A Framework for Empirical Evaluation of Malware Detection Resilience Against Behavior Obfuscation

Sebastian Banescu Tobias Wüchner Aleieldin Salem
Marius Guggenmos Martín Ochoa Alexander Pretschner
Technische Universität München, Germany
Email:{banescu, wuechner, salem, guggenmos, ochoa, pretschn}@cs.tum.edu

Abstract

Program obfuscation is increasingly popular among malware creators. Objectively comparing different malware detection approaches with respect to their resilience against obfuscation is challenging. To the best of our knowledge, there is no common empirical framework for evaluating the resilience of malware detection approaches w.r.t. behavior obfuscation. We propose and implement such a framework, called FEEBO that obfuscates the observable behavior of malware binaries targeting Microsoft Windows operating systems. To assess the framework's utility, we use it to obfuscate known malware binaries and then investigate the impact on detection effectiveness of different popular behavior based malware detection approaches. We find that the obfuscation transformations employed by FEEBO can affect the accuracy of such detection approaches significantly. FEEBO is therefore an effective and fair way to test the degree of resilience of behavior-based malware detection approaches against behavior obfuscation.

1 Introduction

Industry and academia continuously devise countermeasures against the thread of malware in form of advanced detection approaches. However, malware developers are often several steps ahead the state of the art. Most commercial antivirus software in principle continues to be some form of signature-based analysis on the persistent representation of potential malware. Not surprisingly, almost all modern malware families employ some means to confuse and hamper signature-based approaches. Such countermeasures range from simple techniques (e.g. code encryption), to more sophisticated techniques (e.g. control-flow obfuscation) [3].

Given control-flow obfuscation (e.g. packers), commonly used in malware, one intuitively appropriate detection strategy is so-called behavioral detection. The idea is to look at the malware's observable behavior rather than its static code. *Observable behavior* or simply *behavior* of a malware instance, refers to the sequence of system calls (syscalls) executed by that instance. Behavioral detection approaches are barely affected by control-flow obfuscation techniques, because they change the malware's static structure but rarely its behavior. Although, behavioral detection techniques compensate the effects of (build-time) control-flow obfuscation to a large extent, they are often vulnerable to more advanced (run-time) behavior obfuscation techniques that change the observable behavior of malware. Examples for such behavior obfuscation techniques include the injection of bogus syscalls or the deliberate randomized re-ordering of call execution sequences.

While control-flow obfuscation of malware and respective countermeasures at the detection side have been well researched [3], the effects of *behavior* obfuscation on the effectiveness of detection approaches so far only received very little attention in the literature. Behavior obfuscation in itself has been discussed from a theoretical perspective by Dalla Preda *et al.* [4], but we are not aware of any empirical investigations of its effects on real-world malware.

To provide a foundation for such empirical evaluations, we propose a behavior obfuscation framework which we call FEEBO. Provided an arbitrary malware sample as input, it applies a set of behavior obfuscation transformations to its observable behavior. This makes it possible to add behavior obfuscation to malware samples in a structured and targeted way, regardless of whether or not the malware sample given as input contains any form of obfuscation itself. Furthermore, FEEBO enables a basis for controlled experiments, where we can simulate different obfuscation transformations with different degrees and precisely investigate their impact on certain detection approaches.

Contributions: **a)** To our best knowledge, we are the first to provide an open source empirical malware behavior obfuscation framework that is able to behaviorally obfuscate standard malware binaries. **b)** With FEEBO we establish a basis for reproducible behavioral obfuscation resilience experiments. **c)** Our evaluation reveals the sensitivity of several popular behavior-based detection approaches [15] towards certain behavioral obfuscation transformations implemented in FEEBO.

Organization: We introduce *behavior obfuscation* and discuss several n-gram-based behavioral detection techniques in §2. Then we describe the design and implementation of FEEBO in §3. We show the effectiveness of a prototypical implementation of FEEBO in §4 and discuss its limitations in §5. We discuss possible application areas of our approach and give an outlook on future work in §6.

2 Related Work

Control-flow obfuscation applies transformations at the source code or intermediate representation levels in order to make a program harder to understand by a human or an automated analysis engine. Collberg [3] describes such transformations including: virtualization obfuscation, bogus code insertion via opaque predicates, function splitting, and control-flow flattening. These transformation will typically not have an effect on the observable behavior of a program. Therefore, in this work we do not employ control-flow obfuscation.

Behavior obfuscation requires changing the observable behavior of the program being obfuscated. This paper is related to work of Péchoux and Ta [8] on malware behavior obfuscation. They divide the behavior, i.e. executed operations of a program (e.g. malware) into (i) internal computations and (ii) syscalls. *Internal computations* operate only on the process memory of the corresponding program and only affect and are affected by the information stored inside this process' memory. *System calls* represent interactions with the operating system (OS) kernel, i.e. there is a transfer of control from the corresponding program to the kernel and back. Therefore, syscalls affect and are affected by the information stored anywhere in OS memory.

Péchoux and Ta [8] show that it is possible to obfuscate the behavior of known malware samples such that the original malware functionality is preserved by: (i) inserting syscalls before and/or after syscalls in the observable behavior, (ii) reordering syscalls in the observable behavior and (iii) substitution of syscalls by other syscalls which maintain the original functionality of the malware. Different from our work, their goal is to obtain a sequence of syscalls (trace) that is similar to a goodware trace in order to perform *mimicry attacks* [14]. We, on the other hand, focus on automatically generating behaviorally obfuscated malware via randomized syscall insertion, reordering and substitution, to assess the resilience of behavioral detection approaches which analyze the syscalls executed by malware. We plan to extend FEEBO with mimicry transformations in future work.

3 Approach for Behavior Obfuscation

Statically transforming (obfuscating) x86 binary programs without debugging symbols is a non-trivial task which involves binary rewriting [10]. This task becomes significantly more challenging when the binaries to transform are malware binaries, which employ packing and anti-disassembly techniques [5]. However, for the purpose of evaluating the resilience of behavior-based malware detection approaches we do not necessarily need a self-contained obfuscated malware binary. We only need to record the observable behavior of a behaviorally obfuscated malware sample. Therefore, this paper takes an alternative approach to static binary rewriting, by obfuscating using runtime binary instrumentation [11].

In a nutshell runtime binary instrumentation does not need to perform x86 disassembly and enables the interception of any syscall executed by the input binary program. One can choose to execute, delay, drop or even swap the intercepted syscall with a different syscall, plus perform other additional instructions including executing more syscalls. Therefore, it is a suitable technique for obfuscating the observable behavior of malware binaries.

The remainder of this section describes the behavior obfuscation transformations implemented by FEEBO: (i) syscall *insertion*, (ii) syscall *reordering*, and (iii) *substitution* of syscalls with functionally equivalent ones. The intuition behind these transformation is that they should hamper behavior-based detection approaches which rely on patterns in sequences of syscalls.

3.1 System Call Insertion

This obfuscation transformation changes the observable behavior of the input program given as input to FEEBO, by inserting a random number of syscalls in various locations of the original executed sequence of syscalls. With a given probability p_i, syscall insertion adds for each syscall executed by the input program, a number of additional syscalls randomly chosen from the previously executed syscalls. The number of inserted syscalls per intercepted syscall is randomly chosen between two input parameters of FEEBO called: min_i and max_i.

To prevent these inserted calls from changing the original functionality of the input program, we modify the values of their parameters in case the syscalls belong to a set of syscalls (denoted S in the rest of this paper) that have side-effects. Therefore, the set S contains syscalls which involve: creating, moving, writing to or deleting: files, pipes or thread message queues, communicating over a socket, loading a library, allocating, writing to or freeing process memory, creating windows and device contexts. The values of the changed parameters are chosen such that they will not collide with existing data, e.g., files. Moreover, we do not reuse previously observed file handles since that may cause crashes or changes in functionality. Instead, we create new handles for different randomly generated file-names. Moreover, syscalls that access a unique system resource are excluded. For instance, inserting a syscall that sets clipboard data, would imply inserting a second call to restore the clipboard data since the clipboard is unique on each system.

For example, if we configure FEEBO with the following input parameters: $p_i = 0.25$, $min_i = 2$ and $max_i = 5$, then every syscall executed by the input program is intercepted with a probability of 25%. For every syscall that is intercepted FEEBO inserts x syscalls after the execution of the intercepted syscall, where x is a number between $min_i = 2$ and $max_i = 5$ chosen uniformly at random.

3.2 System Call Reordering

System call reordering can naïvely be implemented by delaying a sequence of syscalls in a buffer which is randomly permuted before execution. This approach would likely break the functionality of transformed programs or cause a crash. Intuitively, this would frequently occur when delaying syscalls that read data (e.g. from files), because such data is likely used immediately after it was read.

In FEEBO reordering is implemented such that every syscall in the set S (described in §3.1) executed by the input program, is delayed with a probability p_r and placed in a queue (of size n) for later execution. The reason only calls in S were delayed is that calls outside S generally read information, which programs need to continue their proper execution. Moreover, we use a queue for the delayed syscalls, because we want to preserve the original ordering of syscalls that have side effects like first creating and then writing to a file. Once the queue is full, our tool will execute them in their original order. Each of the delayed syscalls can additionally trigger the insertion of additional syscalls with similar parameters as described in §3.1, i.e. the probability of inserting syscalls when executing a delayed syscall is denoted p_{ri} and the minimum and maximum number of inserted syscalls, are denoted by min_{ri}, respectively max_{ri}.

For example, if we configure FEEBO with the following input parameters: $p_r = 0.5$, $n = 5$, $p_{ri} = 0.75$, $min_{ri} = 1$ and $max_{ri} = 2$, every syscall from S made by the input program is delayed with a probability of 50%. Once $n = 5$ calls have been delayed, they will be executed. Each of the delayed syscalls has a 75% probability to insert between $min_{ri} = 1$ and $max_{ri} = 2$ other syscalls.

We successfully tested this obfuscation transformation on a few Windows utilities such as Paint and Notepad. However, after testing the transformation on a larger set of programs including real-world malware, we noticed it caused a large number of crashes (see §4).

```
{"readfile":[
   {"syntax1":  [ "ReadFile" ] },
   {"syntax2":  [ "ReadFileEx" ] },
   {"syntax3":  [ "LockFile", "ReadFile", "UnlockFile" ] },
   {"syntax4":  [ "LockFile", "ReadFileEx", "UnlockFile" ] },
   {"syntax5":  [ "GetFileInformationByHandle", "ReadFileEx" ] },
   {"syntax6":  [ "GetFileAttributes", "ReadFileEx" ] },
   {"syntax7":  [ "GetFileSize", "ReadFileEx" ] },
   {"syntax8":  [ "GetFileType", "ReadFileEx" ] },
   {"syntax9":  [ "CancelIo", "ReadFileEx" ] },
   {"syntax10": [ "CancelIo", "ReadFile" ] },
   {"syntax11": [ "EncryptFile", "DecryptFile", "ReadFileEx" ] },
   {"syntax12": [ "CancelIo", "CreateSymbolicLink", "OpenFile", "ReadFileEx
       ", "DeleteFile" ] },
   {"syntax13": [ "CancelIo", "CreateSymbolicLink", "OpenFile", "ReadFile",
        "DeleteFile" ] }
],
"writefile":[
   {"syntax1":  [ "WriteFile" ] },
   {"syntax2":  [ "WriteFileEx" ] },
   {"syntax3":  [ "LockFile", "WriteFile", "UnlockFile" ] },
   {"syntax4":  [ "LockFile", "WriteFileEx", "UnlockFile" ] },
   {"syntax5":  [ "GetFileInformationByHandle", "WriteFileEx" ] },
   {"syntax6":  [ "GetFileAttributes", "WriteFileEx" ] },
   {"syntax7":  [ "GetFileSize", "WriteFileEx" ] },
   {"syntax8":  [ "GetFileType", "WriteFileEx" ] },
   {"syntax9":  [ "CancelIo", "WriteFileEx" ] },
   {"syntax10": [ "CancelIo", "WriteFile" ] },
   {"syntax11": [ "EncryptFile", "DecryptFile", "ReadFileEx" ] },
   {"syntax12": [ "CancelIo", "CreateSymbolicLink", "OpenFile", "
       WriteFileEx", "DeleteFile" ] },
   {"syntax13": [ "CancelIo", "CreateSymbolicLink", "OpenFile", "WriteFile"
       , "DeleteFile" ] }
]}
```

Figure 1: FEEBO Equivalence Classes

3.3 System Call Substitution

This obfuscation transformation is implemented by intercepting a sequence of syscalls and substituting their execution with that of a sequence of different syscalls, such that the resulting functionality of the program is functionally equivalent to the original one. This substitution occurs with a given probability p_s. The sets of functionality equivalent syscalls are grouped together in an *equivalence class*. Whenever FEEBO intercepts a sequence of syscalls from an equivalence class, it substitutes this sequence with another sequence (chosen randomly) from the same equivalence class.

An equivalence class is specified as a configuration file in JSON format, therefore, it can be extended without source code modifications to FEEBO. In this work we have developed equivalence classes for reading and writing files shown in Figure 1 as `readfile`, respectively `writefile`. Each of these equivalence classes contains 13 equivalent sequences of syscalls indexed by the keyword `syntax` and followed by a list of Microsoft Windows syscall names. We have created these equivalence classes manually by looking-up the description of each of the syscalls and mapping their parameters such that any two entries from an equivalence class are functionally equivalent. An example sequence from the `writefile` equivalence class is `syntax3` which first locks a file, then writes to it and finally unlocks it. The `readfile` equivalence class contains similar rules. These syntax rules are functionally equivalent under the assumption that the obfuscated malware samples are not multi-threaded, which we believe is the case for the dataset used in our experiments [7].

4 Evaluation

The evaluation of FEEBO is divided into two parts. Firstly, §4.1 presents the feasibility of the implemented obfuscation transformations, i.e. we check whether the behaviorally obfuscated malware instances crash for specific input parameter configurations of FEEBO. Secondly, in §4.2 we only use the obfuscation transformations which were found to be stable (i.e. produce a low number of crashes) during the feasibility evaluation. We measure the impact that various input parameter configurations of FEEBO have on the accuracy of a classifier used by behavior-based malware detection approaches.

4.1 Feasibility Evaluation

To check which of the obfuscation transformations implemented in FEEBO reliably produce running malware samples, we first randomly selected a subset of 100 malware samples from the Malicia dataset [7], spread across 16 malware families.

We used FEEBO to generate 300 obfuscated versions (variants) of each of the 100 malware instances from our chosen subset, totaling 30,000 log files from executions of behaviorally obfuscated malware instances. To obtain this data we executed each of the 100 malware samples separately within an installation of the Cuckoo malware analysis sandbox[1], where we replaced the behavior monitor with FEEBO to obtain a variety of obfuscated behavior logs of those malware samples. To capture a critical mass of syscalls sufficiently large to allow training a classifier with good accuracy, we need to monitor a malware sample for at least 3 minutes. With one input parameter configuration capturing the obfuscated logs of 100 malware samples would then take 300 minutes which, with help of parallel execution of multiple VMs on 5 cores, we could cut down to about one hour per input configuration.

The first 10,000 logs were obtained by iterating over 100 different input parameter configurations for FEEBO, where we only applied syscall insertion. The following 20 values were used for the insertion probability $p_i \in \{0.05, 0.10, 0.15, \cdots, 1.00\}$. For each of these probability values we iterated over the following values for the maximum number of inserted syscalls $max_i \in \{1, 2, 4, 8, 16\}$. The minimum number of inserted syscalls was fixed to 1, i.e. $min_i = 1$.

The next 10,000 logs were obtained by iterating over 100 different input configurations for FEEBO, where we applied syscall reordering. The following 5 values were used for the reordering probability $p_i \in \{0.20, 0.40, 0.60, 0.80, 1.00\}$. For each of these probability values we iterated over the following values for the size of the queue $n \in \{1, 2, 4, 8\}$. For each different value of the queue size (n), we iterated over the following values for the maximum number of inserted syscalls for each delayed call $max_{ri} \in \{1, 2, 4, 8, 16\}$. The minimum number of inserted syscalls was fixed to 1, i.e. $min_{ri} = 1$.

The final 10,000 logs were obtained by iterating over 100 different input parameter configurations for FEEBO, where we only applied semantic substitution of equivalent syscalls. There 100 input configurations were obtained by iterating over the following values for the system call substitution probability $p_s \in \{0.01, 0.02, 0.03, \cdots, 1.00\}$.

We automatically inspected these 30,000 log files to see which ones resulted in crashes of the instrumented malware binary. Our results showed that only 3% of the logs corresponding to syscall insertion involved a crash, which we believe is negligible since these crashes occurred for random malware samples. For syscall reordering around 54% of logs involved crashes. We noticed that a large proportion of crashes occur for larger values of the queue size n, which means that the more syscalls are delayed the higher the probability of a crash. For semantic substitution of equivalent syscalls around 14% of the logs involved crashes, which we find to be a tolerable number given the purpose of behavioral obfuscation of malware binaries. Due to these results, we decided to only use syscall insertion and semantic substitution in our effectiveness evaluation from §4.2.

4.2 Effectiveness Evaluation

From the results of §4.1 we concluded that syscall insertion and substitution would produce a tolerable number of crashes (<15%) on the remaining malware instances from the Malicia dataset [7]. Therefore, we implemented a FEEBO-simulator for those two obfuscation transformations, but which operated directly on recorded execution logs of the unobfuscated malware samples from the Malicia dataset. The rationale behind implementing the FEEBO-simulator is that generating 200 variants for each of the 11,688 malware samples from the malicia dataset would have taken $200 \times 11,688/100 = 23376$ hours (about $2\frac{1}{2}$ years) on our evaluation setup described in §4.1, while with the FEEBO-simulator it takes under a week of computation. Note that by construction, the two approaches (simulation and instrumentation) produce outputs with the same probability distribution for identical input configurations, under the hypothesis that the variants wouldn't crash.

Using the FEEBO-simulator we were able to generate 200 variants for each of the 11,688 samples of the Malicia dataset, totaling 2,337,600 logs. Half of these logs correspond to the 100 input configurations for syscall insertion, while the other half correspond to the 100 input configurations for semantic substitution described in §4.1. In addition, we collected the logs of 730 known goodware samples

[1]http://www.cuckoosandbox.org/.

(including programs shipped together with Microsoft Windows 7), using our Cuckoo sandbox, which we did not obfuscate, to use as comparison baseline for later training the detection classifiers.

Measuring the Actual Degree of Obfuscation Since the input configuration of both FEEBO and FEEBO-simulator are specified as probabilities and ranges of numbers of syscalls to insert, we cannot use these values to measure the actual degree of obfuscation between the original malware instance and the variant obtained from FEEBO. Therefore, as a measure of the actual degree of obfuscation, we calculated the *Levenshtein distance* between the original and the obfuscated logs, as it represents the number of atomic insertion, deletion, and substitution operations needed to transform one event log into the other. For computing the Levenshtein distance we abstracted our logs to only the name of the syscalls (excluding their parameters), which are elements of the Latin alphabet.

Classifier Used for Malware Detection There is a large body of work regarding behavior-based malware detection approaches [2, 12, 13, 9]. In this work we use the insights given by Canali *et al.* [2], which indicate that simple n-gram approaches perform poorly with respect to variations in syscall logs. Therefore we chose to use support vector machines (SVMs) for our evaluation, which is a more advanced classifier, often employed by state of the art malware detection approaches [12, 13, 9]. SVMs can only process numerical feature vectors. In order to extract numerical features from our syscall logs, we made use of the spectrum kernel [6] which enables us to classify our test logs in linear time. The spectrum kernel is originally meant to detect homology, i.e. the existence of common ancestry, between two protein sequences. We can match this setting to ours by considering the samples from the Malicia dataset as the ancestors of the behaviorally obfuscated malware versions. The objective of our experiment is to investigate whether the SVM can still associate the obfuscated versions of the malware instances with their ancestors by classifying them correctly. If our experiments show that FEEBO affects the accuracy of this classifier, then we are confident that it is a suitable tool for the evaluation of other behavior-based detection approaches.

Classifier Accuracy Baseline Given a number $k \geq 1$, the spectrum kernel computes the frequency of occurrence of all subsequences of length k in the original sequence. Therefore, we want to select proper parameter configurations for the SVM classifiers, i.e. ensure they have good accuracy and detection rate. To do this, we consider different values of $k = \{3, 6, 9\}$. To also ensure that the classifiers do not over-fit, we perform 10-fold cross-validation (10-FCV) using all 11,688 malware samples from the Malicia dataset.

Table 1: Average accuracy of classifiers for different subsequence lengths (k) after 10-FCV on the Malicia dataset

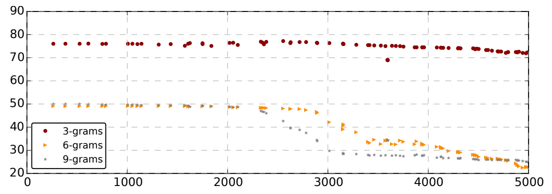
	$k = 3$	$k = 6$	$k = 9$
SVM-alpha accuracy	80.36%	40.49%	41.37%
SVM-raw accuracy	90.03%	53.98%	38.29%

We use two types of input data for our SVM classifier. For the first type of input data we convert the syscall logs into alphabetical character sequences, similar to protein sequences, by mapping every syscall from an execution log to one letter of the alphabet. The second type of input data we consider the entire raw log of syscalls including parameter values. We call the results obtained from the SVM with the first and the second types of input data: *SVM-alpha*, respectively *SVM-raw*, throughout the rest of this paper.

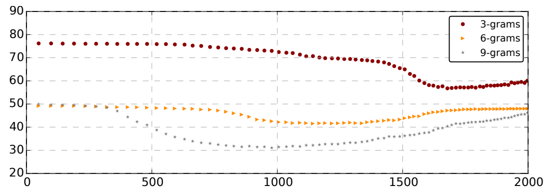
The detection rates scored by the SVM with the spectrum kernel on the non-behaviorally obfuscated Malicia dataset can be seen on the first row of Table 1 for the alphabetical sequences as input data and on the second row of Table 1 for the raw logs as input data. The accuracy of both classifiers degrades as the length of the subsequences (k) increases. However, we notice that the accuracy of SVM-raw for $k = 3$ is a bit above 90%, which is 10% higher than the accuracy of SVM-alpha for $k = 3$. Also for $k = 6$ the accuracy of SVM-raw is higher than that of SVM-alpha. However, for $k = 9$ they are almost the same. In the experiments presented later in this section we want to see if and how these accuracy values are affected by the behavior obfuscation transformations employed by FEEBO.

Classifier Training Time Training the SVM-raw classifier is very time-consuming, because of the character-wise way the spectrum kernel analyzes the raw logs [6]. We noticed that the time needed for training the SVM-raw classifier on the entire Malicia dataset is around 3.5 hours for each fold of the 10-FCV on our experiment setup, whereas training the SVM-alpha classifier was more than one order of magnitude faster. We explain this by the difference in size between the SVM-alpha and the SVM-raw input data sets that also is of about one order of magnitude.

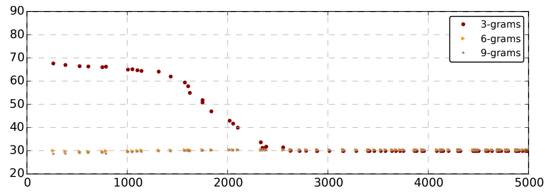
To reduce the time needed for training, we sampled 5 times 1000 log samples recorded using the Malicia dataset. Trained each of these sets separately using the same classifiers yielded a training time of about 17 minutes per 1000 log sample. Furthermore, we compared the accuracy of the 10-FCV of the same SVM classifier on the entire set of logs from the Malicia dataset and on the five sets of 1000 malware log samples. We noticed that the accuracy only differs by at most 2% for all the 5 sets of 1000 malware log sam-



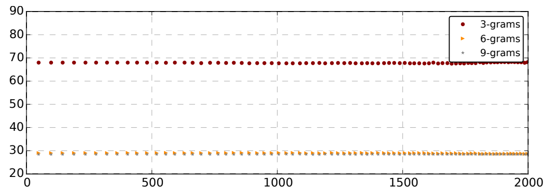
(a) SVM-alpha for syscall insertion



(b) SVM-alpha for semantic substitution



(c) SVM-raw for syscall insertion



(d) SVM-raw for semantic substitution

Figure 2: Training with original malware (x-axis: degree of obfuscation, y-axis: classifier accuracy)

ples. Therefore, in the experiments described in the following paragraph, we used one of these 1000 log samples to obtain an evaluation set with a balanced malware and goodware distribution.

Training only with Original Malware In this experiment we used logs recorded for a period of 3 minutes for all samples from the Malicia dataset and our 730 goodware samples. To obtain balanced malware and the goodware sets to avoid classification bias, we then sampled 1000 logs from the 11,688 logs generated from the Malicia dataset. These 1000 logs yielded files containing between 64.2 KBs to 5.3 MBs of recorded syscalls. The 730 logs generated by the goodware samples ranged from 94.3 KBs to 3.6 MBs.

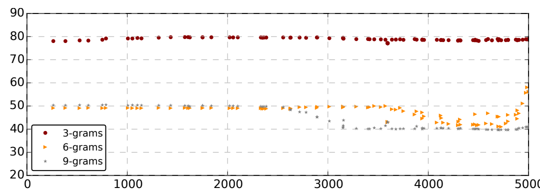
We trained the SVM-alpha and the SVM-raw classifiers with the 1000 logs of malware and the 730 logs of goodware. Afterward, we obfuscated the 1000 logs with the 100 input parameter configurations of FEEBO-simulator for syscall insertion and the 100 input configurations for semantic substitution which were presented in §4.1. These log files along with 200 different input configurations were then given to the FEEBO-simulator. This resulted in 200,000 obfuscated log files, grouped in 200 folders containing 1000 instances each, labeled with the input parameter configuration of FEEBO used to generate the respective logs.

We then used the SVM-alpha and SVM-raw classifiers on each of these 200 folders. For each folders we recorded the average degree of obfuscation (Levenshtein distance) and the average classifier accuracy. The results of this experiment for SVM-alpha on the 100 folders obfuscated with different input parameter configurations of FEEBO involving only syscall insertion can be seen in Figure 2a. From this figure we can clearly see that the accuracy of the SVM-alpha classifier for all values of $k = \{3, 6, 9\}$ decrease inversely proportional to the degree of obfuscation by behavioral obfuscation via syscall insertion. For $k = 3$ the accuracy for an obfuscation degree of 5000 dropped to 72%, while for $k = \{6, 9\}$ the accuracy dropped under 25%.

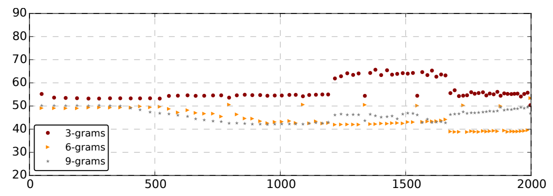
The results for SVM-alpha on the 100 folders obfuscated with different input parameter configurations of FEEBO involving only semantic substitution can be seen in Figure 2b. This figure also shows a decreasing trend of the accuracy starting from obfuscation degree 0 up to the obfuscation degrees around 1500, 1000 and 800 for values of k: 3, 6, respectively 9. Also, we can see an increasing trend that does not go higher than the initial accuracy. The reason for the increasing trend is related to the spectrum kernel [6], used by the SVM classifier. For lower degrees of obfuscation there are only a few semantic substitutions to create confusion and lower accuracy. However, after a certain degree of obfuscation the classifier maps the frequent occurrences of the small set of possible syntax rules within an equivalence class (see Figure 1) to each other and thus can easily factor out the obfuscation effects.

The results for SVM-raw on the 100 folders obfuscated with different input parameter configurations of FEEBO involving only syscall insertion can be seen in Figure 2c. It is apparent that the accuracies of all the classifiers are affected by syscall insertion. However, only for $k = 3$ we see a significant decrease in accuracy around obfuscation degree 2000. Afterward, the accuracy level stabilizes.

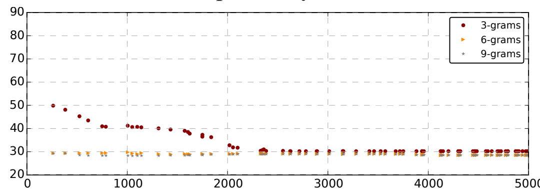
The results for SVM-raw on the 100 folders obfuscated with different input parameter configurations of FEEBO involving only semantic substitution can be seen in Figure 2d. At first glance it seems like the accuracies remain constant independent of the applied obfuscation degree. However, we can see a similar phenomenon to that observed in Figure 2b, i.e. the detection accuracy decreases slightly and then comes back up to its original value. In sum we can conclude that SVM-raw classifiers seem to be much more resilient to semantic substitution than SVM-alpha ones.



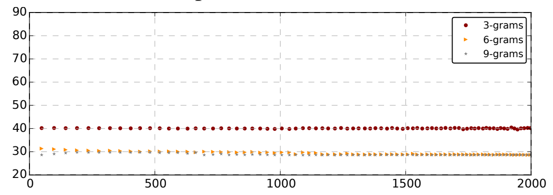
(a) SVM-alpha for syscall insertion



(b) SVM-alpha for semantic substitution



(c) SVM-raw for syscall insertion



(d) SVM-raw for semantic substitution

Figure 3: Training with both original and obfuscated malware (x-axis: degree of obfuscation, y-axis: classifier accuracy)

Training with both Original and Behaviorally Obfuscated Malware In this experiment we used the same logs as in our previous experiment. However, instead of training only on logs of non-behaviorally obfuscated malware samples, we mixed the 730 goodware samples, the 1000 non-obfuscated malware samples and 1000 behaviorally obfuscated samples corresponding to one input parameter configuration of FEEBO-simulator. Hence, for each of the 200 different input configurations of the FEEBO-simulator, we performed 10-FCV using the SVM-alpha and SVM-raw classifiers. The accuracy of the classifiers decreases, because adding the 1000 obfuscated malware samples creates an imbalance between the goodware and the malware sets.

Through this experiment we reason about classifier stability which captures the situation in which a classifier is trained with logs from behaviorally obfuscated malware samples. We measure the stability of a classifier as the standard deviation of accuracy points in a certain interval of obfuscation degrees.

In the ideal case the applied obfuscations not having any effect on the externally visible behavior, the detection rate should remain 100%. With this setting we could investigate the effects of the applied obfuscation transformations with respect to detection accuracy. Figure 3a shows the results of this experiment for SVM-alpha on the 100 folders obfuscated with different configurations of syscall insertion. For $k = 3$ this classifier is slightly affected by syscall insertion. However, for $k = \{6, 9\}$ the standard deviation of the SVM-alpha classifier increases directly proportional to the obfuscation degree.

Figure 3b shows the results of this experiment for SVM-alpha on the 100 folders obfuscated with different configurations of semantic substitution. For this obfuscation transformation we see that the standard deviation of the accuracy grows significantly around obfuscation degree 1200 for $k = 3$, it grows less for $k = 6$ and even less for $k = 9$.

Figure 3c shows the results of this experiment for SVM-raw on the 100 folders obfuscated with different configurations of syscall insertion. Figure 3d shows the results of this experiment for SVM-raw on the 100 folders obfuscated with different configurations of semantic substitution. These last two figures show that the standard deviation of the SVM-raw classifier is barely affected by both syscall insertion and semantic substitution. Thus, we conclude that SVM-raw classifiers are more stable w.r.t. obfuscation than SVM-alpha ones, but with lower overall detection accuracy.

5 Discussion and Threats to Validity

In §4.2 we learned that the proposed obfuscation transformations can have a significant effect on the detection accuracy of the SVM-alpha and SVM-raw approaches. In Figures 2 we can see a clear correlation between the increase of obfuscation degree and the detection accuracy of the two classifiers for certain values of k and certain obfuscation transformations. In Figure 3 we can see, that the spread in classification accuracy, i.e. the standard deviation of the accuracy, increases with the obfuscation degree. Furthermore, we noticed that syscall insertion has a higher impact on detection accuracy than semantic substitution which we explain by the fact that the respective transformations are more diverse and thus harder to factor out by learning.

The functionality of any obfuscated program should include the functionality of the original (non-obfuscated) program. For many software transformation engines such as optimizing compilers, this is a strict requirement. However, even very widely used compilers such as GCC or Clang have been found to contain optimizations that break the functionality of the original source code [16]. The transformations described in §6 suffer from the same issue, i.e. they may change the functionality of malware such that it becomes ineffective. We however argue that the primary goal of malware obfuscation is hampering detection by anti-malware software at almost any cost. Therefore, the small risk of the obfuscation engine producing ineffective samples is acceptable in most cases. We thus, in particular from a pragmatic perspective, consider our results valuable given that checking for behavioral equality in general is undecidable (cf. Rice's theorem) and in our experiments few of the obfuscated malware samples crashed during execution.

In sum, we can draw the following conclusions from our experiments: a) FEEBO is able to effectively obfuscate the behavior of real-world malware with significant impact on the effectiveness of behavioral detection approaches; b) FEEBO can be used to test the resilience of various malware detection approaches with various input configurations against different forms of behavior obfuscation.

6 Conclusions and Future Work

We have introduced FEEBO, a framework to conduct empirical experiments on the effects of behavior obfuscation on behavior-based malware detection approaches. To this extent we developed a prototype that can apply certain obfuscation transformations to the observable behavior of malware samples. To evaluate the effectiveness of the implemented obfuscation transformations and of our approach in general, we investigated the effects of a wide range of behavior obfuscation transformations on the detection capabilities of an SVM classifier with different input types. We showed that FEEBO is an effective tool for analyzing the resilience of the SVM classifier with various input parameters, against different types of behavior obfuscation.

In future work we plan to improve the implementation of system call reordering and also extend the semantic substitution transformation with more equivalence classes and syntax rules. We release FEEBO [1] to parties from academia and industry. For ethical reasons we only provide a version not capable of generating self-contained obfuscated binaries to avoid misuse by malware developers.

References

[1] S. Banescu, T. Wüchner, A. Salem, M. Guggenmos, M. Ochoa, and A. Pretschner. FEEBO. `https://www22.in.tum.de/tools/feebo/`.

[2] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. A quantitative study of accuracy in system call-based malware detection. In *ISSTA '12*, pages 122–132. ACM, 2012.

[3] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2009.

[4] M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. *ACM TOPLAS*, 30(5):1–54, 2008.

[5] C. Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.

[6] C. S. Leslie, E. Eskin, and W. S. Noble. The spectrum kernel: A string kernel for svm protein classification. In *Pacific Symposium on Biocomputing*, 2002.

[7] A. Nappa, M. Z. Rafique, and J. Caballero. Driving in the Cloud: An Analysis of Drive-by Download Operations and Abuse Reporting. In *Proc. of DIMVA*, 2013.

[8] R. Péchoux and T. D. Ta. A categorical treatment of malicious behavioral obfuscation. In *Theory and Applications of Models of Computation*, pages 280–299. Springer, 2014.

[9] J. Pfoh, C. Schneider, and C. Eckert. Leveraging string kernels for malware detection. In *Network and System Security*, LNCS volume 7873, pages 206–219. 2013.

[10] M. Prasad and T.-c. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX*, pages 211–224, 2003.

[11] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. Pin: a binary instrumentation tool for computer architecture research and education. In *Proc. of Workshop on Computer architecture education*, page 22, 2004.

[12] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *Proc. of DIMVA*, 2008.

[13] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *Inf. Secur. Tech. Rep.*, 14(1):16–29, 2009.

[14] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proc. of CCS*, pages 255–264. ACM, 2002.

[15] C. Wressnegger, G. Schwenk, D. Arp, and K. Rieck. A close look on n-grams in intrusion detection: anomaly detection vs. classification. In *Workshop on Artificial intelligence and security*, pages 67–76, 2013.

[16] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.