Malware-Aware Processors: A Framework for Efficient Online Malware Detection

Meltem Ozsoy[1], Caleb Donovick[1], Iakov Gorelik[1], Nael Abu-Ghazaleh[2], and Dmitry Ponomarev[1]

[1]State University of New York at Binghamton
[2]University of California, Riverside
{mozsoy,cdonovi1,igoreli1,dima}@cs.binghamton.edu, naelag@ucr.edu

Abstract

Security exploits and ensuant malware pose an increasing challenge to computing systems as the variety and complexity of attacks continue to increase. In response, software-based malware detection tools have grown in complexity, thus making it computationally difficult to use them to protect systems in real-time. Therefore, software detectors are applied selectively and at a low frequency, creating opportunities for malware to remain undetected. In this paper, we propose Malware-Aware Processors (MAP) - processors augmented with an online hardware-based detector to serve as the first line of defense to differentiate malware from legitimate programs. The output of this detector helps the system prioritize how to apply more expensive software-based solutions. The always-on nature of MAP detector helps protect against intermittently operating malware. Our work improves on the state of the art in the following ways: (1) We define and explore the use of sub-semantic features for online detection of malware. (2) We explore hardware implementations and show that simple classifiers appropriate for such implementations can effectively classify malware. We also study different classifiers, develop implementation optimizations, and explore complexity to performance trade-offs. (3) We propose a two-level detection framework where the hardware classifier prioritizes the work of a more accurate but more expensive software defense mechanism. (4) We integrate the MAP implementation with an open-source x86-compatible core, synthesizing the resulting design to run on an FPGA.

1. Introduction

Computing systems are under continuous attacks by increasingly motivated and sophisticated adversaries. These attackers use vulnerabilities to compromise systems and deploy malware. Malware is a general term for malicious software encompassing several types of programs that vary in their intent and propagation methods [58, 41]. Malware threat to systems continues to increase: according to McAfee (fourth quarter report of 2013 [40]), their malware zoo has nearly 200 million malware samples, with over 25 million new samples added in the quarter; this is a rate of over 3 new samples per second! The same report shows that mobile malware has also arrived in force — there are 3.7 million Android malware samples, with over 800 thousand of those added in the quarter.

Although significant effort continues to be directed at making systems more difficult to attack, the number of exploitable vulnerabilities is overwhelming. Attackers obtain privileged access to systems in a variety of ways, such as drive-by-downloads with websites exploiting browser vulnerabilities [9], network-accessible vulnerabilities [57] or even social engineering attacks [5]. Attackers need only to succeed in exploiting a single vulnerability to compromise a system completely. Thus, it is essential to invest in approaches to detect malware so that infections can be stopped and damage contained.

Because malware is increasing in sophistication, its detection has become more difficult. An increasing challenge faced by malware detection is resource related — the resource requirements make it prohibitive to monitor every application all the time. Typical techniques proposed for online malware detection include VM introspection [27], dynamic binary instrumentation [21], information flow tracking [68], and software anomaly detection [29]. These solutions each have coverage limitations and introduce substantial overhead (e.g., 10x slowdown for information flow tracking is typical in software [67]). The problem is especially critical for mobile environments where the energy cost of detection imposes limits on the effort that a system can dedicate to online malware detection. These difficulties often limit malware detection to static signature-based virus scanning tools [22] which have known limitations [44] that allow attackers to bypass them and remain undetected.

Demme et al. [19] recently showed that machine learning approaches can successfully classify malware from normal programs based on features obtained from sampling CPU performance counters. They used off-line analysis based on complex data mining algorithms to show that this classification is possible after the fact, with the complete trace of the program behavior available for analysis. Building on this evidence, we motivate and present MAP (Malware-Aware Processor) — a hardware-supported sub-semantic detector that can classify malware from normal programs in real-time. We use the term sub-semantic to mean architectural information about an executing program that does not require modeling or detecting program semantics. Sub-semantic information includes architectural events such as cache miss rates, branch prediction outcomes, dynamic instruction mixes, and data reference patterns.

MAP builds on the work by Demme et al. in the following ways:

- **Real-time malware detection:** real-time detection includes a new time-series component where successive decisions from the classifier are evaluated to detect anomalous

978-1-4799-8930-0/15/$31.00 ©2015 IEEE

behavior. We explore simple Exponentially Weighted Moving Average (EWMA) approach for detecting malware. In contrast, the offline problem uses after-the-fact analysis with the benefit of the complete data for the process lifetime. Thus, the online detection results demonstrate (for the first time) that classification over windows of execution can also separate malware from normal programs.

- **Hardware implementation using simpler classifiers:** a hardware implementation has significant benefits over software detection for this problem. First, direct access to hardware features is possible at low cost. Hardware detection can be always on, for all programs, with low complexity and power overhead. In contrast, software implementations require additional resources, are limited by the available performance counters, and incur significant costs. On the other hand, hardware implementations necessitate simpler classifiers than those available in software. This paper demonstrates that such simple classifiers can be effectively used to detect malware.
- **Exploration of complexity/detection tradeoffs:** we investigate both linear classifiers as well as neural network based classifiers. We explore the tradeoff between complexity and classification effectiveness. We also study a number of optimizations to the hardware implementation of both the base classifier and the time-series detector.
- **Two level detection framework:** False positives are likely to occur due to simple classification algorithms and the low-level features used. Thus, hardware detection is not sufficient on its own. We propose a two-level detection framework with MAP being the first line of defense. The goal of MAP is to prioritize running processes such that a heavy-weight software solution can be guided to protect or scan more suspicious processes first, reducing the effort and time to detection as compared to using the second level for all processes. To avoid building complex and stateful semantic models in hardware, the first-level hardware detector is based on the sub-semantic features that are easily collectable in hardware. In contrast, the slow second-level software detector can be an IDS that is using full semantic information.

A major advantage of MAP is that it can react to a malware quickly, acting as a low-level alert system for further software protection. The hardware detector of MAP is always on, without affecting the available resources and with minimal energy consumption. At the same time, it can be built to use architectural events that are expensive and difficult to obtain at the software level (e.g., through performance counters).

We develop a fully functional hardware description of MAP hardware detector using Verilog, and integrated it within an open source x86-compatible core implementation. Our evaluations show that MAP data collection delay fits within a single cycle of the processor. Moreover, for features related to instructions, the logic is located at the commit stage of the processor pipeline, therefore avoiding any negative impact on the cycle time, instruction throughput and execution time of the program. At a time where CPU manufacturers are showing increasing willingness to invest in hardware support for security [35, 62, 66, 56, 26], MAP offers an attractive mixture of significant impact on security and low complexity.

In this paper, we did not consider how the detector should evolve to the changing nature of malware: a practical deployment will require a secure channel to update the detector configuration. Our contribution is to study the use of online hardware detection of existing malware. In particular, we did not explore how attackers will react to the presence of such a detector to attempt to hide the behavior of malware. Adversarial classification is a branch of machine learning that can assist with the evolution of attackers over time as commonly occurs in a security context [18]. Techniques from this space (such as feature randomization [65]) can be integrated into our design to make it more resilient to attacker evolution.

The remainder of the paper is organized as follows. Section 2 and Section 3 overview the malware detection approaches and examine a number of candidate sub-semantic features. Section 4 presents the proposed online detectors. Section 5 presents the implementations of the proposed detectors, and evaluates their timing and complexity. Section 6 presents an evaluation of the real-time detection system based on MAP. In Section 7 we present the related work. Finally, Section 8 offers our concluding remarks.

2. Background and preliminaries: sub-semantic malware detection

Malware detectors typically use high-level information such as behavior models of programs based on system calls, accessed/created files and thread creation events [22] to capture common features of malware. In contrast, MAP uses low-level information that can be collected during the execution of programs such as architectural events, instructions and memory addresses, and the mix of executed instruction types.

In this section, we show that sub-semantic information collected and processed in hardware can effectively distinguish malware from normal programs using simple classifiers. The classification in this section is done after-the-fact, similar to prior work [19], but differs in that the classifiers are simpler and more suitable for hardware implementation. Moreover, the section introduces the set of features that we use as representatives of the different available classes of sub-semantic information.

We study two different classification algorithms for MAP: (1) Logistic Regression (LR), which is a simple linear classification algorithm. LR attempts to linearly separate malware from normal programs in the feature space. In general, the programs are not linearly separable so LR provides a probability between 0 to 1 for the likelihood of a program being malware. To convert this likelihood to a binary decision, we pick a threshold above which programs are considered malware; and (2) Neural Network (NN) which consist of a network of perceptrons that when trained, approximates a classification function that most likely could have generated the training data. LR is equivalent to a single perceptron in a NN; thus, we expect NNs to perform better than LR but also to have higher implementation complexity.

For this experiment, the classifiers are trained based on the chosen sub-semantic features collected using a PINtool [15]. In a hardware implementation these features would be collected directly from the hardware; for example, opcode frequencies can be collected directly at the commit stage of the processor pipeline.

2.1. Data Set & Data Collection

We used the University of Mannheim malware dataset for this study [3]. We downloaded the corresponding samples of 1,087 malware programs from the Offensive Computing website [47]. Using the VirusTotal [64] malware classification interface, we identified different types and families of these programs. We followed Microsoft's classification [4] which identified 9 malware families in total which are shown in Table 1. For normal program samples, we used a variety of programs including system programs, browsers, text editing applications and the SPEC2006 benchmarks. Overall, we analyzed 467 regular programs in our evaluations.

Family	Train	Test-1	Val	Test-2	Total
Vundo	14	2	5	21	42
Emerleox	10	5	4	33	52
Virut	8	3	7	46	64
Sality	12	2	4	46	64
Ejik	7	6	4	101	118
Looper	10	3	6	145	164
AdRotator	14	1	2	119	136
PornDialer	11	6	4	196	217
Boaxxe	13	6	0	211	230

Table 1: Malware Dataset

In order to collect the data, we used a virtual machine running a 32-bit Windows 7 operating system. We disabled the firewall and Windows Security Services on this machine and connected it to the network to support malware operations.

The collected data was divided into training, testing, and validation sets as shown in Table 1. For machine learning, typical ratio of training-test-validation set is 60%-20%-20% and we followed the same rule for our model selection. We used a balanced training set (roughly equal number of malware and normal programs). The table shows two test sets: Test-1 contains 34 randomly selected malware programs and was used for feature evaluation. Test-2 contains 918 malware which was used for online detection model evaluation. The remaining malware are contained in the validation set which was used for exploring the detection and training settings.

3. Feature Selection

One of the most important decisions in setting up a classifier is the choice of features used for training and detection. Clearly, there is a large number of different candidate sub-semantic features–features that are directly available at the microarchitecture level. We explore this space by evaluating three types of features: (1) features based on executed instructions; (2) features based on memory address patterns; (3) features based on architectural events. We selected candidates from each category driven by both ease of collection through binary instrumentation as well as estimated implementation complexity. We introduce these selected features in the remainder of this section. We also evaluate their off-line detection performance using our candidate classifiers to allow comparison to prior work [19] which used more complex classifiers and in some cases different features.

3.1. Features Related to Architectural Events

One group of features is based on microarchitectural events which are not directly visible to the program. Demme et al. [19] used performance counters on the ARM chip to capture architectural features including the number of memory reads, memory writes, software updates to the program counter, unaligned memory accesses, immediate branches and taken branches. We explore these same features for the x86 instruction set with the exception of software updates to the PC which are not possible on x86. We call the collection of these features ARCH.

The value of the architectural features is collected once every 10,000 committed instructions [19]. At the end of each period, the detection algorithm classifies whether this execution period is representative of malware or of a normal program based on the collected feature data. These architectural features attempt to capture the similarity of the architectural events between malware.

Feature	Description
ARCH	Frequency of memory read/writes, taken & immediate branches and unaligned memory accesses

Table 2: Features based on Architectural Events

3.2. Features Related to Memory Addresses

The typical operations of malware include accessing files and updating/reading windows registry entries. This type of behavior results in similar access patterns to memory addresses during program execution. In order to capture this behavior, we examined the use of memory addresses as a detection feature. Specifically, we calculated the distance between the memory address of the current load/store instruction and the memory address of the first load/store operation in the group of 10K instructions (again, the collection is done at the granularity of 10K committed instructions). We used two different approaches for memory address features: (i) We created a histogram of read distances and write distances separately quantized into bins. At every 10K instructions, we store the frequency of each bin to create the feature vector (MEM1 in Table 3); and (ii) this feature is similar to MEM1, but in this case we only use a binary existence vector for the read/write histogram features. The feature bits are set to one if a distance that falls into that bin is encountered during the execution

(MEM2). In summary, the memory address features capture the similarity between the memory access patterns of malware and regular programs.

Feature	Description
MEM1	Frequency of memory address distance histogram
MEM2	Memory address distance histogram mix

Table 3: Features based on Memory Addresses

3.3. Features Related to Instructions

Executed instructions are another sub-semantic indicator of high level actions during the execution of the program. We use two different aspects of the instructions, the first one is instruction opcode and the second one is instruction category. Instruction opcode is one of the features previously used for static malware detection [52, 54, 10, 67]. However, it is not common to use the opcodes for dynamic detection of malware. We constructed our opcode features in two ways.

First, we created a list of most frequently used opcodes from malware and regular programs, we combined the top 35 opcodes that showed the largest difference (delta) in frequency between malware and regular programs (INS2 in Table 4). The combined top 10 opcodes (*mov*, *cmp*, *push*, *add*, *inc*, *jnz*, *movzx*, *xor*, *jz*, *test*) are used in both malware and regular programs. By using delta opcode features, we added to the feature vector one extra opcode ($fild$) that is mostly used in regular programs and two extra opcodes ($fnclex, fadd$) mostly used in malware programs. We also used the same opcode features in the form of a binary vector, where each element indicates if an instruction with that opcode has been executed (INS4).

Feature	Description
INS1	Frequency of instruction categories
INS2	Frequency of opcodes with largest difference
INS3	Existence of categories
INS4	Existence of opcodes

Table 4: Features based on Instructions

The instruction category features are based on Intel XED2[13] instruction category classes. Instead of tracking individual opcodes, we track frequencies of the instruction categories. There are 58 different instruction categories and the feature vector has one entry for each category. For example, all arithmetic instructions are in the *BINARY* category, all bit manipulation instructions are in *LOGICAL* category and data movement instructions are in *DATAXFER* category. We use frequency of categories (INS1) and existence of categories (INS3) as separate feature vectors. Using categories as features generalizes the instruction types such that many similar instructions are counted only with one feature. In contrast, INS2 tracks frequency of opcodes that are commonly encountered either in malware or regular programs, while INS4 tracks the existence of these opcodes in the period.

3.4. Offline detection evaluation

Evaluation of classification performance is based on the sensitivity and specificity of the model. *Sensitivity (S)* is the fraction of malware that are classified correctly and *Specificity (C)* is the fraction of normal programs classified correctly (1-C is the fraction of false positives). To evaluate classification performance and to select the best performing thresholds and features, Receiver Operating Characteristics (ROC) graphs[6] are used. We present the ROC graph for each feature in Figure 1.

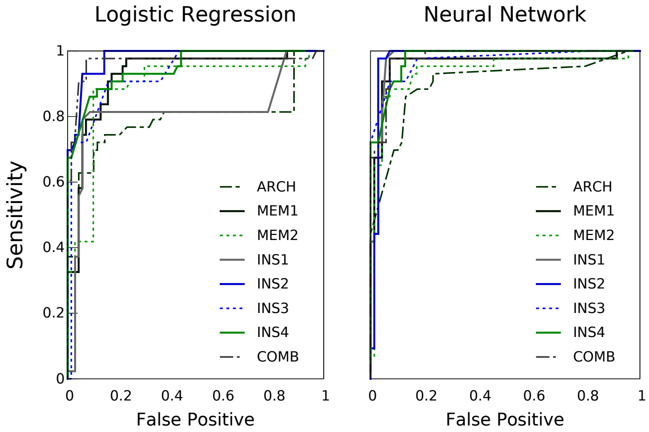

Figure 1: Detection Performance of all Features

In order to evaluate the features, we use *after-the-fact* detection performance: simply, if the majority of classifier decisions show malicious behavior then the program is labeled as malware, otherwise it is labelled as regular. The threshold for each feature selected at the point where (S+C) sum is maximized.

Figure 1 shows the Receiver-Operating Characteristics (ROC) graph for the two classifiers across the different features we studied. In an ROC graph, S is plotted as a function of *FP rate*. FP rate is calculated by dividing the number of false positives by the number of actual negative instances ($FPrate = FP/(FP+TN)$, where TN is the number of True Negatives). The upper left corner of an ROC graph (0,1) provides the best classification performance with no false positives and 100% Sensitivity. We discuss the performance of the different features in more detail below.

Architectural Features ARCH feature can correctly identify 70% of malware with only 10% false positives with the basic LR model. For the more complex NN model, the classification rate increases to 88%; however, the false positives also increase to 20%. Architectural features have already been shown to be effective for Android malware[19] using complex machine learning classifiers; they are also somewhat effective for detecting malware on x86 using simpler classifiers. Because of their modest classification performance, we did not pursue these features further.

Memory Address Features Detection performance of both MEM1 and MEM2 features significantly outperforms the ARCH feature. Both the NN and the LR models can detect 90% of malware with the NN model having only 4% false positives for MEM1. The frequency based feature (MEM1) not only classifies better than the histogram mix feature (MEM2), but also achieves the best false positive rate among all features using the NN. However, the mix features (MEM2) are easier to collect and are simpler to classify (they do not require multiplication), allowing low complexity hardware implementations.

Instruction Mix Features Instruction traces provide significant information about program execution. These features provide the highest accuracy among the set we considered: Figure 1 shows that most of the instruction based features achieve nearly 100% sensitivity with around 10% false positive rate using the NN model. The LR model is less effective than NN model for all features. Our hardware implementation is based on the INS2 feature which can detect all malware in our test set with only 9% and 16% false positive rates for NN and LR respectively.

Combining Features Finally, we evaluated the use of combinations of features to attempt to combine their strengths. All features can be combined together to create a powerful detection. This design point is marked as COMB in Figure 1. As expected, both models perform best when all features are used together. However, this significantly increases the implementation complexity of MAP.

4. Online Malware Detection

In this section, we introduce the online detection component of MAP. Detecting malware execution during runtime is a time-series analysis problem where the time-series consists of the successive decisions of the classifier. To be effective, the detection algorithm must filter out occasional false positives and quickly detect true malicious behavior.

To make a decision that considers past behavior of programs, but is not dominated by them, we use Exponentially Weighted Moving Average (EWMA) [32]. EWMA is a form of a low-pass filter commonly used to smooth out transients in a time-series signal, giving more weight to more recent inputs. EWMA computation requires floating point operations and is not suitable for efficient hardware implementation. Instead, we use a fixed-point implementation by first considering binary decisions from the base classifier (making the time-series consist of 1's for malware and 0's for normal decisions). We then use a window of these decisions with integer weights that best correspond to the chosen smoothing factor (α).

In Figure 2 we show the precise EWMA result (for $\alpha = 0.2$) and a fixed point hardware implementation for an arbitrary binary input stream. For the results in Figure 2, the input stream is assumed to have 20 bits and the window size for fixed point implementation is 8. As seen from the graph, the approximate hardware implementation closely tracks the precise EWMA estimate. The hardware implementation has a weight for each input in a window: the weight of an input in k^{th} order (W_k) is calculated by $W_k = 2^{\lfloor n/2 \rfloor} + \sum_{i=1}^{\lfloor k/2 \rfloor} 2^i$ where n is the window size and $0 \le k < n$. There are two accumulators, one for regular labels and one for malware labels. The last step performs a subtraction operation and obtains the absolute difference between the summations.

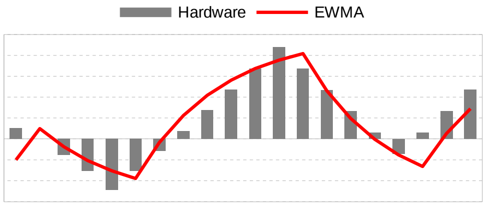

Figure 2: EWMA vs. Fixed-point Approximation

Figure 3 shows the impact of the window size on the detection performance for the LR-based model with a trained threshold. While small windows cause around 100% false positive rate, the number of false positives decreases significantly with larger windows. As the window size continues to increase, false negatives also increase because malware behavior is more likely to be missed with larger windows. We use a window size of 16 to balance these two effects.

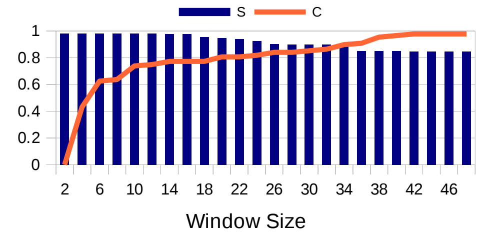

Figure 3: Effect of Window Size on Detection Performance

5. Implementation

In this section, we describe the design of MAP. We also present the implementation of the LR and the NN classifiers and evaluate the performance and complexity of the design. In addition, we introduce some optimizations to simplify the implementation and evaluate their effect. The MAP logic is located at the end of the processor pipeline after the instruction commit stage; for instruction-based features, we only consider committed instructions. For the NN classifier, we consider the trade-offs between performance and complexity: increasing the number of neurons improves detection at the cost of more complex hardware implementation.

5.1. The MAP Microarchitecture

The general MAP microarchitecture is depicted in Figure 4. The Feature Collection (FC) component collects and prepares the feature being used for classification and provides it as an input to the Prediction Unit (PU). The PU implements the classifier (the LR or the NN) that provides a binary decision on one feature vector with 1 indicating malware, and 0 indicating normal program. The output of the PU is therefore a time-series consisting of the sequence of the PU decisions over time. This time-series is the input to the Online Detection (OD) module that carries out the time-series moving average analysis to provide a real-time decision on the currently executing program as explained in Section 4.

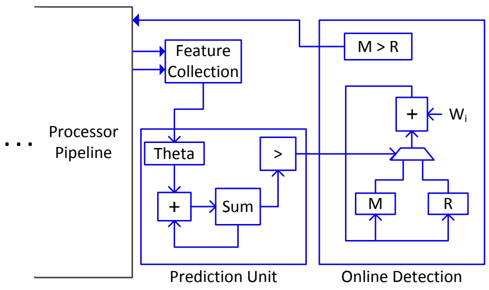

Figure 4: MAP Microarchitecture with LR

For implementation analyzed in this paper, we use the INS2 feature. Thus, the FC unit collects the committed instruction trace from the commit stage of the core pipeline. Other features require collection from the appropriate source of the feature events, such as the branch prediction unit, the memory management unit, or the fetch logic.

The MAP logic operates as follows. The FC unit collects and sends the features to the PU. The PU classifies the collected feature vector every classification period (we used 10K instruction period as with prior work [19]). The predictions are sent to the online detection module which applies the time-series algorithm as described in Section 4 to make a decision about the process. The counters in the OD module are treated as part of the process state; they are stored, restored and reset along with the process state on a context switch. A more secure option would be to store these counters in hardware. Since there are only two 32-bit registers in the OD module, it can synchronize with running processes without creating extra complexity.

5.1.1. Logistic Regression Prediction Unit We implemented the logistic regression prediction unit using INS2 feature. The feature vector has 50 elements to represent selected opcodes. The Θ vector represents the weight of each feature as a floating point number based on the detector training. In the future, we envision a secure process that allows the update of Θ to allow the detector to evolve with evolving malware.

In a standard implementation of logistic regression [31], the features are multiplied with their weights (Θ) and accumulated to calculate the hypothesis. As a final step, the hypothesis is translated to a value between 0 and 1 by *sigmoid* function and the input is labeled according to the threshold. In theory, updating the feature vector for every commit and calculating the result at the checking granularity (10K instructions) is sufficient. However, in our implementation it is not necessary to wait for the end of the period. For every new committed instruction, we set the corresponding element of the feature vector to 1 and add its weight to the total value. However, we only send the detection signal to the OD unit when 10K instructions have committed. Therefore, in our implementation, the multiplication operation is not required. The feature weights (Θ), created after training, are all floating point numbers, but they are converted to 16-bit fixed point numbers with 3 integer and 13 fractional bits. The use of fixed-point arithmetic instead of floating point significantly reduces the complexity of our design [12]. For our studies, we used scalar pipeline. For a superscalar pipeline, there will be multiple bits set for each committed instruction and multiple adders will be required.

The final step of logistic regression is the sigmoid function and prediction. Sigmoid is an asymptotic function that creates values between 0 and 1. We discretize the prediction to produce a boolean classification using simple thresholding: if the classification threshold is 0.5, then all hypothesis values larger than 0 ($sigmoid(0) = 0.5$) will be classified as class 1 (malicious programs). The implementation of actual sigmoid function is not necessary since the threshold can be compared to the sum, instead of the sigmoid of the sum. In the last step of our LR implementation, we only compare this value with the predetermined threshold and send the result to the OD module.

It is important to note that even though MAP is a hardware detection mechanism, it is possible to design a configurable version. The configurable detection mechanism can edit the Theta (Θ) values and the detection threshold through privileged operations such as firmware updates or verified accesses. This capability makes the online detector effective and flexible and can accommodate defenses for future malware types.

5.1.2. Neural Network Prediction Unit We implemented the neural network classifier as a multi-layer perceptron (MLP) with 50 input features and a single hidden layer with 19 neurons. This configuration provides the best detection performance in the feature space we explored. In parallel to our machine learning model [55], we use *tanh* as an activation function. An MLP with a single hidden layer operates by training a set of weights for each hidden neuron and the output neuron. Each hidden neuron calculates the dot product of their weights and feature vector, this value is then passed to a sigmoid function (in our case *tanh*). The output neuron operates like the hidden neurons except the output neuron uses the outputs of the hidden neurons as inputs, instead of using the feature vector.

We evaluated two designs with the same functionality. Our base design was implemented with performance constraints so that the neural network calculations are done in parallel. We then optimized this design for space constraints by serializing the operation of the neural network, which significantly reduced the number of operational units.

Similar to our LR implementation, both NN designs accumulate feature weights as feature data becomes available. Next, we calculate $\sum_{i=1}^{L} \tanh(a_i) \cdot w_i$ where L is the number of hidden neurons, a_i are the accumulated neuron values and w_i are the weights for each neuron in the output layer. Notice that we could not emit the actual implementation of the *tanh* function while implementing the NN logic, because this time the output neuron requires the actual *tanh* of the values calculated in the hidden layer. To reduce the complexity of both designs *tanh* is approximated by a Look-up Table (LUT) [43]. In particular, the lookup table based implementation of *tanh* function has a total absolute error of 0.062425 (error integrated over all input values of *tanh*). To further reduce complexity, we used fixed-point operations instead of floating point ones. To prevent the loss of precision and to reduce overflows, we use 16-bit values (3 integer plus 13 fractional bits) prior to multiplication and 32-bit values (6 integer plus 26 fractional bits) post multiplication. Finally we do not perform the final *sigmoid* operations, opting instead to simply compare the resulting sum to a precalculated threshold.

Base Design The base neural network design operates by calculating $tanh(a_i)$ for each a_i in parallel. Next, each $tanh(a_i)$ is multiplied by w_i (the corresponding weight) to generate the inputs to the ouput neuron in parallel. Finally, the products are summed using a reduction tree of adders to compute the sum in $\log_2(L)$ cycles. The final sum is compared with the threshold to produce the prediction. This design allows the classifier to be activated every cycle and produce a prediction in $T(tanh) + T(mul) + T(add) \cdot \log_2(L) + T(compare)$ cycles, where $T(x)$ is the number of cycles needed to perform x. However, the design requires L 16 bit accumulators, *tanh* LUTs and multipliers, along with $\lceil \frac{L}{2} \rceil$ 32 bit adders.

Optimized Serial Design The serial design operates by storing the accumulated values in a buffer, then multiplexing the values through a pipeline consisting of *tanh*, multiply, and accumulate. The final sum is compared to the threshold to produce the prediction. This design requires $T(setup) + T(tanh) + T(mul) + T(add) + L + T(compare)$ cycles to complete. While this unit is active, the accumulation of the feature data continues. However, another classification cannot be initiated until the previous feature set has been fully processed. Similar to the base parallel design, the serial design requires L 16-bit accumulators. However, as shown in Figure 5, the serial design requires only 1 *tanh* LUT, 1 multiplier and 1 32 bit accumulator.

5.2. FPGA Implementation and Cycle Time Impact

We implemented MAP on an open source x86 processor (AO486) [2] using Verilog. The processor is a 32-bit in-order pipelined implementation of the Intel 80486 ISA. We synthesized the core with the MAP logic at the end of the pipeline on an Altera DE2-115 FPGA board [1] using Quartus II 13.1 software. We evaluated three different prediction unit options for MAP and summarized their time, area and power impact in Table 5. The MAP design with the LR prediction unit is extremely light-weight in terms of complexity and its impact on the core power and area is under 1%. The increase of the cycle time is caused by the exception transfer to the processor pipeline. However, it can be easily eliminated if the MAP exception transfer is performed over two cycles. For the NN prediction units, the base design requires substantial area and consumes significant power; in contrast, the optimized design uses only 5.67% of the core area. The cycle time impact of the NN designs could be reduced by deepening their pipelines. The processor area breakdown is shown in Figure 6 and MAP takes up 0.28-5% of the logic cells depending on the prediction unit choice.

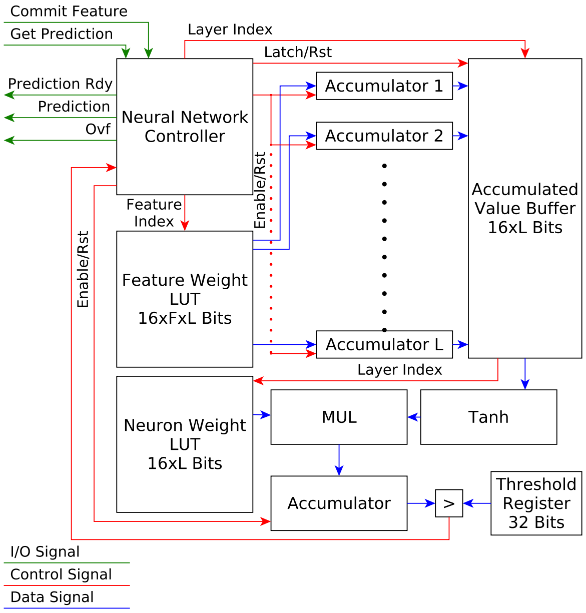

Figure 5: Neural Network Serial Design

	LR	NN Base	NN Serial
Logic Cells	+0.28%	+13.12%	+5.67%
Frequency	-1.93%	-2.28%	-5.53%
Power Usage	+0.08%	+5.23%	+1.66%

Table 5: MAP's effect on core

Our goal of implementing MAP on an FPGA was to show that it has minimal impact on the processor cycle time, power and area for a realistic system implemented within an x86 processor.

6. Effectiveness of MAP in Online Detection

In this section, we present the online detection results showing both conventional detection effectiveness (such as the ROC graph), as well as the translation from prediction unit outputs to online detection signals at runtime.

Our hardware implementation of online detection is based on INS2 feature, as it showed the best performance during offline analysis. In Figure 7, we show the detection success using the ROC graphs. The first graph shows the sensitivity of the detector that is based on an LR prediction unit. As seen from the results, it can detect almost 90% of the malware

AO486 Processor Core

Execute :	20.02 %
Writeback :	16.26 %
I-cache + D-cache :	12.61 %
TLB :	11.88 %
Register read :	8.94 %
Memory read :	7.09 %
Decode :	6.01 %
Uop decode :	3.83 %
Fetch :	0.7 %
MAP :	0.2-5 %
*Other :	12.66 %

*exception handling, register file, module interconnections, prefetching unit, ...

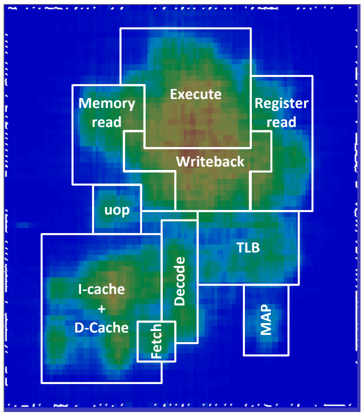

Figure 6: MAP integrated into AO486 processor core

with 6% false positive rate at its most optimal configuration. The same feature can detect 93% of malware with the same false positive rate, if after-the-fact detection was possible. The second ROC graph in Figure 7 shows detection performance of the detector with an NN-based prediction unit. While the INS2 feature can detect all malware with 7% false positive rate with after-the-fact detection, it can still detect 94% of malware at runtime with the same false positive rate.

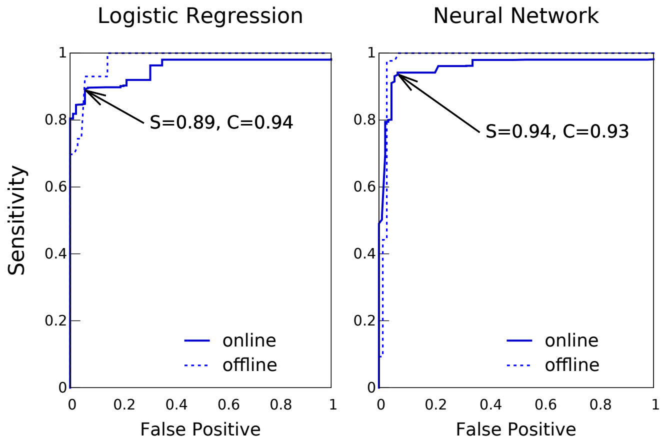

Figure 7: Online Detection Performance

Next, we show how periodic signals from Prediction Unit (PU) are translated into a detection signal at runtime by the Online Detection (OD) counters. In Figures 8 and 9, we show the first 200 instances of 10K instruction periods for a malware sample from *Virut* family and one of the Spec2K6 benchmarks (*mcf*). In Figure 8, the prediction unit is implemented using the LR model. For *Virut* sample, the PU output shows that the executed program is a malware in the beginning. However, after some period of time the output becomes indicative of a regular program, causing PU to output zeros. The online detection logic smooths these infrequent signals and correctly predicts that the executed program is a malware. Similarly, for *mcf*, the "malicious program" output signals are smoothened by the OD unit.

In Figure 9, we show the generation of the detection signal by the OD unit from the periodic outputs of the PU that implements the NN model. As seen from the figure, the NN prediction is more sensitive to the behavior of the program compared to the LR. For *Virut*, NN generates some "regular program" outputs even in the first phase of *Virut*. Again, smoothing these discrete signals from the PU output successfully creates a continuous correct detection result at runtime. For *mcf*, the NN model generates less ones than LR, because of the sensitivity of the model is higher.

The optimal design for MAP is dependent on the hardware budget. With a neural network, it is possible to get better sensitivity than with logistic regression; however, the hardware requirements for the LR implementation are almost negligible. Therefore, manufacturers are likely to consider LR a more attractive candidate for production unless further optimizations to the NN design can be found.

7. Related Work

The related work section is organized into two parts. More related to this paper, we first overview research in malware detection. The second part of this section reviews protection approaches, including those with architectural support.

Malware Detection Malware detection is an area that has attracted extensive research and commercial interest over the past decade. In general, malware detection techniques are either static (focusing on the structure of a program or system) or dynamic (analyzing the behavior during execution) [33]. Detection approaches are also classified as signature-based (looking for signatures of known malware) or anomaly-based (modeling the normal structure/behavior of programs or systems and detecting deviations from this model).

Static approaches including virus and spyware scanners are the first line of defense in malware detection. Originally, these scanners are operated using pattern matching to look for signatures of known malware. However, these 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 [45]. More advanced detectors based on semantic signatures have been proposed, and significantly improved the performance of static scanners [14]. Static approaches are limited and can be bypassed by sophisticated attackers [44]. In particular, code obfuscation techniques (polymorphic malware), and malware encryption (packing or metamorphic malware) are both sufficient to hide even from these more advanced detectors [44].

Dynamic detection observes the behavior of the program (or the system) as it runs and interacts with the environment. Dynamic behavior-based detection attempts to detect deviations from normal behavior of a program as it operates. It detects anomalies in the observed behavior compared to its model of normal behavior, which is often program-specific, to identify malware. A large number of software malware detectors have been investigated that vary in terms of the monitored events, the normal behavior model, and the detection

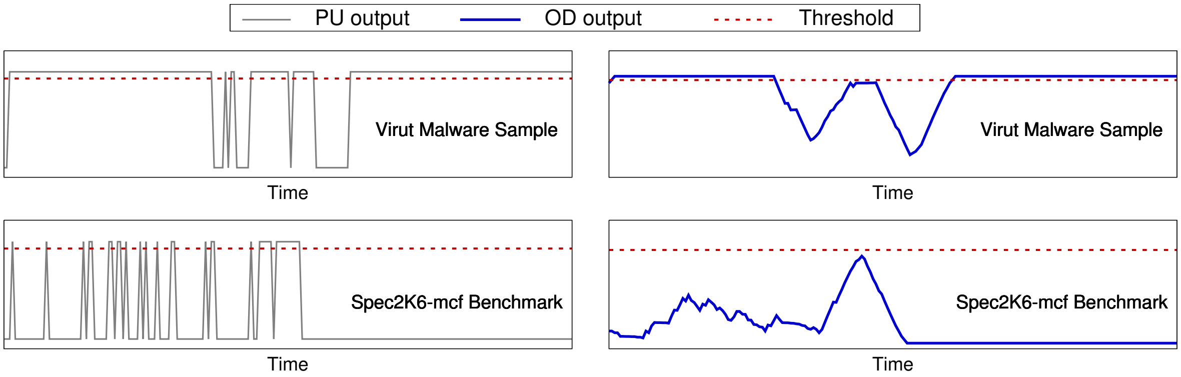

Figure 8: Translation of Prediction Unit Output to Online Detection Signal at Runtime with LR-Based Detector

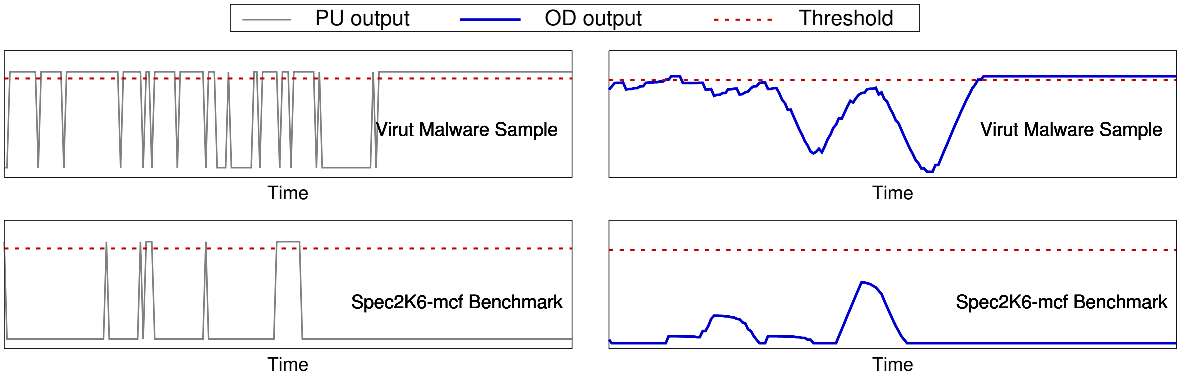

Figure 9: Translation of Prediction Unit Output to Online Detection Signal at Runtime with NN-Based Detector

algorithm [30, 51, 34, 33, 38]. The advantage of dynamic detection is that it is resilient to metamorphic and polymorphic malware [44, 39]; it can even detect previously unknown malware. However, disadvantages include a typically high false positive rate, and the high cost of monitoring during run-time. Moreover, since detection is a one time (or periodic) process, malware can evade detection either probabilistically or by recognizing that it is being observed and acting normally for that period.

Most similar to our work, RiskRanker uses a rule-based lightweight detection pass to rank the risk posed by different Android based Apps [28]. The analysis requires around 4 days of processing time, to identify a high risk set (comprising about 3% of the scanned 118,000 Apps). About one fourth of this set was found to actually have malware, including 322 zero-day exploits. MAP uses the same premise of a two-level monitoring; however, we do so in real-time for live systems.

Use of Subsemantic Features A number of earlier works explored sub-semantic features for malware detection. Bilar et al [10] examine the frequency of opcode use in malware. Santos et al and Yan et al evaluate opcode sequence signatures [54, 67], while in particular, opcode sequence signatures were found to effectively classify metamorphic malware. Runwal et al [52] study opcode sequence similarity graphs. These techniques obtain this information from running programs and malware inside heavyweight profiling tools such as Pin [15]. Moreover, all of these works consider offline analysis, rather than online detection.

Demme et al [19] collect performance counter statistics for programs and malware under execution. They show that offline machine learning tools can effectively classify malware. They conjecture that an online detector can therefore be built but do not explore this idea further. Our work builds on this evidence to develop a lightweight online hardware-supported malware detector. Tang et al [60] demonstrated that unsupervised learning on sub-semantic feature can also successfully classify malware offline; unsupervised learning may be more amenable to detecting novel malware and attacker evolution. However, unsupervised learning also requires more sophisticated analysis implying more complex hardware implementations.

Protection Approaches In this part of the related work, we overview protection approaches that make it more difficult to attack systems to install malware. We first discuss buffer overflows (as an example important vulnerability type) and defenses that have attempted to address it. We follow with a description of more comprehensive solutions that attempt to more generally protect the system.

Malware requires a vulnerability to be exploited to provide the attackers with access to the victim machine. In particular, buffer overflows are a major attack vector exploited by attackers [7]. There are several approaches for protecting against buffer overflows[61, 16, 24, 63, 53, 46]. ASLR (Address Space Layout Randomization)[61], implemented on current operating systems, adds a random offset to the starting address of the different segments in the process address space, to make it more difficult for attackers to initiate their attack. However, the unchanged library addresses, format string vulnerabilities and other data disclosure attacks make it possible to bypass ASLR or even deeper randomization[17, 50, 69]. C compiler extensions that promote correct memory allocations have also been proposed[16, 24, 63]. For example, StackGuard[16] is a compiler extension that places a canary value on top of the stack and checks this value to detect buffer overflow. CRED[53] and CCured[46] are other extensions to GNU C compiler that dynamically check bounds of allocated memory objects.

Comprehensive solutions for protecting against buffer overflows include dynamic information flow tracking [59, 49], and dynamic bounds checking [20]. However, these techniques involve significant hardware modifications (if implemented with hardware support), or incur performance losses (if implemented in software) which complicates their adoption in commercial systems. Indeed, despite these efforts, attacks based on exploiting buffer overflows continue to occur.

In the past few years, all major CPU manufacturers introduced the `W ⊕ X` memory permission bit which marks a memory page to be either writable or executable but not both [8]. This bit prevents conventional code injection attacks such as those described above since the attack code on the stack is not executable. In response, attackers have evolved to use the so-called code-reuse attacks (CRAs). CRAs, including both return-oriented [57] and jump-oriented [11] variations remain open vulnerabilities and active research topics, despite some promising solutions [48, 70, 36, 37]. An orthogonal line of research pursues protection of application secrets even in the presence of compromised system software layers and malware [23, 25, 42].

8. Concluding Remarks

This paper contributes an always-on hardware malware detection engine called MAP. MAP is integrated at the commit stage of a conventional processor, which enables it to collect sub-semantic features with low power consumption, and without software interference. MAP builds on recent important work that showed that hardware counters can be used to classify malware from normal programs off-line [19]. We explore the use of different sub-semantic features for online detection, and show that these features using logistic regression can achieve excellent sensitivity and reasonable false positive rates.

Because of the false positives which are common in anomaly-based malware detection approaches, we propose to use MAP in combination with a heavier-weight software-based detector. In particular, MAP prioritizes the scanning order of processes such that those processes that are most anomalous are scanned first. Moreover, the always-on nature of MAP makes it difficult for malware to avoid detection. We developed the hardware design for MAP and showed that its delay, complexity and energy consumption are small.

9. Acknowledgement

We would like to thank the anonymous reviewers and our shepherd John Demme for their insightful comments. This material is based on research sponsored by the National Science Foundation grant CNS-1018496. Caleb Donovick was partially supported through the REU supplement award CNS-1338672. Iakov Gorelik was partially supported by the REU Site Award CCF-1005153.

References

[1] "De2-115 development and education board," 2010, http://www.altera.com/education/univ/materials/boards/de2-115/unv-de2-115-board.html.

[2] "The ao486 project," 2014, accessed May 2014 at http://opencores.org/project,ao486.

[3] "Laboratory for dependable distributed systems university of mannheim," 2014, accessed Feb. 2014 at http://pi1.informatik.uni-mannheim.de/malheur/.

[4] "Malware protection center," 2014, accessed May 2014 at http://www.microsoft.com/security/portal/mmpc/shared/malwarenaming.aspx.

[5] S. Abraham and I. Chengalur-Smith, "An overview of social engineering malware: Trends, tactics, and implications," *Technology in Society*, vol. 32, no. 3, pp. 183–196, 2010.

[6] Y. Abu-Mostafa, M. Magdon-Ismail, and H. Lin, *Learning from Data: A short course*. AMLBook, 2012.

[7] Aleph One, "Smashing the stack for fun and profit," Nov. 1996.

[8] S. Andersen, "Part 3: Memory protection technologies," in *Changes to Functionality in Microsoft Windows XP Service Pack 2*. Microsoft Corp., 2004, http://technet.microsoft.com/en-us/library/bb457155.aspx.

[9] S. Bandhakavi, S. King, P. Madhusudan, and M. Winslett, "Vex: Vetting browser extensions for security vulnerabilities." in *Proc. USENIX Security Symposium*, 2010.

[10] D. Bilar, "Opcode as predictor for malware," 2007.

[11] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of ASIACCS*. ACM, 2011, pp. 30–40. [Online]. Available: http://doi.acm.org/10.1145/1966913.1966919

[12] J. Cavanagh, *Computer Arithmetic and Verilog HDL Fundamentals*. CRC Press, 2009.

[13] M. Charney, "Xed2 user guide," 2011, http://software.intel.com/sites/landingpage/pintool/docs/56759/Xed/html/main.html.

[14] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant, "Semantics-aware malware detection," in *Proc. IEEE Symposium on Security and Privacy*, 2005, pp. 32–46.

[15] C.Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proc. PLDI*, 2005.

[16] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of USENIX Security*, vol. 7, 1998.

[17] F. Crew, "Aslr bypassing method on 2.6.17/20 linux kernel," 2008, available online at http://www.exploit-db.com/papers/13030/.

[18] N. Dalvi, P. Domingos, M. Sumit Sanghai, and D. Verma, "Adversarial classification," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004, pp. 99–108.

[19] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 559–570. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485970

[20] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the c programming language," in *Proceedings of the ASPLOS*. New York, NY, USA: ACM, 2008, pp. 103–114. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346295

[21] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM conference on Computer and communications security (CCS)*, 2008, pp. 51–62.

[22] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys (CSUR)*, vol. 44, no. 2, 2012.

[23] J. Elwell, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "A non-inclusive memory permissions architecture for protecting against cross-layer attacks," in *Proc. International Symposium on High Performamce Computer Architecture (HPCA)*, Feb. 2014.

[24] H. Etoh and K. Yoda, "Propolice: Improved stack-smashing attack detection," *IPSJ SIG notes on computer security*, Oct 2001.

[25] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, and R. Riley, "Iso-x: A flexible architecture for hardware-managed isolated execution," in *Proc. International Symposium on Microarchitecture (MICRO)*, Dec. 2014.

[26] "Intel architecture instruction set extensions programming reference," 2014, accessed Feb. 2014 at http://download-software.intel.com/sites/default/files/319433-015.pdf.

[27] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Usenix Symposium on Network and Distributed System Security (NDSS)*, 2003.

[28] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th international conference on Mobile systems, applications, and services (MobiSys)*, 2012, pp. 281–294.

[29] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee, "Bothunter: Detecting malware infection through ids-driven dialog correlation," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007.

[30] S. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of computer security*, vol. 6, no. 3, pp. 151–180, 1998.

[31] D. W. Hosmer Jr. and S. Lemeshow, *Applied Logistic Regression*. John Wiley & Sons, 2004.

[32] R. J. Hyndman, A. B. Koehler, J. K. Ord, and R. D. Snyder, *Forecasting with exponential smoothing*. Springer, 2008.

[33] N. Idika and A. Mathur, "A survey of malware detection techniques," technical Report, Departemnt of Computer Science, Purdue University. Accessed Feb. 2014 at: http://cyberunited.com/wp-content/uploads/2013/03/A-Survey-of-Malware-Detection-Techniques.pdf.

[34] G. Jacob, H. Debar, and E. Filiol, "Behavioral detection of malware: from a survey towards an established taxonomy," *Journal in computer Virology*, vol. 4, no. 3, pp. 251–266, 2008.

[35] V. G. Jim Guilford, Kirk Yap, "Fast SHA-256 Implementations on Intel Architecture Processors," Intel Corporation, Tech. Rep., May 2012.

[36] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low overhead mitigation of code reuse attacks," in *Proceedings of ISCA*, 2012.

[37] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "Scrap: Architecture for signature-based protection from code reuse attacks," *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, vol. 0, pp. 258–269, 2013.

[38] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang, "Effective and efficient malware detection at the end host." in *USENIX Security Symposium*, 2009, pp. 351–366.

[39] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," in *IEEE Annual Computer Security Applications Conference (ACSAC)*, 2007, pp. 431–441.

[40] "McAfee labs threats report, q4, 2013," 2014, accessed May 2014 from http://mcaf.ee/qw7fe.

[41] G. McGraw and G. Morrisett, "Attacking malicious code: Report to the infosec research council," *IEEE Software*, vol. 17, no. 5, pp. 33–41, Sep. 2000.

[42] F. McKeen, I. Alexandrovich, A. Berenzon, C.Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Wkshp. on Hardware and Architectural Support for Security and Privacy, with ISCA'13*, 2013.

[43] P. Meher, "An optimized lookup-table for the evaluation of sigmoid function for artificial neural networks," in *VLSI System on Chip Conference (VLSI-SoC), 2010 18th IEEE/IFIP*, Sept 2010, pp. 91–95.

[44] A. Moser, c. Kruegel, and E. Kirda, "Limits of static analysis of malware detection," in *IEEE Annual Computer Security Applications Conference (ACSAC)*, 2007, pp. 421–430.

[45] C. Nachenberg, "Computer virus-antivirus coevolution," *Communications of the ACM*, vol. 40, no. 1, pp. 46–51, Jan. 1997.

[46] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: type-safe retrofitting of legacy code," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '02. New York, NY, USA: ACM, 2002, pp. 128–139. [Online]. Available: http://doi.acm.org/10.1145/503272.503286

[47] "Open Malware," accessed Feb. 2014 at: http://www.offensivecomputing.net/.

[48] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "Gfree: Defeating return-oriented programming through gadget-less binaries," in *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2010, pp. 49–58.

[49] M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, and T. Suri, "SIFT: A low-overhead dynamic information flow tracking architecture for smt processors," in *Proceedings of the ACM International Conference on Computing Frontiers*, May 2011.

[50] V. Pappas, M. Polychronakis, and A. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012, pp. 601–615.

[51] M. Roesch, "Snort: Lightweight intrusion detection for networks." in *Proc. Usenix System Adminsitration Conference (LISA)*, 1999, pp. 229–238.

[52] N. Runwal, R. M. Low, and M. Stamp, "Opcode graph similarity and metamorphic detection," *J. Comput. Virol.*, vol. 8, no. 1-2, pp. 37–52, May 2012. [Online]. Available: http://dx.doi.org/10.1007/s11416-012-0160-5

[53] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *NDSS*, 2004.

[54] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: Opcode-sequence-based malware detection," in *Engineering Secure Software and Systems*. Springer, 2010, pp. 35–43.

[55] M. Schmid, "A feed forward multi-layer neural network," 2010.

[56] "Software Guard Extensions Programming Reference," 2014, accessed Feb. 2014 at http://download-software.intel.com/sites/default/files/319433-015.pdf.

[57] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of CCS*. ACM Press, Oct. 2007, pp. 552–61.

[58] E. Skoudis, *Malware: Fighting Malicious Code*. Prentice Hall, 2003.

[59] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of ASPLOS*. ACM, 2004, pp. 85–96. [Online]. Available: http://doi.acm.org/10.1145/1024393.1024404

[60] A. Tang, S. Sethumadhavan, and S. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Research in Attacks, Intrusions and Defenses*, ser. Lecture Notes in Computer Science, 2014, vol. 8688, pp. 109–129.

[61] P. Team, "Pax address space layout randomization (aslr)," http://pax.grsecurity.net/docs/aslr.txt.

[62] ——, "Pax non-executable pages design & implementation," http://pax.grsecurity.net/docs/noexec.txt.

[63] Vendicator, "Stack shield technical info file v0.7," January 2001, http://www.angelfire.com/sk/stackshield/.

[64] "VirusTotal," accessed Feb. 2014 at: https://www.virustotal.com/en/.

[65] Y. Vorobeychik and B. Li, "Optimal randomized classification in adversarial settings," in *Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014)*, 2014.

[66] "Crimeware protection: 3rd generation intel core vpro processors," 2014, accessed Feb. 2014 at http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/3rd-gen-core-vpro-security-paper.pdf.

[67] G. Yan, N. Brown, and D. Kong, "Exploring discriminatory features for automated malware classification," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2013, pp. 41–61.

[68] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, 2007, pp. 116–127.

[69] Y. Yu, "Dep/aslr bypass without rop/jit."

[70] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Proc. 22nd Usenix Security Symposium*, 2013.