Large-Scale Identification of Malicious Singleton Files

Bo Li
University of Michigan
bbbli@umich.edu

Kevin Roundy
Symantec Research Labs
Kevin_Roundy@symantec.com

Chris Gates
Symantec Research Labs
Chris_Gates@symantec.com

Yevgeniy Vorobeychik
Vanderbilt University
yevgeniy.vorobeychik@vanderbilt.edu

ABSTRACT

We study a dataset of billions of program binary files that appeared on 100 million computers over the course of 12 months, discovering that 94% of these files were present on a single machine. Though malware polymorphism is one cause for the large number of singleton files, additional factors also contribute to polymorphism, given that the ratio of benign to malicious singleton files is 80:1. The huge number of benign singletons makes it challenging to reliably identify the minority of malicious singletons. We present a large-scale study of the properties, characteristics, and distribution of benign and malicious singleton files. We leverage the insights from this study to build a classifier based purely on static features to identify 92% of the remaining malicious singletons at a 1.4% percent false positive rate, despite heavy use of obfuscation and packing techniques by most malicious singleton files that we make no attempt to de-obfuscate. Finally, we demonstrate robustness of our classifier to important classes of automated evasion attacks.

CCS Concepts

•Security and privacy → Software security engineering;

Keywords

Singleton files; malware detection; robust classifier

1. INTRODUCTION

Despite continual evolution in the attacks used by malicious actors, labeling software files as benign or malicious remains a key computer security task, with nearly 1 million malicious files being detected per day [29]. Some of the most reliable techniques label files by combining the context provided by multiple instances of the file. For example, Polonium judges a file based on the hygiene of the machines on which it appears [4], while Aesop labels a file by inferring its software-package relationships to known good or bad files, based on file co-occurrence data [31]. These detection technologies are unable to protect customers from early instances of a file because they require the context from multiple instances to label malware reliably, only protecting customers from later instances of the file. Thus, the hardest instance of a malware file to label is its first, and regrettably, the first instance is also the last in most cases, as most malware samples appear on a single machine. In 2015 around 89% of all program binary files (such as executable files with .EXE and .DLL extensions on Windows computers) reported through Norton's Community Watch program existed on only one machine, a rate that has increased from 81% since 2012. To make matters worse, real-time protection must label files that have been seen only once even though they may eventually appear on many other machines, putting the effective percentage of unique files at any given time at 94%.

We present the first large-scale study of *singleton* files and identify novel techniques to label such files as benign or malicious based on their contents and context. We define a singleton file as any file that appears on exactly 1 machine. We consider two files to be distinct when a cryptographic hash taken over their contents (such as SHA-256) yields a different result, meaning that two files that differ by a single bit are considered distinct even though they may be functionally equivalent.

Due to the fact that malware is often polymorphic, many malicious files are among these singletons. However, singleton executable files do not trend towards being malicious; in fact the opposite is true: the ratio of benign to malicious singleton files is 80 to 1, resulting in a skewed dataset. This ratio gives low prevalence malware a large set of files to hide amongst and it makes effective classification models difficult to train, as most machine learning models require relatively balanced data sets for effective training. We study the root causes behind the large numbers of benign singleton files in Section 2.2 and study malicious singletons in Section 2.3. We study the properties of machines that are prolific sources of benign singleton files in Section 3.1. We filter obviously benign singletons by profiling the prominent categories of benign singleton files that appear on such systems (Section 3.2). We present the full machine learning pipeline and the features we use to classify these samples in Section 3.3. We present experimental results in Section 4.

Since the phenomenon of malicious singleton files was largely driven by the arms race between security vendors and malicious adversaries in the first place, it is important to analyze robustness of our model against evasion attacks, and we do so in Section 4.3. We form the interactions between and ad-

CODASPY'17, March 22-24, 2017, Scottsdale, AZ, USA

DOI: http://dx.doi.org/10.1145/3029806.3029815

versary and our malware detection system as a Stackelberg game [34] and simulate evasion attacks on real singleton files to demonstrate that our proposed pipeline performs robustly against attacker interference.

In summary, we make the following contributions:

1. We provide the first detailed discussion of the role that benign polymorphism plays in making singleton file classification a challenging problem.
2. We identify root causes of benign polymorphism and leverage these to develop a method for filtering the most "obvious" benign files prior to applying malware classification methods.
3. We develop an algorithm that classifies 92% of malicious singletons as such, at a 1.4% false positive rate. We do so purely on the basis of static file properties, despite extensive obfuscation in most malware files, which we make no attempt to reverse.
4. We explore the adversarial robustness of multiple classification models to an important class of automated evasion/mimicry attacks, demonstrating the robustness of a performant set of features derived from static file properties.

2. SINGLETON FILES IN THE WILD

To address the paucity of information about singleton files, we study their causes, distribution patterns, and internal structure. We describe the predominant reasons for which software creators produce benign and malicious singleton files. For benign singletons, we identify the software packages that are the strongest predictors of the presence of benign singleton files on a machine. For malicious software, many singletons are produced from a relatively much smaller base of malware families. Thus, to better understand the nature of the polymorphism that is present in practice across a large body of singleton malware, we study the static properties of malicious singleton files across all malware families and within individual families.

2.1 Dataset Description

In the interest of performing a reproducible study, we perform the following study over data that is voluntarily contributed by members of the Norton Community Watch program, which consists of metadata about the binary files on their computers and an identifier of the machines on which they appear. Symantec shares a representative portion of this anonymized dataset with external researchers through its WINE program [10]. We use an extended window of time from 2012 through 2015 to generate high-level statistics about singleton data, and refer to this dataset as $D0$. We also use an 8-month window of data from 2014 for a more in depth analysis of the properties of singleton files and machines on which they appear, we call this $D1$. A portion of the files in $D1$ is labeled with high-confidence benign or malicious labels. We form dataset $D2$ by selecting a subset of the previous data that consists of labeled singleton files, and for which the file itself is available, allowing us to extract additional static features from the files that we describe in Section 3.3. The ground truth labels are generated by manually inspection and other high confident evidence. This dataset comprises 200,000 malicious and 16 million benign singleton files, and is the basis of the experimental evaluation of Section 4.

2.2 Benign Singleton Files

The abundance of benign singletons may be surprising given that there are not obvious benefits to distributing legitimate software as singleton files. Of course, some software is rare simply because it attracts few users, as in the case of software on a build machine that performs daily regression tests. However, there are also less obvious, but no less significant reasons behind the large numbers of singleton files, including the following:

1. The .NET Framework seeks to enable localized performance optimizations by distributing software in Microsoft Intermediate Language code so it can be compiled into native executable code by the .NET framework on the machine where it will execute, in a way that is specific to the machine's architecture. This is evident in practice, as .NET produces executables that are unique in most cases. Its widespread use makes it the largest driver of benign singleton files in our data.
2. Many classes of binary rewriting tools take a program binary file as input, producing a modified version as output, typically to insert additional functionality. For instance, tools such as Themida and Armadillo add resistance to tampering and reverse engineering, frequently to protect intellectual property and preserve revenue streams, as in the example of freemium games that require payment to unlock in-game features and virtual currency. Other examples of binary rewriting tools include the RunAsAdmin tool referenced in Table 1, which modifies executables so that administrative privileges are required to run them.
3. In many cases, software embeds product serial numbers or license keys in its files, resulting in a different hash-based identifier for otherwise identical files.
4. Singleton files can be generated by software that produces executable files in scenarios where other file formats are more typically used. For instance, Microsoft's Active Server Pages framework generates at least one DLL for every ASP webpage that references .NET code. Another example is ActiveCode's Building Information Modeling software that creates project files as executables rather than as data files. It is not uncommon for these frameworks to generate thousands of singleton binaries on a single machine.
5. Interrupted or partial downloads can result in files that appear to be singletons, even though they are really prefixes of a larger more complete file. If the entire file is available for inspection, this can be checked, but our dataset includes metadata for many files that have not been physically collected.

In Figure 1 we show the most common substring used in benign singleton filenames as extracted from dataset $D1$, many of which hint at the above factors. In particular, the most-observed filename pattern is "app-web-", which is seen in DLL files supporting web-pages created by ASP Web Applications. These files are often singletons because they are compiled from .NET code.

Using a subset of the data from dataset $D0$, we demonstrated in Figure 2 (a) that singleton files are not uniformly distributed across systems. The figure shows the number of machines that possess specific counts of singleton and non-singleton files. Figure 2 (b) is another way to view the same data, showing that almost 40% of machines have few or no singleton files and more than 94% of the systems have fewer than 100 singletons. Thus, the majority of singleton files

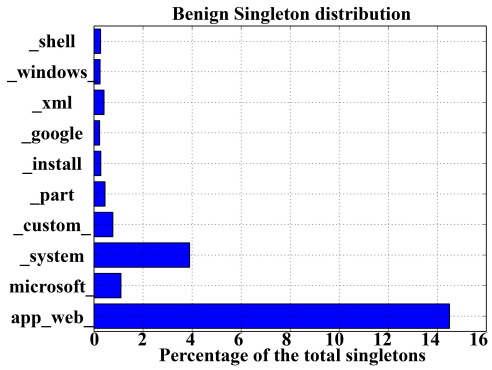

Figure 1: Percent of singleton files containing a specific substring.

come from the heavy tail of the distribution representing relatively few systems. Note that this data is from a specific period in time, and so machines with low numbers of non-singleton files indicate machines that experienced minimal changes/updates during the period when data was collected.

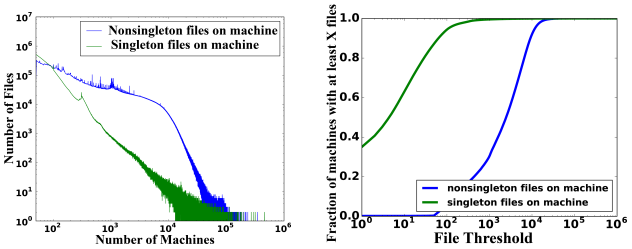

Figure 2: (a) Number of machines with a specific number of singleton/non-singleton files, (b) percent of machines that report more than X singleton and non-singleton files.

To help us work towards a solution that could identify benign singletons as such, we seek to better understand the machines on which they are most likely to exist. To identify software packages that could be responsible for the creation of singletons, we turn to the clustering approach proposed by Tamersoy et al. [31], which identifies software packages indirectly by clustering software files that are nearly always installed together on a machine, or not at all (see Section 3.1 for more details). Henceforth, we refer to these clusters as *software packages*. Once files are so clustered, we proceed by identifying the software packages that are most indicative of the presence of absence of singletons on a machine. Let S denote a specific software package (cluster). We identified a set of 10 million machines from $D1$, each of which contains at least 10 benign singleton files, which we denote by H (for *HasSingletons*). Likewise, we identified 10 million machines from $D1$ with no singleton files, which we denote by N, for *NoSingletons*. We identify the predictiveness of each software package S by counting its number of occurrences in each H and N, and use these counts to compute the odds ratios (OR) of a machine containing singletons given S, $OR(S) = H/N$. Intuitively, the higher $OR(S)$ is for a particular software package S, the more likely it is that this (benign) package generates many singletons. An $OR(S)$ ratio that is close to 1 is indicative of a software package that is equally likely to appear on machines that do and do not contain singletons, and therefore probably does not generate singletons itself. On the other hand, an $OR(S)$ that is significantly lower than 1 indicates that machines on which S is installed are tightly controlled or special-use systems unlikely to contain singleton files.

Table 1 shows software packages that are strong predictors for the presence (or absence) of benign singletons on a machine. Software packages that correlate with increased numbers of singletons include compiler-related tools (Visual Studio, SoapSuds, SmartClient), tools that wrap or modify executables (RunAsAdmin, App-V), and software packages that include numerous signed singletons (Google Talk Plugin). Interestingly, there are also many software packages that correlate strongly with an absence of singletons on the system. These are indicative of tightly controlled or minimalist special-purpose systems.

Our ability to identify software packages that lead to presence/absence of many benign singleton files is a critical step towards developing a method for classifying malicious vs. benign singletons. In particular, as described in Section 3, it enables us to prune a large fraction of files as benign before applying machine learning methods, dramatically reducing the false positive rate.

2.3 Malicious Singleton Files

Malware files skew heavily towards low-prevalence files, and towards singleton files in particular. Using $D0$ we can see that this trend has increased in recent years: 75% of known malware files were singletons in 2012, and the rate increased to 86% by 2015. There are readily apparent reasons why malware files skew towards low-prevalence files, including the following:

1. Avoiding signature-based detection: Users typically want to prevent malware from running on their systems, and blocking a single high-prevalence file is much easier than blocking large numbers of distinct yet functionally equivalent files. Polymorphism is a widespread technique for producing many functionally equivalent program binaries, which aims to reduce the effectiveness of traditional Anti-Virus signatures over portions of the file.
2. Resistance to reverse engineering and tampering: Many malware authors pack, obfuscate or encrypt their binaries, often with the assistance of third-party tools that are inexpensive or free. Polymorphism is often a welcome byproduct of these techniques, though it is not necessarily the primary objective.
3. Malware attribution resistance: The ease with which malware authors can create many functionally equivalent malware files makes the problem of attributing a malicious file to its author much harder than it would be if the same file was used in all instances. For the same reason, polymorphism makes it difficult for security researchers to assess a malware family's reach. Modularity also allows for specific components to be used as needed, without unnecessarily exposing the binary to detection.

Despite the widespread availability and use of tools that can inexpensively apply polymorphism and obfuscation to malware binaries, the security industry has developed effective techniques to counter these. Much of the polymorphism seen in malware binaries is superficially applied by post-compilation binary obfuscation tools that "pack" the original contents of the malware file (by compressing or encrypting

Have singleton: Control set OR	Representative Filename	Software Name
13770:1	Appvux.dll	Microsoft App-V
11792:1	Soapsuds.ni.exe	SoapSuds Tool for XML Web Services
110501:2	Blpsmarthost.exe	SmartClient
36515:2	gtpo3d host.dll	Google Talk Plugin
13868:1	Runasadmin.exe	Microsoft RunAsAdmin Tool
8511:1	Microsoft.office.tools.ni.dll	Visual Studio
...	...	...
1:1702	Policy.exe	???
1:4392	vdiagentmonitor.exe	Citrix VDI-in-a-Box

Table 1: Software packages that are most predictive of presence/absense of benign singleton files. For succinctness, we represent each software package by its most prevalent filename.

the code), and add layers of additional obfuscation-related code [28]. There are some obfuscation tools that are far more complex than this, but most of them are used almost exclusively by either malicious or by benign software authors. Techniques used by the anti-virus industry to combat these obfuscations are discussed at the end of this section.

To provide additional insight into the nature of malware polymorphism, we study the use of polymorphism by 800 malware families that were observed in the wild in our $D1$ dataset. Overall, we found that 31% of these families are distributed exclusively as singletons, accounting for over 80% of all singleton malware files, while 60% of families rely exclusively on non-singletons. There is a subtle difference here, that by volume, the 60% of families account for many detections since they are higher prevalence, while the 80% of singletons account for a lower percent of all detections even though there are more of them, since they only occur on a single system.

To identify malware families that exhibit a high degree of polymorphism, we extracted about 200 static features from files belonging each malware family. Our features include most fields in the Portable Executable file header of Windows Executable files (such as file size, number of sections, etc.), as well as entropy statistics taken from individual binary sections, and information about dynamically linked external libraries and functions that are listed in the file's Import Table. For each malware family, we calculate variability scores as the average variance of our static features for the files belonging to that family. The families with the highest variability scores are:

- *Adware.Bookedspace*
- *Backdoor.Pcclient*
- *Spyware.EmailSpy*
- $Trojan.Usuge!gen3$
- $W32.Neshuta$
- $W32.Pilleuz$
- $W32.Svich$
- $W32.Tu1ik$

These malware families vary greatly in form, function, and scale, though they do share properties that help account for their high variance. In particular, all of these families are modular, infecting machines with multiple functionally different files that are of similar prevalence and have dramatically different characteristics. In all cases, there is at least an order of magnitude difference in file size between the largest and smallest binary. Furthermore, all samples apply binary packing techniques sporadically rather than in all instances.

Backdoor.Pcclient is a Remote Access Trojan and the lowest prevalence family that has high variance in the static features. Polymorphism is not evident in this family; its elevated variance is a reflection of a modular design, multiple releases of some of those modules, and large differences from one module to another. By contrast, $W32.Pilleuz$ is a very prevalent worm family, but its Visual Basic executables achieve high variance through extensive obfuscation and highly variable file sizes, which add to the worm's modularity and occasional use of packing. $W32.Neshuta$ is particularly interesting in that it infects all *.exe* and *.com* files on the machines that it compromises, resulting in many detected unique executables of differing sizes, in addition to its own modular and polymorphic code.

API Purpose	*API Function*
Anti-Analysis	$IsDebuggerPresent$ $GetCommandLineW$ $GetCurrentProcessId$ $GetTickCount$ $Sleep$ $TerminateProcess$
Unpack Malware Payload	$GetProcAddress$ $GetModuleHandleW$ $GetModuleFileNameW$
Load/Modify Library Code	$CreateFileMappingA$ $CreateFileMappingW$ $MapViewOfFile$ $SetFilePointer$ $LockResource$
Propagation	$GetTempPathW$ $CopyFileA$ $CreateFileW$ $WriteFile$

Table 2: Categories of Windows API functions that are disproportionately used by malware

The Windows API functions imported by malware files provide interesting insights into their behavior, and are useful as static features, because they are reasonably adversarially resistant. Though malware authors can easily add imports for API functions that they do not need, removing APIs is significantly harder, as these may be needed to compromise the system (e.g., CreateRemoteThread). The only inexpensive way in which a malware file can hide its use of API functions from static analysis is to use a binary packing tool so that its Import Table is not unpacked until runtime, when it is used to dynamically link to Windows API functions. However, this technique completely alters the file's static profile and introduces the static fingerprint of the obfuscation tool, offering an indication that the file is probably malicious. In addition, as discussed at the end of this section, these obfuscations can be reversed by anti-virus vendors.

Table 2 lists the API functions that are most disproportionately used by malware, categorized by the purpose for which malware authors typically use them. Many of these

APIs support analysis resistance, either by detecting an analysis environment, hiding behavior from analysis, or by actively resist against analysis. Most other APIs that are indicative of malware have to do with linking or loading to additional code at runtime, typically because the malware payload is packed, but also for more nefarious purposes, such as malicious code injection and propagation.

Anti-Virus Industry Response to Obfuscation:

The anti-virus industry has sought to adapt to malware's widespread use of obfuscation tools by applying static and dynamic techniques to largely reverse the packing process in a way that preserves many of the benefits of static analysis. In particular, these techniques allow malicious code to be extracted, along with the contents of the Import Address Table, which contains the addresses of functions imported from external dynamically linked libraries. Unpacking techniques include the "X-Ray" technique, which may be used to crack weak encryption schemes or recognize the use of a particular compression algorithm and reverse its effects [27]. Most unpacking techniques, however, have a dynamic component and can be broadly classified into emulators and secure sandboxes. Emulators do not allow malicious files to execute natively on the machine or to execute real system calls or Windows API calls, but provide a good approximation of a native environment nonetheless. They are frequently deployed on client machines so that any suspicious file can be emulated long enough to allow unpacking to occur, after which the program's malicious payload can be extracted from memory and the de-obfuscated code can be recovered and analyzed. Offline analysis of suspicious program binaries typically uses a near-native instrumented environment where the malware program can be executed and its dynamically unpacked malicious payload can be extracted [12]. Though there are more elaborate obfuscation schemes that can make executable files difficult to unpack with the aforementioned techniques, these are either not widely deployed (e.g., because they are custom-built for the malware family) or are used predominantly by benign or malicious software, but not both. Thus, effective benign vs. malicious determinations can be made even in these cases, because the obfuscation toolkits leave a recognizable fingerprint.

Though the effectiveness of the above de-obfuscation techniques is open to debate, in our methodology for this paper, we make the deliberate choice to use no de-obfuscation techniques at all in our attempts to classify singleton files. We demonstrate that malware classification based purely on static features can be successful, even in the face of extensive polymorphism, by good and bad files alike. The success we achieve demonstrates that the obfuscation techniques that are widely used by malware are themselves recognizable, and appreciably different from the kinds of polymorphism that are common in benign files. We expect that the classification accuracy of our methodology would improve when applied to files that have been de-obfuscated, given that other researchers have found this to be the case [13].

3. LEARNING TO IDENTIFY MALICIOUS SINGLETONS

Most prior efforts for identifying malicious files have either relied on the context in which multiple instances of the file appear (e.g., Polonium [4] and Aesop [31] systems) or have relied exclusively on static or dynamic features derived from the file itself (e.g., MutantX-S [13]). The context that is available for a singleton file is necessarily limited, making the aforementioned context-dependent techniques not applicable. Making matters worse is the fact that the ratio of benign to malicious singleton files is nearly 80:1, which has the effect of multiplying the false positive rate of a malware detector by a factor of 80, and presents a significant class imbalance problem that makes effective classifier training difficult.

To address the lack of context for singleton files and the preponderance of benign singleton files, we leverage insights gleaned from our empirical observations about singleton files in the wild. In particular, as discussed in Section 2.2, a handful of software packages generate the lion's share of benign singletons, while other packages correlate with their absence. Furthermore, the toolchains that generate benign singletons in large numbers imbue them with distinctive static properties that make them easy to label with high confidence. We use these insights to develop a pipeline that filters benign singleton files with high confidence, yielding a more balanced dataset of suspicious files.

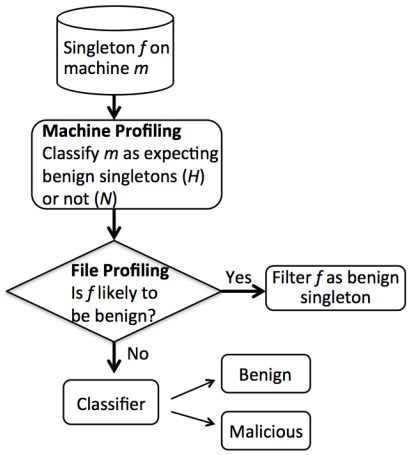

Figure 3: Pipeline of the singleton classification system.

Figure 3 presents a diagram of our pipeline. We take as input a pair (f, m), where f is a file and m is the machine on which it resides. The first step of the pipeline, which we call *machine profiling*, determines whether m is likely to host many benign singleton files. The second step is *file profiling*, in which we label obviously benign files, primarily from many-singleton machines, by determining that they closely match the benign files that are common on such systems. The final step, *classification*, uses a supervised classification algorithm (we explore the use of Support Vector Machines [30] and Recursive Neural Networks [21]) to render a final judgment on the remaining files. We proceed by describing each of our pipeline's components in detail.

3.1 Machine profiling

Machine profiling operationalizes the following insight gleaned from our empirical observations: since the distribution of benign singletons is highly non-uniform, singleton classification will benefit from identifying machines that are likely to host many benign singletons. As discussed in Section 2.2,

the software packages present on a machine are highly predictive of the presence or absence of benign singletons.

The first challenge we face is that of automatically identifying software packages from telemetry about installations of individual program binary files. In mapping individual files to software packages, we wish to proceed in an automated way that is inclusive of rare software that is not available for public download. Our approach adopts the clustering portion of the Aesop system described by Tamersoy et al. [31], in which they leverage a dataset consisting of tuples of file and machine identifiers, each of which indicates the presence of file f on machine m. Specifically, let F be a set of (high-prevalence) files (in the training data). For each file $f \in F$, let $M(f)$ be the set of machines on which f appears. As Aesop did, we use locality sensitive hashing [11] to efficiently and approximately group files whose $M(f)$ sets display low `Jaccard` distance to one another. The Jaccard distance between two sets X and Y is defined as: $J(X,Y) = 1 - \frac{X \cap Y}{X \cup Y}$, and we define the distance between two files f and f' in terms of Jaccard distance as $d(f,f') = J(M(f), M(f'))$. We tune locality sensitive hashing to cluster files with high probability when the Jaccard distance between the files is less than 0.2, and to cluster them very rarely otherwise. We obtain a collection of clusters $\mathcal{C}$, such that each cluster $C \in \mathcal{C}$, serves as an approximation of a software package, since C represents a collection of files that are usually installed together on a machine or not at all.

We proceed by identifying the approximate software packages that are the best predictors for the presence of singleton files. We formulate this task as a machine learning problem. We define a feature vector for each machine m that encodes the set of software packages that exist on m. Specifically, given n clusters (software packages), we create a corresponding binary feature vector s_m of length n, where $s_{mj} = 1$ iff cluster j is present on machine m. Next, we append a label l_m to our feature vector such that we have $\{s_m, l_m\}$ for each machine, with feature vectors s_m corresponding to machines and labels $l_m \in \{H, N\}$ representing whether the associated machine ***h***as benign singletons (label H) or has ***n***o singletons (label N). With this dataset in hand, we are able to train a simple, interpretable classifier to predict l_m to good effect. Had we used individual files as predictors, we would have to choose a machine learning algorithm that behaves well in the presence of strongly correlated features, but software package identification dramatically reduces feature correlation. Thus, we select Naive Bayes as our classifier $g(s)$, which performs well and gives us significant insight into the software packages that are the best indicators of the presence or absence of benign singleton files, as reported in Table 1. Our classifier takes as input a feature vector s that represents the software packages on a given machine, and outputs a prediction as to whether or not the machine has benign singletons. To achieve a balanced dataset, we randomly selected 2,000,000 uninfected machines, half of which contain singletons and half of which do not.

3.2 File profiling

Given a classifier $g(m)$ that determines whether a machine m is expected to host benign many singletons, the next step in our pipeline—*file profiling*—uses this information to identify files that can be confidently labeled benign. The result is both a more balanced dataset that makes our pipeline's classifier easier to train, as well as a high-confidence labeling technique that reduces classifier's false positive rate. The main intuition behind our proposed file profiling method is that benign singleton files bear the marks of the specific benign software packages that generate them. Of course, different software generates singletons with dramatically different file structures and file-naming conventions. Consequently, we seek to identify prototypical benign singletons by clustering them based on their static properties, and filter benign files that closely match these prototypes. Since the information we have about the software installed on any given machine is typically incomplete, we filter benign files that closely match benign-file prototypes on all machines, but require much closer matches on machines where benign singletons are not expected. This point is operationalized below through the use of a less aggressive filtering threshold for machines m labeled as N (no benign singletons) than for machines labeled H (having benign singletons).

The full path, filename, and size of singleton files are the primary static attributes that we use in our file profiling study. We had little choice in this case because large collections of labeled benign singleton files that security companies share with external researchers are extremely hard to come by, and are limited in the telemetry they provide. In the interest of conducting a reproducible experiment, we limit ourselves to the metadata attributes provided for files in Dataset $D1$ (see Section 2.1) that Symantec shares with external researchers through its WINE program [10]. Though $D1$ gives us a representative dataset of singleton files, it also limits us to a small collection of metadata attributes about files, of which the path, filename, and size are the most useful attributes. In modest defense of the use of filename and path as a feature, though it is true that a malicious adversary can trivially modify the malware's filename (and the path, to a lesser extent), the malware author would frequently have to do so at the cost of losing the social engineering benefit of choosing a filename that entices the user to install the malware.

Due to the feature limitations of the file profiling step, we proceed by developing techniques to maximize the discriminative value of the path and filename. We seek to leverage the observation that a handful of root causes create a significant majority of benign singletons, and these origins are often strongly evident in the filename and path of benign singletons. Although malware files display significantly more diversity in their choice of filenames, these filenames typically bear the marks of social engineering, and their paths are frequently reflective of the vector by which they managed to install themselves on the machine, or are demonstrative of attempts to hide from scrutiny. Accordingly, we engineer features from filenames and paths to capture the naming conventions used by benign singletons. Given a file f, we divide its filename into words using chunking techniques. Specifically, we identify separate words within each file name that are demarcated by whitespace or punctuation, and separate words based on CamelCase capitalization transitions, and so on. Subsequently, we represent the filename and path components in a "bag of words" feature representation that is physically represented as a binary vector, where the existence of a word in the filename or path corresponds to a 1 in the associated feature, and a 0 indicates that the word is not a part of its name. In addition, we capture the relative frequencies of the words that appear in filenames by measuring the term frequency (TF) of each word. Term fre-

quency is then used as a part of weighted Jaccard distance measure used to cluster files, as described below. More formally, let $T \subseteq \mathbb{R}^n$ represent the feature space of the singleton files, with n the number of features. Each singleton file f can be represented by a feature vector t, which is the dot product of a binary bag of words vector w and the normalized term frequency vector q corresponding to each word, $t = w \cdot q$, where t^j is the jth feature value. Note that we exclude words that appear extremely frequently, such as *exe, dll, setup*, as stop words, to prevent the feature vector t from becoming dominated by these. For any two files f_1 and f_2, the weighted Jaccard distance between them is then calculated as $J(f_1, f_2) = 1 - \frac{\sum_k \min({f_1}^k, {f_2}^k)}{\sum_k \max({f_1}^k, {f_2}^k)}$.

We use the weighted Jaccard distance to cluster benign singleton files in the training data using the scalable NN Descent algorithm [9] implemented on Spark [33], which efficiently approximates K-Nearest Neighbors and produces clusters C of of highly similar files.[1] We gain further efficiency and efficacy gains by choosing a bag of words representation over edit distance when making filename comparisons. This approach also has the benefit of producing an understandable model that identifies the most frequent filename patterns present in benign singleton files, such as those highlighted in Figure 1.

The final step in the file profiling process is to use the clusters derived above to filter benign files that align closely with the profile of benign singletons. To this end, for each benign singleton cluster $c \in C$, we compute the cluster mean $\bar{c} = \frac{1}{|c|} \sum_{t_j \in c} t_j$. For a given file f, we then find the cluster ; let c^* whose mean $\bar{c}$ is least distant from f, where distance is again measured based on weighted Jaccard distance: $J(\bar{c}, f)$. Then, if file f resides on a machine m that is expected to have singletons (that is, $g(m) = H$ as defined in Section 3.1), we filter it as benign iff $J(c^*, f) \leq \theta_H$; otherwise, it is filtered iff $J(c^*, f) \leq \theta_N$, where θ_H and θ_N are the corresponding filtering thresholds.

We select different θ values for the training and final versions of our pipeline. For training, our primary goal is to reduce the 80:1 benign to malicious class imbalance ratio so that we can train an effective classifier, whereas for testing, our goal is to achieve a high true positive rate while minimizing false positives. For purposes of creating a balanced training set, we select $\theta_N = 0.1$ and $\theta_H = 0.3$, which filters 91.8% of benign singletons, resulting in a more manageable 9:1 class imbalance ratio, at the cost of 7% of malware samples being thrown out of our training set. However this does not affect the performance of our model adversely, since during testing we can be less aggressive with the thresholds and pass more files to the classifier. In practice, we found values around $\theta_N = 0.07$ and $\theta_H = 0.13$ result in the best performance over the test data.

3.3 Malicious singleton detection

Having filtered out a large portion of predicted benign file instances, we are left with a residual data set of benign and malicious files that we classify using supervised-learning techniques. Though the filtering of benign files by the previous stages of our pipeline provide better class balance, we found that significant improvements in classification accuracy result when the residual data set is augmented by including 3 benign files that we sample randomly from each cluster C generated in the file profiling step. Doing so improves the classifier by adding additional benign files that are representative of the overall population of benign singleton files. We trained multiple classification algorithms with different strengths to determine which would be most effective at singleton classification.

Feature engineering is also key to the performance of our classifiers. Whereas machine and file profiling were designed for a backend system where a global view of the distribution of benign and malicious singleton files is available, here we design a classifier that we can deploy on client machines, based entirely on the static features of the file. Hence, we assume direct access to the files themselves and can build rich feature sets over the files, so long as they are not expensive to compute. This is in contrast to the telemetry used for machine and file profiling, for which network bandwidth constraints and privacy concerns limited the telemetry that could be collected. As mentioned in Section 2.3, we make no attempt to reverse the effects of obfuscation attempts employed by malware, finding that the use of the obfuscation techniques themselves provides strong discriminative power that helps us to disambiguate between benign and malicious singletons.

Features.

The features used by our learning algorithms to classify singleton program binary files fall into four categories.

1. The first category of features corresponds to features of file name and path. For these we used the same file name and path bag-of-words feature representation here as in the file profiling step of Section 3.2. To reduce the number of features included in our model, we applied a chi-squared feature selection to choose the most discriminative features [19].
2. The second category of features are derived from the header information of the executable file. We include all fields in the headers that are common to most windows executable files that exhibit some variability (some header fields never change). These header fields include the MS-DOS Stub, Signature, the COFF File Header, and the Optional Header (which is optional but nearly always present) [6].
3. We derive features from the Section Table found in the file's header, which describes each section of the binary, and also compute the entropy of each of the file's sections as features.
4. Our third category of features is derived from the external libraries that are dynamically linked to the program binary file. To determine which libraries the file links to, we create a feature for each of the most popular Windows library files (primarily Windows API libraries) that represents the number of functions imported from the library. We also create binary features for the individual functions in common Windows libraries that are most commonly used by malware. These take a value of 1 when the function is imported and 0 otherwise.

In all, category 1's bag of words features for filename and path consist of 300 features, while category 2,3, and 4 features together comprise close to 1000 features.

[1]Note that this clustering of files is entirely distinct from the clustering of files in machine profiling, where non-singleton files are clustered based on machines that they appear on.

Classification.

We apply two learning models, a Recurrent Neural Network (RNN) [23] and a Support Vector Machine with a radial basis function as its kernel [3], and compare their performance and ability to withstand adversarial manipulation in Section 4. The RNN model is particularly suited for textual data, so we train it solely using file names and path information as features. Given the sequential properties of the file name text, RNNs aim to make use of the dependency relationship among characters to classify malicious vsbenign singletons. The goal of the character-level learning model is to predict the next character in a sequence and thereby classify the entire sequence based on the character distribution. Here, given a training sequence of characters $(a_1, a_2, ..., a_m)$, the RNN model uses the sequence of its output vectors $(o_1, o_2, ..., o_m)$ to obtain a sequence of distributions $P(a_{k+1}|a_{\leq k}) = softmax(o_k)$, where the softmax distribution is defined by $P(softmax(o_k) = j) = exp({o_k}^{(j)})/\sum_k exp({o_k}^{(l)})$. The learning model's objective is to maximize the total log likelihood of the training sequence, which implies that the RNN learns a probability distribution over the character sequences used in a full path + filename.

For the SVM model, we apply the text chunking technique described in Section 3.2, and use the bag-of-words representation as described above, concatenated with static and API-based features, where relevant. While numerous other classification algorithms could be used here, our purpose of exploring RNN and SVM specifically is to contrast an approach specifically designed for text data (making use of filename and path information exclusively) with a general-purpose learning algorithm that is known to perform well in malware classification settings [16].

Putting Everything Together.

The high-level algorithm for the entire training pipeline is shown in Algorithm 1. The input to this algorithm is a collection of tuples $\{s_i, z_i, m_i, y_i\} \in \{S, Z, M, Y\}$ describing file instances on machines, which are partitioned into training (tr) and testing (te) for the pipeline. Each file instance is represented by s_i, the 256-bit digest of a SHA-2 hash over its contents and the size z_i of the file in bytes. The machine is represented by a unique machine identifier m_i, and each instance of the file receives a label y_i, which designates a file as benign, malicious, or unknown. Machine profiling processes the file-instance data to identify singleton files (those for which only one instance exists) from more prevalent software that it groups into packages and uses to predict the presence or absence of singletons. The end result of training the pipeline includes the two classifiers: g classifies machines into H (has benign singletons) and N (no benign singletons), while h classifies files as malicious or benign, trained based on the selected representative data D. Additional by-products include, the clusters of benign files C and the thresholds θ_H and θ_N that determine how aggressively files projected to be benign are filtered before the classifier h is applied.

Algorithm 1 Train($\{S_{tr}, Z_{tr}, M_{tr}, Y_{tr}\}$):

1: $g = \text{machineProfiling}(\{S_{tr}, Z_{tr}, M_{tr}, Y_{tr}\})$
2: $(D, \theta_H, \theta_N, C) = \text{fileProfiling}(\{S_{tr}, Z_{tr}, M_{tr}, Y_{tr}\}, g)$
3: $h = \text{learnClassifier}(D)$
4: **return** $g, h, \theta_H, \theta_N, C$

Our test-time inputs include a set of singleton files that we withheld from training and our model parameters, and it returns simply whether or not to label f as benign or malicious. The specifics of the associated testing process, which use of our training pipeline, are given in Algorithm 2.

Algorithm 2 Predict($\{S_{te}, M_{te}\}$, $g, h, \theta_H, \theta_N, C$):

1: $l = g(M_{te})$: label the machine as H or N
2: $c^* = \arg\min_{c \in C} J(S_{te}, c)$ // find closest cluster center to S_{te}
3: **if** $J(S_{te}, c) \leq \theta_l$ **then**
4: **return** B // "benign" if S_{te} is close to a benign cluster center
5: **end if**
6: **return** $h(S_{te})$ // otherwise, apply the classifier

4. EXPERIMENTAL EVALUATION

We conduct experiments on a large real-world dataset, dataset $D2$ as described in Section 2.1, to evaluate the proposed pipeline as well as analyze the robustness of learning system. As mentioned above, in implementing and deploying such a system in practice, we face a series of tradeoffs. The first is how much information about each file we should be collecting. On the one hand, more information will likely improve learning performance. On the other hand, collecting and analyzing data at such scale can become extremely expensive, both financially and computationally. Moreover, collection of detailed data about files on end-user machines can become a substantial privacy issue. For all of these reasons, very little information is traditionally collected about files on end-user systems, largely consisting of file name and an anonymized path, as well as file hashes and machines they reside on. For a subset of files, deeper information is available, including static features as well as API calls, as discussed above. However, these involve a significant cost: for example, extracting API calls requires static analysis. Our experiments are therefore designed to assess how much value these additional features have in classification, and whether or not it is truly worthwhile to be collecting them at the scale necessary for practical deployment. Since we are the first work to deal with the singleton malware detection problem, here we compare our proposed method with standard machine learning algorithms in various settings. Our evaluation applies Machine Profiling (MP), File Profiling (FP), an RNN based on only file name features, a SVM based on file name features, a SVM based on both file name and the static features (SVMS), and a SVM based on file name, static features, and API function features (SVMSF).

4.1 Baseline Evaluation

Our first efficacy study demonstrates the benefit provided by our machine learning pipeline as compared to two natural baselines. Our first baseline applies machine and file profiling, ranking all examples based on their similarity to benign files, and identifying the samples that are furthest from benign cluster centers as malicious. Our second baseline is our best-performing classifier trained over our entire feature set (SVMSF), but trained without the benefit of an initial machine/file profiling step, which reduces the ratio of

benign to malicious files from an 80:1 ratio to a 9:1 ratio. This baseline is similar to prior work in malware classification based on static features [13]. As seen in Figure 4, our full pipeline demonstrates clear improvement over the two baselines, with a significantly higher AUC score. The spot on the curve with the maximal $F_{0.5}$ score achieves a 92.1% true positive rate at a 1.4% false positive rate, a dramatic improvement over applying FP or SVMSF on its own. Different locations on the ROC curve are achieved by selecting increasing values for θ_N and θ_H. The maximal $F_{0.5}$ score is achieved with $\theta_H = 0.13$ and $\theta_N = 0.07$.

Though uninformed downsampling of benign files may reasonably be suggested as an alternative means to reduce the class imbalance and achieve better classification results with SVMSF, our attempts to do so resulted in classifiers that perform worse than the SVMSF classifier of Figure 4. The reason for this is likely that downsampling decimates small clusters of benign files, resulting in a model that represents benign singletons only by its most massively populated clusters. Our pipeline can be thought of as providing an informed downsampling of benign files that reduces massively populated clusters of benign files to a few prototypes, allowing the SVM to train a model that represents the full gamut of benign singletons with the additional benefit of doing so over a more balanced dataset.

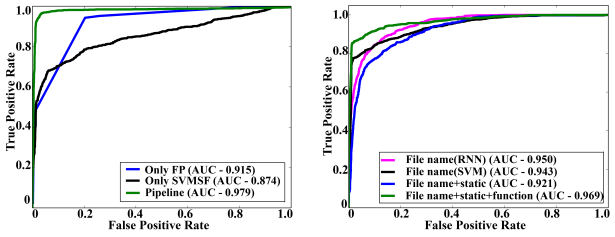

Figure 4: (a) ROC-curve comparison of the pipeline performance with the two baselines: no machine/file profiling, and only machine/file profiling. (b) Comparisons for models with different features without attacker.

4.2 Evaluating Performance of the Classification Step

To assess the relative importance of the three classes of features (text, static, and API) used by our model, we analyze the relative performance of just the last classification step of four models on the dataset produced by MP and FP filtering: 1) RNN (using text features only), 2) SVM (using text features only), 3) SVM with both text and static features, and 4) SVM with text, static, and API features.

To highlight the performance differences between these classifiers, we evaluate them over a test set of singletons from which obviously benign singletons have been pre-filtered by file profiling (for this reason this figure does not reflect the overall performance of our pipeline as reported in Figure 4). Our first observation is that RNN outperforms SVM when only textual features are used, which is not surprising, given that RNN's are particularly well suited to text data. Second, our model's performance drops when training over filename and anonymized path plus static features, which demonstrates the high discriminative value of the filename and anonymized path relative to features derived from header information in the executable. However, these static features do offer value when we account for the potential for adversarial manipulation, as discussed in Section 4.3. Third, the value of features based on imported API functions is evident in the performance of the SVMSF model compared to all other models, particularly when we choose a threshold that limits the false positive rate, as security vendors are prone to do: The precision and recall scores that produce a maximal $F_{0.5}$ score for SVMSF are 83% recall at a 1% false positive rate, as compared to 76% recall at a 5% false positive rate for RNN, which is this model's closest competitor on an Area Under the Curve (AUC) basis. Note that the performance of the full pipeline is better than either of these classifiers alone (see Figure 4), because many of the benign files that are causing the FPs are labeled correctly using the machine and file profiling steps. Finally, our adversarial evaluation of these classifiers (Section 4.3) offers additional justification for incorporating static and imported function-based features into our model.

We evaluated the run-time required to train each step of our pipeline, including Machine Profiling (MP), File Profiling (FP), and the selected classifier, which is one of the following: RNN, SVM (based on only file name), SVMS, and SVMSF. The run-time of each step, when performed on a single powerful machine, is illustrated in Figure 5. Training Machine Profiling and File Profiling is fairly expensive, However, these two steps can be done offline, and updated incrementally as new data arrives. Training the SVM classifiers is inexpensive, whereas training the RNN takes on the order of three hours with GPU acceleration. Though we do not believe that this is a cause for concern, the inferior performance of the RNN as compared to SVMSF makes it less appealing for inclusion in the final version of our pipeline. We do not include test-time performance evaluation since the cost to test a single file is negligible for all stages of the pipeline.

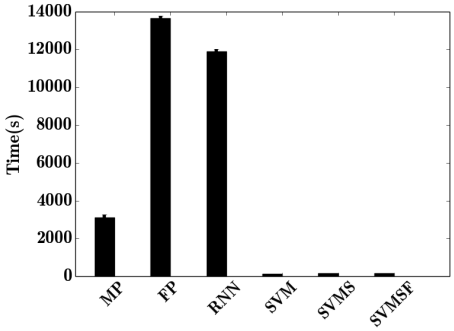

Figure 5: Comparisons of the runtime of different components within the pipeline.

4.3 Adversarial Evaluation

Though the evaluation of our classifiers, presented in Figure 4 (b) is fairly typical for a malware classification tool, it is not necessarily indicative of the long-term performance of a classifier once it has been massively deployed in the wild. In particular, what is missing is an evaluation of the ability of our classifier to withstand the inevitable attempts of malware authors to respond to its deployment by modifying their malicious singleton files to mimic benign file patterns in order to evade detection. Whereas researchers have

traditionally discussed an algorithm's robustness to evasion based on subjective arguments about the strength or weakness of individual features, the now well-developed body of research on *adversarial machine learning* provides more rigorous methods for evaluating the adversarial robustness of a machine learning method [8, 20], and provides guidelines for developing more adversarially robust learning techniques [18, 32].

We proceed by providing an evaluation of our model's adversarial robustness. The adversarial resistance of a classifier evaluation presupposes a given classifier, h, that outputs for a given feature vector x, a label $h(x) \in \{-1, +1\}$, where in our case, -1 represents a benign prediction and $+1$ represents a malicious prediction. Given h, the adversary is modeled as aiming to minimize the cost of evasion,

$$x^* = \arg \min_{x' | h(x')=-1} c(x, x'),$$

where $c(x, x')$ is the cost of using a malicious instance x' in place of x to evade h (by ensuring that $h(x') = -1$, that is, that the malicious file will be classified as benign). The optimal evasion is represented by x^*. Because this model always results in a successful evasion, no matter its cost, we follow a more realistic model presented by Li and Vorobeychik [17], where the evasion only occurs when its cost is within a fixed adversarial budget B, thus: $c(x, x^*) \leq B$. Similarly, we mainly focus on the binary features here and prioritize the ones that have the most distinguished values for malicious and benign to modify, focusing the adversaries budget on the features that will be most useful for them to modify under the assumption that they known how to mimic benign software. In effect, we assume that the adversary will evade detection only if the gains from doing so outweigh the costs. The budget represents the percentage of the total number of features that the attacker is able to modify. A natural measure of the evasion cost $c(x, x')$ is the weighted l_1 distance between x and x': $c(x, x') = \sum_i a_i \|x_i - x'_i\|$. The choices of weights can be difficult to determine in a principled way, although some features will clearly be easier for an adversary to modify than others. We use $a_i = 1$ for all features i below as a starting point. As we will see, this already provides us with substantial evidence that a classifier using solely filename-based features is extremely exploitable by an adversary, without even accounting for the fact that such features are also easier to modify for malware authors than, say, the functions they import from the Windows API and other libraries.

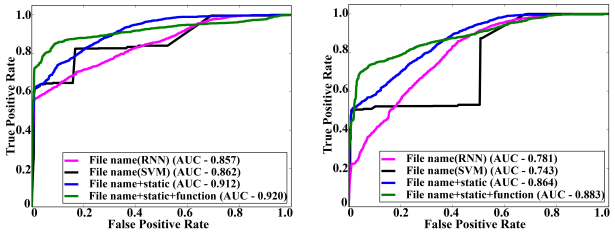

Figure 6: Comparisons for models with attacker budget as (a) 5 (b) 10.

We now perform a comparison of the same classifier and feature combinations presented in Section 4.2, but we now evaluate these classifiers using evasion attacks, as shown in Figures 6 (a) and (b) with budgets $B = 5\%$ and $B = 10\%$, respectively. These figures highlight a significant trend: whereas the RNN's performance was previously rather close to that of the SVM with filename, static, and imported function features, the former has displays poor adversarial resistance, while the latter is far more robust. The RNN's AUC drops to 0.857 under pressure from a weaker attacker, and to 0.78 when pressured by a stronger one, whereas the AUC for the SVM with the largest feature set only drops to 0.92 under a smaller adversarial budget, and to 0.88 with a larger one). The SVM based only on filename features performed even worse than the RNN. Interestingly, while adding static features (and not imported function features) to the SVM degrades its adversary-free performance, the classifier performs considerably better than the RNN and SVM with filename features, in the presence of an adversary.

In summary, our experimental results point consistently to the use of a Support Vector Machine with features derived from the filename, path, static properties of the file, and imported functions, as the model that performs the best, even against an active adversary. Thus, the best version of our overall pipeline leverages this support vector machine as its classifier, achieving the overall performance results shown in Figure 4.

5. RELATED WORK

The problem of detecting malicious files has been studied extensively. Perdisci *et al.* have dealt with the static detection of malware files [25] and malware clustering using HTTP features [26]. Other malware detection systems have also been proposed [15, 7, 5]. Particularly relevant is work that is designed to deal with low-prevalence malware. This prior art includes work designed to reverse the effect of packing-based obfuscation tools by either statically decompressing or decrypting the malicious payload [27], or simply executing the program until it has unrolled its malicious payload into main memory [12]. At this point, traditional anti-virus signatures may be applied [22], and clustering may serve to identify new malicious samples based on their similarity to known malicious samples [13, 14]. By contrast, we make no effort to undo obfuscation attempts, which are frequently evidence of malicious intent. Whereas these researchers have focused on the causes behind low-prevalence malware, we augment this by providing the first detailed study of benign singleton files.

The importance of an adversarially robust approach to malicious singleton detection is evident, given that the high volume of singleton malware is largely the byproduct of adaptations to anti-virus technology [1, 17, 2]. Researchers have formalized the notion of evasion attacks on classifiers through game theoretic modeling and analysis [8, 24]. In one of the earliest such efforts, Dalve et al. [8] play out the first two steps of best response dynamics in this game. However, there has been a disconnect between the learner-attacker game models and real world dataset validation in these prior work. We bridge this gap by considering a very general adversarial learning framework based on an evaluation of a real, large-scale dataset.

6. CONCLUSIONS

We analyzed a large dataset to extract insights about the properties and distribution of singleton program binary files and their relationships to non-singleton software. We leverage the *context* in which singletons appear to filter benign files from our dataset, allowing us to train a model over a

more balanced set of positive and negative examples. We build a classifier and feature set over the *static contents* of the file to effectively label benign and malicious singletons, in a way that is adversarial robust. Together, these components of our pipeline classify singletons much more effectively than either a context or a content-based approach can do on its own.

7. ACKNOWLEDGMENTS

This research was partially supported by the NSF (CNS-1238959, IIS-1526860), ONR (N00014-15- 1-2621), ARO (W911NF-16-1-0069), AFRL (FA8750-14-2-0180), Sandia National Laboratories, and Symantec Labs Graduate Research Fellowship.

8. REFERENCES

[1] Brückner, M., and Scheffer, T. Nash equilibria of static prediction games. In *Advances in neural information processing systems* (2009), pp. 171–179.

[2] Bruckner, M., and Scheffer, T. Stackelberg games for adversarial prediction problems. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining* (2011), ACM, pp. 547–555.

[3] Chang, C.-C., and Lin, C.-J. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology 2* (2011), 27:1–27:27.

[4] Chau, D. H., Nachenberg, C., Wilhelm, J., Wright, A., and Faloutsos, C. Polonium: Tera-scale graph mining and inference for malware detection. In *SIAM International Conference on Data Mining* (2011), vol. 2.

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

[6] Corporation, M. Microsoft portable executable and common object file format specification. Revision 6.0.

[7] Dahl, G. E., Stokes, J. W., Deng, L., and Yu, D. Large-scale malware classification using random projections and neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on* (2013), IEEE, pp. 3422–3426.

[8] Dalvi, N., Domingos, P., Sanghai, S., Verma, D., et al. Adversarial classification. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining* (2004), ACM, pp. 99–108.

[9] Dong, W., Moses, C., and Li, K. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web* (2011), ACM, pp. 577–586.

[10] Dumitras, T., and Shou, D. Toward a standard benchmark for computer security research: the worldwide intelligence network environment (wine). In *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)* (Salzburg, Austria, 2011).

[11] Gionis, A., Indyk, P., and Motwani, R. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)* (Edinburgh, Scotland, UK, 1999).

[12] Guo, F., Ferrie, P., and Chiueh, T. A study of the packer problem and its solutions. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (Cambridge, MA, 2008), Springer Berlin / Heidelberg.

[13] Hu, X., Shin, K. G., Bhatkar, S., and Griffin, K. Mutantx-s: Scalable malware clustering based on static features. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 187–198.

[14] Jang, J., Brumley, D., and Venkataraman, S. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 309–320.

[15] Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X.-y., and Wang, X. Effective and efficient malware detection at the end host. In *USENIX security symposium* (2009), pp. 351–366.

[16] Kolter, J. Z., and Maloof, M. A. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research 7* (2006), 2721–2744.

[17] Li, B., and Vorobeychik, Y. Feature cross-substitution in adversarial classification. In *Advances in Neural Information Processing Systems* (2014), pp. 2087–2095.

[18] Li, B., and Yevgeniy, V. Scalable optimization of randomized operational decisions in adversarial classification settings. In *Proc. International Conference on Artificial Intelligence and Statistics* (2015).

[19] Liu, H., and Setiono, R. Chi2: Feature selection and discretization of numeric attributes. In *Proceedings of 7th Intenational Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 388–391.

[20] Lowd, D., and Meek, C. Adversarial learning. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining* (2005), ACM, pp. 641–647.

[21] LukošEvičIus, M., and Jaeger, H. Reservoir computing approaches to recurrent neural network training. *Computer Science Review 3*, 3 (2009), 127–149.

[22] Martignoni, L., Christodorescu, M., and Jha, S. Omniunpack: Fast, generic, and safe unpacking of malware. In *Annual Computer Security Applications Conference (ACSAC)* (Miami Beach, FL, 2007).

[23] Mikolov, T., Kombrink, S., Burget, L., Cernocky, J., and Khudanpur, S. Extensions of recurrent neural network language model. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (ICASSP)* (Prague, Czech Republic, 2011).

[24] Parameswaran, M., Rui, H., and Sayin, S. A game theoretic model and empirical analysis of spammer strategies. In *Collaboration, Electronic Messaging, AntiAbuse and Spam Conf* (2010), vol. 7.

[25] Perdisci, R., Ariu, D., and Giacinto, G. Scalable fine-grained behavioral clustering of http-based malware. *Computer Networks 57*, 2 (2013), 487–500.

[26] Perdisci, R., Lanzi, A., and Lee, W. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual* (2008), IEEE, pp. 301–310.

[27] Perriot, F., and Ferrie, P. Principles and practise of x-raying. In *Virus Bulletin Conference* (Chicago, IL, 2004).

[28] Roundy, K. A., and Miller, B. P. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys (CSUR) 46*, 1 (2013).

[29] Security, and Group, R. Internet security threat report, 2015.

[30] Suykens, J. A., and Vandewalle, J. Least squares support vector machine classifiers. *Neural processing letters 9*, 3 (1999), 293–300.

[31] Tamersoy, A., Roundy, K., and Chau, D. H. 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* (2014), ACM, pp. 1524–1533.

[32] Vorobeychik, Y., and Li, B. Optimal randomized classification in adversarial settings. In *International Joint Conference on Autonomous Agents and Multiagent Systems* (2014), pp. 485–492.

[33] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010), vol. 10, p. 10.

[34] Zhang, J., and Zhang, Q. Stackelberg game for utility-based cooperative cognitiveradio networks. In *Proceedings of the tenth ACM international symposium on Mobile ad hoc networking and computing* (2009), ACM, pp. 23–32.