RHMD: Evasion-Resilient Hardware Malware Detectors

Khaled N. Khasawneh
University of California, Riverside
kkhas001@ucr.edu

Nael Abu-Ghazaleh
University of California, Riverside
nael@cs.ucr.edu

Dmitry Ponomarev
Binghamton University
dima@cs.binghamton.edu

Lei Yu
Binghamton University
lyu@cs.binghamton.edu

ABSTRACT

Hardware Malware Detectors (HMDs) have recently been proposed as a defense against the proliferation of malware. These detectors use low-level features, that can be collected by the hardware performance monitoring units on modern CPUs to detect malware as a computational anomaly. Several aspects of the detector construction have been explored, leading to detectors with high accuracy. In this paper, we explore the question of how well evasive malware can avoid detection by HMDs. We show that existing HMDs can be effectively reverse-engineered and subsequently evaded, allowing malware to hide from detection without substantially slowing it down (which is important for certain types of malware). This result demonstrates that the current generation of HMDs can be easily defeated by evasive malware. Next, we explore how well a detector can evolve if it is exposed to this evasive malware during training. We show that simple detectors, such as logistic regression, cannot detect the evasive malware even with retraining. More sophisticated detectors can be retrained to detect evasive malware, but the retrained detectors can be reverse-engineered and evaded again. To address these limitations, we propose a new type of Resilient HMDs (RHMDs) that stochastically switch between different detectors. These detectors can be shown to be provably more difficult to reverse engineer based on resent results in probably approximately correct (PAC) learnability theory. We show that indeed such detectors are resilient to both reverse engineering and evasion, and that the resilience increases with the number and diversity of the individual detectors. Our results demonstrate that these HMDs offer effective defense against evasive malware at low additional complexity.

CCS CONCEPTS

• **Security and privacy** → **Hardware security implementation**; **Malware and its mitigation**;

KEYWORDS

HMDs, malware detection, adversarial machine learning

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123972

ACM Reference format:
Khaled N. Khasawneh, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Lei Yu. 2017. RHMD: Evasion-Resilient Hardware Malware Detectors. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3123972

1 INTRODUCTION

Malicious attackers compromise systems to install malware [33, 48]. Preventing the compromise of systems is practically impossible: attackers obtain privileged access to systems in a variety of ways, such as drive-by-downloads with websites exploiting browser vulnerabilities [2] as well as network-accessible vulnerabilities [46]. Similarly, social engineering attacks including Phishing attacks, and malicious email attachments allow user-authorized installation of malicious programs [1].

While preventing compromise is difficult, detecting malware is also becoming increasingly more complicated. Indeed, modern malware is increasing in sophistication, challenging the abilities of software detectors. Typical techniques for malware detection such as VM introspection [17], dynamic binary instrumentation [13], information flow tracking [57], and software anomaly detection [19] have both coverage limitations and introduce substantial overhead (e.g., 10x slowdown for information flow tracking is typical in software [56]). These difficulties typically limit malware detection to static signature-based virus scanning tools [15] which have known limitations [35], allowing the attackers to bypass them and remain undetected.

In response to these trends, Hardware Malware Detectors (HMDs) have recently been proposed to make systems more malware-resistant. Several studies have shown that malware can be classified based on low-level hardware features such as instruction mixes, memory reference patterns, branch distributions, and architectural state information such as cache miss rates and branch prediction rates [12, 25, 27, 39, 50]. Demme et al. showed that features collected from the ARM performance counters can be used to classify malware from normal programs after the fact [12]. Tang et al. showed that unsupervised learning approaches could also successfully classify malware. Ozsoy et al. built a hardware accelerator that allows continuous real-time monitoring of malware [39]. Khasawneh et al. improved the design to apply ensemble learning techniques [27]. Kazdagli et al. proposed an environment to systematically evaluate HMDs [26]. The SnapDragon processor from Qualcomm appears to be using HMDs for online malware detection, although the technical details are not published [43]. At a time when malware developers appear to have the upper hand over defenders, hardware supported malware

detection can offer a substantial advantage to defenders because it is always on and has little impact on performance or power [39, 40].

In this paper, we first explore whether attackers can adapt malware to continue to operate while avoiding detection by HMDs. Should HMDs become widely deployed, it is natural to expect that attackers will attempt to evade detection. Intuitively, it should be possible for the attacker to evade detection if there are no restrictions placed on them, by running a mostly normal program and advancing the attack very slowly. However, we assume that the attacker would like to minimize the impact on the malware execution time since some attacks are time sensitive (e.g., covert and side-channels [10, 16, 21, 23, 37, 42]) or computationally intensive (e.g., Spam or Click Fraud [14, 54]). We describe the threat model and limitations in Section 2.

We approach the question of whether malware can evade HMD, and whether HMDs can be made resilient to evasion, in the following steps:

(1) Can HMDs be reverse-engineered? Recent results in adversarial classification [52] imply that arbitrarily complex but deterministic classifiers can be reverse-engineered. We confirm that this is the case for HMDs by reverse-engineering a number of detectors under realistic assumptions. We describe our dataset and methodology in Section 3, and present and analyze the reverse-engineered detectors in Section 4.
(2) Having a model of the detector, can malware developers modify malware to avoid detection? Evading the detection by changing the behavior of the malware is known as mimicry attacks [8, 53]. We show (in Section 5) that existing HMDs can be rendered ineffective using simple modifications to the malware binary.
(3) Can the malware evade detection even if the detector is retrained with some samples of the evasive malware? We show in Section 6 that for simple evasion strategies that can fool a given detector, retraining a logistic regression (LR) detector does not result in effective classification of evasive malware, unless the detection performance on normal malware is sacrificed. On the other hand, more sophisticated detectors such as Neural Networks (NN) can be successfully retrained, but the attacker is still able to reverse-engineer the retrained detector and evade it again.
(4) After showing that the current generation of HMDs is vulnerable to evasion, we explore whether new HMDs can be constructed that are robust to evasion. In particular, we propose in Section 7 a new resilient HMD (RHMD) organization that uses multiple diverse detectors and switches between them unpredictably. We show that RHMDs that use even simple base detectors are resilient to both reverse-engineering and evasion. Furthermore, this resilience increases with the number and diversity of the base detectors.
(5) Finally, we explore whether RHMDs fundamentally increase the difficulty of evasion or simply present another hurdle that can be bypassed by attackers. To this end, in Section 8 we overview recent results in Probably Approximately Correct (PAC) learnability theory that proves that RHMDs provide a measurable advantage in increasing the difficulty of reverse-engineering and complicate evasion. By making HMDs resilient to evasion, we bring them closer to practical deployment.

The problem of evasive malware detection has been considered in the context of software malware detectors [8, 49, 53]. Moreover, some existing HMD proposals discuss the possibility of malware evasion [12, 25]. However, ours is the first study that explores this important question regarding HMDs in detail and develops solutions to it. We note that while our experiments target HMDs, the underlying evasion problem exists in the context of any adversarial classification problem [52]. Our work advances the state of the art in general, not just for HMDs: we show systematically that reverse-engineering is possible, we develop techniques that use the result of reverse-engineered detectors to efficiently evade detection, and we introduce evade-retrain games and study their resilience to evasion.

In summary, the contributions of the paper are:

- We show that it is possible to accurately reverse engineer HMDs, regardless of their complexity.
- We show that once an HMD has been reverse engineered, malware can effectively evade it using low overhead evasion strategies. This result brings into question the effectiveness of existing HMDs.
- We show that simple linear HMDs such as LR cannot be retrained to adapt to evasive malware. More complex classifiers such as NN can adapt better, but may break down after several generations of evasion and retrain. Moreover, new malware can still reverse-engineer and evade even such classifiers.
- We develop a new class of evasion-resilient HMDs (RHMDs) that operates by randomizing detection responsibility across different diverse detectors. RHMDs cannot effectively be reverse-engineered to enable evasion, which we support both experimentally and using recent results from PAC learnability theory. The number and diversity of the individual classifiers used increases the resilience to reverse-engineering and evasion. In addition, we study implementation complexity of such classifiers in hardware.

2 THREAT MODEL AND LIMITATIONS

We assume an adversarial attack model which starts with the adversary attempting to reverse engineer the classifier. We assume that the attacker has access to a machine with a similar detector as the victim machine. This allows the attacker to observe the behavior of the classifier for given programs (whether malware or normal programs). With a model of the detector, the attacker can attempt to generate evading malware that hide themselves by changing some of their characteristics (feature values). Such evading mechanism is known as *mimicry* attacks [8, 53], which can be in the form of no-op insertion, code obfuscation by the attackers, or calling benign functions in the middle of the malicious payload [24].

We assume that the attacker that undertakes malware rewriting as part of a mimicry attack is interested in maintaining reasonable performance of the malware. If this assumption is not true, an attacker can simply run a normal program with embedded malware, that advances the malware program arbitrarily slow (e.g., 1 malware instruction every N normal instructions where N is arbitrarily large),

making detection impossible. Note that this is a limitation of all anomaly detectors, and not only HMDs. This assumption is also reasonable for important segments of malware such as: malware that is time sensitive (e.g., that performs covert or side-channel attacks [16, 23, 37, 42]) and malware that is computationally intensive such as that executing on botnets being monetized under a pay-per-install model [9] (e.g., Spam bots or Click fraud). Such malware have a utility to the malware writer proportional to their performance.

3 DATA AND METHODOLOGY

We collected samples of malware (from the MalwareDB malware set [31]) and normal programs to use in our study. The downloaded malware data set consisted of 3000 malware programs. For regular program samples, we used Windows programs since the malware programs that we use are Windows-based. The regular program set contains a variety of applications including browsers, text editing tools, system programs, SPEC 2006 benchmarks [20], and other popular applications such as Acrobat Reader, Notepad++, and Winrar. In total, the non-malware data set contains 554 programs. Both malware and regular programs data sets were divided into 60% *victim training*, 20% *attacker training* for the reverse engineered detector, and 20% *attacker testing* of the detector. To ensure that there is no bias in the distribution of malware programs across the sets, each set includes a randomly selected subset of malware samples from each type of malware in the overall data set.

The data was collected by running both malware and regular programs on a virtual machine with a Windows 7 operating system. To allow malware programs to carry out their intended functionality, the Windows security services and firewall were disabled. Furthermore, the dynamic trace of executed programs was collected using Pin instrumentation tool [30]. Unlike mobile malware where many malware samples require user interaction and necessitate special efforts to ensure correct behavior [26], we observed that the vast majority of our windows/desktop malware operated correctly (through manual inspection and examination malware behavior during run-time); several malware samples tripped the intrusion detection monitoring systems on our network as they attempted to discover and attack other machines, until we separated the environment into an independent subnet.

The collected trace duration for each executed program was 5000 system calls or 15 million of committed instructions, starting after a warm-up period, whichever is reached first. While ideally, we would have liked to run each program longer, we are limited by the computational overhead; since we are collecting run-time behavior of the programs using dynamic profiling information through Pin within a virtual machine, collection requires several weeks of execution on a small cluster and produces several terabytes of compressed profiling traces. Training and testing are also extremely computationally intensive. We believe that this data set is sufficiently large to establish the feasibility and provide trustworthy experimental results.

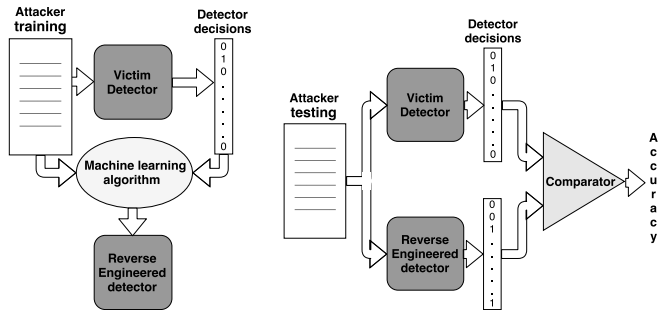
We collected different feature vectors, specifically:

- Executed instruction mixes (called *Instructions* in the rest of the paper): this feature tracks the frequency of instructions that show the most different frequency (delta) between normal programs and malware in the training set;
- Memory address patterns (called *Memory* in the rest of the paper): this feature tracks the distribution of memory references organized in bins based on the address difference between consecutive memory accesses; and
- Architectural events (called *Architectural* in the rest of the paper): tracks the numbers of different architectural events occurring in an execution period such as unaligned memory accesses, and taken branches.

These features are modeled after those used in prior HMD studies [12, 39].

4 REVERSE-ENGINEERING HMDS

In this section, we demonstrate that we can successfully reverse-engineer HMDs based on supervised learning (e.g., similar to those presented by Demme et al. [12], Ozsoy et al. [39], and some of the detectors used by Kazdagli et al. [26]). Reverse-engineering the malware detector allows the adversary to construct a model of the HMD. The model is necessary to be able to methodically develop evasive malware. We assume that the adversary has the ability to query the targeted detector; if they do not, the problem becomes NP-Hard [52].



(a) Reverse-engineering of victim **(b) Evaluating reverse-engineered detector**

Figure 1: Overview of the reverse-engineering process

Figure 1a shows the steps in reverse-engineering a detector. First, the adversary prepares a training data set that is composed of both regular and malware programs: this is a separate data set from the one used to train the classifier, which is unknown to the attacker. Next, the adversary uses this data set to query the victim detector and records the victim's detection decisions. The decisions are used as the label for the data as we construct the reverse-engineered detector. Finally, the adversary may use different machine learning classification algorithms trained with the labeled data to build the new reverse-engineered detector.

Figure 1b shows the evaluation of the reverse-engineered detector. The adversary first prepares an attacker testing data set as described above. Next, both the original detector and reverse-engineered detector are queried using the attacker testing data set. Finally, the percentage of equivalent decisions made by the two detectors is calculated. Note that from the adversary point of view, it does not matter if the detector is classifying malware and regular programs

correctly; rather, the attacker desires to mimic the classification of the victim detector and it evaluates success on that basis.

For most of our studies, we evaluate baseline detectors that use logistic regression (LR) and neural networks (NN); the methodology naturally generalizes to other classification algorithms. We implemented the NN classifier as a multi-layer perceptron (MLP) with a single hidden layer that has a number of neurons equal to the number of features in the feature vector. We use the *tanh* function as the activation function. The rationale for selecting these two algorithms is that prior studies showed that LR performs well and has low complexity, facilitating hardware implementations [39]. NN features more complex classification ability, capable of producing a non-linear classification boundary. These detectors allow us to contrast the impact of the detector complexity on both reverse-engineering process and mimicry attacks. For some studies, we use other classifiers to illustrate some generalizations of our conclusions.

The victim data set is used to train different detector instances using each of the two algorithms for each of the three different features, resulting in six detectors. The detectors take a feature vector as an input in order to produce a 0 or 1 decision indicating whether the program is a regular or malware program respectively.

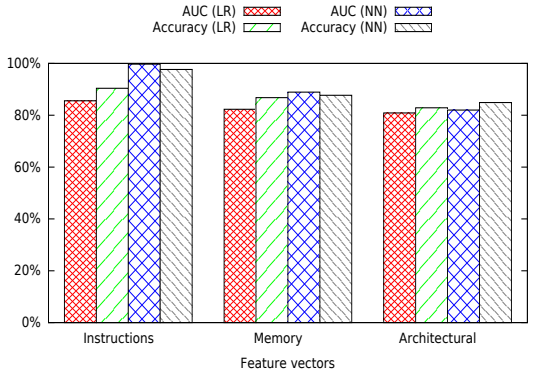


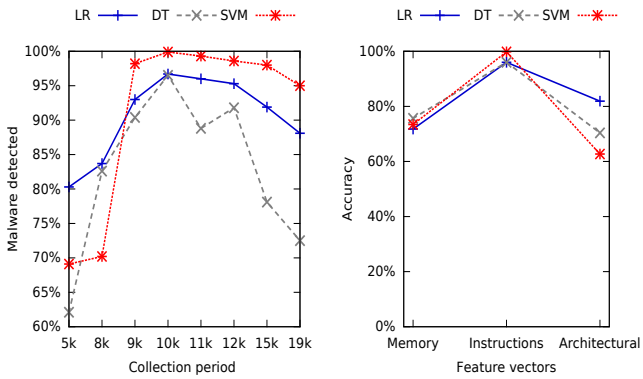
Figure 2: Performance of individual detectors

Figure 2 shows the performance of the detectors in classifying malware and regular programs using area under the curve (AUC) and the accuracy of the classification metrics for each of the detectors. Note that this figure shows the performance of the baseline HMD which we will be attempting to reverse-engineer. AUC is the area under the Receiver Operating Characteristics (ROC) curve which plots the sensitivity of the detector against the percentage of false positives; the larger the AUC, the better the classification. Accuracy refers to the point on the ROC which maximizes the accuracy (percentage of decisions that are made correctly). It is a more direct measure of performance since the HMD classification threshold will be typically set to perform at or near this optimal point.

We assume that the attacker does not know the details of how the target victim detector was trained. Thus, they do not know important configuration parameters of the detector including: (1) the size of the instruction window that is used to collect the features; the detector collects the feature over a collection window, typically measured in thousands of instructions, and then uses these features for classification; (2) the specific feature used for the classification. However, we assume that the attacker has a set of candidate features that includes the feature used by the target detector; and (3) the classification algorithm used by the target detector. Importantly, the attacker has access to a machine with a similar detector so they can test hypotheses and evaluate the success of the mimicry attacks. Next, we show how the attacker can reverse-engineer the detection period and the features used in training the target detector.

4.1 Target Detector Classification Period

The classification period refers to the size of the instruction window used to collect the classification features. Prior work [12] has shown that a classification period of about 10K instructions works well for supervised learning classifiers, but a detector may be trained with a different classification period. For this experiment, we used a classifier built using the Instruction mix feature which we assume the attacker knows (later we relax this assumption). The target detector collection period is 10K. We prepare multiple pairs of attacker testing and training datasets, using different collection periods. Next, we train a reverse-engineered detector using different data sets and evaluate its accuracy. For each of the attacker data sets, we construct three reverse engineered detectors using three different machine learning algorithms, which are: LR, decision tree (DT), and support vector machine (SVM). The results of this experiments are shown in Figure 3a. The results show that the highest accuracy for reverse-engineering for each of the machine learning algorithms used is when the collection period is the same as the victim's collection period (10K). Thus, by using an experiment such as this one, the attacker can infer the victim's collection period.



(a) Collection periods **(b) Features**

Figure 3: Reverse-engineer configurations

4.2 Target Detector Feature

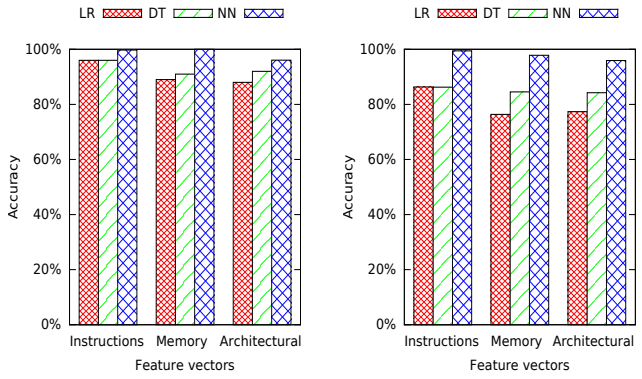
Malware detectors can be built using different features. In this subsection, we explore the possibility of reverse-engineering the feature vector used by the victim detector only by querying the victim detector. We use a detector based on the Instruction mix feature with a classification window of 10K instructions. We prepared multiple

pairs of attacker testing and training data sets using the same collection period (10K), but using different feature vectors. Next, we construct reverse-engineered detectors using the attacker training data sets labeled with the output from the victim detector as malware or regular program. For each of the attacker data sets, we constructed three detectors using different machine learning algorithms, which are: LR, Decision Tree (DT), and Support Vector Machine (SVM). The results of this experiment are shown in Figure 3b. The results show that the highest accuracy is achieved when the feature vector is the same as the victim's feature vector (Instructions). We conclude that the victim HMD features can be successfully reverse-engineered.

Note that at the correct value of the feature and period, it is possible to obtain 0-error reverse-engineering in our experiments. This is consistent with results from PAC learning theory which we overview in Section 8. Although we showed how to separately reverse-engineer the classification period (assuming that the classification feature is known) and the classification feature (assuming the classification period is known), we can also jointly reverse-engineer them both. The process involves constructing detectors with different classification features and periods, and finding the detector that maximizes the reverse-engineering accuracy.

4.3 Performance of Reverse-engineered HMD

In the next set of experiments, we evaluate the performance of the reverse-engineered detectors. We reverse-engineer LR and NN detectors, but the reverse-engineered detector is constructed using three different machine learning algorithms, which are LR, Decision Trees (DT), and NN. The results for reverse-engineering of the LR and NN detectors are shown in Figure 4a and Figure 4b respectively. The results show that NN can reverse-engineer both types of detectors with high accuracy (e.g., less than 1% error for all features for LR). The performance is somewhat lower for NN since the separation criteria used in the classification is more complex and therefore more difficult to reverse-engineer accurately.As can be expected, the simpler linear detector (LR) cannot effectively capture the non-linear behavior of NN, consistent with PAC learning theory as we discuss in Section 8.



(a) Logistic regression (b) Neural networks

Figure 4: Reverse-engineering efficiency

5 DEVELOPING EVASIVE MALWARE

After reverse-engineering the victim detectors, the next step that attackers are likely to take is to develop systematic transformations of their malware that can evade detection by these detectors. The malware developers may modify their malware in any way, to attempt to produce behavior in the feature space of the detector that causes them to be classified as normal. Possible strategies to accomplish this goal include using polymorphism to produce different binaries [58], or adding instructions that do not affect the malware state.

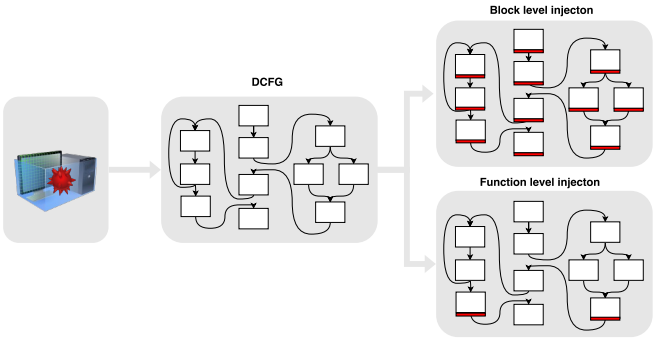


Figure 5: Methodology for generating evasive malware

Since we are working with actual malware binaries, we do not have the source available to apply general transformations. Moreover, most of the malware is obfuscated making decompilation difficult and challenge binary rewriting tools. To address these challenges, we developed a methodology to dynamically insert instructions into the malware execution in a controllable way (Figure 5). In particular, we construct the Dynamic Control Flow Graph (DCFG) of the malware by instrumenting it through the PIN tool [30]. Next, we add instructions into the control flow graph in a way that does not affect the execution state of the program.

The injected instructions must change feature vector in a controlled way based on the reversed-engineered classifier to attempt to move the malware across the classification decision boundary to be classified as normal. For the Instruction feature, injection of opcodes increases the weight of the corresponding feature directly. For the memory feature, insertion of load and store instructions with controlled distances changes the histogram of memory reference frequencies. For architectural features, the effects may not be directly controllable. For example, increasing the cache hit rate or the branch predictor success rate requires inserting code segments that will generate cache misses or predictable branches respectively. Without loss of generality, all of our experiments use the instruction feature. When attempting to change multiple features, the evasion code must combine these strategies (for example, alternating their use at different injection points).

We explored two approaches for inserting the instructions: (1) Block level: insert instructions before every control flow altering instruction. Note that the instructions inserted at that point in the program are executed every time that control flow instruction is reached (i.e., we do not change the instructions that are injected at

a particular point in the program, once they are injected); and (2) Function level: we insert instructions before every return instruction.

Random instruction injection: Although we do not expect this strategy to succeed, we first check if injecting randomly chosen instructions in the malware programs is sufficient to evade detection to establish that the injection must be specific to the detector. Each malware program data set is divided into two sets based on whether the victim detector successfully detected them without modification. Each of the data sets is modified using our framework to inject the additional instructions and retested; the results are shown in Figure 6. Clearly, injecting random instructions at the basic block level or at the function call level does not help in evading detection.

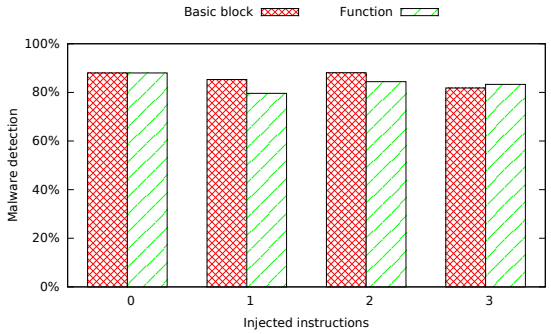


Figure 6: Detection with random instruction injection

Reverse-engineering driven instruction injection: The next set of strategies we use is to exploit the information in the structure of the reverse-engineered detector to attempt to avoid detection. Ideally, we would like to make the malware appear close to regular programs in terms of behavior so that the detector cannot successfully identify it. Based on the parameters of the reverse-engineered detector (from Section 4), we form a strategy for which instructions we can inject to make the detection difficult.

To understand the rationale behind the instruction selection, we must explain some details about the operation of LR. LR is defined by a vector θ that identifies the linear separation between the points being classified. The weights of the elements of the vector determine the relative importance of these elements. Since we are only adding instructions, we pick the instructions whose weights are negative to move the malware towards the other side of the separation line. In this first strategy, we inject only the instruction with the least weight in the vector.

Figure 8a, shows the percentage of malware detected by both the original and reverse-engineered detectors, after injecting the malware using the information for the reverse engineered detector at the basic block and at the function call levels. We observe that the modified malware evades detection by both detectors.

We conduct a similar experiment for the NN detectors, where the classifier is not clearly defined by a single vector and the separation plane is not linear. We develop a heuristic approach to identify candidate instructions for insertion. Figure 7 shows a NN with one hidden layer. Each circle in the Figure represents a neuron in the network; input, hidden, and output neurons. To compute the overall weight contributed by a single input we multiple out its contributions

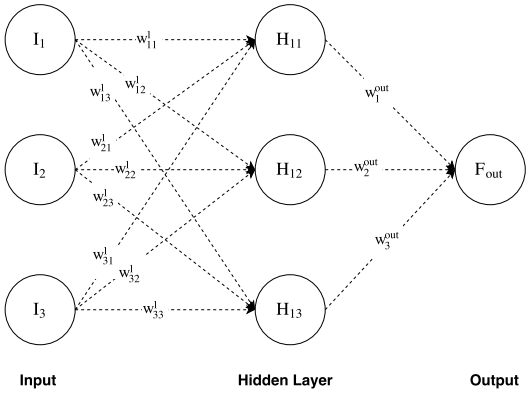


Figure 7: Neural network with one hidden layer overview

to the eventual output of the network and sum out these products. For the example in Figure 7, the weight of input I1 can be estimated as:

$$w_1 = w^1_{11} \times w^{out}_1 + w^1_{12} \times w^{out}_2 + w^1_{13} \times w^{out}_3$$

More generally, for input j, the weight is:

$$w_j = \Sigma^n_{i=1} w_{ji} \times w^{out}_i$$

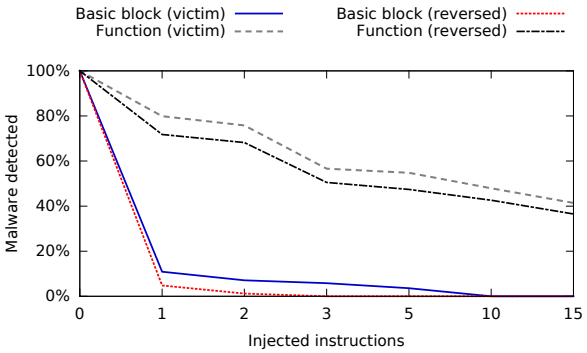
With multiple hidden layers, we must add all the factors on all the paths to which a given input contributes.

The procedure above allows us to collapse the description of the NN into a single vector that summarizes the contributions of each feature. This allows us to use the same strategies for instruction selection that we used in LR; for example, we can select the instruction with the most negative weight for insertion. However, for NN this is an approximate strategy; for LR, if we inject more of the negative weight instructions we are guaranteed to monotonically decrease as the dot product of θ and the collected feature from the malware execution becomes increasingly more negative. However, since the separation surface of NN is non-linear, the same cannot be guaranteed.

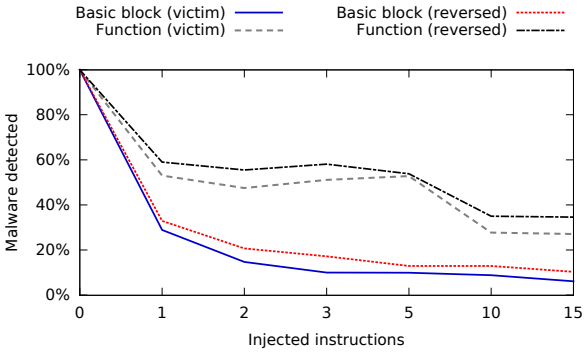
Figure 8b shows the percentage of modified malware detected by the NN victim and reverse-engineered detectors. While the evasive strategy also works in this case, it is slightly less effective; with 2 injected instructions per basic block, we can evade detection 80% of the time.

As we explained in our threat model (Section 2), we assume that the attacker is interested in maintaining the performance of the malware, and does not want to arbitrarily slow it down to evade detection. Figure 9, shows the static and dynamic overhead of injecting instructions both at the basic block level and the function call level. Inserting a single instruction at the basic block level was effective in evading detection for most malware for LR; both the static overhead (increase in the text segment of the executable) and the dynamic overhead (increase in execution time) are about 10%.

We also consider selecting the instruction for injection among all the instructions with a negative weight, with a probability proportional to the weight; we call this strategy the *weighted injection strategy*. Figure 10, shows the percentage of malware detected by the victim, after weighted injection of the malware using the information for both the reverse-engineered detector and the victim detector at



(a) Logistic regression



(b) Neural networks

Figure 8: Detection with least weight injection

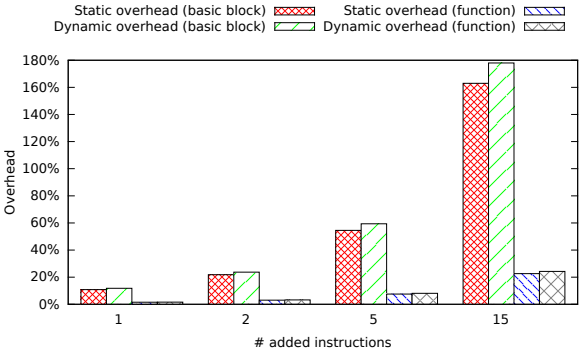


Figure 9: Injection static and dynamic overhead

the basic block and at the function call levels. The evasion success using the reverse-engineered detector is almost equal to the success when using the actual victim detector. The advantage of this strategy is that it makes it more difficult to detect the evasion if the detector is retrained as explained in the next section.

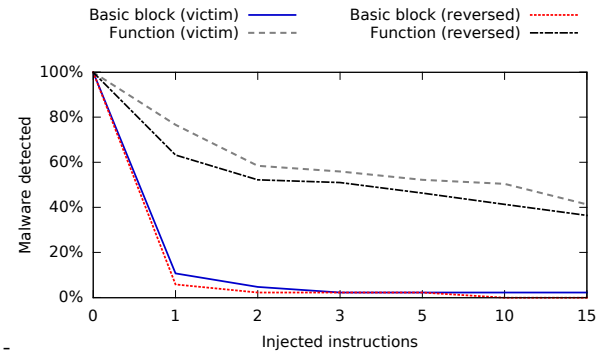


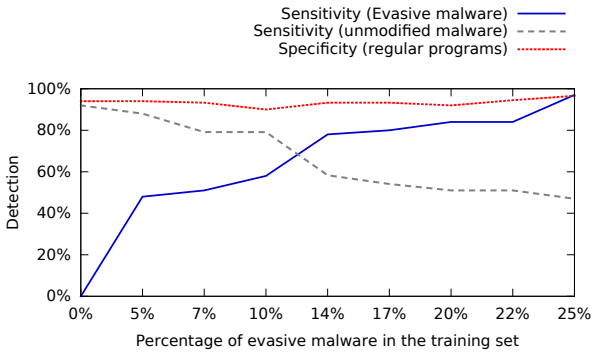
Figure 10: Detection with weighted injection (LR)

6 RETRAINING VICTIM DETECTORS

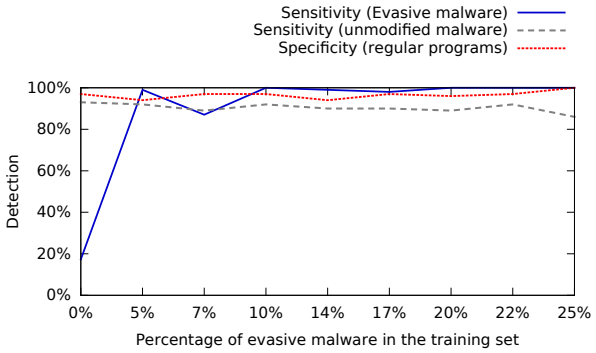
The results of the previous section demonstrate that existing HMDs that use supervised learning [12, 39] can be fairly easily evaded. The next question we consider is: if a detector is retrained with the addition of sample evasive malware, would it be able to classify them correctly? If the answer is yes, then perhaps the weights can be updated regularly to allow the detector to adapt to emerging malware. However, there is still a possibility that the retrained detector could itself be reverse-engineered and evaded again. Moreover, as attackers continue to evolve, it is not clear if the detector will eventually become ineffective due to the number of classes it is attempting to separate, or if it will converge to a classification setting that is impossible to evade.

Figure 11a, shows the effect of increasing the percentage of evasive malware programs in the training data that we use to retrain the simple LR detector. For example, the point with 10% indicates that 10% of the malware part of the training set consists of evasive malware modified with one of our evasion strategies. We see that in general, increasing the percentage of evasive malware leads to more accurate detection. Unfortunately, this comes at the price of loss of accuracy (sensitivity) for non-evasive malware, making simple retraining an ineffective strategy. It's interesting to note that the correct classification of normal programs (specificity) does not suffer.

Figure 12a illustrates why linear detectors such as LR have to sacrifice accuracy when retrained. Figure on the left shows that there is a linear separation between the malware and regular programs. Figure in the middle demonstrates that in order to evade detection, the evasive malware have to cross the separation boundary. Figure on the right shows that with retrained detector it may be impossible to find a linear separation between malware (including evasive malware) and regular programs. In contrast, non-linear classifiers such as NN (Figure 11b) are able to detect this new form of malware with high accuracy even with a low percentage of evasive malware in the retraining set. This can be achieved without affecting the detection accuracy of non-evasive malware or regular programs. Figure 12b illustrates why non-linear detectors are more effective when retrained. Even when evasive malware crosses the separation boundary of the original classification, a new non-linear boundary can be found that separates the two malware classes from normal programs. Thus, HMDs must be non-linear if we want to allow them to be retrained in response to evasive malware.

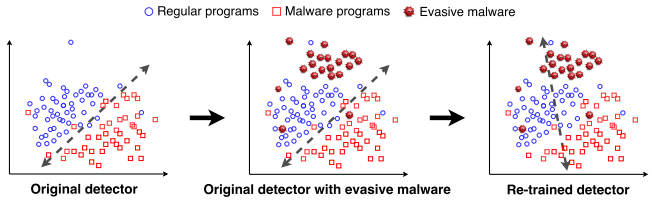


(a) Logistic regression

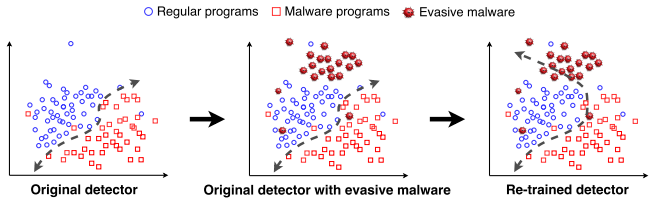


(b) Neural networks

Figure 11: Effectiveness of retraining



(a) Linear model



(b) Non-Linear model

Figure 12: Illustration of effect of retraining a linear and non-linear classifiers with data that includes evasive malware

Figure 13 shows the detection of several generations of NN detectors. In each generation, we repeat retraining of the detector by adding malware from the previous generations to the training set. The original detector in generation 1 is first evaded successfully as we see low detection for evasive malware. After we retrain, we see that evasive malware developed to evade detector 1 is now detected successfully (rightmost bar for detector 2). However, if we reverse-engineer the detector and evade it again, we can do so successfully as evidenced by the low detection of the evasive malware in the third bar for detector 2. As the retrain-evade process is continued, we expected one of two outcomes: (1) the detector will no longer be able to classify; or (2) the decision boundary will tighten and malware will no longer be able to evade. The second outcome occurred: after 7 generations, the detector can no longer be trained successfully as malware and normal programs became inseparable using our NN. There are two possible explanations: (1) the feature is not sufficiently discriminative, and its possible to turn malware to be similar to normal programs with respect to this feature. Note that in each successive generation, the overhead is increased, and this level of overhead may not be acceptable to the attacker; or (2) NN could no longer represent the complex decision boundary between the different classes of evasive malware and normal programs, similar to how LR failed after one generation.

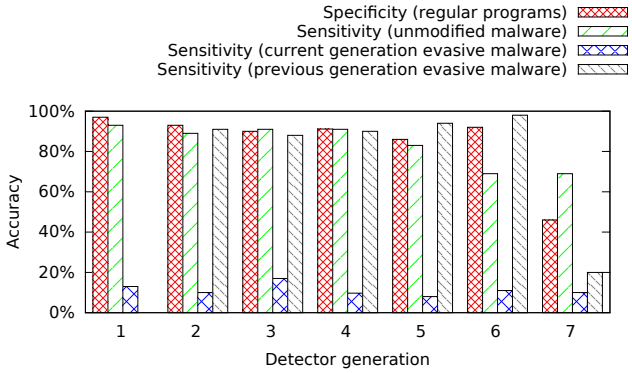


Figure 13: Performance of NN detector for each retraining on evasive malware generation

7 EVASION-RESILIENT HMDS

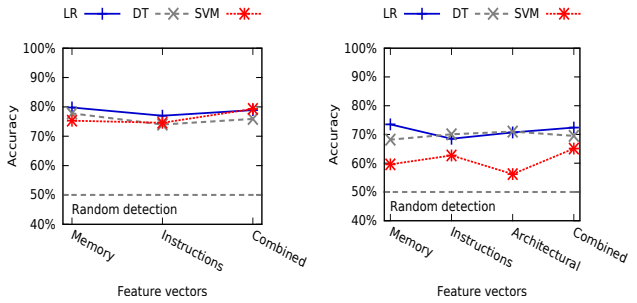
Although retraining the detectors can allow the updated detectors to detect evasive malware that has representatives in the training set, we showed retraining may eventually fail as attackers continue to evade. Moreover, retraining cannot detect novel evasive malware: even after retraining they can be reverse-engineered and evaded.

In this section, we introduce a new class of evasion-resilient HMDs (RHMDs). RHMDs leverage randomization to make detectors resistant to reverse-engineering and therefore evasion. In particular, this is a strong advantage in the sense that randomization introduces an error to the reverse engineering that is bounded by a function of how often the detectors disagree; we show a proof for this claim in Section 8 based on PAC learnability theory.

We randomize two settings of the detectors: i) The feature vectors used for detection; and ii) The collection periods used in the detection. In particular, we construct detectors with these heterogeneous features and switch between them stochastically in a way that cannot be predicted by the attacker.

Our first study examines the effect of randomizing the feature vectors used for detection. We start with two detectors using the same detection period. The results of this experiment are shown in Figure 14a. We reverse-engineer the detector using two of the original feature vectors as well as a combination of them. In particular, the point in the figure marked "combined" represents reverse-engineering with a combined detector using the union of the two feature vectors. Using an RHMD with two detectors, reverse-engineering the detector becomes substantially more difficult because the model now includes two diverse detectors which are selected randomly. The diversity can be further expanded by using a pool of three detectors — the results of this approach are shown in Figure 14b. Again, the combined point on the figure refers to a reverse-engineering attempt using the union of the three feature vectors of the three individual detectors. As seen from the results, reverse-engineering becomes harder with increased diversity.

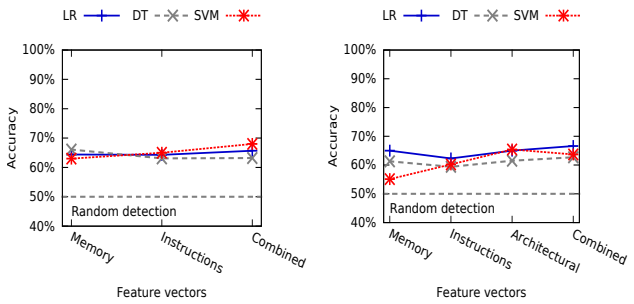
To further increase detector diversity, we construct detectors with two different collection periods (10K cycles period and 5K cycles period), resulting in a pool of six detectors, which are randomly chosen by the detection logic. The results are presented in Figure 15. Consistent with the previous trend, additional diversity makes reverse-engineering even more difficult. Note that having detectors operating on the same features with different period does not substantially increase the hardware complexity; the different weight for the two detectors must be kept separately, but the collection logic and the detector evaluation logic is shared.



(a) Two feature vectors (b) Three feature vectors

Figure 14: RHMD reverse engineering (features)

Having reverse-engineered the detector, we use our evasion framework to inject instructions to evade it. Given that the reverse-engineering becomes inaccurate in RHMDs and given the random switch between the individual detectors, the constructed evasive malware can no longer hide from detection (Figure 16). It is interesting to note that the higher the diversity of the detector, the more resilient it is to evasion, consistent with PAC learnability theory discussed in the next section. These results demonstrate that this approach to constructing HMDs provides resilience to evasive malware. The average detection accuracy of the RHMD without evasion (Figure 16 with 0 injected instructions) is equal to the average accuracy of its base detectors since the randomization selects between the detectors with equal probability. Thus, the average loss of detection due to randomization is the difference of accuracy between the most accurate detector and the average of all base detectors.



(a) Two feature vectors (b) Three feature vectors

Figure 15: RHMD reverse engineering (features and periods)

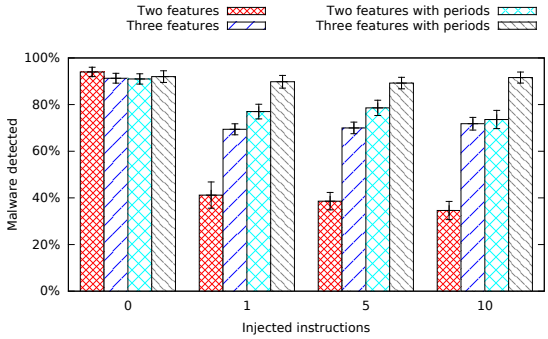


Figure 16: RHMD evasion resilience

To evaluate the overhead of implementing the detectors for online detection [39], the proposed resilient detectors were implemented using Verilog, as an extension of an open source x86-compatible core (AO486) [38] to estimate the overhead. The detectors collect information from the commit stage of the pipeline and apply the detection logic at the detection period. After synthesizing the new core implementation on an FPGA board for a configuration with three detectors corresponding to the three features with the same period, we observed that the area and power increase is modest: 1.72% and 0.78% respectively. Note that the resilient detectors can also be used to make offline detection [12] resilient to evasion.

8 THEORETICAL BASIS FOR RHMD

This section provides theoretical support for the resilience of RHMD for evasion based on probably approximately correct (PAC) learnability theory [34]. In particular, we show that randomized classification is inherently more difficult to be reverse-engineered than a deterministic classifier (even one with arbitrarily high complexity).

8.1 Learnability of Deterministic Classification

Consider a learning system (a defender) that uses a single classifier to detect malware from normal programs. Consider also another *reverse engineering* learning system (an attacker) that uses data of past classification collected, for example, by repeatedly querying the defender classifier, to determine with high accuracy the nature of the defender classifier.

Formally, let H be the class of possible classifiers (also called hypothesis class) a learning system considers. Let P be the probability distribution over instances (x, y), where x is an input feature vector, and y is a label in $\{0, 1\}$. We assume below that $y = \bar{h}(x)$, i.e., a deterministic function that gives the true label of x. For any $h \in H$, let $e(h) = \Pr_{x \in P}[h(x) \neq \bar{h}(x)]$ be the expected error of h w.r.t. P. We define $e_H = \inf_{h \in H} e(h)$ as the optimal (smallest) error achievable by any function $h \in H$. Let $D = \{(x_1, y_1), ..., (x_m, y_m)\}$ be a training data set of size m generated according to P and $\mathscr{D}$ be the set of all possible D. A learning algorithm is a function $L : \mathscr{D} \rightarrow H$ which produces a classifier $\hat{h} \in H$ given a training set D.

DEFINITION 1. *A hypothesis class H is learnable if there is a learning algorithm L for H with the property that for any $\varepsilon, \delta \in (0, 1/2)$ and distribution P there exists a training sample size $m_0(\varepsilon, \delta)$, such that for all $m \geq m_0(\varepsilon, \delta)$, $\Pr_{D \in \mathscr{D}}[e(L(D)) \leq e_H + \varepsilon] \geq 1 - \delta$, i.e., L will with probability at least $(1 - \delta)$ output a hypothesis $\hat{h} \in H$, whose error on P is almost $(e_H + \varepsilon)$. H is efficiently learnable if $m_0(\varepsilon, \delta)$ is polynomial in $1/\varepsilon$ and $1/\delta$, and L runs in time polynomial in m, $1/\varepsilon$ and $1/\delta$.* [1]

The definition of PAC learnability above says that a hypothesis class H is efficiently learnable (i.e., learning is easy) if we can compute with high probability an approximately optimal candidate from this class given a polynomial number of samples. The error bound $(e_H + \varepsilon)$ for an approximately correct classifier $\hat{h} \in H$ consists of two components. ε becomes arbitrarily small and hence $e(\hat{h})$ approaches e_H when the number of training samples increases polynomially w.r.t. $1/\varepsilon$. The other component e_H depends on the learning bias [34] about H. That is the set of assumptions that the learner makes (e.g., the type of classifiers and the underlying features). With a good choice of H, e_H can be arbitrary small; e_H is zero if H contains the true classifier $\bar{h}$. We observed the implications of this result in Section 4 when the correct feature and detection period lead to the highest accuracy reverse engineered classifier.

The concept of PAC learnability applies to the learning tasks by both the defender and the attacker, with one caveat: for the defender, the goal is to correctly predict the *true* label of an instance (i.e., $y = \bar{h}(x)$); while for the attacker, the goal is to correctly predict the label of an instance *assigned* by the defender's classifier (i.e., $y = \hat{h}(x)$). As shown in [52], efficient learning for $\bar{h}$ by the defender implies efficient learning for $\hat{h}$ (called efficient reverse-engineering) by the attacker. Suppose that the defender has learned $\hat{h}$ from an efficiently learnable H. Provided the attacker identifies the type and features of the classifiers in H, then $\hat{h}$ is contained in the hypothesis class used by the attacker, and $e_H = 0$, i.e., the distribution P over $(x, \bar{h}(x)$ can be efficiently reverse-engineered with arbitrary precision [52]. Without prior knowledge of H, the attacker can tune its hypothesis class based on the error rate on the training data collected over repeated queries.

These results support the reverse-engineering experiments in Section 4. In particular, the analysis shows that *reverse-engineering a deterministic classifier is "easy" in practice regardless of the complexity of the defender's classifier*. Increasing the complexity of the defender's classifier can make it more costly to reverse-engineer it, but will not change the outcome of the arms race between the defender and attacker with respect to the difficulty of evasion.

8.2 Learnability of Randomized Classification

We now consider a defender that uses randomized classification such as the model used in RHMD. As before, consider a distribution P over instances (x, y), with $y = \bar{h}(x)$ as the ground truth. Suppose that we have n hypothesis classes H_i, all efficiently learnable. Let $\hat{h}_i \in H_i$ be the classifier learned from these classes, respectively, with the corresponding error rate $e(\hat{h}_i)$. Additionally, let $\triangle_{i,j} = \Pr_{x \in P}[\hat{h}_i(x) \neq \hat{h}_j(x)]$ for all i, j: that is, $\triangle_{i,j}$ measures the difference between two classifiers $\hat{h}_i(x)$ and $\hat{h}_j(x)$ over the data distribution. Consider a space of policies parametrized by $p_i \in [0, 1]$ with $\sum_i p_i = 1$, where we choose $\hat{h}_i \in H_i$ with probability p_i. Let $\vec{p}$ denote the corresponding probability vector. Then, a policy $\vec{p}$ induces a distribution $Q_{\vec{p}}$ over (x, z), where $z = \hat{h}_i(x)$ with probability p_i. The defender will incur a baseline error rate of $e_{\vec{p}}(h) = \Pr_{x \in Q_{\vec{p}}}[\hat{h}_i(x) \neq \bar{h}(x)] = \sum_i p_i e(\hat{h}_i)$ if there is no reverse-engineering effort.

Suppose the attacker observes a sequence of data points from $Q_{\vec{p}}$, and tries to efficiently learn the hypothesis class $H = \cup_i H_i$. For any $h \in H$, let $e_{\vec{p}}(h) = \Pr_{x \in Q_{\vec{p}}}[h(x) \neq \hat{h}_i(x)] = \sum_i p_i e(h)$, the expected error of h w.r.t. $Q_{\vec{p}}$, and we define $e_{\vec{p},H} = \inf_{h \in H} e_{\vec{p}}(h)$ as the optimal (smallest) error achievable by any function $h \in H$ under a policy $\vec{p}$. Definition 1 naturally extends to the randomized setting: in particular, the distribution P becomes $Q_{\vec{p}}$ and the error bound $(e_H + \varepsilon)$ becomes $(e_{\vec{p},H} + \varepsilon)$.

THEOREM 1. *Suppose that each H_i is efficiently learnable, and $\hat{h}_i \in H_i$ be the classifier learned from these classes by a defender, respectively, with the corresponding error rate $e(\hat{h}_i)$. Then, any distribution $Q_{\vec{p}}$ over (x, z) can be efficiently reverse engineered, with $e_{\vec{p},H}$ bounded by $\min_i \sum_{j \neq i} p_i \triangle_{i,j} \leq e_{\vec{p},H} \leq 2(\max_i e(\hat{h}_i))$.* [2]

This theorem shows that on the one hand, even with randomization, reverse-engineering is easy as long as all classifiers among which the defender randomizes accurately predict the target - that is $\max_i e(\hat{h}_i)$, the maximal error among the n classifiers, is arbitrarily small. On the other hand, the attacker's error depends directly on the difference among the classifiers, which can be significant if at least some of the classifiers are not very accurate, allowing them to disagree more often. According to the error bound $(e_{\vec{p},H} + \varepsilon)$, even though ε becomes arbitrarily small as the number of queried samples increases, the defender will inevitably suffer from an error caused by $e_{\vec{p},H}$. This error can be high; for example, when randomizing two classifiers of error 0.2 and 0.1 with $p_1 = p_2 = 0.5$, $e_{\vec{p},H}$ is in [0.15, 0.4]. In contrast, in the deterministic setting, e_H can be 0. For our experiments with the pool of six detectors we measured the error to be around 25% on our testing dataset.

The above theorem also suggests a trade-off between the accuracy of the defender under no reverse-engineering vs. the susceptibility to being reverse-engineered: using low-accuracy but high-diversity classifiers allow the defender to induce a higher error rate on the attacker, but will also degrade the baseline performance against the target. To combat reverse engineering effort, we propose to randomize among a set of low-complexity, low-accuracy classifiers (e.g.,

[1] This definition is taken, in a slightly extended form, from [34].

[2] This theorem is formed by combining Theorem 2.2 and Corollary 2.3 (with detailed proofs) from [52].

logistic regression), rather than deploying a single high-complexity, high-accuracy classifier (e.g., deep neural network or random forest). The former is also more suitable for a hardware implementation than the latter. Although this low accuracy applies to the classification of each individual period, we raise the overall accuracy of the detector by averaging the decisions across multiple intervals.

8.3 Evasion Without Reverse Engineering

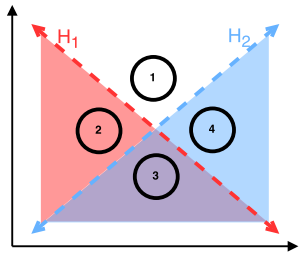


Figure 17: Impact of Randomization on Evasion

Our threat model assumes that an attacker needs to reverse engineer a detector before evading it. The theoretical resilience claims on RHMDs rely on the difficulty of this reverse engineering. In this section, we consider whether it is possible to evade the detector without reverse engineering. To provide intuition, we start with Figure 17, which shows the decision boundaries of two diverse base detectors learned from hypothesis classes H_1 and H_2. The two decision boundaries are not mutually exclusive (H_1 malware regions are 2, 3 and H_2 malware regions are 3, 4). To fool both detectors, the malware has to move to region 1 which both detectors treat as normal. Note that these decision boundaries represent hyperplanes in an n-dimensional feature space for LR, and complex surfaces in the same space for NN. Therefore, as we increase diversity the target area for evasion gets smaller. Thus, provided that detectors are diverse, making random insertion guesses is unlikely to succeed and expensive to validate. Note that evasion must succeed continuously across consecutive detection windows, which complicates attempts to incrementally evade the detector.

This example also provides intuition on why randomization complicates reverse engineering (as shown by Theorem 1). The attacker has to suffer a significantly increased error ($e_{\vec{p},H}$) if she tries to learn a decision boundary from the same hypothesis classes adopted by the defender. Otherwise, she has to learn a decision boundary of a higher complexity class which requires exponentially larger number of samples.

If the attacker knows precisely the configuration of the base detectors of an RHMD, we verified that it is possible to evade it, for example, by iteratively evading each. This approach incurs a high overhead since instructions need to be injected to evade each of the detectors. We do not consider this case as part of our threat model. Resilience in this case may be achieved if we make the decision boundary of the RHMD non-stationary. This can be accomplished by having a large set of candidate features and periods, of which a random subset is used for the RHMD at any given time. This is an interesting area for future research.

9 RELATED WORK

In this section, we discuss related work organized into two main parts. First, we review related work in malware anomaly detection. In the second part, we discuss adversarial classification and some important recent results in that domain. We show that these results are consistent with our results both in terms of reverse engineering and the use of randomization as a defense.

9.1 Malware Detection

In general, malware detection techniques are either static (focusing on the structure of a program or system) or dynamic (analyzing the behavior during execution) [22]. Static approaches can be easily evaded using program obfuscation or simple code transformations that preserve the function of the malware but make it not match the patterns known to the scanner [35, 36]. On the other hand, the advantage of dynamic malware detection is that it is resilient to metamorphic and polymorphic malware [32, 35]; it can even detect previously unknown malware. However, disadvantages include a typically high false positive rate, and a high monitoring cost during run-time.

A number of works have looked at using low-level architecture features for malware detection such as frequency of opcodes use in malware [7], evaluation of opcode sequence signatures [45, 56] and opcode sequence similarity graphs [44], which consider offline analysis. Further, Demme et al. [12] proposed collecting performance counter statistics for programs and malware under execution and used them to show that offline detection of malware is effective. Then, a real-time hardware malware detector was built by Ozsoy et al. [39]. Tang et al. [50] used unsupervised learning to detect malware exploits, which will make the regular program deviate from the baseline execution model. Kazdagli et al [25] identified some pitfalls in training and evaluating HMDs for mobile malware, and proposed several improvements to them.

Khasawneh et al. used ensemble learning to improve the accuracy of HMDs [27]. Superficially, ensemble learning is similar to RHMD since it combines the output of multiple diverse detectors through a combiner function such as majority voting to improve the overall detection performance. However, since ensemble classifiers are deterministic, they can be reverse engineered and evaded. In contrast, the *stochastic switching* between individual detectors in RHMD makes both reverse-engineering and evasion difficult with a difficulty that increases with the number and diversity of the individual detectors. Smutz et al. also studied the use of an ensemble for PDF malware detection [49]; when the baseline detectors disagree, they consider this a possible indicator of evasive malware.

Although hardware-supported malware detection offers many advantages as it can be always on and has a low overhead on both power and performance, malware developers will try to come up with evasive techniques to cross this line of defense (HMDs). In this paper, we show that without hardening HMDs, it is possible to reverse engineer and evade them, bringing into question the effectiveness of HMDs. Our work also explores whether retraining can be used to continue to track malware evasion, as well as the construction of resilient hardware malware detectors. With these results we believe evasion resilient HMDs become practical, bringing such solutions closer to practical deployment.

9.2 Adversarial Classification

Several other studies have looked at attacking machine learning models. Attacks can be classified into two types: poisoning and evasion attacks [3]. In poisoning attacks, the adversary focuses on injecting malicious samples in the training data as an attempt to influence the accuracy of the model [6, 28]. For evasion attacks, similar to our own, the adversary crafts input samples that aim to be misclassified by the model [4, 18, 29, 41, 55]. Several evasion attacks were studied in the image classification field. An adversary can make changes to the pixels of an image to cause miss-classification of the image but will not change the visibility of the image to the human eye [4, 18, 41]. Since images have high entropy they can be easily manipulated without changing the appearance of the image. On the other hand, in the malware detection domain, manipulating malware programs have different challenges since the functionality of the malware needs to be preserved. Evasion attacks in contexts outside image classification have also been considered. Such attacks are called *mimicry* attacks [8, 53]. Although recent studies [5, 11, 52] have provided theoretical grounds for randomization as a possible solution in adversarial classification, practical algorithms have yet to be developed for this problem.

Related to our evasion attack on malware detectors, researchers recently proposed evasive attacks on PDF malware detectors [29, 55]. These works consider static classifiers using structural features present in the PDF image. In contrast, our contribution targets detectors for a wide range of malware and we consider run-time anomaly detection using microarchitectural features. Besides the different nature of the application, our work makes a number of contributions relative to these recent papers including showing how to reverse engineer the classifiers, reverse-engineering driven instruction injection to evade detection (they use random modifications), exploring the impact of retraining, and providing theoretical insights based on PAC theory into the structure of the problem. Moreover, these studies do not explore resilient classification.

Similar to the reverse engineering component of our work, Tramèr et al. [51] were able to reverse-engineer machine learning models against production Machine Learning-as-a-service (MLaaS) providers. However, they assumed that they know the features used by the target classifier. In addition, Shokri et al. [47] were able to use reverse-engineered models to perform a membership inference attack (given a data record and black-box access to a model, determine if the record was in the model's training dataset) against MLaaS providers. In both works, they attempt reverse engineering using random noise [47, 51]. We believe that this approach does not work in our threat model where we do not have access to the classifier confidence, and where classification is a continuous process, which make it difficult to assess incremental changes.

10 CONCLUDING REMARKS

Recently proposed Hardware Malware Detectors have demonstrated remarkable accuracy in classifying malware using low-level features. In one model, when implemented in hardware, they can help monitor all programs as they run to detect malware with high accuracy, and at a cost orders of magnitude lower than software monitoring. There is evidence that these detectors will be incorporated in commercial processors [43].

If HMDs are widely deployed, we must expect that attackers will attempt to evade detection as is the case in any adversarial setting. We believe that this is the first paper that considers this issue in detail for HMDs. In particular, we showed that the attacker can accurately reverse-engineer earlier proposed detectors. Moreover, once the detector is reverse-engineered, we demonstrated simple evasive techniques that can successfully hide malware from detection. Although existing HMDs admit that these detectors may be vulnerable to evasion, these results conclusively confirm that malware can evade such detectors, rendering them ineffective.

We considered that detectors can be retrained to capture evasive malware once samples become known (similar to the current practice of updating virus signatures). For LR, such retraining was not effective: considering evasive malware compromised the classification performance on normal malware; due to linear separation, it is not possible to produce LR detectors that successfully catch both. In contrast, NN could easily be retrained to detect the evasive malware. Thus, non-linear classifiers need to allow the detectors to be retrained to capture emerging malware. However, after several rounds of evade-retrain game, the NN classifier could also no longer be effectively retrained.

We proposed resilient HMDs that switch randomly between a diversity of detectors. We showed that such detectors can resist reverse-engineering and complicate evasion. We showed both empirically, and from PAC learnability theory, that their resilience increases with the number and diversity of the individual detectors available to select from. With this class of resilient HMDs, hardware malware detection becomes a promising direction in malware detection.

11 ACKNOWLEDGEMENT

The work in this paper is partially supported by National Science Foundation grants CNS-1422401, CNS-1619322 and CNS-1617915.

REFERENCES

[1] Sherly Abraham and InduShobha Chengalur-Smith. 2010. An overview of social engineering malware: Trends, tactics, and implications. *Technology in Society* 32, 3 (2010), 183–196.
[2] Sruthi Bandhakavi, Samuel T King, Parthasarathy Madhusudan, and Marianne Winslett. 2010. VEX: Vetting Browser Extensions for Security Vulnerabilities.. In *USENIX Security Symposium*, Vol. 10. 339–354.
[3] Marco Barreno, Blaine Nelson, Anthony D Joseph, and JD Tygar. 2010. The security of machine learning. *Machine Learning* 81, 2 (2010), 121–148.
[4] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. 2013. Evasion attacks against machine learning at test time. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 387–402.
[5] Battista Biggio, Giorgio Fumera, and Fabio Roli. 2008. Adversarial pattern classification using multiple classifiers and randomisation. *Structural, Syntactic, and Statistical Pattern Recognition* (2008), 500–509.
[6] Battista Biggio, Blaine Nelson, and Pavel Laskov. 2012. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389* (2012).
[7] Daniel Bilar. 2007. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics* 1, 2 (2007), 156–168.
[8] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. 2007. An efficient technique for preventing mimicry and impossible paths execution attacks. In *Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa*. IEEE, 418–425.
[9] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. 2011. Measuring Pay-per-Install: The Commoditization of Malware Distribution.. In *Usenix security symposium*. 15.
[10] Jie Chen and Guru Venkataramani. 2014. Cc-hunter: Uncovering covert timing channels on shared processor hardware. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 216–228.

[11] Richard Colbaugh and Kristin Glass. 2012. Predictive defense against evolving adversaries. In *Intelligence and Security Informatics (ISI), 2012 IEEE International Conference on*. IEEE, 18–23.

[12] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. 2013. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 559–570.

[13] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS)*. 51–62.

[14] Harris Drucker, Donghui Wu, and Vladimir N Vapnik. 1999. Support vector machines for spam categorization. *IEEE Transactions on Neural networks* 10, 5 (1999), 1048–1054.

[15] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2012. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)* 44, 2 (2012).

[16] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016), 10.

[17] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. 2003. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review*, Vol. 37. ACM, 193–206.

[18] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[19] Guofei Gu, Phillip Porras, Vinod Yegneswaran, Martin Fong, and Wenke Lee. 2007. BotHunter: Detecting Malware Infection Through IDS-driven Dialog Correlation. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*.

[20] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[21] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. 2015. Understanding contention-based channels and using them for defense. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 639–650.

[22] Nwokedi Idika and Aditya P Mathur. 2007. A survey of malware detection techniques. *Purdue University* 48 (2007).

[23] Mehmet Kayaalp, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Aamer Jaleel. 2016. A high-resolution side-channel attack on last-level cache. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 1–6.

[24] Mehmet Kayaalp, Timothy Schmitt, Junaid Nomani, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2013. SCRAP: Architecture for signature-based protection from code reuse attacks. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 258–269.

[25] Mikhail Kazdagli, Ling Huang, Vijay Reddi, and Mohit Tiwari. 2016. EMMA: A New Platform to Evaluate Hardware-based Mobile Malware Analyses. *CoRR* abs/1603.03086 (2016). http://arxiv.org/abs/1603.03086

[26] Mikhail Kazdagli, Vijay Janapa Reddi, and Mohit Tiwari. 2016. Quantifying and improving the efficiency of hardware-based mobile malware detectors. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–13.

[27] Khaled N. Khasawneh, Meltem Ozsoy, Caleb Donovick, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2015. Ensemble Learning for Low-Level Hardware-Supported Malware Detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404 (RAID 2015)*. Springer-Verlag New York, Inc., New York, NY, USA, 3–25. https://doi.org/10.1007/978-3-319-26362-5_1

[28] Marius Kloft and Pavel Laskov. 2010. Online anomaly detection under adversarial impact. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 405–412.

[29] Pavel Laskov and Nedim Šrndić. 2014. Practical evasion of a learning-based classifier: A case study. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 197–211.

[30] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[31] Malwaredb. 2010. Liste Malware. (2010). Available online (last accessed, May 2015): www.malwaredb.malekal.com.

[32] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. 2007. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 431–441.

[33] Gary McGraw and Greg Morrisett. 2000. Attacking malicious code: A report to the Infosec Research Council. *IEEE software* 17, 5 (2000), 33–41.

[34] Tom M Mitchell. 1997. *Machine Learning*. McGraw-Hill.

[35] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*. IEEE, 421–430.

[36] Carey Nachenberg. 1997. Computer virus-antivirus coevolution. *Commun. ACM* 40, 1 (1997), 46–51.

[37] Hoda Naghibijouybari and Nael Abu-Ghazaleh. 2017. Covert Channels on GPGPUs. *IEEE Computer Architecture Letters* 16, 1 (2017), 22–25.

[38] Aleksander Osman. 2014. The AO486 project. (2014). http://opencores.org/project,ao486

[39] Meltem Ozsoy, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2015. Malware-aware processors: A framework for efficient online malware detection. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 651–661.

[40] Meltem Ozsoy, Khaled N Khasawneh, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2016. Hardware-Based Malware Detection Using Low-Level Architectural Features. *IEEE Trans. Comput.* 65, 11 (2016), 3332–3344.

[41] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. 2016. Practical black-box attacks against deep learning systems using adversarial examples. *arXiv preprint arXiv:1602.02697* (2016).

[42] Colin Percival. 2005. Cache missing for fun and profit. (2005).

[43] Qualcomm 2016. Qualcomm Smart Protect Technology. (2016). Last Accessed July 2016 from https://www.qualcomm.com/products/snapdragon/security/smart-protect.

[44] Neha Runwal, Richard M Low, and Mark Stamp. 2012. Opcode graph similarity and metamorphic detection. *Journal in Computer Virology* 8, 1-2 (2012), 37–52.

[45] Igor Santos, Felix Brezo, Javier Nieves, Yoseba K Penya, Borja Sanz, Carlos Laorden, and Pablo G Bringas. 2010. Idea: Opcode-sequence-based malware detection. In *Engineering Secure Software and Systems*. Springer, 35–43.

[46] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of CCS 2007*. ACM Press, 552–61.

[47] Reza Shokri, Marco Stronati, and Vitaly Shmatikov. 2016. Membership inference attacks against machine learning models. *arXiv preprint arXiv:1610.05820* (2016).

[48] Ed Skoudis and Lenny Zeltser. 2004. *Malware: Fighting malicious code*. Prentice Hall Professional.

[49] Charles Smutz and Angelos Stavrou. 2016. When a Tree Falls: Using Diversity in Ensemble Classifiers to Identify Evasion in Malware Detectors. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*.

[50] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. 2014. Unsupervised anomaly-based malware detection using hardware features. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*. 109–129.

[51] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2016. Stealing machine learning models via prediction apis. In *USENIX Security*.

[52] Yevgeniy Vorobeychik and Bo Li. 2014. Optimal randomized classification in adversarial settings. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*. International Foundation for Autonomous Agents and Multiagent Systems, 485–492.

[53] Ke Wang, Janak Parekh, and Salvatore Stolfo. 2006. Anagram: A content anomaly detector resistant to mimicry attack. In *Recent Advances in Intrusion Detection*. Springer, 226–248.

[54] Kenneth C Wilbur and Yi Zhu. 2009. Click fraud. *Marketing Science* 28, 2 (2009), 293–308.

[55] Weilin Xu, Yanjun Qi, and David Evans. 2016. Automatically evading classifiers. In *Proceedings of the 2016 Network and Distributed Systems Symposium*.

[56] Guanhua Yan, Nathan Brown, and Deguang Kong. 2013. Exploring discriminatory features for automated malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 41–61.

[57] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*. 116–127.

[58] Ilsun You and Kangbin Yim. 2010. Malware Obfuscation Techniques: A Brief Survey.. In *Proc. International Conference on Broadband, Wireless Computing, Communications and Applications (BWCCA)*. 297–300.