/abs/1502.07661

R. Moskovitch, D. Stopel, C. Feher, N. Nissim, N. Japkowicz, and Y. Elovici, "Unknown malcode detection and the imbalance problem," *Journal in Computer Virology*, vol. 5, no. 4, pp. 295–308, 11 2009. [Online]. Available: http://link.springer.com/10.1007/s11416-009-0122-8

A. Vezhnevets and V. Vezhnevets, "'Modest AdaBoost' – Teaching AdaBoost to Generalize Better," in *GraphiCon*, Novosibirsk Akademgorodok, Russia, 2005. [Online]. Available: http://www.inf.ethz.ch/personal/vezhneva/Pubs/ModestAdaBoost.pdfhttp://graphics.cs.msu.ru/en/publications/text/gc2005vv.pdf

G. Hulten, L. Spencer, and P. Domingos, "Mining time-changing data streams," in *seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2001, pp. 97–106. [Online]. Available: http://dl.acm.org/citation.cfm?id=502529

I. Goodfellow, Y. Bengio, and A. Courville, "Deep Learning," 2016. [Online]. Available: http://www.deeplearningbook.org

D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 10 1986. [Online]. Available: http://www.nature.com/doifinder/10.1038/323533a0

S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a Next-Generation Open Source Framework for Deep Learning," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015. [Online]. Available: http://learningsys.org/papers/LearningSys_2015_paper_33.pdf

F. Chollet, "Keras," 2015. [Online]. Available: https://github.com/fchollet/keras

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *arXiv:1603.04467v2*, p. 19, 3 2016. [Online]. Available: http://download.tensorflow.org/paper/whitepaper2015.pdfhttp://arxiv.org/abs/1603.04467

M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, Q. V. Le, and A. Y. Ng, "Building high-level features using large scale unsupervised learning," in *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, J. Langford and J. Pineau, Eds. New York, NY, USA: ACM, 2012, pp. 81–88. [Online]. Available: http://icml.cc/2012/papers/73.pdf

I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, 2014, pp. 2672–2680.

A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 369–376.

A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

I. Firdausi, C. Lim, A. Erwin, and A. S. Nugroho, "Analysis of Machine learning Techniques Used in Behavior-Based Malware Detection," in *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on*, 12 2010, pp. 201–203.

C. Liangboonprakong and O. Sornil, "Classification of malware families based on N-grams sequential pattern features," in *8th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, 6 2013, pp. 777–782.

W. Hardy, L. Chen, S. Hou, Y. Ye, and X. Li, "DL4MD : A Deep Learning Framework for Intelligent Malware Detection," in *International Conference on Data Mining (DMIN)*, 2016, pp. 61–68.

W. Huang and J. W. Stokes, "MtNet: A Multi-Task Neural Network for Dynamic Malware Classification," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 399–418. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-40667-1_20

R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, "Malware classification with recurrent networks," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 4 2015, pp. 1916–1920. [Online]. Available: http://ieeexplore.ieee.org/document/7178304/

A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "WaveNet: A Generative Model for Raw Audio," 2016. [Online]. Available: http://arxiv.org/abs/1609.03499

M. Z. Shafiq, S. A. Khayam, and M. Farooq, "Embedded Malware Detection Using Markov n-Grams," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 88–107. [Online]. Available: http://link.springer.com/10.1007/978-3-540-70542-0_5

W. Wong and M. Stamp, "Hunting for metamorphic engines," *Journal in Computer Virology*, vol. 2, no. 3, pp. 211–229, 2006. [Online]. Available: http://dx.doi.org/10.1007/s11416-006-0028-7

L. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989. [Online]. Available: http://ieeexplore.ieee.org/document/18626/

Z. Ghahramani, "An introduction to hidden Markov models and Bayesian networks," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 15, no. 1, pp. 9–42, 2001.

A. Y. Ng and M. I. Jordan, "On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes," in *Advances in Neural Information Processing Systems 14*, T. G. Dietterich, S. Becker, and Z. Ghahramani, Eds. MIT Press, 2002, pp. 841–848. [Online]. Available: http://papers.nips.cc/paper/2020-on-discriminative-vs-generative-classifiers-a-comparison-of-logistic-regression-and-naive-bayes.pdf

Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo, "On the infeasibility of modeling polymorphic shellcode," *Machine Learning*, vol. 81, no. 2, pp. 179–205, 2010. [Online]. Available: http://dx.doi.org/10.1007/s10994-009-5143-5

V. Harichandran, F. Breitinger, and I. Baggili, "Bytewise Approximate Matching: The Good, The Bad, and The Unknown," *Journal of Digital Forensics, Security and Law*, vol. 11, no. 2, pp. 59–78, 2016. [Online]. Available: http://commons.erau.edu/jdfsl/vol11/iss2/4/

V. Roussev, "Building a Better Similarity Trap with Statistically Improbable Features," in *Proceedings of the 42Nd Hawaii International Conference on System Sciences*, ser. HICSS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/HICSS.2009.97

——, "Data Fingerprinting with Similarity Digests," in *Advances in Digital Forensics VI: Sixth IFIP WG 11.9 International Conference on Digital Forensics, Hong Kong, China, January 4-6, 2010, Revised Selected Papers*, K.-P. Chow and S. Shenoi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 207–226. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15506-2_15

J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation*, vol. 3, pp. 91–97, 2006.

Y. Li, S. C. Sundaramurthy, A. G. Bardas, X. Ou, D. Caragea, X. Hu, and J. Jang, "Experimental Study of Fuzzy Hashing in Malware Clustering Analysis," in *8th Workshop on Cyber Security Experimentation and Test (CSET 15)*. Washington, D.C.: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/cset15/workshop-program/presentation/li

J. Upchurch and X. Zhou, "Malware provenance: code reuse detection in malicious software at scale," in *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 10 2016, pp. 1–9. [Online]. Available: http://ieeexplore.ieee.org/document/7888735/

M. Li, X. Chen, X. Li, B. Ma, and P. M. Vitanyi, "The Similarity Metric," *IEEE Transactions on Information Theory*, vol. 50, no. 12, pp. 3250–3264, 2004.

M. Cebrián, M. Alfonseca, A. Ortega, and others, "Common pitfalls using the normalized compression distance: What to watch out for in a compressor," *Communications in Information & Systems*, vol. 5, no. 4, pp. 367–384, 2005.

R. S. Borbely, "On normalized compression distance and large malware," *Journal of Computer Virology and Hacking Techniques*, pp. 1–8, 2015.

E. Raff and C. Nicholas, "An Alternative to NCD for Large Sequences, Lempel-Ziv Jaccard Distance," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '17*.   New York, New York, USA: ACM Press, 2017, pp. 1007–1015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3097983.3098111

A. Z. Broder, "On the Resemblance and Containment of Documents," in *Proceedings of the Compression and Complexity of Sequences 1997*, ser. SEQUENCES '97.   Washington, DC, USA: IEEE Computer Society, 1997, pp. 21–29. [Online]. Available: http://dl.acm.org/citation.cfm?id=829502.830043

E. Raff and C. Nicholas, "Malware Classification and Class Imbalance via Stochastic Hashed LZJD," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '17.   New York, NY, USA: ACM, 2017, pp. 111–120. [Online]. Available: http://doi.acm.org/10.1145/3128572.3140446

K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*.   New York, New York, USA: ACM Press, 2009, pp. 1113–1120. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1553374.1553516

P. Li, A. B. Owen, and C.-H. Zhang, "One Permutation Hashing," in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, ser. NIPS'12.   USA: Curran Associates Inc., 2012, pp. 3113–3121. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999325.2999482

E. Raff, J. Aurelio, and C. Nicholas, "PyLZJD: An Easy to Use Tool for Machine Learning," in *Proceedings of the 18th Python in Science Conference*, C. Calloway, D. Lippa, D. Niederhut, and D. Shupe, Eds., 2019, pp. 97–102. [Online]. Available: http://conference.scipy.org/proceedings/scipy2019/pylzjd.html

E. Raff, C. Nicholas, and M. McLean, "A New Burrows Wheeler Transform Markov Distance," in *The Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020, pp. 5444–5453. [Online]. Available: http://arxiv.org/abs/1912.13046

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Comput.*, vol. 1, no. 4, pp. 541–551, 12 1989. [Online]. Available: http://dx.doi.org/10.1162/neco.1989.1.4.541

Y. LeCun and Y. Bengio, "The Handbook of Brain Theory and Neural Networks," M. A. Arbib, Ed.   Cambridge, MA, USA: MIT Press, 1998, ch. Convolutio, pp. 255–258. [Online]. Available: http://dl.acm.org/citation.cfm?id=303568.303704

Z. C. Lipton, J. Berkowitz, and C. Elkan, "A Critical Review of Recurrent Neural Networks for Sequence Learning," 2015.

Y. Bengio, P. Simard, and P. Frasconi, "Learning Long-term Dependencies with Gradient Descent is Difficult," *Trans. Neur. Netw.*, vol. 5, no. 2, pp. 157–166, 3 1994. [Online]. Available: http://dx.doi.org/10.1109/72.279181

H. Jaeger, "The "echo state"approach to analysing and training recurrent neural networks," German National Research Center for Information Technology, Tech. Rep., 2001. [Online]. Available: http://www.faculty.jacobs-university.de/hjaeger/pubs/EchoStatesTechRep.pdf

M. Lukoševičius, "A Practical Guide to Applying Echo State Networks," in *Neural Networks: Tricks of the Trade: Second Edition*, G. Montavon, G. B. Orr, and K.-R. Müller, Eds.   Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 659–686. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35289-8_36

S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 11 1997. [Online]. Available: http://www.mitpressjournals.org/doi/abs/10.1162/neco.1997.9.8.1735

P. J. Werbos, "Generalization of backpropagation with application to a recurrent gas market model," *Neural Networks*, vol. 1, no. 4, pp. 339–356, 1988.

S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi, "Malware Detection with Deep Neural Network Using Process Behavior," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 6 2016, pp. 577–582. [Online]. Available: http://ieeexplore.ieee.org/document/7552276/

E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware Detection by Eating a Whole EXE," in *AAAI Workshop on Artificial Intelligence for Cyber Security*, 10 2018. [Online]. Available: http://arxiv.org/abs/1710.09435

S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of The 32nd International Conference on Machine Learning*, vol. 37, 2015, pp. 448–456.

M. Krčál, O. Švec, M. Bálek, and O. Jašek, "Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only," in *ICLR Workshop*, 2018.

P. Chaovalit, A. Gangopadhyay, G. Karabatis, and Z. Chen, "Discrete Wavelet Transform-based Time Series Analysis and Mining," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 6:1–6:37, 2 2011. [Online]. Available: http://doi.acm.org/10.1145/1883612.1883613

M. Wojnowicz, G. Chisholm, M. Wolff, and V. K. Ave, "Suspiciously Structured Entropy : Wavelet Decomposition of Software Entropy Reveals Symptoms of Malware in the Energy Spectrum," in *Proceedings of the Twenty-Ninth International Florida Artificial Intelligence Research Society Conference*, Z. Markov and I. Russell, Eds. Key Largo, Florida: {AAAI} Press, 2016, pp. 294–298. [Online]. Available: http://www.aaai.org/ocs/index.php/FLAIRS/FLAIRS16/paper/view/12978

J. S. Walker, *A Primer on WAVELETS and Their Scientific Applications on WAVELETS and Their Scientific Applications*, 2nd ed. Eau Claire, Wisconsin, U.S.A: CRC Press, 2008.

Z. R. Struzik and A. Siebes, "The Haar Wavelet Transform in the Time Series Similarity Paradigm," in *Proceedings of the Third European Conference on Principles of Data Mining and Knowledge Discovery*, ser. PKDD '99. London, UK, UK: Springer-Verlag, 1999, pp. 12–22. [Online]. Available: http://dl.acm.org/citation.cfm?id=645803.669368

Y. Pati and P. Krishnaprasad, "Analysis and synthesis of feedforward neural networks using discrete affine wavelet transformations," *IEEE Transactions on Neural Networks*, vol. 4, no. 1, pp. 73–85, 1993. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/18267705http://ieeexplore.ieee.org/document/182697/

V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," in *Soviet physics doklady*, vol. 10, 1966, p. 707.

R. Bellman, "The theory of dynamic programming," *Bulletin of the American Mathematical Society*, vol. 60, no. 6, pp. 503–516, 11 1954. [Online]. Available: http://www.ams.org/journal-getitem?pii=S0002-9904-1954-09848-8

G. Shanmugam, R. M. Low, and M. Stamp, "Simple Substitution Distance and Metamorphic Detection," *J. Comput. Virol.*, vol. 9, no. 3, pp. 159–170, 8 2013. [Online]. Available: http://dx.doi.org/10.1007/s11416-013-0184-5

D. J. Berndt and J. Clifford, "Using Dynamic Time Warping to Find Patterns in Time Series," in *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, ser. AAAIWS'94. AAAI Press, 1994, pp. 359–370. [Online]. Available: http://dl.acm.org/citation.cfm?id=3000850.3000887

S. Naval, V. Laxmi, N. Gupta, M. S. Gaur, and M. Rajarajan, "Exploring Worm Behaviors Using DTW," in *Proceedings of the 7th International Conference on Security of Information and Networks*, ser. SIN '14. New York, NY, USA: ACM, 2014, pp. 379:379–379:384. [Online]. Available: http://doi.acm.org/10.1145/2659651.2659737

C. Kang, N. Park, B. A. Prakash, E. Serra, and V. S. Subrahmanian, "Ensemble Models for Data-driven Prediction of Malware Infections," in *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, ser. WSDM '16. New York, NY, USA: ACM, 2016, pp. 583–592. [Online]. Available: http://doi.acm.org/10.1145/2835776.2835834

M. R. Thakur, D. R. Khilnani, K. Gupta, S. Jain, V. Agarwal, S. Sane, S. Sanyal, and P. S. Dhekne, "Detection and prevention of botnets and malware in an enterprise network," *International Journal of Wireless and Mobile Computing*, vol. 5, no. 2, p. 144, 2012. [Online]. Available: http://www.inderscience.com/link.php?id=46776

E. Keogh and C. A. Ratanamahatana, "Exact indexing of dynamic time warping," *Knowledge and Information Systems*, vol. 7, no. 3, pp. 358–386, 2005. [Online]. Available: http://dx.doi.org/10.1007/s10115-004-0154-9

S. Salvador and P. Chan, "Toward Accurate Dynamic Time Warping in Linear Time and Space," *Intell. Data Anal.*, vol. 11, no. 5, pp. 561–580, 10 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1367985.1367993

F. Petitjean, G. Forestier, G. I. Webb, A. E. Nicholson, Y. Chen, and E. Keogh, "Dynamic Time Warping Averaging of Time Series Allows Faster and More Accurate Classification," in *2014 IEEE International Conference on Data Mining*. IEEE, 12 2014, pp. 470–479. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7023364http://ieeexplore.ieee.org/document/7023364/

C. A. Ratanamahatana and E. Keogh, "Making Time-series Classification More Accurate Using Learned Constraints," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 4 2004, no. April 2004, pp. 11–22. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9781611972740.2

S. Alam, R. Horspool, I. Traore, and I. Sogukpinar, "A framework for metamorphic malware analysis and real-time detection," *Computers & Security*, vol. 48, pp. 212–233, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404814001576

H. Hashemi, A. Azmoodeh, A. Hamzeh, and S. Hashemi, "Graph embedding as a new approach for unknown malware detection," *Journal of Computer Virology and Hacking Techniques*, pp. 1–14, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11416-016-0278-y

M. Eskandari and S. Hashemi, "A graph mining approach for detecting unknown malwares," *Journal of Visual Languages & Computing*, vol. 23, no. 3, pp. 154–162, 2012.

J.-w. Jang, J. Woo, J. Yun, and H. K. Kim, "Mal-netminer: Malware Classification Based on Social Network Analysis of Call Graph," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14 Companion. New York, NY, USA: ACM, 2014, pp. 731–734. [Online]. Available: http://doi.acm.org/10.1145/2567948.2579364

M. Lazarescu, H. Bunke, and S. Venkatesh, "Graph Matching: Fast Candidate Elimination Using Machine Learning Techniques," in *Proceedings of the Joint IAPR International Workshops on Advances in Pattern Recognition*. London, UK, UK: Springer-Verlag, 2000, pp. 236–245. [Online]. Available: http://dl.acm.org/citation.cfm?id=645889.673411

B. Luo, R. C. Wilson, and E. R. Hancock, "Spectral embedding of graphs," *Pattern Recognition*, vol. 36, no. 10, pp. 2213–2230, 10 2003. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0031320303000840

B. Luo and E. R. Hancock, "Structural graph matching using the EM algorithm and singular value decomposition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, no. 10, pp. 1120–1136, 2001.

M. Slawinski and A. Wortman, "Applications of Graph Integration to Function Comparison and Malware Classification," in *2019 4th International Conference on System Reliability and Safety (ICSRS)*. IEEE, 11 2019, pp. 16–24. [Online]. Available: https://ieeexplore.ieee.org/document/8987703/

Haoran Guo, Jianmin Pang, Yichi Zhang, Feng Yue, and Rongcai Zhao, "HERO: A novel malware detection framework based on binary translation," in *2010 IEEE International Conference on Intelligent Computing and Intelligent Systems*. IEEE, 10 2010, pp. 411–415. [Online]. Available: http://ieeexplore.ieee.org/document/5658586/

C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and Efficient Malware Detection at the End Host," in *18th USENIX Security Symposium (USENIX Security 09)*. Montreal, Quebec: {USENIX} Association, 8 2009. [Online]. Available: https://www.usenix.org/conference/usenixsecurity09/technical-sessions/presentation/effective-and-efficient-malware

Y. Park and D. Reeves, "Deriving Common Malware Behavior Through Graph Clustering," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: ACM, 2011, pp. 497–502. [Online]. Available: http://doi.acm.org/10.1145/1966913.1966986

X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale Malware Indexing Using Function-call Graphs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 611–620. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653736

Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel, "Fast Malware Classification by Automated Behavioral Graph Matching," in *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, ser. CSIIRW '10. New York, NY, USA: ACM, 2010, pp. 45:1–45:4. [Online]. Available: http://doi.acm.org/10.1145/1852666.1852716

J. Lee, K. Jeong, and H. Lee, "Detecting Metamorphic Malwares Using Code Graphs," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 1970–1977. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774505

M. Eskandari, Z. Khorshidpour, and S. Hashemi, "HDM-Analyser: a hybrid analysis approach based on data mining techniques for malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 2, pp. 77–93, 2013. [Online]. Available: http://link.springer.com/10.1007/s11416-013-0181-8

D. M. W. Powers, "Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation," *Journal of Machine Learning Technologies*, vol. 2, no. 1, pp. 37–63, 2011.

A. Marx, "A guideline to anti-malware-software testing," in *European Institute for Computer Anti-Virus Research (EICAR)*, 2000, pp. 218–253.

K. H. Brodersen, C. S. Ong, K. E. Stephan, and J. M. Buhmann, "The Balanced Accuracy and Its Posterior Distribution," in *Proceedings of the 2010 20th International Conference on Pattern Recognition*, ser. ICPR '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 3121–3124. [Online]. Available: http://dx.doi.org/10.1109/ICPR.2010.764

A. P. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern Recognition*, vol. 30, no. 7, pp. 1145–1159, 1997.

M. Alazab, S. Venkatraman, P. Watters, and M. Alazab, "Zero-day Malware Detection Based on Supervised Learning Algorithms of API Call Signatures," in *Proceedings of the Ninth Australasian Data Mining Conference - Volume 121*, ser. AusDM '11. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2011, pp. 171–182. [Online]. Available: http://dl.acm.org/citation.cfm?id=2483628.2483648

Y. Zhou and W. M. Inge, "Malware Detection Using Adaptive Data Compression," in *Proceedings of the 1st ACM Workshop on Workshop on AISec*, ser. AISec '08. New York, NY, USA: ACM, 2008, pp. 53–60. [Online]. Available: http://doi.acm.org/10.1145/1456377.1456393

G. Yan, "Be Sensitive to Your Errors: Chaining Neyman-Pearson Criteria for Automated Malware Classification," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. New York, NY, USA: ACM, 2015, pp. 121–132. [Online]. Available: http://doi.acm.org/10.1145/2714576.2714578

AV-TEST, "The best antivirus software for Windows Client Business User," 2016. [Online]. Available: https://www.av-test.org/en/antivirus/business-windows-client/windows-10/april-2016/

V. Roussev and C. Quates, "Content triage with similarity digests: The M57 case study," *Digital Investigation*, vol. 9, pp. S60–S68, 2012.

H. Khan, F. Mirza, and S. A. Khayam, "Determining malicious executable distinguishing attributes and low-complexity detection," *Journal in Computer Virology*, vol. 7, no. 2, pp. 95–105, 2010. [Online]. Available: http://dx.doi.org/10.1007/s11416-010-0140-6

M. Wojnowicz, D. Zhang, G. Chisholm, X. Zhao, and M. Wolff, "Projecting "better than randomly": How to reduce the dimensionality of very large datasets in a way that outperforms random projections," in *IEEE International Conference on Data Science and Advanced Analytics*. IEEE, 2016.

J. Gama, P. Medas, G. Castillo, and P. Rodrigues, "Learning with Drift Detection," in *Advances in Artificial Intelligence – SBIA 2004*, A. Bazzan and S. Labidi, Eds. Springer Berlin Heidelberg, 2004, pp. 286–295. [Online]. Available: http://link.springer.com/chapter/10.1007%2F978-3-540-28645-5_29

M. Baena-Garcia, J. d. Campo-Avila, R. Fidalgo, A. Bifet, R. Gavalda, and R. Morales-Bueno, "Early Drift Detection Method," in *4th ECML PKDD International Workshop on Knowledge Discovery from Data Streams*, Berlin, Germany, 2006, pp. 77–86.

Bo-Heng Chen and Kun-Ta Chuang, "A model-selection framework for concept-drifting data streams," in *2014 International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 10 2014, pp. 290–296. [Online]. Available: http://ieeexplore.ieee.org/document/7058087/

J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A Survey on Concept Drift Adaptation," *ACM Computing Surveys*, vol. 46, no. 4, 2014.

B. Biggio and F. Roli, "Wild patterns: Ten years after the rise of adversarial machine learning," *Pattern Recognition*, vol. 84, pp. 317–331, 12 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0031320318302565

W. Fleshman, E. Raff, R. Zak, M. McLean, and C. Nicholas, "Static Malware Detection & Subterfuge: Quantifying the Robustness of Machine Learning and Current Anti-Virus," in *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 10 2018, pp. 1–10. [Online]. Available: http://arxiv.org/abs/1806.04773https://ieeexplore.ieee.org/document/8659360/

H. S. Anderson, B. Filar, and P. Roth, "Evading Machine Learning Malware Detection," in *Black Hat USA*, 2017. [Online]. Available: https://www.blackhat.com/docs/us-17/thursday/us-17-Anderson-Bot-Vs-Bot-Evading-Machine-Learning-Malware-Detection-wp.pdf

B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, "Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables," in *26th European Signal Processing Conference (EUSIPCO '18)*, 2018. [Online]. Available: https://arxiv.org/pdf/1803.04173.pdf

F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet, "Adversarial Examples on Discrete Sequences for Beating Whole-Binary Malware Detection," *arXiv preprint*, 2018. [Online]. Available: http://arxiv.org/abs/1802.04528

W. Fleshman, E. Raff, J. Sylvester, S. Forsyth, and M. McLean, "Non-Negative Networks Against Adversarial Attacks," *AAAI-2019 Workshop on Artificial Intelligence for Cyber Security*, 2019. [Online]. Available: http://arxiv.org/abs/1806.06108

G. Poulios, C. Ntantogian, and C. Xenakis, "ROPInjector: Using Return- Oriented Programming for Polymorphism and AV Evasion," in *Black Hat USA*, 2015. [Online]. Available: https://github.com/gpoulios/ROPInjector

Thomas Kittel, S. Vogl, J. Kirsch, and C. Eckert, "Counteracting Data-Only Malware with Code Pointer Examination," in *Research in Attacks, Intrusions, and Defenses. Lecture Notes in Computer Science*, vol. 9404, 2015, pp. 177–197.

T. Singh, F. Di Troia, V. A. Corrado, T. H. Austin, and M. Stamp, "Support vector machines and malware detection," *Journal of Computer Virology and Hacking Techniques*, pp. 1–10, 2015. [Online]. Available: http://dx.doi.org/10.1007/s11416-015-0252-0

D. H. Wolpert, "Stacked generalization," *Neural networks*, vol. 5, pp. 241–259, 1992. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608005800231

R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive Mixtures of Local Experts," *Neural Computation*, vol. 3, no. 1, pp. 79–87, 2 1991. [Online]. Available: http://www.mitpressjournals.org/doi/10.1162/neco.1991.3.1.79

R. Islam, R. Tian, L. M. Batten, and S. Versteeg, "Classification of malware based on integrated static and dynamic features," *Journal of Network and Computer Applications*, vol. 36, no. 2, pp. 646–656, 2013.

P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, vol. 1. IEEE Comput. Soc, 2001, pp. I–511–I–518. [Online]. Available: http://ieeexplore.ieee.org/document/990517/

X. Zhu, "Semi-Supervised Learning Literature Survey," University of Wisconsin-Madison, Tech. Rep., 2008. [Online]. Available: http://pages.cs.wisc.edu/~jerryzhu/pub/ssl_survey.pdf

B. Miller, A. Kantchelian, M. C. Tschantz, S. Afroz, R. Bachwani, R. Faizullabhoy, L. Huang, V. Shankar, T. Wu, G. Yiu, A. D. Joseph, and J. D. Tygar, "Reviewer Integration and Performance Measurement for Malware Detection," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 122–141. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-40667-1_7

R. Moskovitch, N. Nissim, and Y. Elovici, "Malicious Code Detection Using Active Learning," in *Privacy, Security, and Trust in KDD*, 2009, pp. 74–91. [Online]. Available: http://link.springer.com/10.1007/978-3-642-01718-6_6

H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 9, pp. 1263–1284, 2009.

O. Patri, M. Wojnowicz, and M. Wolff, "Discovering Malware with Time Series Shapelets," in *Proceedings of the 50th Hawaii International Conference on System Sciences*, 2017.

J. S. Cross and M. A. Munson, "Deep PDF Parsing to Extract Features for Detecting Embedded Malware," no. September, 2011.

G. Yan, N. Brown, and D. Kong, "Exploring Discriminatory Features for Automated Malware Classification," in *Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 41–61. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39235-1_3

J. Laurikkala, "Improving Identification of Difficult Small Classes by Balancing Class Distribution," in *Proceedings of the 8th Conference on AI in Medicine in Europe: Artificial Intelligence Medicine*, ser. AIME '01. London, UK, UK: Springer-Verlag, 2001, pp. 63–66. [Online]. Available: http://dl.acm.org/citation.cfm?id=648155.757340

G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning," *Journal of Machine Learning Research*, vol. 18, no. 17, pp. 1–5, 2017. [Online]. Available: http://jmlr.org/papers/v18/16-365.html

R. C. Prati, G. E. Batista, and M. C. Monard, "Data Mining with Imbalanced Class Distributions: Concepts and Methods," in *Indian International Conference on Artificial Intelligence (IICAI)*, Tumkur, India, 2009, pp. 359–376.

M. Kubat and S. Matwin, "Addressing the Curse of Imbalanced Training Sets: One Sided Selection," in *Proceedings of the Fourteenth International Conference on Machine Learning*, vol. 97, 1997, pp. 179–186.

H. M. Nguyen, E. W. Cooper, and K. Kamei, "Borderline over-sampling for imbalanced data classification," *International Journal of Knowledge Engineering and Soft Data Paradigms*, vol. 3, no. 1, pp. 4–21, 4 2011. [Online]. Available: http://dx.doi.org/10.1504/IJKESDP.2011.039875

H. Han, W.-Y. Wang, and B.-H. Mao, "Borderline-SMOTE: A New Over-sampling Method in Imbalanced Data Sets Learning," in *Proceedings of the 2005 International Conference on Advances in Intelligent Computing - Volume Part I*, ser. ICIC'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 878–887. [Online]. Available: http://dx.doi.org/10.1007/11538059_91

N. Chawla, K. Bowyer, L. Hall, and P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique," *Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002. [Online]. Available: http://arxiv.org/abs/1106.1813

X.-Y. Liu, J. Wu, and Z.-H. Zhou, "Exploratory Undersampling for Class Imbalance Learning," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 39, no. 2, pp. 539–550, 2009.

P. A. Gutierrez, M. Perez-Ortiz, J. Sanchez-Monedero, F. Fernandez-Navarro, and C. Hervas-Martinez, "Ordinal Regression Methods: Survey and Experimental Study," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 1, pp. 127–146, 1 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7161338/

S. K. Lim, A. O. Muis, W. Lu, and C. H. Ong, "MalwareTextDB: A Database for Annotated Malware Articles," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, 7 2017, pp. 1557–1567. [Online]. Available: https://www.aclweb.org/anthology/P17-1143

A. Pingle, A. Piplai, S. Mittal, A. Joshi, J. Holt, and R. Zak, "RelExt: Relation Extraction Using Deep Learning Approaches for Cybersecurity Knowledge Graph Improvement," in *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ser. ASONAM '19.   New York, NY, USA: Association for Computing Machinery, 2019, p. 879–886. [Online]. Available: https://doi.org/10.1145/3341161.3343519

A. T. Nguyen and E. Raff, "Heterogeneous Relational Kernel Learning," in *5th KDDWorkshop on Mining and Learning from Time Series*, 2019. [Online]. Available: http://arxiv.org/abs/1908.09219

C. Steinruecken, E. Smith, D. Janz, J. Lloyd, and Z. Ghahramani, "The Automatic Statistician," in *Automated Machine Learning*, ser. Series on Challenges in Machine Learning, F. Hutter, L. Kotthoff, and J. Vanschoren, Eds.   Springer, 5 2019.

Y. Hwang, A. Tong, and J. Choi, "Automatic construction of nonparametric relational regression models for multiple time series," *33rd International Conference on Machine Learning, ICML 2016*, vol. 6, pp. 4419–4433, 2016.

R. Grosse, R. Salakhutdinov, W. T. Freeman, and J. B. Tenenbaum, "Exploiting compositionality to explore a large space of model structures," in *Proceedings of the 28th Conference in Uncertainty in Artificial Intelligence*, N. de Freitas and K. Murphy, Eds.   Corvallis, Oregon, USA: AUAI Press, 2012.

J. R. Lloyd, D. Duvenaud, R. Grosse, J. B. Tenenbaum, and Z. Ghahramani, "Automatic Construction and Natural-Language Description of Nonparametric Regression Models," in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, ser. AAAI'14.   AAAI Press, 2014, p. 1242–1250.