Exploiting an Antivirus Interface

Kevin W. Hamlen[a,*], Vishwath Mohan[a], Mohammad M. Masud[a], Latifur Khan[a], Bhavani Thuraisingham[a]

[a] *Computer Science Department, University of Texas at Dallas, 800 W. Campbell Rd., Richardson, Texas 75080, USA*

Abstract

We propose a technique for defeating signature-based malware detectors by exploiting information disclosed by antivirus interfaces. This information is leveraged to reverse engineer relevant details of the detector's underlying signature database, revealing binary obfuscations that suffice to conceal malware from the detector. Experiments with real malware and antivirus interfaces on Windows operating systems justifies the effectiveness of our approach.

Key words: Security, Signature-based malware detection, Data mining, Binary obfuscation

1. Introduction

Traditional *signature-based* malware detectors identify malware by scanning untrusted binaries for distinguishing byte sequences or *features*. Features unique to malware are maintained in a *signature database*, which must be continually updated as new malware is discovered and analyzed.

Signature-based malware detection generally enforces a static approximation of some desired dynamic (i.e., behavioral) security policy. For example, access control policies, such as those that prohibit code injections into operating system executables, are statically undecidable and can therefore only be approximated by any purely static decision procedure such as signature-matching. A signature-based malware-detector approximates these policies by identifying syntactic features that tend to appear only in binaries that exhibit policy-violating behavior when executed. This approximation is both unsound and incomplete in that it is susceptible to both false positive and false negative classifications of some binaries. For this reason signature databases are typically kept confidential, since they contain information that an attacker could use to craft malware that

*Corresponding author

Email addresses: hamlen@utdallas.edu (Kevin W. Hamlen), vishwath.mohan@utdallas.edu (Vishwath Mohan), mehedy@utdallas.edu (Mohammad M. Masud), lkhan@utdallas.edu (Latifur Khan), bhavani.thuraisingham@utdallas.edu (Bhavani Thuraisingham)

the detector would misclassify as benign, defeating the protection system. The effectiveness of signature-based malware detection thus depends on both the comprehensiveness and confidentiality of the signature database.

Traditionally, signature databases have been manually derived, updated, and disseminated by human experts as new malware appears and is analyzed. However, the escalating rate of new malware appearances and the advent of self-mutating, polymorphic malware over the past decade have made manual signature updating less practical. This has led to the development of automated data mining techniques for malware detection (e.g., [1, 2, 3]) that are capable of automatically inferring signatures for previously unseen malware.

In this article we show how these data mining techniques can also be applied by an attacker to discover ways to obfuscate malicious binaries so that they will be misclassified as benign by the detector. Our approach hinges on the observation that although malware detectors keep their signature databases confidential, all malware detectors reveal one bit of signature information every time they reveal a classification decision. This information can be harvested particularly efficiently when it is disclosed through a public interface. The classification decisions can then be delivered as input to a data mining malware detection algorithm to infer a model of the confidential signature database. From the inferred model we derive feature-removal and feature-insertion obfuscations that preserve the behavior of a given malware binary but cause it to be misclassified as benign. The result is an obfuscation strategy that can defeat any purely static signature-based malware detector.

We demonstrate the effectiveness of this strategy by successfully obfuscating several real malware samples to defeat malware detectors on Windows operating systems. Windows-based antivirus products typically support Microsoft's `IOfficeAntivirus` interface [4], which allows applications to invoke any installed antivirus product on a given binary and respond to the classification decision. Our experiments exploit this interface to obtain confidential signature database information from several commercial antivirus products.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 provides an overview of our approach, Section 4 describes a data mining-based malware detection model, and Section 5 discusses methods of deriving binary obfuscations from a detection model. Section 6 then describes experiments and evaluation of our technique. Section 7 concludes with discussion and suggestions for future work.

2. Related work

Both the creation and the detection of malware that self-modifies to defeat signature-based detectors are well-studied problems in the literature (c.f. [5, 6]). Self-modifying malware has existed at least since the early 1990's and has subsequently become a major obstacle for modern malware protection systems. For example, Kaspersky Labs reported three new major threats in February 2009 that use self-modifying propagation mechanisms to defeat existing malware detection products [7]. Propagation and mutation rates for such malware can

be very high. At the height of the *Feebs* virus outbreak in 2007, Commtouch Research Labs reported that the malware was producing over 11,000 unique variants of itself per day [8].

Most self-modifying malware uses encryption or packing as the primary basis for its modifications. The majority of the binary code in such *polymorphic malware* exists as an encrypted or packed *payload*, which is unencrypted or unpacked at runtime and executed. Signature-based protection systems typically detect polymorphic malware by identifying distinguishing features in the small unencrypted code stub that decrypts the payload (e.g., [9]). More recently, *metamorphic malware* has appeared, which randomly applies binary transformations to its code segment during propagation in order to obfuscate features in the unencrypted portion. An example is the MetaPHOR system (c.f., [10]), which has become the basis for many other metamorphic malware propagation systems. Reversing these obfuscations to obtain reliable feature sets for signature-based detection is the subject of much current research [9, 11, 12], but case studies have shown that current antivirus detection schemes remain vulnerable to simple obfuscation attacks until the detector's signature database is updated to respond to the threat [13].

To our knowledge, all existing self-modifying malware mutates randomly. Our work therefore differs from past approaches in that it proposes an algorithm for choosing obfuscations that target and defeat specific malware defenses. These obfuscations could be inferred and applied fully automatically in the wild, thereby responding to a signature update without requiring re-propagation by the attacker. We argue that simple signature updates are therefore inadequate to defend against such an attack.

Our proposed approach uses technology based on data mining-based malware detectors. Data mining-based approaches analyze the content of an executable and classify it as malware if a certain combination of features are found (or not found) in the executable. These malware detectors are first trained so that they can generalize the distinction between malicious and benign executables, and thus detect future instances of malware. The training process involves feature extraction and model building using these features. Data mining-based malware detectors differ mainly on how the features are extracted and which machine learning technique is used to build the model. The performance of these techniques largely depends on the quality of the features that are extracted.

Schultz et al. [2] extract DLL call information (using *GNU binutils*) and character strings (using *GNU strings*) from the headers of Windows PE executables, as well as 2-byte sequences from the executable content. The DLL calls, strings, and bytes are used as features to train models. Models are trained using two different machine learning techniques—RIPPER [14] and Naïve Bayes (NB) [15]—to compare their relative performances. Kolter et al. [1] extract binary n-gram features from executables and apply them to different classification methods, such as k-nearest neighbor (KNN) [16], NB, Support Vector Machines (SVM) [17], decision trees [18], and boosting [19]. Boosting is applied in combination with various other learning algorithms to obtain improved models (e.g., *boosted decision trees*).

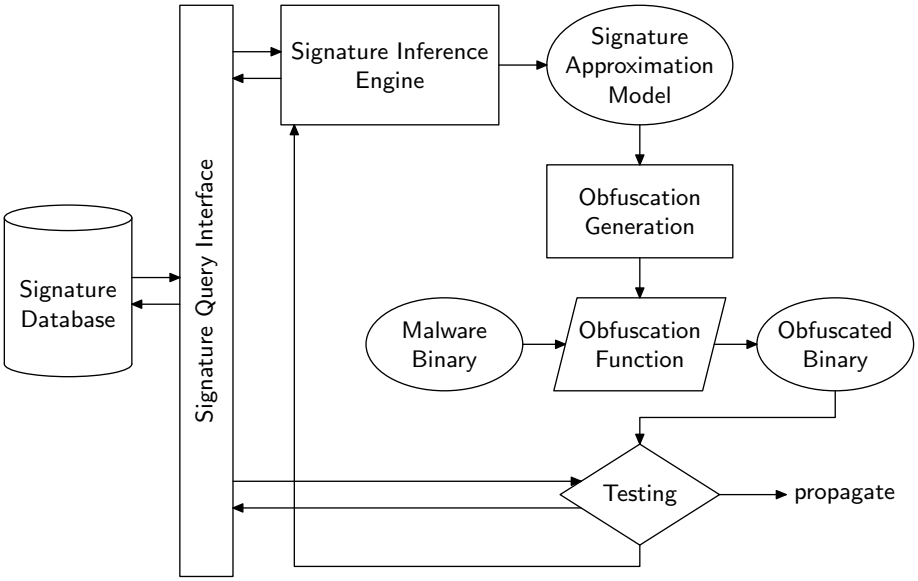

Figure 1: Binary Obfuscation Architecture

Our previous work on data mining based malware detection [3] extracts binary n-grams from the executable, assembly instruction sequences from the disassembled executables, and DLL call information from the program headers. The classification models used in this work are SVM, decision tree, NB, boosted decision tree, and boosted NB. In the following sections we show how this technology can also be applied by an attacker to infer and implement effective attacks against malware detectors using information divulged by antivirus interfaces.

3. Overview

The architecture of our binary obfuscation methodology is illustrated in Figure 1. We begin by submitting a diverse collection of malicious and benign binaries to the victim signature database via the signature query interface. The interface reveals a classification decision for each query. For our experiments we used the `IOfficeAntivirus` COM interface that is provided by Microsoft Windows operating systems (Windows 95 and later) [4]. The `Scan` method exported by this interface takes a filename as input and causes the operating system to use the installed antivirus product to scan the file for malware infections. Once the scan is complete, the method returns a success code indicating whether the file was classified as malicious or benign. This allows applications to request virus scans and respond to the resulting classification decisions.

We then use the original inputs and resulting classification decisions as a training set for an inference engine. The inference engine learns an approximat-

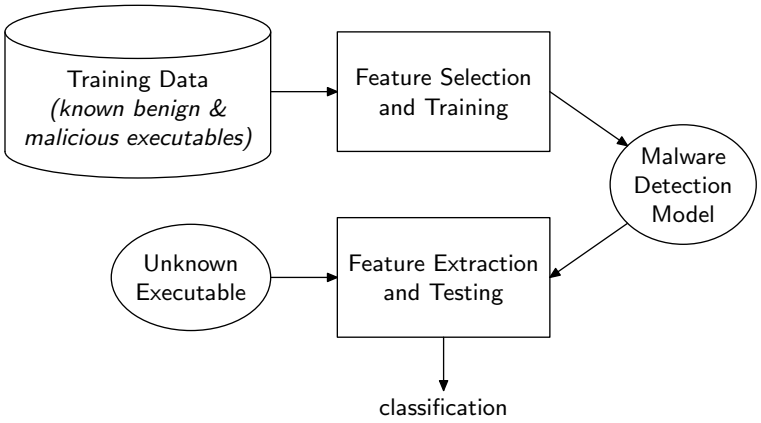

Figure 2: A data mining-based malware detection framework

ing model for the signature database using the training set. In our implementation, this model was expressed as a decision tree in which each node tests for the presence or absence of a specific binary n-gram feature that was inferred to be security-relevant by the data mining algorithm.

This inferred model is then reinterpreted as a recipe for obfuscating malware so as to defeat the model. That is, each path in the decision tree encodes a set of binary features that, when added or removed from a given malware sample, causes the resulting binary to be classified as malicious or benign by the model. The obfuscation problem is thus reduced to finding a binary transformation that, when applied to malware, causes it to match one of the benignly-classified feature sets. In addition, the transformation must not significantly alter the behavior of the malware binary being obfuscated. Currently we identify suitable feature sets by manual inspection, but we believe that future work could automate this process.

Once such a feature set is identified and applied to the malware sample, the resulting obfuscated sample is submitted as a query to the original signature database. A malicious classification indicates that the inferred signature model was not an adequate approximation for the signature database. In this case the obfuscated malware is added to the training set and training continues, resulting in an improved model, whereupon the process repeats. A benign classification indicates a successful attack upon the malware detector. In our experiments we found that repeating the inference process was not necessary; our obfuscations produced misclassified binaries after one round of inference.

4. A data mining based malware detection model

A data mining-based malware detector first trains itself with known instances of malicious and benign executables. Once trained, it can predict the

proper classifications of previously unseen executables by testing them against the model. The high-level framework of such a system is illustrated in Figure 2.

The predictive accuracy of the model depends on the given training data and the learning algorithm (e.g., support vector machine, decision tree, naïve bayes, etc.) Several data mining-based malware detectors have been proposed in the past [1, 2, 3]. The main advantage of these models over the traditional signature-based models is that data mining-based models are more robust to changes in the malware. Signature-based models fail when new malware appears with an unknown signature. On the other hand, data mining-based models generalize the classification process by learning a suitable malware model dynamically over time. Thus, they are capable of detecting malware instances that were not known at the time of training. This makes it more challenging for an attacker to defeat a malware detector based on data mining.

Our previous work on data mining-based malware detection [3] has developed an approach that consists of three main steps:

1. feature extraction, feature selection, and feature-vector computation from the training data,
2. training a classification model using the computed feature-vector, and
3. testing executables with the trained model.

These steps are detailed throughout the remainder of the section.

4.1. Feature extraction

In past work we have extracted three different kinds of features from training instances (i.e., executable binaries):

1. **Binary n-gram features:** In order to extract these features, we consider each executable as a string of bytes and extract all possible n-grams from the executables, where n ranges from 1 to 10.
2. **Assembly n-gram features:** We also disassemble each executable to obtain an assembly language program. We then extract n-grams of assembly instructions.
3. **Dynamic link library (DLL) call features:** Library calls are particularly relevant for distinguishing malicious binaries from benign binaries. We extract the library calls from the disassembly and use them as features.

When deriving obfuscations to defeat existing malware detectors we found that restricting our attention only to binary n-gram features sufficed for our experiments reported in Section 6. However, in future work we intend to apply all three feature sets to produce more robust obfuscation algorithms. Next, we describe how these binary features are extracted.

Binary n-gram feature extraction. First, we apply the UNIX `hexdump` utility to convert the binary executable files into textual *hexdump files*, which contain the hexadecimal numbers corresponding to each byte of the binary. This process is performed to ensure safe and easy portability of the binary executables. The

feature extraction process consists of two phases: (1) feature collection, and (2) feature selection.

The feature collection process proceeds as follows. Let the set of hexdump training files be $\mathcal{H} = \{h_1, ..., h_b\}$. We first initialize a set L of n-grams to empty. Then we scan each hexdump file h_i by sliding an n-byte window over its binary content. Each recovered n-byte sequence is added to L as an n-gram. For each n-gram $g \in L$ we count the total number of positive instances p_g (i.e., malicious executables) and negative instances n_g (i.e., benign executables) that contain g.

There are several implementation issues related to this basic approach. First, the total number of n-grams may be very large. For example, the total number of 10-grams in our dataset is 200 million. It may not possible to store all of them in computer's main memory. Presently we solve this problem by storing the n-grams in a large disk file that is processed via random access. Second, if L is not sorted, then a linear search is required for each scanned n-gram to test whether it is already in L. If N is the total number of n-grams in the dataset, then the time for collecting all the n-grams would be $O(N^2)$, an impractical amount of time when $N = 200$ million. In order to solve the second problem, we use an Adelson-Velsky-Landis (AVL) tree [20] to index the n-grams. An AVL tree is a height-balanced binary search tree. This tree has a property that the absolute difference between the heights of the left sub-tree and the right sub-tree of any node is at most one. If this property is violated during insertion or deletion, a balancing operation is performed, and the tree regains its height-balanced property. It is guaranteed that insertions and deletions are performed in logarithmic time. Inserting an n-gram into the database thus requires only $O(\log_2(N))$ searches. This reduces the total running time to $O(N \log_2(N))$, making the overall running time about 5 million times faster when N as large as 200 million. Our feature collection algorithm implements these two solutions.

Feature selection. If the total number of extracted features is very large, it may not possible to use all of them for training. Aside from memory limitations and impractical computing times, a classifier may become confused with a large number of features because most of them would be noisy, redundant, or irrelevant. It is therefore important to choose a small, relevant and useful subset of features for more efficient and accurate classification. We choose information gain (IG) as the selection criterion because it is recognized in the literature as one of the best criteria isolating relevant features from large feature sets. IG can be defined as a measure of effectiveness of an attribute (i.e., feature) in classifying a training data [21]. If we split the training data based on the values of this attribute, then IG gives the measurement of the expected reduction in entropy after the split. The more an attribute can reduce entropy in the training data, the better the attribute is for classifying the data.

The next problem is to select the best S features (i.e., n-grams) according to IG. One naïve approach is to sort the n-grams in non-increasing order of IG and select the top S of them, which requires $O(N \log_2 N)$ time and $O(N)$ main memory. But this selection can be more efficiently accomplished using a heap that requires $O(N \log_2 S)$ time and $O(S)$ main memory. For $S = 500$ and

$N = 200$ million, this approach is more than 3 times faster and requires 400,000 times less main memory. A heap is a balanced binary tree with the property that the root of any sub-tree contains the minimum (maximum) element in that sub-tree. First we build a min-heap of size S. The min-heap contains the minimum-IG n-gram at its root. Then each n-gram g is compared with the n-gram at the root r. If $IG(g) \leq IG(r)$ then we discard g. Otherwise, r is replaced with g, and the heap is restored.

Feature vector computation. Suppose the set of features selected in the above step is $\mathcal{F} = \{f_1, ..., f_S\}$. For each hexdump file h_i, we build a binary feature vector $h_i(\mathcal{F}) = \{h_i(f_1), ..., h_i(f_S)\}$, where $h_i(f_j) = 1$ if h_i contains feature f_j, or 0 otherwise. The training algorithm of a classifier is supplied with a tuple $(h_i(\mathcal{F}), l(h_i))$ for each training instance h_i, where $h_i(\mathcal{F})$ is the feature vector and $l(h_i)$ is the class label of the instance h_i (i.e., positive or negative).

4.2. Training

We apply SVM, Naïve Bayes (NB), and decision tree (J48) classifiers for the classification task. SVM can perform either linear or non-linear classification. The linear classifier proposed by Vapnik [17] creates a hyperplane that separates the data points into two classes with the maximum margin. A maximum-margin hyperplane is the one that splits the training examples into two subsets such that the distance between the hyperplane and its closest data point(s) is maximized. A non-linear SVM [22] is implemented by applying a kernel trick to maximum-margin hyperplanes. This kernel trick transforms the feature space into a higher dimensional space where the maximum-margin hyperplane is found, through the aid of a kernel function.

A decision tree contains attribute tests at each internal node and a decision at each leaf node. It classifies an instance by performing the attribute tests prescribed by a path from the root to a decision node. Decision trees are rule-based classifiers, allowing us to obtain human-readable classification rules from the tree. J48 is the implementation of the C4.5 Decision Tree algorithm. C4.5 is an extension of the ID3 algorithm invented by Quinlan [18]. In order to train a classifier, we provide the feature vectors along with the class labels of each training instance that we have computed in the previous step.

4.3. Testing

Once a classification model is trained, we can assess its accuracy by comparing its classification of new instances (i.e., executables) to the original victim malware detector's classifications of the same new instances. In order to test an executable h, we first compute the feature vector $h(\mathcal{F})$ corresponding to the executable in the manner described above. When this feature vector is provided to the classification model, the model outputs (predicts) a class label $l(h)$ for the instance. If we know the true class label of h, then we can compare the prediction with the true label, and check the correctness of the learned model. If the model's performance is inadequate, the new instances are added to the training set resulting in an improved model, and testing resumes.

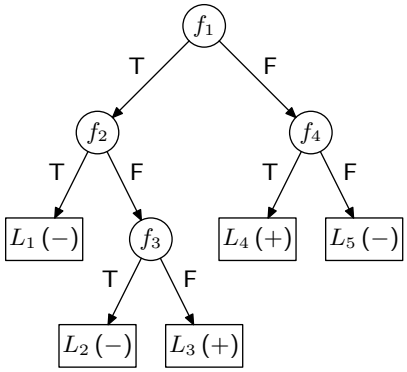

Figure 3: An example of a decision tree-based malware detection model

In the next section, we describe how the model yielded by the above process can be used to derive binary obfuscations that defeat the model.

5. Model-reversing Obfuscations

Malware detectors based on static data-mining attempt to learn correlations between the syntax of untrusted binaries and the (malicious or benign) behavior that those binaries exhibit when executed. This learning process is necessarily unsound or incomplete because most practically useful definitions of "malicious behavior" are Turing-undecidable. Thus, every purely static algorithm for malware detection is vulnerable to false positives, false negatives, or both. Our obfuscator exploits this weakness by discovering false negatives in the model inferred by a static malware detector.

The decision tree model inferred in the previous section can be used as a basis for deriving binary obfuscations that defeat the model. The obfuscation involves adding or removing features (i.e., binary n-grams) to and from the malware binary so that the model classifies the resulting binary as benign. These binary transformations must be carefully crafted so as to avoid altering the runtime behavior of the malware program lest they result in a policy-adherent or non-executable binary.

A simple example will illustrate. Figure 3 shows a simple decision tree model for malware detection. Each internal node in the tree denotes a feature test. For example, the root has feature test f_1. A test instance x (i.e., executable) is first tested against the root. If x has feature f_1, then the detector follows the left branch (True); otherwise it follows the right branch (False). This process continues until the detector arrives at a leaf node. Leaf nodes denote classification decisions, expressed as a minus sign (benign) or plus sign (malicious). For example, instances that cause the detector to arrive in leaf node L_1 will be classified as benign.

According to the model in Figure 3, malware having features f_1 and f_2 will be classified as benign. Likewise, malware lacking both f_1 and f_4 will also be

classified as benign. Thus, an obfuscation that inserts features f_1 and f_2 into a binary, or removes both f_1 and f_4 from the binary, without altering the runtime behavior of the binary, suffices to conceal malware from the detector.

5.1. Path Selection

We begin the obfuscation process by searching for a candidate path through the decision tree that ends in a benign leaf node. Our goal will be to add and remove features from the malicious executable x so as to cause the detector to follow the chosen decision tree path during classification. Since the path ends in a benign-classifying decision node, this will cause the malware to be misclassified as benign by the detector.

Each path from the root to a leaf node in a decision tree can be thought of as a classification rule composed of the conjunction of the conditions encoded by each node in the path. For example, in Figure 3 the path from the root to leaf node L_2 encodes the following rule (R_2):

$$R_2 : f_1 \wedge \neg f_2 \wedge f_3 \implies \textit{benign}$$

which says that if a test instance has feature f_1 and does not have feature f_2, and has feature f_3, then the instance is benign. Here the negated term $\neg f_2$ indicates that this pattern must not be present in the test instance.

While we believe that it is possible in theory to obfuscate any executable binary to satisfy any given rule at classification-time, some rules are significantly easier to realize than others for any given binary. In general, feature removal tends to be more difficult to implement than feature addition. Thus, to minimize the number of feature additions, for each benign-classifying rule in the decision tree we count the number of negated conjunct terms corresponding to features that are present in x. The path with the fewest such terms is likely to be easiest to implement and is therefore chosen as the candidate rule.

Once a candidate rule R is selected, we evaluate each conjunct term of R against binary x. Each such test may either succeed or fail for x. Negated feature tests $\neg f$ may fail because feature f is present in x, and non-negated feature tests f may fail because feature f is not present in x. Let $\mathcal{F}_r$ be the set of features f such that $\neg f$ is a term in rule R and test $\neg f$ fails for x, and let $\mathcal{F}_a$ be the set of features f such that f is a term in rule R and test f fails for x. Adding all features in $\mathcal{F}_a$ to x and removing all features in $\mathcal{F}_r$ from x would cause the resulting binary to satisfy rule R. If these feature-additions and feature-removals can be implemented without changing the runtime behavior of x, then these transformations suffice to successfully obfuscate x so that its malicious behavior is not detected. In the following sections we discuss strategies for implementing feature-removals and feature-additions to successfully obfuscate x in this way.

5.2. Feature Insertion

Inserting new features into executable binaries without significantly altering their runtime behavior tends to be a fairly straightforward task. We discuss

several strategies for adding features to x86 Portable Executable (PE) binary files. While the discussion is specific to PE files, we believe these strategies can also be extended to other binary formats.

The x86 PE binary format is composed of sections of binary data. The starting file offset of each section is linked from a header at the beginning of the file, or from fields within other sections reachable via the header. PE files can therefore be thought of as tree data structures rooted at the file header; the system loader does not typically process them sequentially from beginning to end. Bytes appearing outside of any section are therefore completely ignored by the system loader.

Thus, one easily implementable strategy for transparently inserting a new feature into a PE file is to simply append the feature bytes to the end of the file, or to insert them between existing sections. These bytes will be ignored by the system loader and will not be present in the process memory image when the binary is loaded. They will therefore have no effect upon the runtime behavior of the process.

In our tests reported in Section 6 we found that this simple feature-insertion sufficed to defeat the detectors we tested. However, a more sophisticated detector might limit its feature-selection process to reachable PE sections in order to defeat this attack. To counter this defense, we could have introduced the features to existing sections in one of the following ways:

- *Add the feature to a non-loaded section.* Each section in a PE file includes a flag that specifies whether the section is loaded into memory at runtime. Non-loaded sections typically contain meta-data that is useful for tools such as debuggers but that is not used during execution. Features can therefore be safely added to new or existing non-loaded sections without affecting the program's runtime behavior.

- *Append the feature to a dynamically-sized section.* Some loaded sections, such as the heap, grow at runtime. These sections have two different length specifiers in PE files—one specifies the size of any statically initialized data that is loaded from the PE file into the segment at process start, and the other specifies the amount of memory to allocate (but not initialize) for the entire section at load time.

  Feature data can be safely added to these growable sections by appending it to the statically initialized data for the section. This feature data will be loaded into the section at process start but will be overwritten as the section grows. It will only affect runtime behavior of programs that read uninitialized heap memory before it is allocated. Since no standard memory manager does this, this is a simple and safe way to add features to loaded sections without changing the behavior of most programs.

- *Insert the feature into the code segment as dead (i.e., unreachable) code.* This strategy involves inserting the feature bytes into unused portions of the code segment, possibly by shifting existing code blocks to make room. We discuss this technique in more detail below.

Dead code insertion is more difficult to implement than the other feature insertion strategies, but has the advantage that it is provably undecidable for any purely static detector to reliably isolate these features from the rest of the code. Dead code identification is a well known undecidable problem for any architecture that includes conditional or computed jumps. Thus, the feature will not be safely discardable during feature selection, and will therefore be included in the decision tree by any classifier that depends on a static analysis for feature selection.

Dead code insertion cannot be computably implemented for arbitrary binaries, but it can be implemented as long as a control-flow graph for the binary code is known. This is a reasonable assumption if we assume that the attacker has access to the malware source code. Most existing compilers for x86 architectures already insert small blocks of static data or padding between methods using this information, and assemblers have directives for doing this as well. Dead code can be safely inserted using either of these established techniques.

5.3. Feature Removal

Removal of a feature from an executable binary is more difficult to implement without changing the program's runtime behavior. There are two major techniques for doing this implemented by existing malware:

- encryption (polymorphic malware), and
- code mutation (metamorphic malware).

Polymorphic malware encrypts the majority of its code and data using a random key. This payload is then decrypted at runtime and executed. The payload can be re-encrypted using a different key during propagation, creating many syntactically different but functionally identical variants.

Features found in the encrypted payload cyphertext of a polymorphic virus or worm can typically be removed simply by choosing a different encryption key. With a large enough key space and a sufficiently diverse collection of cyphertexts, the probability of finding a cyphertext that includes none of the disallowed features can be raised arbitrarily high. Non-polymorphic malware can be made polymorphic by wrapping it in a polymorphic propagation system.

Thus, polymorphic malware propagation reduces the feature space available to a detector to the two remaining file portions that cannot be encrypted: the file meta-data, and the decryption kernel. File meta-data cannot be encrypted because the system loader must be able to parse it in order for the binary to be executable. The decryption kernel that decrypts the payload cannot be encrypted without introducing a new decryption kernel, hence reintroducing the issue.

Although PE meta-data cannot be safely encrypted, the meta-data of typical x86 PE files can easily be crafted to be identical to that of known benign executables. A detector that does not reject important benign programs cannot, therefore, reliably distinguish malicious instances from non-malicious instances

based on features found in this standard meta-data. Thus, we henceforth limit our attention to removing features from the decryption kernel.

The decryption kernel of a polymorphic worm is a relatively small code stub at or near the program entrypoint that decrypts the encrypted payload and then branches to the newly decrypted code. Removing a feature from the decryption kernel requires replacing it with a functionally equivalent byte sequence. Metamorphic malware engines achieve this by randomly applying a set of known code equivalence transformations to the decryption code to produce syntactically different but functionally identical code. One of the simplest such transformations is to randomly insert `nop` (no-operation) instructions between various instructions, which will be ignored by the processor at runtime.

While `nop`-insertion is an effective feature-removal strategy for some malware detectors, more sophisticated detectors can defend against such an attack by disregarding all `nop` instructions during feature selection.[1] To defeat such a detector, we could have resorted to a more powerful metamorphic obfuscator such as the MetaPHOR system (c.f., [10]). MetaPHOR disassembles x86 binary code to a simplified intermediate language in which common sequences of instructions are expressed as single operations. It then re-assembles new x86 binary code from the intermediate representation pseudo-randomly. That is, for any given intermediate code there exist many possible equivalent x86 instruction sequences, which are chosen randomly to create a syntactically different but functionally equivalent instruction sequence.

While polymorphism and metamorphism are powerful existing techniques for obfuscating malware against signature-based detectors, it should be noted that existing polymorphic and metamorphic malware mutates randomly. Our attack therefore differs from these existing approaches in that we choose obfuscations that are derived directly from signature database information leaked by the malware detector being attacked. Our work therefore builds upon this past work by showing how antivirus interfaces can be exploited to choose an effective obfuscation, which can then be implemented using these existing techniques.

6. Experiments

To test our approach, we conducted two sets of experiments. In the first experiment we attempted to collect classification data from several commercial antivirus products by querying their public interfaces automatically. In the second experiment we obfuscated a malware sample in order to defeat the data mining-based malware detector we developed in past work [3], and that is described in Section 4. In future work we intend to combine these two results to test fully automatic obfuscation attacks upon commercial antivirus products.

[1] Reliably identifying `nop` instructions in arbitrary x86 binary code is not decidable in general due to the non-aligned nature of the instruction set and the resulting instruction sequence aliasing decision problems. However, a feature-selector based on n-grams could simply disregard all 0x90 bytes (the `nop` op-code) to defeat `nop`-insertion at the expense of losing a marginal amount of decision information.

6.1. Dataset

We have two non-disjoint datasets. The first dataset (dataset1) contains a collection of 1,435 executables, 597 of which are benign and 838 are malicious. The second dataset (dataset2) contains 2,452 executables, having 1,370 benign and 1,082 malicious executables. The distribution of dataset1 is hence 41.6% benign and 58.4% malicious, and that of dataset2 is 55.9% benign and 44.1% malicious. This distribution was chosen intentionally to evaluate the performance of the feature sets in different scenarios. We collect the benign executables from different Windows XP, and Windows 2000 machines, and collect the malicious executables from VX Heavens [23], which contains a large collection of malicious executables. The benign executables contain various applications found in the Windows system folder (e.g. `C:\Windows`), as well as other executables drawn from the default program installation directory (e.g., `C:\Program Files`) of various machines. Malicious executables contain viruses, worms, trojans, and back-doors. We select only the Win32 Portable Executables (PE) in both the cases. We would like to experiment with other executable formats (e.g., ELF) in the future.

6.2. Interface Exploit Experiment

To test the feasibility of collecting confidential signature database information via the antivirus interface on Windows operating systems, we wrote a small utility that queries the `IOfficeAntivirus` [4] COM interface on Windows XP and Vista machines. The utility uses this interface to request virus scans of instances in dataset1. We tested our utility on four commercial antivirus products: Norton Antivirus 2009, McAfee VirusScan Plus, AVG 8.0, and Avast Antivirus 2009.

In all but Avast Antivirus we found that we were able to reliably sample the signature database using the interface. Our utility required no elevated privileges to successfully harvest this data on Windows XP and Windows Vista systems. Benign classifications had no observable effect (other than to solicit the appropriate return code from the interface), while malicious classifications had the side-effect of quarantining the executable named in the query. The quarantining process typically involved GUI activity (e.g., a pop-up window warning the user) and file activity (e.g., moving the file to a safe location), which slowed down the detection process slightly. However, we found that this activity did not prevent ongoing scan requests. On average we were able to obtain classification decisions at a rate of 2 MB/sec (4.6 files per second) on a 2Ghz Windows Vista desktop machine with a standard 5400 RPM SATA harddrive. We therefore expect that a large amount of classification data could be gathered in this way fairly easily on victim systems.

In the case of Avast Antivirus 2009 we found that the return code yielded by the interface was not meaningful—it did not distinguish between different classifications. Thus, Avast Antivirus 2009 was not vulnerable to our attack. However, in Section 7 we discuss possible methods of circumventing this limitation that could be implemented in future work.

6.3. Model-driven Obfuscation Experiment

We next used the techniques described in Section 5 to obfuscate a malicious executable so as to conceal it from a data mining-based malware detector [3]. To train the classifier we used the two datasets described above. Each dataset has different sizes and distributions of benign and malicious executables. We select the classification-relevant binary n-gram features using the techniques explained in Section 4. Then we build decision tree classifiers using the selected feature sets. Our implementation is developed in Java with JDK 1.5. We use Weka ML toolbox [24] for training the decision tree classifier (the C4.5 algorithm).

In order to evaluate our technique on malware obfuscation, we chose to obfuscate the *Win32.Navidad.a* virus using our technique. Our malware detection model M sucessfully classified this as malware. In order to defeat the model, the malware was obfuscated via the following steps:

1. Generate the binary feature vector corresponding to the malware x using our technique described in Section 4.3. Let the feature vector be $\mathcal{F}(x) = \{f_1(x), ..., f_n(x)\}$, where $f_i(x)$ is either 0 or 1 depending on whether the feature (i.e., n-gram) f_i is absent or present in x.
2. Analyze the decision tree model M and identify a candidate rule R as described in Section 5.1.
3. Identify the features that must be inserted or remove to satisfy R.
4. Insert and/or remove the necessary features to/from the malware using a hexadecimal editor.

In the case of *Win32.Navidad.a*, we only needed to insert features in order to successfully defeat the model. The resulting obfuscated binary was misclassified by the detector as benign. We also verified through informal testing that the obfuscated malware still had identical functionality to the original malware.

Although we performed this obfuscation manually, we believe it would be fairly easy to fully automate this process. One obvious approach would be to apply existing polymorphic and metamorphic malware obfuscation engines such as MetaPHOR (c.f., [10]) to feature-containing portions of the malware using a succession of randomly chosen cryptographic keys and seed values until the unwanted features are removed. We intend to investigate such an approach in future work.

7. Conclusion

In this paper we have outlined a technique whereby antivirus interfaces that reveal classification decisions can be exploited to infer confidential information about the underlying signature database. These classification decisions can be used as training inputs to data mining-based malware detectors. Such detectors will learn an approximating model for the signature database that can be used as a basis for deriving binary obfuscations that defeat the signature database. We conjecture that this technique could be used as the basis for effective, fully automatic, and targeted attacks against signature-based antivirus products.

Our experiments justify this conjecture by demonstrating that classification decisions can be reliably harvested from several commercial antivirus products on Windows operating systems by exploiting the Windows public antivirus interface. We also demonstrated that effective obfuscations can be derived for real malware from an inferred model by successfully obfuscating a real malware sample using our model-reversing obfuscation technique. The obfuscated malware defeated the detector from which the model was derived.

Our signature database inference procedure was not an effective attack against one commercial antivirus product we tested because that product did not fully support the antivirus interface. In particular, it returned the same result code irrespective of its classification decision for the submitted binary file. However, we believe this limitation could be overcome by an attacker in at least two different ways:

First, although the return code did not divulge classification decisions, the product did display observably different responses to malicious binaries, such as opening a quarantine pop-up window. These responses could have been automatically detected by our query engine. Determining classification decisions in this way is a slower but still fully automatic process.

Second, many commercial antivirus products also exist as freely distributed, stand-alone utilities that scan for (but do not necessarily disinfect) malware based on the same signature databases used in the retail product. These lightweight scanners are typically implemented as Java applets or ActiveX controls so that they are web-streamable and executable at low privilege levels. Such applets could be executed in a restricted virtual machine environment to effectively create a suitable query interface for the signature database. The execution environment would provide a limited view of the filesystem to the victim applet and would infer classification decisions by monitoring decision-specific system calls, such as those that display windows and dialogue boxes.

From the work summarized in this article, we conclude that effectively concealing antivirus signature database information from an attacker is important but difficult. Current antivirus interfaces such as the one currently supported by Windows operating systems invite signature information leaks and subsequent obfuscation attacks. Antivirus products that fail to support these interfaces are less vulnerable to these attacks, however they still divulge confidential signature database information through covert channels, such as graphical responses and other side-effects.

Fully protecting against these confidentiality violations might not be feasible; however there are some obvious steps that defenders can take to make these attacks more computationally expensive for the attacker. One obvious step is to avoid implementing or supporting interfaces that divulge classification decisions explicitly and on-demand through return codes. While this prevents benign applications from deteting and responding to malware quarantines, this reduction in functionality seems reasonable in the (hopefully uncommon) context of a malware attack. Protecting against signature information leaks through covert channels is a more challenging problem. Addressing it effectively might require leveraging anti-piracy technologies that examine the current execution environ-

ment and refuse to divulge classification decisions in restrictive environments that might be controlled by an attacker. Without such protection, attackers will continue to be able to craft effective, targeted binary obfuscations that defeat existing signature-based malware detection models.

References

[1] J. Z. Kolter, M. A. Maloof, Learning to Detect Malicious Executables in the Wild, in: Proc. of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 470–478, 2004.

[2] M. G. Schultz, E. Eskin, E. Zadok, S. J. Stolfo, Data Mining Methods for Detection of New Malicious Executables, in: Proc. of the IEEE Symposium on Security and Privacy, 38, 2001.

[3] M. M. Masud, L. Khan, B. M. Thuraisingham, A Scalable Multi-level Feature Extraction Technique to Detect Malicious Executables, Information System Frontiers 10 (1) (2008) 33–35.

[4] Microsoft Developer Network (MSDN) Digital Library, `http://msdn.microsoft.com/en-us/library/ms537369(VS.85).aspx`, 2009.

[5] C. Nachenberg, Computer Virus-antivirus Coevolution, Communications of the ACM 40 (1) (1997) 47–51.

[6] P. Ször, The Art of Computer Virus Research and Defense, Addison-Wesley Professional, 2005.

[7] Kaspersky Labs, Monthly Malware Statistics, `http://www.kaspersky.com/news?id=207575761`, 2009.

[8] Commtouch Research Labs, Q1 Malware Trends Report: Server-Side Malware Explodes Across Email, Whitepaper, Alt-N Technologies, Grapevine, Texas, 2007.

[9] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, G. Vigna, Polymorphic Worm Detection Using Structural Information of Executables, in: Proc. of the 8th Symposium on Recent Advances in Intrusion Detection (RAID), 207–226, 2005.

[10] A. Walenstein, R. Mathur, M. R. Chouchane, A. Lakhotia, Normalizing Metamorphic Malware Using Term Rewriting, in: Proc. of the 6th IEEE Workshop on Source Code Analysis and Manipulation (SCAM), 75–84, 2006.

[11] D. Brushi, L. Martignoni, M. Monga, Code Normalization for Self-Mutating Malware, in: Proc. of the IEEE Symposium on Security and Privacy, vol. 5 (2), 46–54, 2007.

[12] A. Walenstein, R. Mathur, M. R. Couchane, A. Lakhotia, Normalizing Metamorphic Malware Using Term Rewriting, in: Proc. of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), 75–84, 2006.

[13] M. Christodorescu, S. Jha, Testing Malware Detectors, in: Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 34–44, 2004.

[14] W. W. Cohen, Learning Rules that Classify E-mail, in: Papers from the AAAI Spring Symposium on Machine Learning in Information Access, 18–25, 1996.

[15] D. Michie, D. J. Spiegelhalter, C. C. Taylor (Eds.), Machine Learning, Neural and Statistical Classification, chap. 5: Machine Learning of Rules and Trees, Ellis Horwood Series in Artificial Intelligence, Morgan Kaufmann, 50–83, 1994.

[16] D. W. Aha, D. Kibler, M. K. Albert, Instance-based Learning Algorithms, Machine Learning 6 (1991) 37–66.

[17] B. E. Boser, I. M. Guyon, V. N. Vapnik, A Training Algorithm for Optimal Margin Classifiers, in: Proc. of the 5th ACM Workshop on Computational Learning Theory, 144–152, 1992.

[18] J. R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann, San Francisco, CA, 5th edn., 2003.

[19] Y. Freund, R. E. Schapire, Experiments with a New Boosting Algorithm, in: Proc. of the 13th International Conference on Machine Learning, 148–156, 1996.

[20] M. T. Goodrich, R. Tamassia, Data Structures and Algorithms in Java, Wiley, New York, 4th edn., 2005.

[21] T. M. Mitchell, Machine Learning, McGraw-Hill, New York, 1997.

[22] C. Cortes, V. Vapnik, Support-vector networks, Machine Learning 20 (3) (1995) 273–297.

[23] VX Heavens, `http://vx.netlux.org`, 2009.

[24] I. H. Witten, E. Frank, Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann, San Francisco, CA, 2nd edn., 2005.