Kalpa Publications in Computing

Volume 7, 2018, Pages 1–15

PROOFS 2018. 7th International Workshop on Security Proofs for Embedded Systems

Side-Channel Assisted Malware Classifier with Gradient Descent Correction for Embedded Platforms

Manaar Alam[1], Debdeep Mukhopadhyay[1], Sai Praveen Kadiyala[2], Siew Kei Lam[2], and Thambipillai Srikanthan[2]

[1] Indian Institite of Technology Kharagpur, Kharagpur, West Bengal, India.
alam.manaar@iitkgp.ac.in, debdeep@cse.iitkgp.ac.in
[2] Nanyang Technological University, Singapore.
saipraveen@ntu.edu.sg, siewkei_lam@pmail.ntu.edu.sg, ASTSRIKAN@ntu.edu.sg

Abstract

Malware detection is still one of the difficult problems in computer security because of the occurrence of newer varieties of malware programs. There has been an enormous effort in developing a generalised solution to this problem, but a little has been done considering the security of resource constraint embedded devices. In this paper, we attempt to develop a lightweight malware detection tool designed specifically for embedded platforms using micro-architectural side-channel information obtained through Hardware Performance Counters (HPCs). The methodology aims to develop a distance metric, called λ, for a given program from a benign set of programs which are expected to execute in the embedded environment. The distance metric is decided based on observations from carefully chosen features, which are tuples of high-level system calls along with low-level HPC events. An ideal λ-value for a malicious program is 1, as opposed to 0 for a benign program. However, in reality, the efficacy of λ to classify a malware largely depends on the proper assignment of *weights* to the features. We employ a gradient-descent based learning mechanism to determine optimal choices for these weights. We justify through experimental results on an embedded Linux running on an ARM processor that such a side-channel based learning mechanism improves the classification accuracy significantly compared to an ad-hoc selection of the weights, and leads to significantly low false positives and false negatives in all our test cases.

1 Introduction

Embedded systems are of growing importance because of the emerging applications, starting from automotive to Internet-of-Things. These processing units though often expected to perform dedicated and limited computations, as opposed to a general purpose computing platform, can serve as a coveted attack surface for adversaries. The reason could be varied, ranging from the fact that often they are easily accessible to attackers giving opportunities for physical attacks to the running embedded operating systems which are lightweight and usually devoid of sophisticated checks for execution of vulnerable codes, making them easy targets for malware. While developing suitable lightweight mechanisms for malware detection is of great importance, most commercial anti-malware software analyze high-level system profiles. A frequency centred

L. Batina, U. Kühne and N. Mentens (eds.), PROOFS 2018 (Kalpa Publications in Computing, vol. 7), pp. 1–15

model using system calls is presented in [1], which aimed at extracting specific system call patterns pertaining to malware and thereby analyzing runtime programs. Owing to the increase in malware feature database, a model, defining bounds for a database, called Bound of Frequency Model (BOFM) is presented in [2]. Many such software protections are bypassed routinely by smart malware writers and are hence inadequate. For instance, the method of observing frequencies of system calls [3] fail to detect kernel-modifying rootkits. On the other hand, modern processors are enabled with dedicated performance monitoring registers, called Hardware Performance Counters (HPCs) which provide low-level and sensitive information of events occurring beneath the software stack. These HPC events have been traditionally used as side-channels for compromising ciphers, ranging from RSA [4] to more contemporary Elliptic Curve Cryptography [5]. HPC events have recently been used to detect side-channel attacks [6] and ransomwares [7] in the contemporary systems. However, in this work, we investigate whether these rich side-channels can be utilized to classify malware from benign codes which are typically expected to execute on a target embedded platform. The HPC values are also difficult to manipulate, and thus potentially serve a robust source of information on the system. The usage of HPCs in literature to detect the presence of malware codes have been rather limited, and mostly quite ad-hoc. Early work in malware detection using Hardware Performance Counters (HPCs) was discussed in [8]. Attempts to build a malware detector in hardware using performance counters are done in [9]. Approaches to detect malware that modify the system calls are discussed in [10, 11]. In these works, authors focused on counting the hardware events that occur during each system call execution in a guest Virtual Machine thereby identifying the modifications to kernel control flow. However, all of these works either require hardware modification, or employ complex detection architecture which may not be suitable to implement in a resource constraint embedded devices. While few approaches have been based on Machine Learning (ML) [12, 13], others have been performing correlation analysis [14, 15] to distinguish benign and malware codes. The accuracy of the ML-based methods largely depends on the features chosen; however, the reported works do not stress on the selection of these feature vectors. On the other hand, the later is based on thresholding of the correlation, where the estimation of threshold can be vital, and an ad-hoc decision can lead to a drastic adverse result.

In this work, we stress that for proper classification of malware both low-level and high-level information is of high importance, and properly associating one with the other can lead to an effective methodology. In this pursuit, we attempt to choose a combination of high-level and low-level features, comprising standard OS utilities and performance counter events, called *indicators* and *observers* respectively. In order to quantify the relative importance of this association between observers and indicators, we initially assign weights based on their ability to distinguish malwares from benign codes during an offline analysis with a known set of repository. The logic for such classification is based on a statistical hypothesis testing performed by computing a t-test, which is recently studied widely in side-channel community for measuring vulnerability of crypto-systems against side-channel analysis. We use the assigned weights in offline phase to establish a metric for online phase, denoted as λ, measuring the distance of a program from the benign set of programs which execute on target processor. An unknown executable is termed as malware, if the metric associated with it crosses a predefined threshold, estimated by a 3σ analysis. The paper shows that while the formalism expects a high value for λ (ideally one) for malwares, and low (ideally zero) for benign codes, in real life an ad-hoc choice of weights for the features can lead to false positives and false negatives, as the classification is mostly dependent on selection of threshold. We improve the method by performing iterations of gradient descent during training phase to adjust the weights and show that the classifier performs more accurately to classify malwares without depending on any such threshold.

Significance to Embedded Platform

The primary objective of this work is to develop a light-weight malware detection method which targets resource constraint embedded environments. The proposed scheme is ideal for such platforms for the following reasons:

1. Embedded platforms are often aimed at supporting a restricted set of applications. In such a situation, the proposed method is ideally useful, as the reference point for computing the distance metric λ is well-defined.
2. The proposed detection scheme is capable of running in a simple embedded Linux without requiring any sophisticated libraries.

The scheme is also amenable to an on-line implementation which does not require a large memory to store while evaluation and have a small code size.

Contribution

The technical contributions of the paper are summarized as below:

1. We formalize a concrete side-channel assisted methodology for malware classification on an embedded platform. As opposed to existing literature, we attempt to develop a theoretical framework by combining statistical tests using side-channel information with learning techniques to arrive at an optimal model for classification.
2. We present a gradient descent based optimization technique to improve the relative sensitivity of the features which comprise of both high-level (common OS utilities), and low-level (HPC) events. This provides significantly better classification accuracy and reduces the false positives and false negatives to a great extent for our test examples.

The overall organization of the paper is as follows: Section 2 presents necessary preliminary background. Section 3 presents a comprehensive discussion on the proposed methodology with Section 4 demonstrating an extensive set of results for an ARM-based embedded system. Finally, Section 5 concludes our work.

2 Preliminary Background

A fundamental question in several scientific discourses is whether two sets of data are significantly different from each other. A t-test is often used to provide a quantitative value, as a probability, that the mean μ of two sets are different. The objective of the test strategy is thus to detect whether there is any deviation from an expected reference distribution.

2.1 Hypothesis testing using *t*-test

Let us consider two samples X_0 and X_1 having mean and standard deviations as μ_0, s_0 and μ_1, s_1 respectively. A statistical hypothesis called *null hypothesis* on the equality of two means for the two samples (denoted as H_0: $\mu_0 = \mu_1$) is tested for acceptance or rejection based on the sample observations. The *Welch's* t-test is used for testing the null hypothesis when the two samples have unequal sample variances and unequal sample sizes with the test statistic,

$$\boxed{t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}}} \tag{1}$$

where, n_0 and n_1 are the sample sizes of X_0 and X_1 respectively. The degree of freedom (ν) is approximated using *Welch-Satterthwaite* equation, given by,

$$\nu \approx \frac{\left(\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}\right)^2}{\frac{s_0^4}{n_0^2\nu_0} + \frac{s_1^4}{n_1^2\nu_1}}$$

where, $\nu_0 = n_0 - 1$ and $\nu_1 = n_1 - 1$. The null hypothesis H_0 is rejected when the test statistic $|t|$ exceeds a threshold defined by the confidence level (α) and ν.

2.2 Online Computation of *t*-test

Computation of t-test for large distributions on resource-constrained devices, where expending significant memory for storing these distributions is not possible, can be performed as follows:

$$M_{i,k} = M_{i,k-1} + \frac{(x_{i,k} - M_{i,k-1})}{k}$$
$$S_{i,k} = S_{i,k-1} + (x_{i,k} - M_{i,k-1})\,(x_{i,k} - M_{i,k})$$

where $x_{i,k}$, $M_{i,k}$, $S_{i,k}$ and n_i are k^{th} observation, current mean, sum of squares of differences from the current mean, and sample size respectively for the i^{th} distribution. The sample variance of the i^{th} distribution following the calculation is $s_i^2 = \frac{S_{i,k}}{n_i - 1}$. The t-statistic between the two distributions may now be computed as per Equation (1) as described previously.

2.3 Hardware Performance Counters

Hardware Performance Counters (HPCs) are a set of special purpose registers which are present in most of the modern processor's Performance Monitoring Unit. These registers can be programmed easily to collect the number of occurrences of different micro-architectural events (like cache misses, branch mispredictions, retired instructions, etc.) during the execution of a program in the processor. Linux `perf` is a widely used tool, for all Linux 2.6.35+ based systems, which can be invoked to access these performance counters with very low granularity. Every popular operating system has HPC-based profilers, but the number and type of performance counter events vary across different Instruction Set Architectures. Most of the modern processors offer thousands of HPC events to monitor, but, only a selected few of them can be observed in parallel because of the restrictions in the number of built-in HPC registers.

3 Overview of the Proposed Methodology

In this section, we first discuss the intuition behind the proposed method. Next, we introduce a formal definition of a distance metric signifying a functional difference from the set of benign programs. Finally, we propose an approach to improve the metric based on error feedback.

3.1 Intuition behind the Approach

Malware programs perform a sequence of activities like any other programs, i.e., they exhibit phase behavior for their execution. It has already been reported in the literature that these phases correspond to different patterns in low-level micro-architectural events. One important property which helps us to consider these micro-architectural events for detecting the presence of malware is that the patterns differ radically between different categories of programs. Moreover, the correlation between the program execution and the micro-architectural events are hard to formalize. Hence, it is challenging for malware writers to rewrite the programs, which perform the same set of operations but have different low-level event patterns.

The number of malware programs can be unlimited; however, they are expected to perform specific operations like potentially affect the file system, try to hide some files, affect the network, create additional processes, etc. The intuition of our detection strategy is to generate a set of benign operating system library executables, called *indicators*, which would be monitored in run-time by observing a set of low-level hardware events, called *observers*. These low-level hardware events can easily be monitored using hardware performance counters (HPCs). We select those *indicators* and *observers*, which are most likely to be affected by the malwares.

The types of programs which can execute on a specific Embedded Platform are limited. For example, an iPod is expected to run MP3 and video player programs. We call such programs as *benign* programs. The idea is based on the hypothesis that the HPC events would vary significantly when a malicious code runs in the system, compared to when a benign program runs on it. So, we try to quantify the functional distance of an unknown program from the set of benign programs defined by some suitable metric. A smaller distance can be interpreted as closeness in functionality to the benign programs, indicating benign nature of the unknown program. Similarly, a larger distance can be interpreted as a significant deviation between functionalities of the unknown program and the set of benign programs, thus indicating a possible malicious nature of the unknown program. Therefore, initial pre-processing would be to create templates of the benign environment, by observing low-level hardware events for the monitored set of indicators. When a potentially unknown malicious code executes, it is expected that the statistics generated by the observers for the indicators are significantly different.

3.2 Formalization of the Distance Metric

Let the set of indicators be denoted as $\mathcal{R} = \{r_1, r_2, \cdots, r_p\}$ and the observers as $\mathcal{H} = \{h_1, h_2, \cdots, h_q\}$. The combination of observers and indicators are called *tuples*. We have a total of $p \times q$ tuples. For any executable, e, when the indicator r_i is run in its presence, we observe the hardware performance counter h_j using the popular Linux `perf` tool. This is performed for s trials to minimize the effect of system noise. Let the value of hardware performance counter h_j for k^{th} trial when indicator r_i is run in presence of e be denoted as $\alpha^k_{i,j}(e)$. Let $\theta_{i,j}(e)$ be the distribution of h_j in the presence of both r_i and e, containing s observations, i.e., $\theta_{i,j}(e) = \{\alpha^1_{i,j}(e), \alpha^2_{i,j}(e), \cdots, \alpha^s_{i,j}(e)\}$. When done for all $p \times q$ tuples, we obtain a distribution matrix, for the executable e:

$$\mathcal{D}(e) = \begin{bmatrix} \theta_{1,1}(e) & \theta_{1,2}(e) & \theta_{1,3}(e) & \dots & \theta_{1,q}(e) \\ \theta_{2,1}(e) & \theta_{2,2}(e) & \theta_{2,3}(e) & \dots & \theta_{2,q}(e) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \theta_{p,1}(e) & \theta_{p,2}(e) & \theta_{p,3}(e) & \dots & \theta_{p,q}(e) \end{bmatrix}$$

Initially we consider a collection of two sets of programs; one which contains malwares and other which contains benign application programs. Let the malware set be denoted as $\mathcal{M} = \{m_1, m_2, \cdots, m_k\}$ and the benign application programs as $\mathcal{B} = \{b_1, b_2, \cdots, b_l\}$. After the *perf stat* collection for both the sets, we obtain two sets of distribution matrices. We denote the distribution matrices for the set of malwares as $\{\mathcal{D}(m_1), \mathcal{D}(m_2), \cdots, \mathcal{D}(m_k)\}$ and for the set of benign application programs as $\{\mathcal{D}(b_1), \mathcal{D}(b_2), \cdots, \mathcal{D}(b_l)\}$.

In our detection scheme, we proposed the computation of a *Univariate t-test* by applying the two sample *Welch's* t-test. The t-test is applied on the distributions of each of $p \times q$ tuples of each samples in benign set against each samples in malware set. The main objective of the t-test is to assign high influence on the indicator-observer tuple which are capable of detecting more malwares. We say a malware m_x is detected against a benign application program b_y by the tuple (r_i, h_j), if the *Null Hypothesis* (i.e., sample means are same in both distributions) is rejected for the distributions $\theta_{i,j}(m_x)$ and $\theta_{i,j}(b_y)$.

Algorithm 1: Calculation of Sensitivity Matrix

```
Data: Sets Containing Benign (B) and Malware (M) Executables, List of Indicator Programs R, List
      of HPC Events H
Result: Sensitivity Matrix W
forall b_l in B do
    forall indicators in R do
        Collect perf script results for the events in H with b_l in the background;
        D(b_l) = parsed values for the collected data;
    end
end
forall m_k in M do
    forall indicators in R do
        Collect perf script results for the events in H with m_k in the background;
        D(m_k) = parsed values for the collected data;
    end
end
Initialize Sensitivity Matrix W to zero for all (R, H) tuple;
forall r_i in R do
    forall h_j in H do
        forall b_l in B do
            forall m_k in M do
                Calculate t-statistic for D(b_l) and D(m_k) corresponding to tuple r_i and h_j using
                  Equation (1);
                if t-statistic > t_critical then
                    Increment W(r_i, h_j) by 1;
                end
            end
        end
    end
end
Normalize and return W;
```

A basic strategy to assign weights to the tuples is through the count of correct classification of a malware by that tuple. If a tuple (r_i, h_j) correctly identifies a malware, m_x, against a benign program, b_y, then we increase its count. Since we have $p \times q$ tuples, we obtain a count matrix with p rows and q columns. The values of each cell in the count matrix ranges between 0 and total number of benign and malware program combination, i.e., between $[0, k \times l]$. These counts are then normalized to obtain the *sensitivity matrix* created based on these tuples. An illustration of *sensitivity matrix* $(\mathcal{W})$ is shown below.

$$\mathcal{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & \dots & w_{1,q} \\ w_{2,1} & w_{2,2} & w_{2,3} & \dots & w_{2,q} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{p,1} & w_{p,2} & w_{p,3} & \dots & w_{p,q} \end{bmatrix}$$

Here, $w_{i,j}$ signifies a relative importance of indicator r_i and observer h_j to detect a malware program. Calculation of sensitivity matrix is described in details with the help of Algorithm 1.

The quantification of the distance metric for an unknown program $\mathcal{T}$ from the set of benign samples are shown as follows. We collect the perf stat data for $\mathcal{T}$ and obtain the distribution matrix $\mathcal{D}(\mathcal{T})$, as described previously. Now this distribution is compared against all the l benign programs using *Welch's* t-test for all the $p \times q$ tuples. This univariate analysis constructs a *Count Matrix* $(\mathcal{C})$ having p rows and q columns. Each cell of $\mathcal{C}$, i.e., value of c_{ij}, equals the total number of times $\mathcal{T}$ is classified as Malware by the tuple r_i and h_j. Thus the value of c_{ij} ranges between $[0, l]$. The count matrix is normalized for each tuple before further processing. An illustration of normalized $\mathcal{C}$ is given as below:

$$\mathcal{C} = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} & \dots & c_{1,q} \\ c_{2,1} & c_{2,2} & c_{2,3} & \dots & c_{2,q} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{p,1} & c_{p,2} & c_{p,3} & \dots & c_{p,q} \end{bmatrix}$$

Multiplying $\mathcal{C}$ with the sensitivity matrix $\mathcal{W}$, obtained previously, we get a *Score Matrix*, $\mathcal{S}$. The score matrix is obtained by element-wise multiplication of $\mathcal{C}$ and $\mathcal{W}$. Thus, $\mathcal{S} = \mathcal{C} \odot \mathcal{W}$.

Hence, $s_{ij} = c_{ij} w_{ij}$ for $i \in \{1, \cdots, p\}$ and $j \in \{1, \cdots, q\}$. **We define λ as the sum of all the elements in the matrix $\mathcal{S}$.** Thus,

$$\boxed{\lambda = \sum_{i=1}^{p} \sum_{j=1}^{q} s_{i,j} = \sum_{i=1}^{p} \sum_{j=1}^{q} c_{i,j} w_{i,j}} \tag{2}$$

This λ, ranging between 0 and 1, can be interpreted as the quantification of functional distance of the program $\mathcal{T}$ from the benign set $\mathcal{B}$ because:

- If $\mathcal{T}$ is a benign program, distributions of most of its tuples will be similar to that of the programs in set $\mathcal{B}$. As a result, elements of the count matrix $\mathcal{C}$, i.e., $c_{i,j}$, will take lower values. Since, the calculation of λ in Equation (2) depends on $c_{i,j}$ and constant $w_{i,j}$, λ will take lower value.
- Similarly, if $\mathcal{T}$ is a malware program, the distributions of most of its tuples will be different from the programs in set $\mathcal{B}$. Hence, elements of the count matrix $\mathcal{C}$ will take higher values than benign programs and the corresponding λ will be higher than the previous case.

We can quantify lower and higher values by defining a threshold. An unknown program having λ value less than a predefined threshold λ_t, is termed as a benign program, or else a malware program. Threshold λ_t is determined by applying 3σ rule on the distribution of λ values obtained for benign programs in set $\mathcal{B}$. Suppose mean and standard deviation obtained from the distribution containing values λ_1, λ_2, $\dots$, λ_l are $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ respectively, where λ_i represents the λ value for i^{th} benign program in set $\mathcal{B}$. Then the threshold λ_t is defined as:

$$\boxed{\lambda_t = \mu_{\mathcal{B}} + 3\sigma_{\mathcal{B}}} \tag{3}$$

Hence, for an unknown program $\mathcal{T}$, having distance metric $\lambda_{unknown}$, we can conclude:

$$\mathcal{T} \text{ is } \begin{cases} Benign, & \text{if } \lambda_{unknown} < \lambda_t \\ Malware, & \text{otherwise} \end{cases}$$

The decision for an unknown program of being a benign or a malware depends on the threshold of λ_t. Since the threshold is computed using the 3σ analysis, there are chances for some malware programs having λ value less than the threshold and some benign programs having λ value greater than the threshold. In other words, because of the threshold, we could incur false positives as well as false negatives. Moreover, the calculation of λ depends on the sensitivity matrix $\mathcal{W}$, which is computed beforehand based on a set of benign program $\mathcal{B}$ and a set of malware program $\mathcal{M}$. Since $\mathcal{W}$ is a constant matrix, we may not get the optimal value of λ as we intended ideally (0 for benign and 1 for malware) and could incur some errors in determining the λ values. Next, we propose a method to update the matrix $\mathcal{W}$ based on the errors calculated on the initial set of benign and malware programs by giving feedback in terms of errors incurred. Our objective is to optimize the matrix such that the λ values for benign and malware programs are close to the ideal values of 0 and 1 respectively. In such a way, the decision making will be free from the dependency on any threshold.

3.3 Improving the Metric based on Error Feedback

As of now, we formalized the distance metric λ for an executable which is lower for a benign program and higher for a malware program. We consider our problem as a *Binary Classification* in which the output of the classifier will take only two values between $\{0, 1\}$ (0 for benign and 1

for malware). The classification of a sample depends on its Count Matrix $\mathcal{C}$, Sensitivity Matrix $\mathcal{W}$ and the threshold λ_t. The count matrix is constant for a particular sample, as it is obtained by deterministic univariate t-test. Instead of playing around with the threshold λ_t, we propose an approach following the principles of *Least Mean Square Error*, to optimize the sensitivity matrix $\mathcal{W}$, which helps to provide λ values close to 0 for benign and close to 1 for malwares. This updated sensitivity matrix will be more robust to false-positives and false-negatives which may occur because of the selection of threshold.

In the initial phase we considered l benign programs $\{b_1, b_2, \cdots, b_l\}$ and k malware programs $\{m_1, m_2, \cdots, m_k\}$ to create the sensitivity matrix. We assign each sample, r ($r = 1$ to $l + k$), a label $y^{(r)}$, such that, $y^{(r)} = 0$ for benign and $y^{(r)} = 1$ for malware. We calculate distance metric $\lambda^{(r)}$ for each sample using Equation (2). A standard mathematical way to compute error is by summing the squared deviation of the obtained output from the actual label. Thus we define,

$$\boxed{\mathcal{E} = \sum_r \left(y^{(r)} - \lambda^{(r)}\right)^2} \tag{4}$$

Since, $\lambda^{(r)}$ depends on the $w_{i,j}$'s in sensitivity matrix $\mathcal{W}$, intuitively we say that changes in $w_{i,j}$ increases or decreases the error. The main objective is to iteratively adjust the $w_{i,j}$'s such that the error $\mathcal{E}$ is minimized. If a change in weight increases (decreases) the error, then we want to decrease (increase) that weight. Mathematically, this means that we look at the derivative of the error with respect to the weight, i.e., we look at $\frac{\partial \mathcal{E}}{\partial w_{i,j}}$, which represents the change in the error given a unit change in the weight. The weight $w_{i,j}$ is then updated by the term $\delta w_{i,j} = -\eta \frac{\partial \mathcal{E}}{\partial w_{i,j}}$. Changes for each weight should be proportional to the gradient. Hence, η is the proportionality constant known as *learning rate* and the minus sign indicates to adjust the weight in the negative direction of the gradient to minimize error.

The new weight $w_{i,j}^{t+1}$ is updated from the old weight $w_{i,j}^{t}$ using the following equation:

$$\boxed{w_{i,j}^{t+1} = w_{i,j}^{t} + \delta w_{i,j} = w_{i,j}^{t} - \eta \frac{\partial \mathcal{E}}{\partial} w_{i,j}} \tag{5}$$

The weight updation process is described as below. We add a factor of $\frac{1}{2}$ to $\mathcal{E}$ for making the calculation convenient. Hence,

$$\begin{aligned}
\mathcal{E} &= \frac{1}{2} \sum_r \left(y^{(r)} - \lambda^{(r)}\right)^2 \\
\frac{\partial \mathcal{E}}{\partial w_{i,j}} &= -\sum_r \left(y^{(r)} - \lambda^{(r)}\right) \frac{\partial \lambda^{(r)}}{\partial w_{i,j}} \\
&= -\sum_r \left(y^{(r)} - \lambda^{(r)}\right) \frac{\partial}{\partial w_{i,j}} \left(\sum_i \sum_j c_{i,j}^{(r)} w_{i,j}\right) \\
&= -\sum_r \left(y^{(r)} - \lambda^{(r)}\right) c_{i,j}^{(r)}
\end{aligned}$$

Hence,

$$\delta w_{i,j} = -\eta \frac{\partial \mathcal{E}}{\partial w_{i,j}} = \eta \sum_r \left(y^{(r)} - \lambda^{(r)}\right) c_{i,j}^{(r)}$$

The new updated weight for $w_{i,j}^{t}$ will be:

$$\boxed{w_{i,j}^{t+1} = w_{i,j}^{t} + \eta \sum_r \left(y^{(r)} - \lambda^{(r)}\right) c_{i,j}^{(r)}} \tag{6}$$

Here, we notice that, before updating the weights we calculate the errors for each samples, i.e., we feedback the error associated with all the training examples and update the weights accordingly. The weight updation is performed for a desirable number of iterations, such that the total error ($\mathcal{E}$) does not improve much, or in other words the error gets saturated to a

Algorithm 2: Training Step

Data: Sensitivity Matrix $\mathcal{W}$, Set of Labels $y^{(r)}$, Count Matrix $\mathcal{C}$ for all $l+k$ samples, Learning Rate η, Tolerance Level ξ, Maximum Number of Iteration *max_iter*
Result: Updated Sensitivity Matrix $\mathcal{W}_0$
repeat
  for *r=1 to* $l+k$ **do**
    Calculate $\lambda^{(r)}$ using the Equation (2);
  end
  for *i=1 to* p **do**
    for *j=1 to* q **do**
      Update $w_{i,j}$ using the Equation (6);
    end
  end
  Calculate $\mathcal{E}$ using Equation (4);
until $\mathcal{E}$ *does not change by* ξ *in* 10 *successive iterations or max_iter is reached*;

constant value. We quantify the term *desirable number of iterations* by introducing a parameter named *tolerance level* (ξ). We perform the weight updation steps for a maximum of *max_iter* times by monitoring $\mathcal{E}$ obtained in each iteration and inspect whether it has been reduced by an amount ξ. If the error does not improve by ξ in, say, successive 10 iterations, we conclude that, the updated weights have reached to the saturated value and will not improve much. We consider the final updated sensitivity matrix for testing unknown programs.

We define the weight updating steps as training step for the binary classifier, which we describe with the help of Algorithm 2.

3.4 Analysis of an Unknown Program

We collect the data for an unknown program $\mathcal{T}$ and obtain the distribution matrix $\mathcal{D}(\mathcal{T})$, as described previously. Now this distribution is compared against all the l benign programs using *Welch's t*-test for all the $p \times q$ tuples. This univariate analysis constructs a *Count Matrix* ($\mathcal{C}$) having p rows and q columns. Multiplying $\mathcal{C}$ with the updated sensitivity matrix $\mathcal{W}_0$, obtained in the training phase, we get a *Score Matrix*, $\mathcal{S}$, and accordingly obtain $\lambda_{unknown}$ using Equation (2). We say that:

$$\mathcal{T} \text{ is } \begin{cases} Benign, & \text{if } \lambda_{unknown} \approx 0 \\ Malware, & \text{if } \lambda_{unknown} \approx 1 \end{cases}$$

The procedure to analyze an unknown program of being malware or benign is described with the help of Algorithm 3. In the next section, we evaluate our proposed approach with detailed experimental results and comparison with state-of-the-art.

4 Experimental Results

In this section, we first present a detailed overview of the experimental setup and then analyze the efficiency of the proposed methodology using a comprehensive set of results.

4.1 Experimental Setup

The primary objective of this work is to detect Malwares in a Real-Time Operating Systems (RTOS) running on embedded platforms. Almost all of the RTOS on embedded platforms use Linux based OS; hence, we restricted ourselves to detect Linux based malwares. In order to analyze the efficiency, we implemented the proposed approach on an ALTERA SoCKit VEEK-MT model, which consists of a dual-core ARM Cortex-A9 processor with an Altera 28-nm Cyclone V FPGA. The Linux based embedded RTOS implemented on this SoCKit has kernel version 3.10.31-ltsi-05172-g28bac3e.

Algorithm 3: Testing Phase

Data: Distributions of all Benign Executables used for training $\mathcal{D}(\mathcal{B})$, Updated Sensitivity Matrix $\mathcal{W}_0$, List of Indicator Programs $\mathcal{R}$, List of HPC Events $\mathcal{H}$, Executable of the unknown program $\mathcal{T}$
Result: Decision whether $\mathcal{T}$ is Malware or Benign
forall *indicators in* $\mathcal{R}$ **do**
  Collect *perf* script results for the events in $\mathcal{H}$ with $\mathcal{T}$ in the background;
  $\mathcal{D}(\mathcal{T})$ = parsed values for the collected data;
end
Initialize *Count Matrix* $\mathcal{C}$ to zero for all ($\mathcal{R}$, $\mathcal{H}$) tuple
forall r_i *in* $\mathcal{R}$ **do**
  forall h_j *in* $\mathcal{H}$ **do**
    forall b_l *in* $\mathcal{B}$ **do**
      Calculate t-statistic for $\mathcal{D}(b_l)$ and $\mathcal{D}(\mathcal{T})$ corresponding to features r_i and h_j using Equation (1);
      if t-*statistic* $> t_{critical}$ **then**
        Increment $\mathcal{C}(r_i, h_j)$ by 1;
      end
    end
  end
end
Normalize $\mathcal{C}$;
Calculate $\lambda_{unknown}$ using Equation (2);
if $\lambda_{unknown} \approx 0$ **then**
  Return $\mathcal{T}$ as Benign;
end
if $\lambda_{unknown} \approx 1$ **then**
  Return $\mathcal{T}$ as Malware;
end

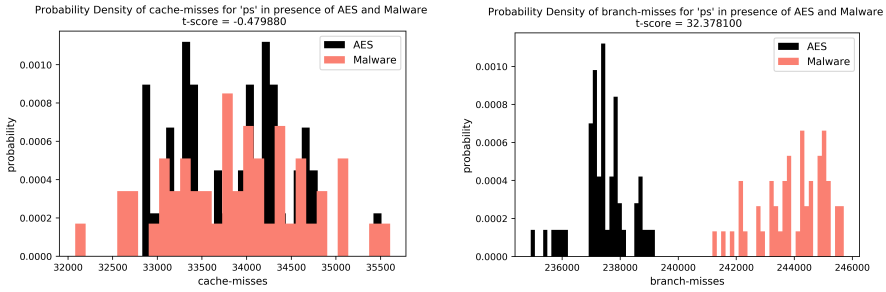
Table 1: List of Malware and Benign Programs chosen for the Experimentation

	Family	Train	Test	Total
Malware	*ARM-Malwares*	152	10	162
Benign[1]	*CHStone*	10	2	12
	UnixBench	12	4	16
	LMBench	18	4	22
Total		**192**	**20**	**212**

We collected 162 latest ARM-based malwares from *Offensive Computing* and *VirusShare* database and used 50 standard Linux benchmark programs, such as *CHStone*, *UnixBench*, and *LMBench* for our analysis. We used these different benchmark programs as a source for the benign reference point. Table 1 shows the separation of all programs into *Train* and *Test* data for offline and online analysis respectively. In this experiment, we consider 5 indicators `ls`, `netstat`, `ps`, `who`, `pwd` and 6 observers `cycles`, `instructions`, `cache-references`, `cache-misses`, `branches`, `branch-misses` to validate the proposed technique. We collected data for each executable 50 times to minimize the effect of noise.

[1]The list of some of the benign programs which are quite relevant to the modern day embedded environments from these benchmark suites are:

1. **CHStone**: Double-precision floating-point operations (DFADD, DFMUL, DFDIV, etc.), Graphics operations (ADPCM, JPEG, MOTION, etc.), Encryption operations (AES, BLOWFISH, SHA, etc.).
2. **UnixBench**: File operations (FSDISK, FSTIME, etc.), Arithmetic operations (SHORT, INT, LONG, etc.), Memory operations (HANOI, DHRY2REG, etc.)
3. **LMBench**: Network operations (LAT_HTTP, LAT_UDP, BW_PIPE, BW_TCP, etc.)



(a) (b)

Figure 1: Distribution of (a) `cache-misses` and (b) `branch-misses` for the indicator program `ps` in presence of *AES* and *Malware*.

Table 2: Initial Sensitivity Matrix $\mathcal{W}$ for Various Indicator-Observer tuples

	cycles	**instructions**	**cache-references**	**cache-misses**	**branches**	**branch-misses**
ls	0.0274	0.0299	0.0285	0.0208	0.0298	0.0371
ps	0.0513	0.0515	0.0512	0.0375	0.0513	0.0514
who	0.0243	0.0382	0.0366	0.0145	0.0405	0.0184
netstat	0.0264	0.0352	0.0352	0.0171	0.0351	0.0194
pwd	0.0259	0.0414	0.0389	0.0171	0.0392	0.0293

4.2 Validation of Assigning Weights to the Tuples

We present an experiment in this subsection to show the effect of (*indicator, observer*) tuple in detecting an unknown program as a benign or a malware. We consider a benign programs, namely *AES* from CHStone Benchmark Suite, and one *malware* as mentioned in Table 1, for this experiment. We choose to monitor one indicator program `ps` and two observers namely `cache-misses` and `branch-misses`. The resulting distributions for both the executables are provided in Figure 1. The univariate t-statistic for the distributions in Figure 1a and Figure 1b are -0.4798 and 32.3781 respectively. The $t_{critical}$ value for these datasets is 1.6605 with 95% confidence level. Figure 1 shows clearly that, while the later is able to present a distinguishable distribution, the former is not. It may be noted that the tuple (`ps`, `cache-misses`) is not effective for this example but can be effective for other pairs of benign and malware programs. Hence, we do not discard this tuple but rather assign a suitable sensitivity value. This sensitivity list provides a quantification of the goodness of the associations of the indicators with the observers in distinguishing a malware from a set of benign programs.

We followed Algorithm 1 considering the data in Offline phase, as mentioned in Table 1, and obtained an initial *sensitivity matrix* as shown in Table 2.

4.3 Analysis of Unknown Programs using Initial Sensitivity Matrix

The generalization of the proposed technique is assessed by testing 20 programs in the Online phase, as mentioned in Table 1, which were not used to create the sensitivity matrix. We followed the procedure mentioned in Algorithm 3 and obtained the $\lambda_{unknown}$ for all the 20 programs, which is shown in Figure 2. The threshold λ_t, calculated using Equation (3), is in this case 0.7261. The λ values for benign and malware examples are indicated using *Blue* and *Red* lines respectively. The threshold λ_t is shown using *Green* line. We can easily observe from Figure 2 that, the 9^{th} malware sample is treated as benign since it has λ value less than λ_t. One important observation from the figure is that we could have adjusted the threshold

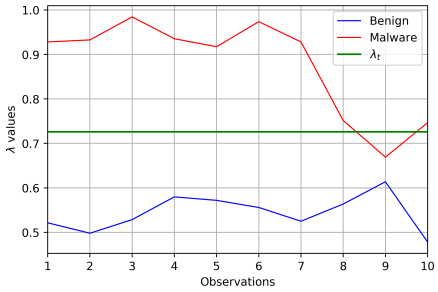


Figure 2: λ values for 20 Test Programs

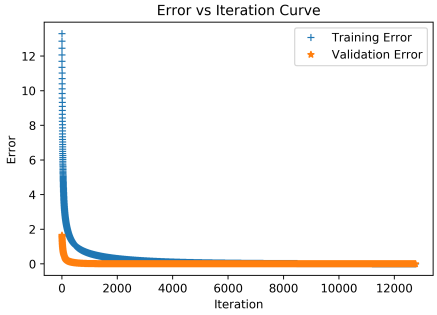


Figure 3: Errors in each Iteration

Table 3: Updated Sensitivity Matrix $\mathcal{W}_0$ for Various Indicator-Observer tuples

	cycles	**instructions**	**cache-references**	**cache-misses**	**branches**	**branch-misses**
ls	0.6411	0.8974	1.1656	-0.9028	-0.9583	0.2866
ps	0.6371	-0.0825	-0.5154	0.0533	-0.2367	0.5548
who	0.2735	0.0005	0.1963	-0.4748	-0.0066	-0.1982
netstat	-0.3863	-0.2506	-0.0963	0.2996	0.8166	-0.2988
pwd	0.2647	-0.0882	-0.3782	0.3098	0.3533	-1.2136

by assigning some empirical value between $[0.62, 0.65]$ instead of 3σ analysis to deal with *false negative* occurred in this experiment. But, to find the optimal threshold value which generalizes for all the examples, we need to analyze ample of malware programs with this technique. Still, there will be a chance for some malwares to raise false negatives as well as some benign programs which raise false positives. Hence, we update the sensitivity matrix instead of dealing with the threshold and show the results in the next subsection.

4.4 Updating the Sensitivity Matrix

We update the sensitivity matrix following Algorithm 2 and using *learning rate* (η) and *tolerance level* (ξ) to be 10^{-3} and 10^{-6} respectively. One major issue with gradient descent algorithm is that it may overfit[2] for the training sample. In order to verify that the proposed gradient descent model does not overfit for our example, we divided the initial data in online phase into train data and validation data. We update weight based on the new training data and monitor both training and validation error in each iteration. The errors in each iteration is shown in Figure 3. *Blue* line presents errors for training data and *Orange* line presents errors for validation data. It can be seen that the weight updation does not overfit for our example (as both the errors decrease with increase in iterations) and gradually saturates to optimum[3] value. The final sensitivity matrix $\mathcal{W}_0$ obtained after weight updation process is shown in Table 3.

4.5 Analyzing Unknown Programs on Updated Sensitivity Matrix

The test programs mentioned in Table 1 are again selected here, like Section 4.3, to analyze the efficiency of the proposed method. We followed the approach mentioned in Algorithm 3 and obtained the $\lambda_{unknown}$ values for all the 20 programs, but this time considering the updated sensitivity matrix. The resulting $\lambda_{unknown}$ values using both the initial and updated sensitivity

[2]It provides optimum results for data used in trained, but results in poor performance for unseen data.
[3]The sensitivity matrix for which the total error in calculating λ is minimum.

Table 4: Comparison of λ values for the Benign Test Programs using Initial and Updated Sensitivity Matrix

	Benign Programs ($\lambda_{unknown}$)									
Initial Sensitivity Matrix	0.5213	0.4979	0.5284	0.5796	0.5718	0.5558	0.5249	0.5636	0.6136	0.4781
Updated Sensitivity Matrix	0	0	0	0	0	0	0	0	0.0589	0

Table 5: Comparison of λ values for the Malware Test Programs using Initial and Updated Sensitivity Matrix

	Malware Programs ($\lambda_{unknown}$)									
Initial Sensitivity Matrix	0.9281	0.9326	0.9844	0.9353	0.9173	0.9736	0.9281	0.7514	0.6691	0.7465
Updated Sensitivity Matrix	1	1	1	1	1	1	1	1	1	1

Table 6: Comparative Study with State-of-the-Art

	False Positive	**False Negative**
Demme *et al.* [9]	3%	7%
Das *et al.* [1]	2.7%	3%
Ozsoy *et al.* [16]	6%-8%	6%
Wang *et al.* [17]	5%	8%
Elnaggar *et al.* [18]	9%	2%
λ with ad-hoc choices of weights	3.76%	5.03%
Proposed Approach	0.19%	0.01%

matrices are shown in Table 4 and Table 5. It is clear from both the tables that the separability between benign and malware programs has been increased significantly.

The average separability between benign and malware test programs provided in Table 4 and Table 5, based on the initial sensitivity matrix $\mathcal{W}$ is 0.3331 and for updated sensitivity matrix $\mathcal{W}_0$ is 0.9941. Hence, for the above dataset we obtain 199.97% increase in separability between benign and malware programs, which further helps to reduce the confusion in decision making and thereby increases the classification accuracy of the proposed method.

4.6 Comparison of Accuracy with other State-of-the-Art

We evaluated our proposed approach several times by selecting 20 programs randomly during testing as mentioned in Table 1 and compared the result with five other notable approaches mentioned in [9, 1, 16, 17, 18], which also aim to detect and prevent malware executables. We have also presented results of the λ-based classifier based on initial ad-hoc selection of sensitivity matrix to show the effectiveness of weight updation in terms of reducing false positives and false negatives. The comparison is presented in Table 6, which clearly shows that the accuracy of the proposed model is best among the recent literature with the given dataset. The table also presents the reduction in false positives and false negatives because of the weight correction. We didn't consider other notable works in this field to compare with, as they either require hardware modifications or require significant computing resources which do not meet our objective to design a lightweight detection framework.

4.7 Implementation Overhead

The detection method continually runs on the processor core and waits for a new process to arrive, thus consuming some amount of the CPU and Memory usage. We analyzed these

resource requirement using the Linux `top` command for the ARM Cortex-A9 processors and observed that %CPU Usage and %MEM Usage[4] are 16.788% and 0.798% respectively. We observe that the memory usage of the proposed method is almost negligible, though the CPU usage is slightly high, but can be permitted given the performance of the model.

Working of the proposed model depends on the distribution of benign programs, which are collected offline and stored in memory to be used in the online phase. Hence, size of the code and offline dataset are also important considering resource constraint embedded platform. Total virtual memory size used by the model to detect a single program under test is around 9 MB. The value includes the size of all code, data and shared libraries. This low value also establishes the fact that the proposed model is indeed very light-weight in terms of storage requirement.

5 Conclusion

In this paper, we attempt to analyze the side-channel information generated via Hardware Performance Counters (HPCs) and OS calls, to classify malware programs from benign programs which are expected to execute on an embedded processor. We base the classification mechanism on statistical hypothesis, which is a light-weight mechanism and can be easily implemented in low computation devices. The work develops a distance metric, called λ for distinguishing malware programs from benign executions. The work stresses the importance of optimally choosing the weights of the features and proposes a gradient descent based methodology to improve the separability of the malware class from the benign applications. We show through experimentations performed on an embedded Linux on an ARM processor that the combination of the learning technique with side-channel analysis outperforms existing works in reducing false positives and false negatives. We also show that the methodology has very less computational overhead and can be developed in any resource constraint devices running an embedded Linux.

Acknowledgement

We would like to acknowledge Haldia Petrochemicals Ltd. and TCG Foundation for partially supporting the research through the grant entitled “Cyber Security Research in CPS”. We are also grateful to the anonymous reviewers for their insightful comments and suggestions.

References

[1] Sanjeev Das, Yang Liu, Wei Zhang, and Mahintham Chandramohan. Semantics-based online malware detection: towards efficient real-time protection against malware. *IEEE transactions on information forensics and security*, 11(2):289–302, 2016.

[2] Mahinthan Chandramohan, Hee Beng Kuan Tan, Lionel C Briand, Lwin Khin Shar, and Bindu Madhavi Padmanabhuni. A scalable approach for malware detection through bounded feature space behavior modeling. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 312–322. IEEE, 2013.

[3] Federico Maggi, Matteo Matteucci, and Stefano Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(4):381–395, 2010.

[4]The %CPU Usage and %MEM Usage signifies the share of elapsed CPU time and available physical memory respectively calculated per second.

[4] Sarani Bhattacharya and Debdeep Mukhopadhyay. Who watches the watchmen?: Utilizing performance monitors for compromising keys of rsa on intel platforms. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 248–266. Springer, 2015.

[5] Sarani Bhattacharya and Debdeep Mukhopadhyay. Utilizing performance counters for compromising public key ciphers. *ACM Transactions on Privacy and Security (TOPS)*, 21(1):5, 2018.

[6] Manaar Alam, Sarani Bhattacharya, Debdeep Mukhopadhyay, and Sourangshu Bhattacharya. Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks. 2017.

[7] Manaar Alam, Sarani Bhattacharya, Debdeep Mukhopadhyay, and Anupam Chattopadhyay. Rapper: Ransomware prevention via performance counters. *arXiv preprint arXiv:1802.03909*, 2018.

[8] Corey Malone, Mohamed Zahran, and Ramesh Karri. Are hardware performance counters a cost effective way for integrity checking of programs. In *Proceedings of the sixth ACM workshop on Scalable trusted computing*, pages 71–76. ACM, 2011.

[9] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *ACM SIGARCH Computer Architec. News*, volume 41, pages 559–570. ACM, 2013.

[10] Xueyang Wang and Ramesh Karri. Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In *Proceedings of the 50th Annual Design Automation Conference*, page 79. ACM, 2013.

[11] Xueyang Wang and Ramesh Karri. Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(3):485–498, 2016.

[12] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

[13] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. Unsupervised anomaly-based malware detection using hardware features. In *International Workshop on Recent Advances in Intrusion Detection*, pages 109–129. Springer, 2014.

[14] Xueyang Wang, Sek Chai, Michael Isnardi, Sehoon Lim, and Ramesh Karri. Hardware performance counter-based malware identification and detection with adaptive compressive sensing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):3, 2016.

[15] Yanzhi Dou, Kexiong Curtis Zeng, Yaling Yang, and Danfeng Daphne Yao. Madecr: Correlation-based malware detection for cognitive radio. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 639–647. IEEE, 2015.

[16] Meltem Ozsoy, Khaled N Khasawneh, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Hardware-based malware detection using low-level architectural features. *IEEE Transactions on Computers*, 65(11):3332–3344, 2016.

[17] Xueyang Wang, Charalambos Konstantinou, Michail Maniatakos, and Ramesh Karri. Confirm: Detecting firmware modifications in embedded systems using hardware performance counters. In *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*, pages 544–551. IEEE, 2015.

[18] Rana Elnaggar, Krishnendu Chakrabarty, and Mehdi B Tahoori. Run-time hardware trojan detection using performance counters. In *Test Conference (ITC), 2017 IEEE International*, pages 1–10. IEEE, 2017.