A Comparison of Static, Dynamic, and Hybrid Analysis for Malware Detection

Anusha Damodaran* Fabio Di Troia† Visaggio Aaron Corrado†
Thomas H. Austin* Mark Stamp*‡

Abstract

In this research, we compare malware detection techniques based on static, dynamic, and hybrid analysis. Specifically, we train Hidden Markov Models (HMMs) on both static and dynamic feature sets and compare the resulting detection rates over a substantial number of malware families. We also consider hybrid cases, where dynamic analysis is used in the training phase, with static techniques used in the detection phase, and vice versa. In our experiments, a fully dynamic approach generally yields the best detection rates. We discuss the implications of this research for malware detection based on hybrid techniques.

1 Introduction

According to Symantec [44], more than 317 million new pieces of malware were created in 2014, which represents a 26% increase over 2013. Given numbers such as these, malware detection is clearly a significant and worthwhile research topic.

In practice, the most widely used malware detection method is signature scanning, which relies on pattern matching. While signature scanning is effective for many types of malware, it is ineffective for detecting new malware, or even significant variants of existing malware [5].

A wide array of advanced detection techniques have been considered in the literature. Some detection techniques rely only on static analysis [3, 4, 6, 11, 15, 16, 26, 38, 39, 40, 42], that is, features that can be obtained without executing the software. In addition, dynamic analysis has been successfully applied to the malware detection problem [1, 2, 12, 19, 21, 27, 30, 31, 32, 33, 50]. Recently, hybrid approaches have been analyzed, where both static and dynamic features are used [10, 20].

Here, we compare static analysis with dynamic analysis, and also consider hybrid schemes that combine elements of both. We use a straightforward training and scoring

*Department of Computer Science, San Jose State University
†Department of Engineering, Università degli Studi del Sannio
‡stamp@cs.sjsu.edu

technique based on Hidden Markov Models and we consider feature sets consisting of API call sequences and opcode sequences.

In this research, our goal is to gain some level of understanding of the relative advantages and disadvantages of static, dynamic, and hybrid techniques. In particular, we would like to determine whether there is any inherent advantage to a hybrid approach. Note that our goal here is not to optimize the detection accuracy, which would likely require combining a variety of scores and scoring techniques. Instead, we conduct our analysis in a relatively simple setting, by which we hope to reduce the number of potentially confounding variables that tend to appear in more highly optimized systems.

The remainder of this paper is organized as follows. In Section 2, we discuss relevant background information, including related work. Section 3 discusses the experiments conducted and the datasets used. In Section 4, we present our experimental results. The paper concludes with Section 5, where we also mention possible future work.

2 Background

In this section, we first provide a brief discussion of malware detection techniques, with an emphasis on Hidden Markov Models, which are the basis for the research presented in this paper. We also review relevant related work. Finally, we discuss ROC curves, which give us a convenient means to quantify the various experiments that we have conducted.

2.1 Malware Detection

There are many approaches to the malware detection problem. Here, we briefly consider signature-based, behavior-based, and statistical-based detection, before turning our attention to a slightly more detailed discussion of HMMs.

2.1.1 Signature Based Detection

Signature based detection is the most widely used anti-virus technique [5]. A signature is a sequence of bytes that can be used to identify specific malware. A variety of pattern matching schemes are used to scan for signatures [5]. Signature based anti-virus software must maintain a repository of signatures of known malware and such a repository must be updated frequently as new threats are discovered.

Signature based detection is simple, relatively fast, and effective against most common types malware. A drawback of signature detection is that it requires an up-to-date signature database—malware not present in the database will not be detected. Also, relatively simple obfuscation techniques can be used to evade signature detection [7].

2.1.2 Behavior Based Detection

Behavior based detection focuses on the actions performed by the malware during execution. In behavior based systems, the behavior of the malware and benign files are analyzed during a training (learning) phase. Then during a testing (monitoring) phase,

an executable is classified as either malware or benign, based on patterns derived in the training phase [25].

2.1.3 Statistical Based Detection

Malware detection can be based on statistical properties derived from program features. For example, in [49], Hidden Markov Models (HMMs) are used to classify metamorphic malware. This technique has served a benchmark in a variety of other studies [6, 35, 39, 45]. Consequently, we use HMMs as the basis for the malware detection schemes considered in this research.

2.2 Hidden Markov Models

A Hidden Markov Model can be viewed as a machine learning technique, based on a discrete hill climb [43]. Applications of HMMs are many and varied, ranging from speech recognition to applications in computational molecular biology, to artificial intelligence, to malware detection [23].

As the name suggests, a Hidden Markov Model includes a Markov process that cannot be directly observed. In an HMM, we have a series of observations that are related to the "hidden" Markov process by a set of discrete probability distributions.

We use the following notation for an HMM [43]:

$$
\begin{aligned}
T &= \text{length of the observation sequence} \\
N &= \text{number of states in the model} \\
M &= \text{number of observation symbols} \\
Q &= \{q_0, q_1, \ldots, q_{N-1}\} = \text{distinct states of the Markov process} \\
V &= \{0, 1, \ldots, M-1\} = \text{set of possible observations} \\
A &= \text{state transition probabilities} \\
B &= \text{observation probability matrix} \\
\pi &= \text{initial state distribution} \\
\mathcal{O} &= (\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1}) = \text{observation sequence.}
\end{aligned}
$$

A generic Hidden Markov Model is illustrated in Figure 1.

A Hidden Markov Model is defined by the matrices π, A and B, and hence we denote an HMM as $\lambda = (A, B, \pi)$. For simplicity, we often refer to λ simply as a "model".

The practical utility of HMMs derives from the fact that there are efficient algorithms to solve each of the following three problems [43].

Problem 1: Given a model $\lambda = (A, B, \pi)$ and an observation sequence $\mathcal{O}$, determine $P(\mathcal{O} \mid \lambda)$. That is, we can score a given observation sequence against a given model—the better the score, the more closely the observation sequence matches the observations used to train the model.

Problem 2: Given a model $\lambda = (A, B, \pi)$ and an observation sequence $\mathcal{O}$, determine the optimal state sequence X. That is, we can uncover the "best" hidden state sequence. Here, "best" is in the sense of maximizing the expected number of

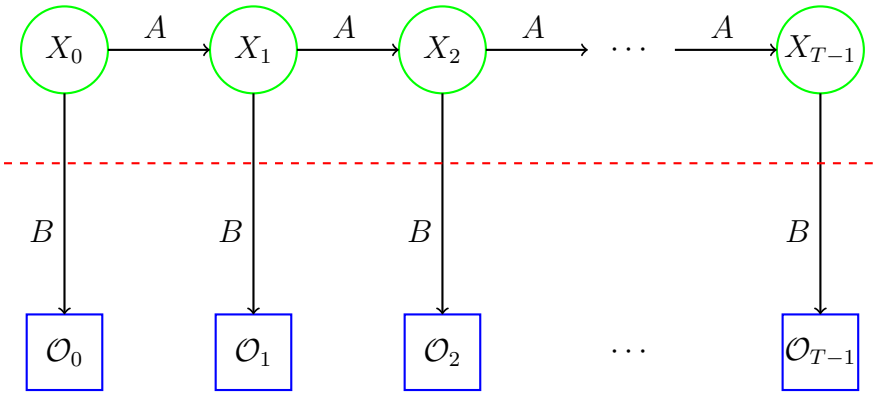

Figure 1: Generic Hidden Markov Model

correct states X_i. This is in contrast to a dynamic program, which yields the X_i corresponding to the highest scoring path.

Problem 3: Given an observation sequence $\mathcal{O}$ and parameters N and M, determine the model $\lambda = (A, B, \pi)$ such that $P(\mathcal{O} \mid \lambda)$ is maximized. That is, we can train a model to fit a given observation sequence $\mathcal{O}$.

In this research, we use the solution to Problem 3 to train a model based on observation sequences extracted from a given malware family. Then we use the solution to Problem 1 to score observation sequences extracted from malware files as well as sequences extracted from benign files. We use the resulting scores to measure the success of each technique.

Details on the HMM algorithms are beyond the scope of this paper. For a thorough discussion of the solutions to HMM Problems 1 through 3, see [43]; for additional information see [23] or the classic introduction [34]. For the application of HMMs to malware detection, see, for example, [49].

2.3 Related Work

Here we discuss some relevant examples of previous work. We group the previous work based on whether it relies on static analysis or dynamic analysis, and we discuss techniques that employ a hybrid approach.

2.3.1 Static Analysis

Static analysis of software is performed without actually executing the program [18]. Examples of the information we can obtain from static analysis include opcode sequences (extracted by disassembling the binary file), control flow graphs, and so on. Such feature sets can be used individually or in combination for malware detection.

In [11], the authors presented a malware detection technique that relies on static analysis and is based on control flow graphs. Their approach focuses on detecting obfuscation patterns in malware and they are able to achieve good accuracy.

Machine learning techniques have been applied to malware detection in the context of static detection. In [49], Hidden Markov Models are used to effectively classify metamorphic malware, based on extracted opcode sequences. A similar analysis involving Profile Hidden Markov Models is considered in [4], while Principal Component Analysis is used in [26] and [16], and Support Vector Machines are used for malware detection in [40]. The paper [3] employs clustering, based on features derived from static analysis, for malware classification.

In [15], function call graph analysis is used for malware detection, while [39] analyzes an opcode-based similarity measure that relies on simple substitution cryptanalysis techniques. API call sequences and opcode sequences are both used in [38] to determine whether a segment of code has similarity to some particular malware.

The papers [6, 28] analyze file structure based on entropy variations. The work in these paper was inspired by the entropy-based score in [42].

2.3.2 Dynamic Analysis

Dynamic analysis requires that we execute the program, often in a virtual environment [18]. Examples of information that can be obtained by dynamic analysis include API calls, system calls, instruction traces, registry changes, memory writes, and so on.

In [27], the authors build fine-grained models that are designed to capture the behavior of malware based on system calls. The resulting behavior models are represented in the form of graphs, where the vertices denote system calls and the edges denote dependency between the calls.

The paper [1] presents a run-time monitoring tool that extracts statistical features based on spatio-temporal information in API call logs. The spatial information consists of the arguments and return values of the API calls, while the temporal information is the sequencing of the API calls. This information is used to build formal models that are fed into standard machine learning algorithms, for use in malware detection.

In [19], a set of program API calls is extracted and combined with control flow graphs to obtain a so-called API-CFG model. In a slightly modified version [21], n-gram methods are applied to the API calls.

Some recent work focuses on kernel execution traces as a means of developing a malware behavior monitor [33]. In [30], the authors present an effective method for malware classification using graph algorithms, which relies on dynamically extracted information. The related work [31] constructs a "kernel object behavioral graph" and graph isomorphism techniques are used for scoring.

API sequences are again used for malware detection in [50]. Also in the paper [32], malware is analyzed based on frequency analysis of API call sequences.

In [12], dynamic instruction sequences are logged and converted into abstract assembly blocks. Data mining algorithms are used to build a classification model that relies on feature vectors extracted from this data.

The authors of [2] propose a malware detection technique that uses instruction trace logs of executables, where this information is collected dynamically. These traces are then analyzed as graphs, where the instructions are the nodes, and statistics from instruction traces are used to calculate transition probabilities. Support Vector Machines are used to determine the actual classification.

2.3.3 Hybrid Approaches

Hybrid techniques combine aspects of both static and dynamic analysis. In this section, we discuss two recent examples of work of this type.

In [10], the authors propose a framework for classification of malware using both static and dynamic analysis. They define features of malware using an approach that they call Malware DNA (Mal-DNA). The heart of this technique is a debugging-based behavior monitor and analyzer that extracts dynamic characteristics.

In the paper [20], the authors develop and analyze a tool that they call HDM Analyser. This tool uses both static analysis and dynamic analysis in the training phase, but performs only static analysis in the testing phase. The goal is to take advantage of the supposedly superior fidelity of dynamic analysis in the training phase, while maintaining the efficiency advantage of static detection in the scoring phase. For comparison, it is shown that HDM Analyser has better overall accuracy and time complexity than the static or dynamic analysis methods in [20]. The dynamic analysis in [20] is based on extracted API call sequences.

Next, we discuss ROC analysis. We use the area under the ROC curve as one of our measures of success for the experiments reported in Section 4.

2.4 ROC Analysis

A Receiver Operating Characteristic (ROC) curve is obtained by plotting the false positive rate against the true positive rate as the threshold varies through the range of data values. An Area Under the ROC Curve (AUC-ROC) of 1.0 implies ideal detection, that is, there exists a threshold for which no false positives or false negatives occur. The AUC-ROC can be interpreted as the probability that a randomly selected positive instance scores higher than a randomly selected negative instance [8, 22]. Therefore, an AUC-ROC of 0.5 means that the binary classifier is no better than flipping a coin. Also, an AUC-ROC that is less than 0.5 implies that we can obtain a classifier with an AUC-ROC greater than 0.5 by simply reversing the classification criteria.

An examples of a scatterplot and the corresponding ROC curve is given in Figure 2. The red circles in the scatterplot represent positive instances, while the blue squares represent negative instances. In the context of malware classification, the red circles are scores for malware files, while the blue squares represent scores for benign files. Furthermore, we assume that higher scores are "better", that is, for this particular score, positive instances are supposed to score higher than negative instances.

For a given experiment, the true positive rate is also known as the sensitivity, while the true negative rate is referred to as the specificity. Then the false positive rate is

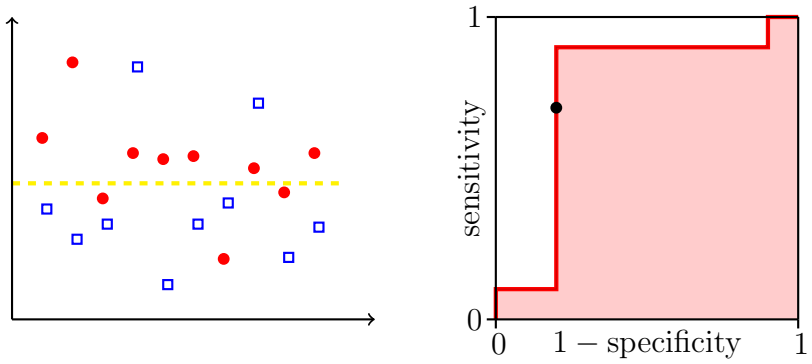

Figure 2: Scatterplot and ROC Curve

given by $1 - \text{specificity}$.

Note that if we place the threshold below the lowest point in the scatterplot in Figure 2, then

$$\text{sensitivity} = 1 \quad \text{and} \quad 1 - \text{specificity} = 1$$

On the other hand, if we place the threshold above the highest point, then

$$\text{sensitivity} = 0 \quad \text{and} \quad 1 - \text{specificity} = 0$$

Consequently, an ROC curve must always include the points $(0, 0)$ and $(1, 1)$. The intermediate points on the ROC curve are determined as the threshold passes through the range of values. For example, if we place the threshold at the yellow dashed line in the scatterplot in Figure 2, the true positive rate (i.e., sensitivity) is 0.7, since 7 of the 10 positive instances are classified correctly, while the false positive rate (i.e., $1 - \text{specificity}$) is 0.2, since 2 of the 10 negative cases lie on the wrong side of the threshold. This implies that the point $(0.2, 0.7)$ lies on the ROC curve. The point $(0.2, 0.7)$ is illustrated by the black circle on the ROC graph in Figure 2. The shaded region in Figure 2 represents the AUC. In this example, we find that the AUC-ROC is 0.75.

2.5 PR Analysis

Precision Recall (PR) curves offer an alternative to ROC analysis for scatterplot data [14]. There are many connections between PR curves and ROC curves,[1] but in certain cases, PR curves can be more informative. In particular, when the nomatch set is large relative to the match set, PR curves may be preferred.

We define recall to be the fraction of the match cases that are classified correctly, and precision is the fraction of elements classified as positive that actually belong to the match set. More precisely,

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad \text{and} \quad \text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

[1] For example, if one curve dominates another in ROC space, it also dominates in PR space, and vice versa.

where TP is the number of true positives, FP is the number of false positives, and FN is the number of false negatives. Note that recall is the true positive rate, which we referred to as sensitivity in our discussion of ROC curves. However, precision is not the same as the false positive rate which is used to compute ROC curves. Also note that TN does not appear in the formula for recall or precision, and hence true negatives play no (direct) role in computing the PR curve. Again, this may be useful if we want to focus our attention on the positive set, particularly when we have a relatively large negative set. As with ROC analysis, we can use the Area Under the PR Curve (AUC-PR) as a measure of the success of a classifier.

To generate the PR curve, we plot the (recall, precision) pairs as the threshold varies through the range of values in a given scatterplot. To illustrate the process, we consider the same data as in the ROC curve example in Section 2.4. This data and the corresponding PR curve is given here in Figure 3.

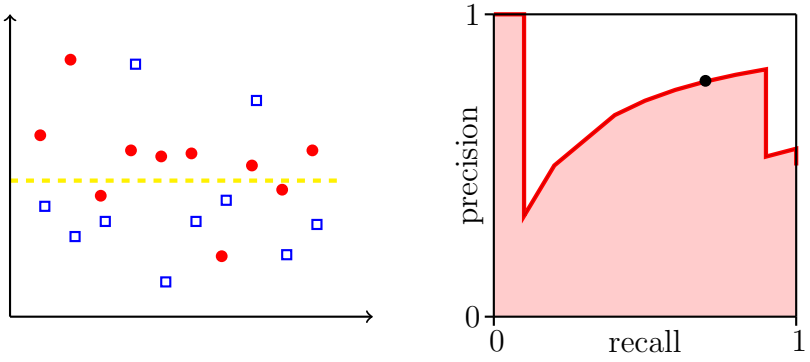

Figure 3: Scatterplot and PR Curve

For the threshold that appears in the scatterplot in Figure 3, we have TP = 7, FP = 2, and FN = 3, and hence

$$\text{recall} = \frac{7}{7+3} = 0.7 \quad \text{and} \quad \text{precision} = \frac{7}{7+2} \approx 0.78$$

This point is plotted on the right-hand side of Figure 3 and the entire PR curve is given. In this example, the AUC-PR is about 0.69.

3 Experiments

In this section, we first discuss the tools that we use to extract features from code—both statically and dynamically. Then we give an overview of the malware dataset used in this project. Finally, we elaborate on the experiments conducted. In Section 4, we provide the results of our experiments.

3.1 Tools for Dynamic and Static Analysis

IDA Pro is a disassembler that generates highly accurate assembly code from an executable. It can also be used as a debugger. IDA is a powerful tool that supports scripting, function tracing, instruction tracing, instruction logging, etc. In this research, we use IDA Pro for static analysis, specifically, to generate `.asm` files from `.exe` files, from which opcodes and windows API calls can be extracted. We also use IDA Pro for dynamic analysis, specifically, to collect instruction traces from executables.

In addition to IDA Pro, for dynamic analysis we use the Buster Sandbox Analyzer (BSA). BSA is a dynamic analysis tool that has been designed to determine if a process exhibits potentially malicious behavior. In addition to analyzing the behavior of a process, BSA keeps track of actions taken by a monitored program, such as registry changes, file-system changes, and port changes [9]. The tool runs inside a sandbox which protects the system from infection while executing malware. The sandbox used by BSA is known as Sandboxie [36].

For dynamic analysis, we also experimented with Ether [17]. Ether is an open source tool that resides completely outside of the target OS, which makes it difficult for a program to detect that emulation is occurring. This is potentially useful for malware analysis, since viruses can, in principle, detect a debugger or a virtual environment during execution. However, for the datasets considered in this paper, we found no significant differences in the API call sequences generated by BSA and Ether. Since BSA is more user-friendly, in this research, we exclusively used BSA to generate our dynamic API call sequences.

3.2 Datasets

The following seven malware families were used as datasets in this research [29].

Harebot is a backdoor that provides remote access to the infected system. Because of its many features, it is also considered to be a rootkit [24].

Security Shield is a Trojan that, like Winwebsec, claims to be anti-virus software. Security Shield reports fake virus detection messages and attempts to coerce the users into purchasing software [37].

Smart HDD reports various problems with the hard drive and tries to convince the user to purchase a product to fix these "errors". Smart HDD is named after S.M.A.R.T., which is a legitimate tool that monitors hard disk drives (HDDs) [41].

Winwebsec pretends to be anti-virus software. An infected system displays fake messages claiming malicious activity and attempts to convince the user to pay money for software to clean the supposedly infected system [48].

Zbot also known as Zeus, is a Trojan horse that compromises a system by downloading configuration files or updates. Zbot is a stealth virus that hides in the file system [46]. The virus eventually vanishes from the processes list and, consequently, we could only trace its execution for about 5 to 10 minutes.

ZeroAccess is a Trojan horse that makes use of an advanced rootkit to hide itself. ZeroAccess is capable of creating a new hidden file system, it can create a backdoor on the compromised system, and it can download additional malware [47].

Table 1 gives the number of files used from each malware family and the benign dataset. For our benign dataset, we use the set of Windows System 32 files listed in Table 2.

Table 1: Datasets

Family	Number of Files
Harebot	45
Security Shield	50
Smart HDD	50
Winwebsec	200
Zbot	200
ZeroAccess	200
benign	40

Table 2: Benign Dataset

notepad	alg	calc	cipher
cleanmgr	cmd	cmdl32	driverquery
drwtsn32	dvdplay	eventcreate	eventtriggers
eventvwr	narrator	freecell	grpconv
mshearts	mspaint	netstat	nslookup
osk	packager	regedit	sndrec32
sndvol32	sol	sort	spider
syncapp	ipconfig	taskmgr	telnet
verifier	winchat	charmap	clipbrd
ctfmon	wscript	mplay32	winhlp32

3.3 Data Collection

For training and scoring, we use opcode sequences and API calls. For both opcode and API call sequences, we extract the data using both a static and a dynamic approach, giving us four observation sequences for each program under consideration. As noted in Section 2.3, opcode sequences and API call traces have been used in many research studies on malware detection.

Table 3: Example Disassembly

```
.text:00401017    call sub_401098
.text:0040101C    push 8
.text:0040101E    lea ecx, [esp+24h+var_14]
.text:00401022    push offset xyz
.text:00401027    push ecx
.text:00401028    call sub_401060
.text:0040102D    add esp, 18h
.text:00401030    test eax, eax
.text:00401032    jz short loc_401045
```

We use IDA Pro to disassemble files and we extract the static opcode sequences from the resulting disassembly. For example, suppose we disassemble an `exe` file and obtain the disassembly in Table 3. The static opcode sequence corresponding to this disassembly is

`call`, `push`, `lea`, `push`, `push`, `call`, `add`, `test`, `jz`

We discard all operands, labels, directives, etc., and only retain the mnemonic opcodes.

For dynamic opcode sequences, we execute the program in IDA Pro using the "tracing" feature. From the resulting program trace, we extract mnemonic opcodes. Note that the static opcode sequence corresponds to the overall program structure, while the dynamic opcode sequence corresponds to the actual execution path taken when the program was traced.

Microsoft Windows provides a variety of API (Application Programming Interface) calls that facilitate requests for services from the operating system [1]. Each API call has a distinct name, a set of arguments, and a return value. We only collect API call names, discarding the arguments and return value. An example of a sequence of API calls is given by

`OpenMutex`, `CreateFile`, `OpenProcessToken`, `AdjustTokenPrivileges`, `SetNamedSecurityInfo`, `LoadLibrary`, `CreateFile`, `GetComputerName`, `QueryProcessInformation`, `VirtualAllocEx`, `DeleteFile`

As with opcodes, API calls can be extracted from executables statically or dynamically. Our static API call sequences are obtained from IDA Pro disassembly. As mentioned in Section 3.1, we use Buster Sandbox Analyser (BSA) to dynamically extract API calls. BSA allows us to execute a program for a fixed amount of time, and it logs all API calls that occur within this execution window. From these logged API calls, we form a dynamic API call sequence for each executable.

3.4 Training and Scoring

For our experiments, four cases are considered. In the first, we use the static observation sequences for both training and scoring. In the second case, we use the dynamically extracted data for both training and scoring. The third and fourth cases are hybrid situations. Specifically, in the third case, we use the dynamic data for training, but the static data for scoring. In the fourth case, we use static training data, but dynamic data for scoring. We denote these four cases as static/static, dynamic/dynamic, static/dynamic, and dynamic/static, respectively.

Our static/static and dynamic/dynamic cases can be viewed as representative of typical approaches used in static and dynamic detection. The dynamic/static case is analogous to the approach used in many hybrid schemes. This approach seems to offer the prospect of the best of both worlds. That is, we can have a more accurate model due to the use dynamic training data, and yet scoring remains efficient, thanks to the use of static scoring data. Since the training phase is essentially one-time work, it is acceptable to spend significant time and effort in training. And the scoring phase can be no better than the model generated in the training phase.

On the other hand, the static/dynamic seems to offer no clear advantage. For completeness, we include this case in our opcode experiments.

We conducted a separate experiment for each of the malware datasets listed in Table 1, for each of the various combinations of static and dynamic data mentioned above. For every experiment, we use five-fold cross validation. That is, the malware dataset is partitioned into five equal subsets, say, S_1, S_2, S_3, S_4, and S_5. Then subsets S_1, S_2, S_3, and S_4 are used to train an HMM, and the resulting model is used to score the malware in S_5, and to score the files in the benign set. The process is repeated five times, with a different subset S_i reserved for testing in each of the five "folds". Cross validation serves to smooth out any bias in the partitioning of the data, while also maximizing the number of scores obtained from the available data.

The scores from a given experiment are used to form a scatterplot, from which an ROC curve is generated. The area under the ROC curve serving as our measure of success, as discussed in Section 2.4.

4 Results

In this section, we present our experimental results. We performed experiments with API call sequences and separate experiments using opcode sequences. All experiments were conducted as discussed in Section 3.4. That is, different combinations of static and dynamic data were used for training and scoring. Also, each experiment is based on training and scoring with HMMs, using five-fold cross validation.

As discussed in Section 2.4, the effectiveness of each experiment is quantified using the area under the ROC curve (AUC). In this section, we present AUC results, omitting the scatterplots and ROC curves. For additional details and results, see [13].

4.1 API Call Sequences

We trained HMM models on API call sequences for each of the malware families in Table 1. The ROC results are given in Table 4, with these same results plotted in the form of a bar graph in Figure 4.

Table 4: AUC-ROC Results for API Call Sequence

Family	Dynamic/ Dynamic	Static/ Static	Dynamic/ Static	Static/ Dynamic
Harebot	0.9867	0.7832	0.5783	0.5674
Security Shield	0.9875	1.0000	0.9563	0.8725
Smart HDD	0.9808	0.7900	0.7760	0.7325
Winwebsec	0.9762	0.9967	0.7301	0.6428
Zbot	0.9800	0.9899	0.9364	0.8879
ZeroAccess	0.9968	0.9844	0.7007	0.9106

Overall, we see that using dynamic training and testing yields the best results, while static training and testing is as effective in all cases, except for Harebot and Smart HDD. Perhaps surprisingly, the hybrid approach of dynamic training with static scoring produces worse results than the fully static case for all families. In fact, the dynamic/static case fares significantly worse than the static/static case for all families except Security Shield and Zbot.

We also computed PR curves for each of the malware families in Table 1. The AUC-PR results are given in Table 5, with these same results plotted in the form of a bar graph in Figure 5.

Table 5: AUC-PR Results for API Call Sequence

Family	Dynamic/ Dynamic	Static/ Static	Dynamic/ Static	Static/ Dynamic
Harebot	0.9858	0.8702	0.7111	0.4888
Security Shield	0.9884	1.0000	0.9534	0.3312
Smart HDD	0.9825	0.8799	0.3768	0.4025
Winwebsec	0.9800	0.9967	0.7359	0.3947
Zbot	0.9808	0.9931	0.9513	0.3260
ZeroAccess	0.9980	0.9879	0.4190	0.3472

Next, we provide results for analogous experiments using opcode sequences. Then we discuss the significance of these results with respect to static, dynamic, and hybrid detection strategies.

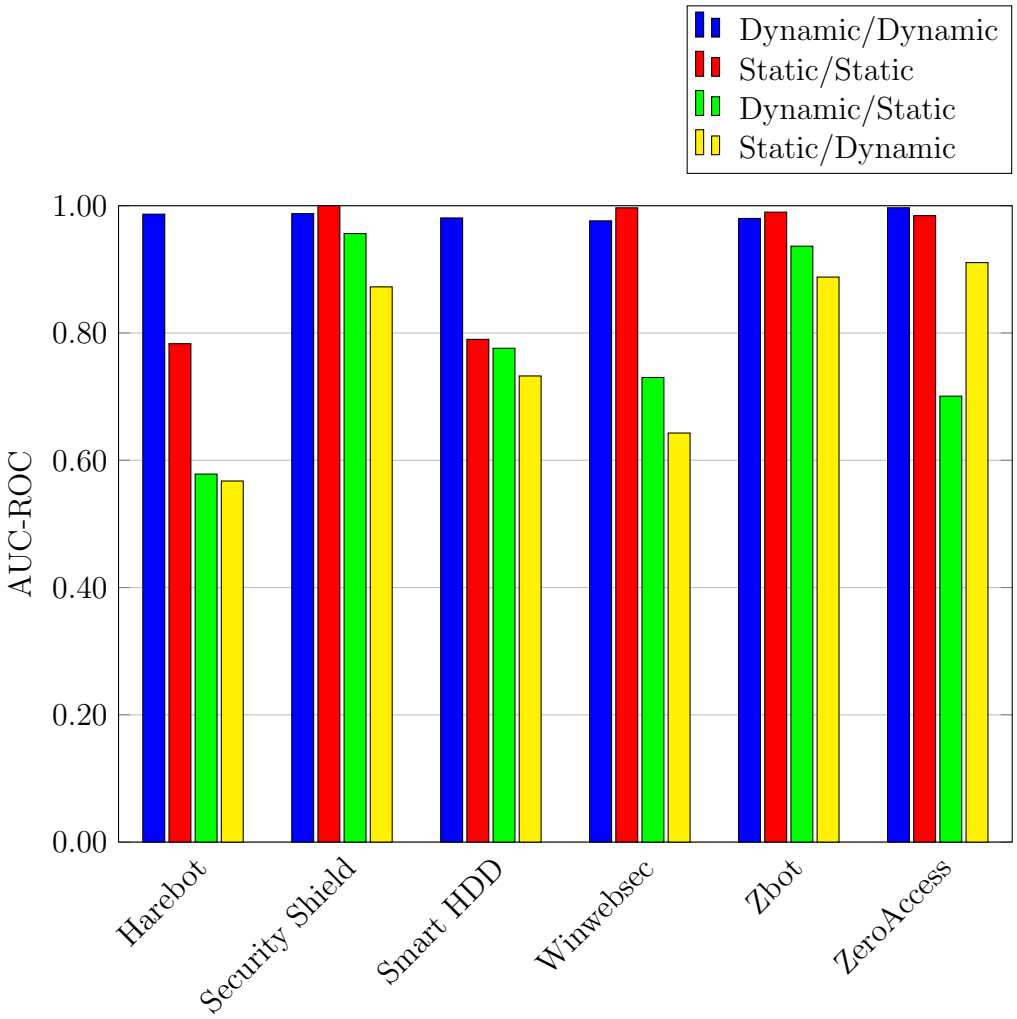

Figure 4: ROC Results for API Call Sequence

4.2 Opcode Sequences

In this set of experiments, we use opcode sequences for training and scoring. As in the API sequence case, we consider combinations of static and dynamic data for training and scoring. Also as above, we train HMMs and use the resulting models for scoring.

Before presenting our results, we note that the opcode sequences obtained in the static and dynamic cases differ significantly. In Figure 6 we give a bar graph showing the counts for the number of distinct opcodes in the static and dynamic cases. From Figure 6, we see that scoring in the dynamic/static case will be complicated by the fact that, in general, many opcodes will appear when scoring that were not part of the training set. While there are several ways to deal with such a situation, when scoring, we simply omit any opcodes that did not appear in the training set.

Our results for training and scoring on opcode sequences are given in Table 6. The

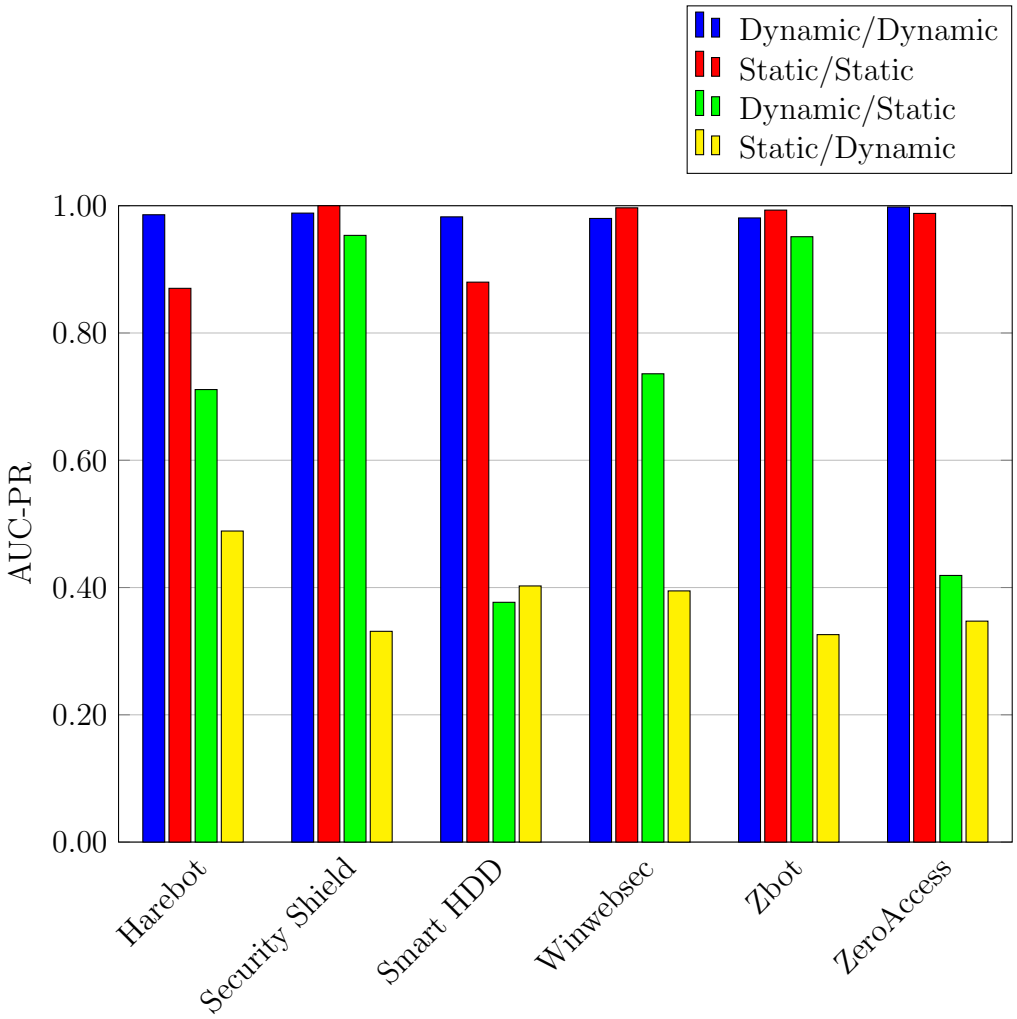

Figure 5: PR Results for API Call Sequence

results in Table 6 are given in the form of a bar graph in Figure 7.

These opcode-based results are generally not as strong as those obtained for API call sequences. But, as with API call sequences, the best results are obtained in the dynamic/dynamic case. However, unlike the API call sequence models, opcode sequences yield results that are roughly equivalent in the static/static and the hybrid dynamic/static case. Additional experimental results can be found in [13].

4.3 Imbalance Problem

In statistical-based scoring, we typically have a primary test that is used to filter suspect cases, followed by a secondary test that is applied to these suspect cases. For malware detection, the primary test is likely to have an imbalance, in the sense that the number of benign samples exceeds the number of malware samples—possibly by a large margin.

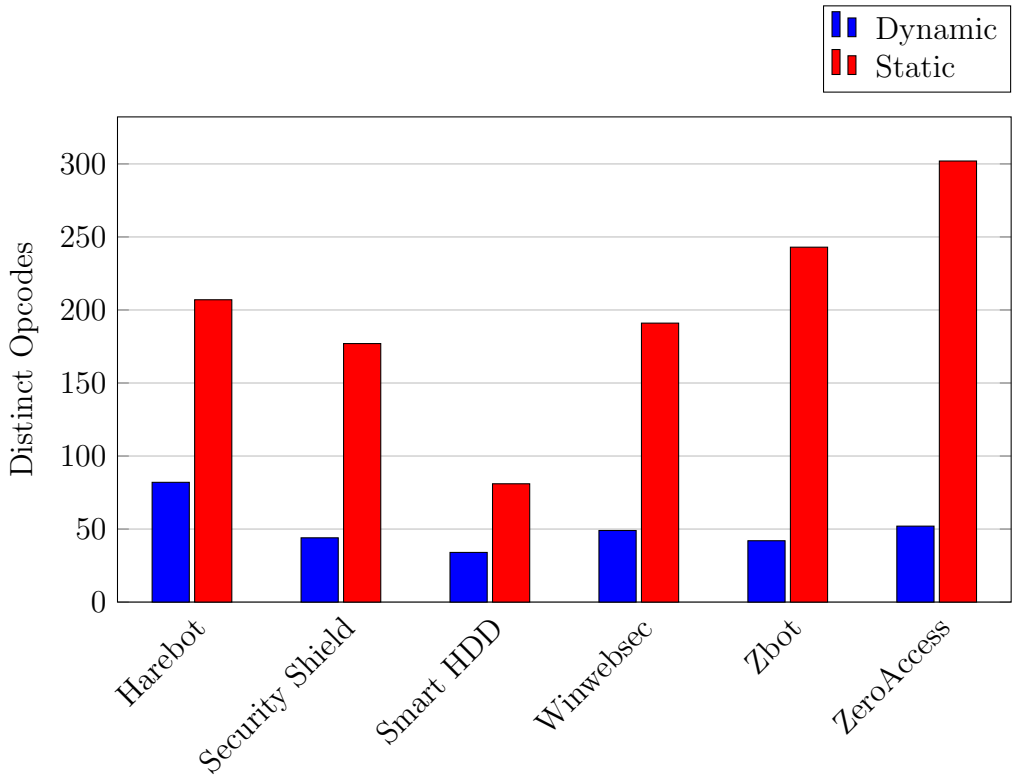

Figure 6: Distinct Opcodes

Table 6: AUC-ROC Results for Opcode Sequences

Family	Dynamic/ Dynamic	Static/ Static	Dynamic/ Static	Static/ Dynamic
Harebot	0.7210	0.5300	0.5694	0.5832
Security Shield	0.9452	0.5028	0.6212	0.5928
Smart HDD	0.9860	0.9952	1.0000	0.9748
Winwebsec	0.8268	0.6609	0.7004	0.6279
Zbot	0.9681	0.7755	0.6424	0.9525
ZeroAccess	0.9840	0.7760	0.8970	0.6890

In the secondary stage, we would expect the imbalance to be far less significant. Due to their cost, malware detection techniques such as those considered in this paper would most likely be applied at the secondary stage. Nevertheless, it may be instructive to consider the effect of a large imbalance between the benign and malware sets. In this section, we consider the effect of such an imbalance on our dynamic, static, and hybrid techniques.

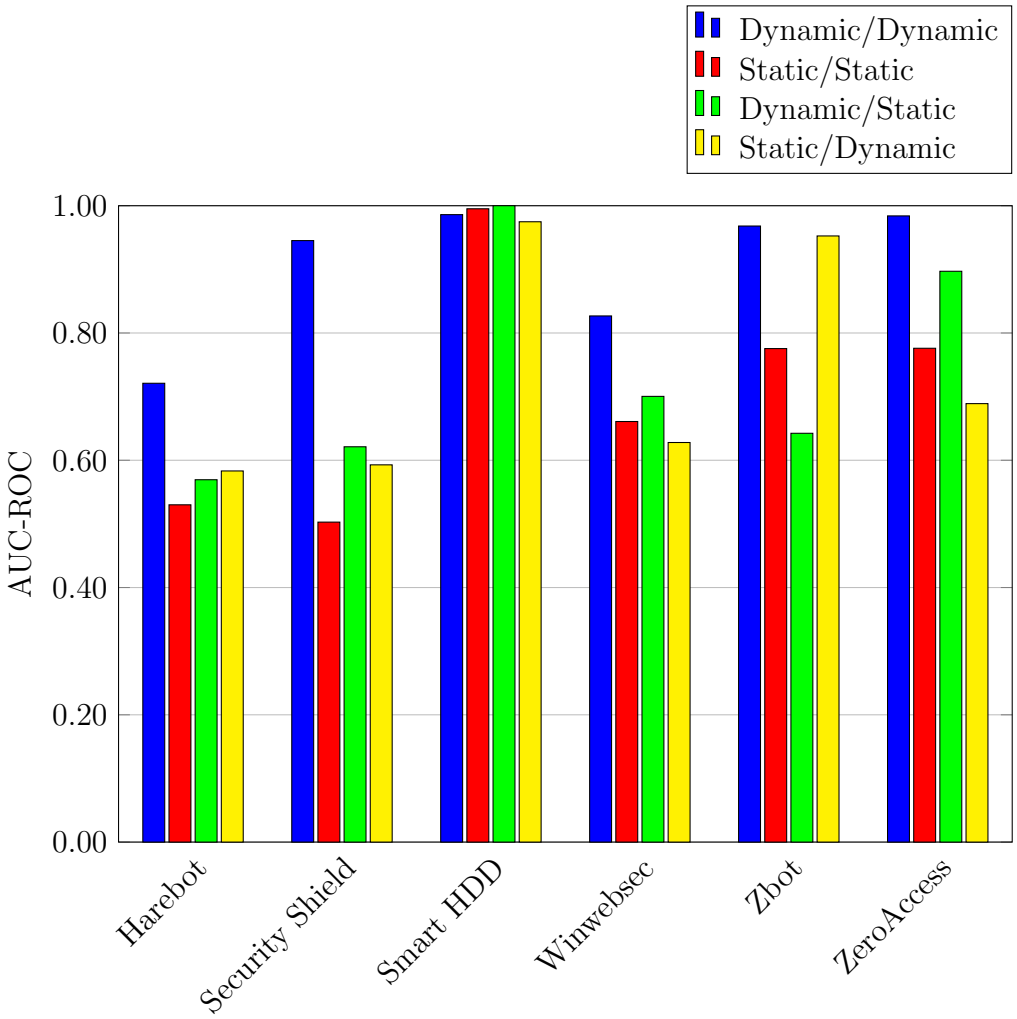

Figure 7: ROC Results for Opcode Sequences

We can simulate an imbalance by simply duplicating each benign score n times. Assuming that the original number of scored benign and malware samples are equal, a duplication factor of n simulates an imbalanced data set where the benign samples outnumber the malware samples by a factor of n. Provided that our original benign set is representative, we would expect an actual benign set of the appropriate size to yield scores that, on average, match this simulated (i.e., expanded) benign set.

However, the AUC-ROC for such an expanded benign set will be the same as for the original set. To see why this is so, suppose that for a given threshold, we have TP $= a$, FN $= b$, FP $= c$, and TN $= d$. Then (x, y) is a point on the ROC curve, where

$$x = \text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} = \frac{c}{c+d} \quad \text{and} \quad y = \text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{a}{a+b} \tag{1}$$

Now suppose that we duplicate each element of the negative (i.e., benign) set n times. Then for the same threshold used to compute (1), we have TP $= a$, FN $= b$, FP $= nc$,

and TN $= nd$, and hence we obtain the same point (x, y) on the ROC curve for this modified dataset.

In contrast, for PR curves, using the same threshold as above we have

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{a}{a+b} \quad \text{and} \quad \text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{a}{a+c}$$

When we expand our dataset by duplicating the benign scores n times, this threshold yields

$$\text{recall} = \frac{a}{a+b} \quad \text{and} \quad \text{precision} = \frac{a}{a+nc}$$

Consequently, we see that simulating an imbalance in this way will tend to flatten the PR curve, and thereby reduce the AUC-PR. In addition, the precise degree of flattening will depend on the relative distribution of the malware and benign scores.

We have shown that the AUC-ROC provides no information on the effect of an imbalance between the malware and benign sets. In some sense, this can be viewed as a strength of the AUC-ROC statistic, although it does render it useless for analyzing the effect of imbalanced data. On the other hand, the AUC-PR is a useful statistic for comparing the effect of an imbalance between these two sets. Consequently, we use the AUC-PR in this section to determine the effect of a (simulated) imbalance between the malware and benign sets. We consider the API sequence results, and we duplicate each benign score by a factor of $n = 1$, $n = 10$, $n = 100$, and $n = 1000$, and plot the results on a logarithmic (base 10) scale. The resulting AUC-PR values for each of the four cases (i.e., dynamic/dynamic, static/static, dynamic/static, and static/dynamic) are plotted as line graphs in Figure 8.

The results in Figure 8 suggest that we can expect the superiority of the a fully dynamic approach, to increase as the imbalance between the benign and malware sets grows. In addition, the advantage of the fully static approach over our hybrid approaches increases as the imbalance increases. We also see that even in those cases where the dynamic/static approach is initially competitive, it fails to remain so for a large imbalance. And finally, the overall weakness of the static/dynamic approach is even more apparent from this PR analysis.

4.4 Discussion

The results in this section show that for API calls and opcode sequences, a fully dynamic strategy is generally the most effective approach. However, dynamic analysis is generally costly in comparison to static analysis. At the training phase, this added cost is not a significant issue, since training is essentially one-time work that can be done offline. But, at the scoring phase, dynamic analysis would likely be impractical, particularly where it is necessary to scan a large number of files.

In a hybrid approach, we might attempt to improve the training phase by using dynamic analysis while, for the sake of efficiency, using only a static approach in the scoring phase. However, such a strategy was not particularly successful in the experiments considered here. For API call sequences, we consistently obtained worse results with the

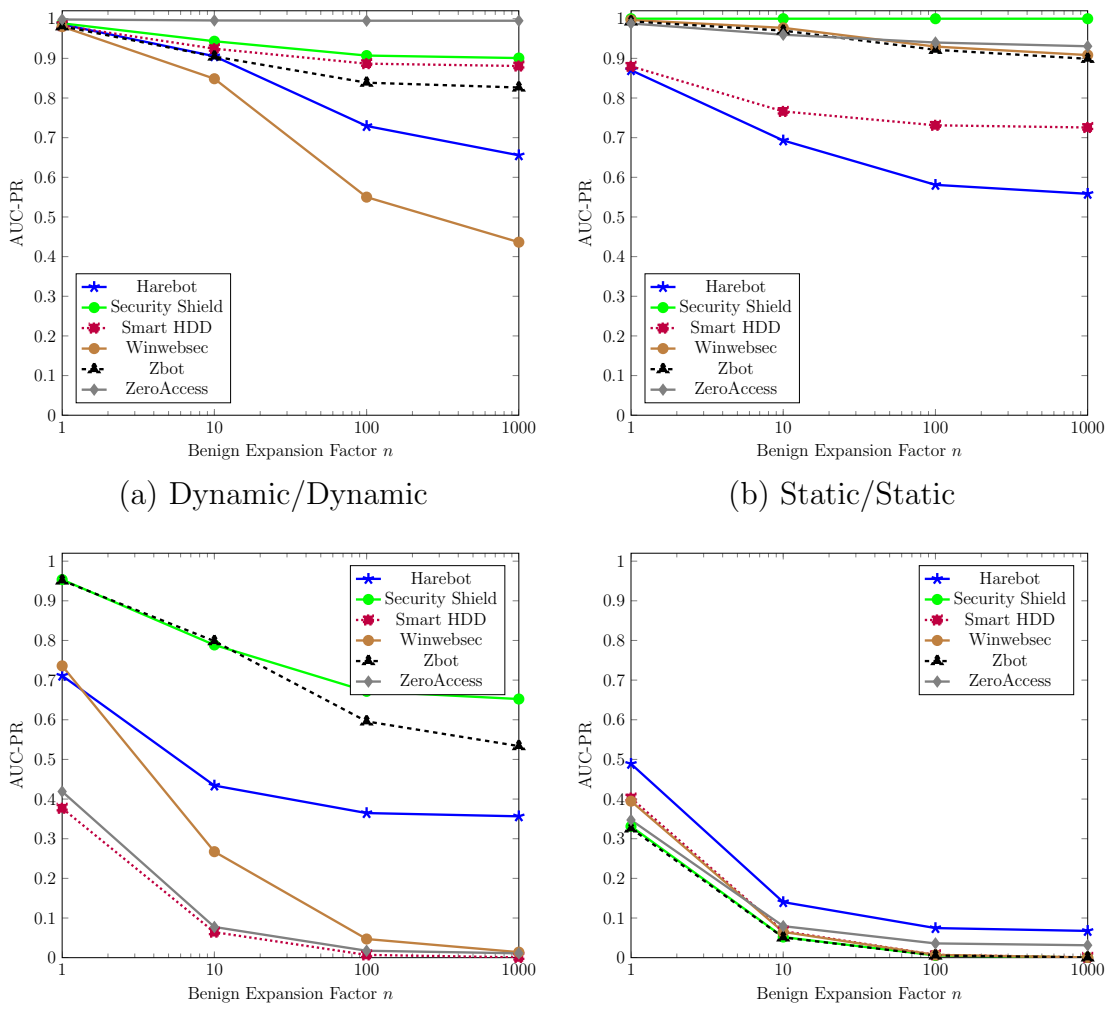

Figure 8: AUC-PR and Imbalanced Data (API Calls)

hybrid dynamic/static as compared to a fully static approach. For opcode sequences, the results were inconsistent—in four of the cases, the hybrid dynamic/static method was marginally better than the fully static approach, but for one case it was significantly worse.

Attempting to optimize a malware detection technique by using hybrid analysis is intuitively appealing. While such a hybrid approach may be more effective in certain cases, our results show that this is not likely to be generically the case. Consequently, when hybrid approaches are proposed, it would be advisable to test the results against comparable fully dynamic and fully static techniques.

5 Conclusion and Future Work

In this paper, we tested malware detection techniques based on API call sequences and opcode sequences. We trained Hidden Markov Models and compared detection rates for models based on static data, dynamic data, and hybrid approaches.

Our results indicate that a fully dynamic approach based on API calls is extremely effective across a range of malware families. A fully static approach based on API calls was nearly as effective in most cases. Our results also show that opcode sequences can be effective in many cases, but for some families the results are not impressive. These results likely reflect the nature of obfuscation techniques employed by malware writers. That is, current obfuscation techniques are likely to have a significant effect on opcode sequences, but little attention is paid to API calls. With some additional effort, API call sequences could likely be obfuscated, in which case the advantage of relying on API call sequences for detection might diminish significantly.

Examples of relatively complex and involved hybrid techniques have recently appeared in the literature. However, due to the use of different data sets, different measures of success, and so on, it is often difficult, if not impossible, to compare these techniques to previous (non-hybrid) work. Further, the very complexity of such detection techniques often makes it difficult to discern the actual benefit of any one particular aspect of a technique. The primary goal of this research was to test the tradeoffs between static, dynamic, and hybrid analysis, while eliminating as many confounding variables as possible.

The experimental results presented in this paper indicate that a straightforward hybrid approach is unlikely to be superior to fully dynamic detection. And even in comparison to fully static detection, our hybrid dynamic/static approach did not offer consistent improvement. Interestingly, the impractical static/dynamic hybrid approach was superior in some cases (by some measures). These results are, perhaps, somewhat surprising given the claims made for hybrid approaches.

Of course, it is certain that hybrid techniques offer significant benefits in some cases. But, the work here suggests that such claims should be subject to careful scrutiny. In particular, it should be made clear whether improved detection is actually due to a hybrid model itself, or some other factor, such as the particular combination of scores used. Furthermore, it should be determined whether these benefits exist over a wide range of malware samples, or whether they are only relevant for a relatively narrow range of malware.

Future work could include a similar analysis invovling additional features beyond API calls and opcodes. A comparison of scoring techniques other than HMMs (e.g., graph-based scores, structural scores, other machine learning and statistical scores) and optimal combinations of static and dynamic scores (e.g., using Support Vector Machines) would be worthwhile. Finally, a more in-depth analysis of imbalance issues in this context might prove interesting.

References

[1] F. Ahmed, et al, Using spatio-temporal information in API calls with machine learning algorithms for malware detection, *ACM Workshop on Security and Artificial Intelligence*, 2009

[2] B. Anderson, et al, Graph-based malware detection using dynamic analysis, *Journal of Computer Virology*, 7(4):247–258, 2011

[3] C. Annachhatre, T. H. Austin, and M. Stamp, Hidden Markov models for malware classification. *Journal of Computer Virology and Hacking Techniques*, 11(2):59–73, 2014

[4] S. Attaluri, S. McGhee, and M. Stamp, Profile Hidden Markov Models and metamorphic virus detection, *Journal in Computer Virology*, 5(2):151–169, 2009

[5] J. Aycock, Computer Viruses and Malware, *Advances in Information Security*, Springer-Verlag, New York, 2006

[6] D. Baysa, R. M. Low, and M. Stamp, Structural entropy and metamorphic malware, *Journal of Computer Virology and Hacking Techniques*, 9(4):179–192, 2013

[7] J. Borello and L. Me, Code obfuscation techniques for metamorphic viruses, *Journal in Computer Virology*, 4(3):211–220, 2008

[8] A. P. Bradley, The use of the area under the ROC curve in the evaluation of machine learning algorithms, *Journal Pattern Recognition*, 30(7):1145–1159, 1997

[9] Buster Sandbox Analyser, `http://bsa.isoftware.nl/`

[10] Y. H. Choi , et al., Toward extracting malware features for classification using static and dynamic analysis, Computing and Networking Technology (ICCNT), 2012

[11] M. Christodorescu, S. Jha, Static analysis of executables to detect malicious patterns, USENIX Security Symposium, 2003

[12] J. Dai, R. Guha, J. Lee. Efficient virus detection using dynamic instruction sequences, *Journal of Computers*, 4(5):405–414, 2009

[13] A. Damodaran, Combining dynamic and static analysis for malware detection, Master's report, Department of Computer Science, San Jose State University, 2015, `http://scholarworks.sjsu.edu/etd_projects/391/`

[14] J. Davis and M. Goadrich, The relationship between precision-recall and ROC curves, `http://www.autonlab.org/icml_documents/camera-ready/030_The_Relationship_Bet.pdf`

[15] P. Deshpande, Metamorphic detection using function call graph analysis, Master's report, Department of Computer Science, San Jose State University, 2013, `http://scholarworks.sjsu.edu/etd_projects/336/`

[16] S. Deshpande, Y. Park, and M. Stamp, Eigenvalue analysis for metamorphic detection, *Journal of Computer Virology and Hacking Techniques*, 10(1):53–65, 2014

[17] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, Ether: Malware analysis via hardware virtualization extensions, CCS 08, October 27–31, 2008, Alexandria, Virginia, `http://ether.gtisc.gatech.edu/ether_ccs_2008.pdf`

[18] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, A survey on automated dynamic malware analysis techniques and tools, *Journal of ACM Computing Surveys*, 44(2):Article 6, 2012

[19] M. Eskandari and S. Hashemi, A graph mining approach for detecting unknown malwares, *Journal of Visual Languages and Computing*, 23(3):154–162, 2012

[20] M. Eskandari, Z. Khorshidpour, and S. Hashemi, HDM-Analyser: A hybrid analysis approach based on data mining techniques for malware detection, *Journal of Computer Virology and Hacking Techniques*, 9(2):77–93, 2013

[21] M. Eskandari, Z. Khorshidpur, and S. Hashemi, To incorporate sequential dynamic features in malware detection engines, Intelligence and Security Informatics Conference (EISIC), pp. 46–52, 2012

[22] T. Fawcett. An introduction to ROC analysis, `http://people.inf.elte.hu/kiss/13dwhdm/roc.pdf`

[23] Z. Ghahramani. An introduction to hidden Markov models and Bayesian networks, *International Journal of Pattern Recognition and Artificial Intelligence*, 15(1):9–42, 2001

[24] Harebot, `http://www.pandasecurity.com/homeusers/security-info/220319/Harebot.M`

[25] G. Jacob, H. Debar, and E. Filiol. Behavioral detection of malware: From a survey towards an established taxonomy. *Journal of Computer Virology*, 4(3):251–266, 2008

[26] R. K. Jidigam, T. H. Austin and M. Stamp, Singular value decomposition and metamorphic detection, to appear in *Journal of Computer Virology and Hacking Techniques*

[27] C. Kolbitsch, et al, Effective and efficient malware detection at the end host, USENIX Security Symposium, 2009

[28] J. Lee, T. H. Austin, and M. Stamp, Compression-based analysis of metamorphic malware, to appear in *International Journal of Security and Networks*

[29] A. Nappa, M. Z. Rafique, and J. Caballero, Driving in the cloud: An analysis of drive-by download operations and abuse reporting, *Proceedings of the 10th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, Berlin, Germany, July 2013

[30] Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel, Fast malware classification by automated behavioral graph matching, *Proceedings of the 6th Annual Workshop on Cyber Security and Information Intelligence Research*, 2010

[31] Y. Park, D. Reeves, and M. Stamp, Deriving common malware behavior through graph clustering, *Computers and Security*, 39(B):419–430, 2013

[32] Y. Qiao, J. He, Y. Yang , and L. Ji, Analyzing malware by abstracting the frequent itemsets in API call sequences, Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 265–270, 2013

[33] J. Rhee, R. Riley, D. Xu, and X. Jiang, Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory, *Recent Advances in Intrusion Detection*, Lecture Notes in Computer Science, Volume 6307, pp. 178–197, 2010

[34] L. R. Rabiner, A tutorial on Hidden Markov Models and selected applications in speech recognition, *Proceedings of the IEEE*, 77(2):257–286, 1989, `http://www.cs.ubc.ca/~murphyk/Bayes/rabiner.pdf`

[35] N. Runwal, R. M. Low, and M. Stamp, Opcode graph similarity and metamorphic detection, *Journal in Computer Virology*, 8(1-2):37–52, 2012

[36] SandBoxie, `http://sandboxie.com/`

[37] Security Shield, `http://www.symantec.com/security_response/glossary/define.jsp?letter=s&word=security-shield`

[38] M. K. Shankarapani, S. Ramamoorthy, R. S. Movva, and S. Mukkamala, Malware detection using assembly and API call sequences, *Journal of Computer Virology*, 2(7):107–119, 2011

[39] G. Shanmugam, R. M. Low, and M. Stamp, Simple substitution distance and metamorphic detection, *Journal of Computer Virology and Hacking Techniques*, 9(3):159–170, 2013

[40] T. Singh, Support Vector Machines and metamorphic malware detection, Master's report, Department of Computer Science, San Jose State University, 2015, `http://scholarworks.sjsu.edu/etd_projects/409/`

[41] Smart HDD, `http://support.kaspersky.com/viruses/rogue?qid=208286454`

[42] I. Sorokin, Comparing files using structural entropy, *Journal in Computer Virology* 7(4):259–265, 2011

[43] M. Stamp, A revealing introduction to hidden Markov models, 2012, `http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf`

[44] Symantec White Paper, Internet Security Report, Volume 20, 2015, `http://www.symantec.com/security_response/publications/threatreport.jsp`

[45] A. H. Toderici and M. Stamp, Chi-squared distance and metamorphic virus detection, *Journal of Computer Virology and Hacking Techniques*, 9(1):1–14, 2013

[46] Trojan.Zbot, `http://www.symantec.com/security_response/writeup.jsp?docid=2010-011016-3514-99`

[47] Trojan.ZeroAccess, `http://www.symantec.com/security_response/writeup.jsp?docid=2011-071314-0410-99`

[48] Win32/Winwebsec, `http://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Win32%2fWinwebsec`

[49] W. Wong and M. Stamp, Hunting for metamorphic engines, *Journal in Computer Virology*, 2(3):211–229, 2006

[50] Y. Ye, D. Wang, T. Li, D. Ye, and Q. Jiang, An intelligent PE-malware detection system based on association mining, *Journal in Computer Virology*, 4(4):323–334, 2008