LAMBDA: Lightweight Assessment of Malware for emBeddeD Architectures

Anonymous Author(s)

1 INTRODUCTION

Contributions:

The contributions of the paper are thus summarized as follows:

(1) We propose a lightweight detection methodology for malware analysis. The detection is based on inspecting the effect of malwares on a set of benign executables and observing the effect from HPC-events. The technique is based on using the availability of multiple cores in modern processors, where one core is used as a watchdog, while another core is the target core being sanitized from malwares being executed.
(2) We propose to make the test on-line and hence lightweight. To be precise, our test is based on evaluation of statistical T-test, and the online formulation does not require storing the entire trace of HPC events.

2 RELATED WORKS

Significant amount of work has been reported in the literature on detection of malware and possible methods to prevent further damage by them. A self organizing map (SOM) based approach is presented by Zhai et al. in [1] to detect the malware. Here the malware are characterized using processorâĂŹs program counters, cycles per instruction etc. Later later these features are used in further detection. Sanjeev et al. in [2] has presented a frequency centered model in tracking down the behavior of running process. This work has aimed at modeling programs based on their system call patterns and had included an early prediction feature. Demonstration of delay gadgets over powering signature based protection schemes is presented in [3]. A software reconfigurable approach along with signature based defense is discussed in this work, which can withstand delay gadgets. The limits of exploring the anomaly detection is presented in [4]. This work made use of opaque constants in developing obfuscation techniques which can evade detection tools. This also highlighted the limits in static analysis based anomaly detection. Developing transparent techniques which does not produce any side effect while detecting the anomaly is discussed in [5]. This method focused on developing hardware virtual extensions which reside outside the target OS environment, giving no hint to anomaly. Work [6] also focused on anomaly detection using contiguous and dis contiguous system call patterns which reflect the intrinsic activities of the process. Their model aimed at reducing the false alarm rate during the detection process.

A machine learning based method for classification of runtime programs in malicious or benign set has been discussed in [7]. This approach detects the malware based on their efficiency to embed function call graphs into selected feature mapping. An OPCODE sequence based ambiguity detection is discussed in [8]. This method takes into consideration both the relevance and the frequency of occurrence of the opcode sequence. A method for characterizing the normal behavior as against malware is presented in [9]. It aimed to observe the perturbations in the system call signatures hence detecting the normal behavior. In work [10], system call arguments along with their inter relation ship are taken into account while detecting the malware. This approach developed a behavioral Markov model to capture the correlations and hence the anomalies. Canali et al. in their work [11] have analyzed the complexity of anomaly detection models against their accuracy. It has been reported that the relationship is non linear across the design space and only analytical reasoning is not sufficient to find the optimal model. With the increase in size of the anomaly, the number of features also increase. In an attempt to keep restriction on the feature count a bounded model (BOFM) is presented in work [12]. Work in [13] used the difference of opcodes in generating the alarm for malware. Difference in statistical distribution of the OPCODE and the rarity of its occurrence are used to detect the malware. Computation of similarity between two executables which have a similar OPCODE frequency is performed in [14]. Along with the relevance of OPCODE the weight of sequence is also taken into account. OPCODE based method is also used in solving metamorphic malware detection [15]. Similarity of various executable files is also explored in this work. Attempts to build an malware detector in hardware using performance counters is done in [16]. This hardware anomaly detector is designed to run securely beneath the system software such that robustness of the detection is increased. Work mentioned in [17] uses hardware low level features in malware detection. This approach also uses an supervised machine learning technique to classify the hardware features into either benign or malware.

Detection of malware that modify the system call is addressed in work [18]. They focused on counting the hardware events that occur during each system call execution. Ozsoy et al. in their work [19] design a hardware malware detector which acts as first line of defense to differentiate between benign and malware. After the prioritization of the hardware classifier further software detection mechanisms are carried out. In work [20], a singular value decomposition (SVD) is used to reduce the hardware event vectors into a set of characterization features. These reduced features are further used by machine learning techniques to identify malware. malware detection in case of embedded systems is addressed in [21], which tries to observe the behavioral differences. The work also aimed to minimize the hardware support so that enabling of detection can be done in real time. A new class of malware called the jump oriented programming are discussed in [22]. As against the return oriented programming (ROP) [23] these new malware do not rely on the stack and fi*ret*fi instruction making them more evasive. In order to handle sophisticated malware a fine grain control flow check is presented in [25],[24]. The check policies cover control transfer of functions, nature of indirect calls and indirect jumps. In this paper, we use a multi processor module to extract the behavior of a process during its runtime. The address space of one of the processor can be

accessed by the other so that the critical information exchange can be uninterrupted. Usage of hardware performance counters (HPC) which cannot be evaded by any obfuscation techniques, gives us an advantage over other software based anomaly detection techniques.

3 MOTIVATION FOR INTRODUCING LAMBDA

In the previous section, we have seen the various domains used in assessing an unknown program, say systems calls, opcodes, performance counters etc. Though systems call based detection methods need less of probing, these approaches are less effective in case of specific malware, like kernel modifying root kits. On contrary Hardware Performance Counters(HPCs) which act at much lower level than systems calls, can effectively track the variations caused by such malware. Hence to design an assessment technique with greater coverage, we present our approach LAMBDA, which relies on HPCs. This approach is capable of measuring HPC counts for each systems call present in the execution trace of a given program. This will help us to analyze deep enough such that Kernel modifying malware is also detected. To substantiate the above claim, we downloaded the commonly available rootkit Adore-Ng and run on Linux virtual Machine version 2.6.35. For a set of basic shell commands (ls, ps, who, pwd) we monitored three performance counters namely Instructions, Branches, Cycles. Percentage change in the value of each of these counters along with change in system calls is shown in Figure 1.

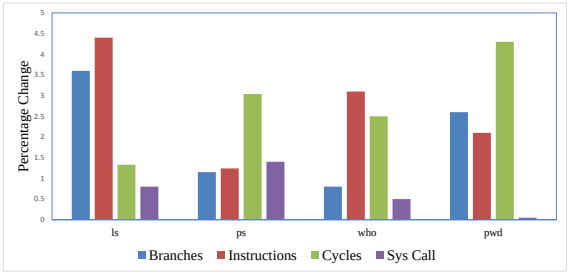

Figure 1: Percentage change in HPC values and number of System calls before and after application of Adore-Ng (ref karri 2013)

Work described in (cite Sanjeev) has extensively relied on finding frequencies of certain system call patterns in unknown programs. Using those frequencies and machine learning methods they predict the nature of unknown program. However this method is less effective when the difference between benign and malicious environment is too narrow. In Fig. 1, it can be clearly seen that for cases 'who' and 'pwd' the percentage difference in system calls (Sys Call) is less than 0.5%. Using the work (cite sanjeev), this case is not possible to detect. It can be observed that, in almost all the cases the HPCs could record a significant difference averaging more than 2.5%. This proves that HPCs could be more effective in assessing the Malware compared to Sys Call.

As mentioned in previous section there are significant works which used HPCs in their detection methodologies (cite karri 2016). However the rationale behind choosing the HPCs that are to be monitored is little focused. Arbitrary choice of HPCs to be monitored, result in large number of False Positives. We can observe from Fig. 1, that different HPCs are better for different shell commands. Our approach LAMBDA addresses the above issue by finding suitable tuples which can keep false positives within threshold and yet lightweight. We elaborate our method in the Section 5.

4 PRELIMINARIES ON STATISTICAL TESTING AND TESTING OF HYPOTHESIS

A fundamental question in several scientific discourses is whether two sets of data are significantly different from each other. The most common approach to answer such a question is Welch's t-test in which the test statistic follows a Student's t-distribution. The aim of a t-test is to provide a quantitative value as a probability that the mean μ of two sets are different. In other words, a t-test gives a probability to examine the validity of the *null hypothesis* as the samples in both sets were drawn from the same population, ie., the two sets are not distinguishable.

Malware detection is an increasingly important field of research, with the failure of prevention techniques, as evidenced by the continued presence of vulnerabilities and exploits. Thus lightweight and online detection mechanisms for detections of malwares are of great interest to the security community. In this work, we focus on improving the state-of-the-art detection mechanism by using the idea of Statistical The objective of the test strategy is thus to detect whether there is any. Thus the purpose is to develop an *estimation* technique, to decide whether the statistical distribution generated by a target program is more distant to the benign distribution or the malware distribution. This will rely on the *Theory of Statistical Hypothesis* and *Theory of Estimation*. The objective of this subsection is to provide a quick overview on the topic.

4.1 Sampling and Estimation

Sampling denotes the selection of a part of the aggregate statistical material with a view to obtaining information of the whole. This totality of statistical information on a particular character of all the members relevant to an investigation is called Population. The selected part which is used to ascertain the characteristics of population is called *Sample*. The main objectives of a good sampling is to obtain maximum information about the population with minimum effort, and to state the limits of accuracy of estimates based on samples.

Any statistical measure calculated on the basis of sample observations is called a *Statistic*, eg. sample mean, sample variance. On the other hand, statistical measures based on the entire population is called a *Parameter*, eg. population mean, population variance. The sample depends on chance, and hence the value of a statistic will vary from sample to sample, while the parameter remains a constant. The probability distribution of a statistic is called *sampling distribution*, and the standard deviation of the sampling distribution is called *standard error*, denoted as SE.

In the sequel, we assume simple random sampling, which implies that in the process of selecting a sample, every unit of the population has an equal probability of being included. Furthermore, we consider that the sampling is such that the probability of selection of any particular member remains constant throughout the selection process, irrespective of the fact that the member has been selected earlier or not. Such a sampling can be obtained by performing a simple random sampling with replacement (SRSWR), and is commonly called as simple sampling. It may be mentioned here that when the population size is infinite, even performing a simple random sampling without replacement (SRSWOR) will also result in simple sampling. Thus in cases where the population size is extremely large (say the plaintext pool in case of a 128 bit block cipher) both SRSWR and SRSWOR will result in simple sampling. Thus, we can assume that each sample members has the same probability distribution as the variable x in the population. Hence, the expectation $\mathbb{E}[x_i] = \mu$, where μ is the population mean. Likewise, the variance $\text{Var}[x_i] = \mathbb{E}\left[(x_i - \mu)^2\right] = \sigma^2$, which is the population variance.

It can be proved that standard error for the mean, $SE(\bar{(x)}) = \frac{\sigma}{\sqrt{n}}$, where n is the size of the sample. We state formally a subsequent result on standard errors which will be useful to understand the subsequent discussion on the detection test.

Theorem 4.1. *Consider two independent simple samples of sizes n_1 and n_2, with means $\overline{x_1}$ and $\overline{x_2}$, and standard deviations σ_1 and σ_2 respectively, then:*

$$SE(\overline{x_1} - \overline{x_2}) \quad = \quad \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}} \tag{1}$$

Different probability distributions are used in sampling theory, and they are all derived from the Normal distribution. They are 1) Standard Normal Distribution, 2) Chi-square (ψ^2) Distribution, 3) Student's t Distribution, and 4) Snedecor's F Distribution.

In the following we provide a quick refresher to the Standard Normal Distribution, and the Student's t Distribution which shall be useful for understanding the remaining part of the sequel and in introducing the T-test.

4.2 Some Statistical Distributions

If a random variable x is normally distributed with mean μ and standard deviation σ, then the variable $z = \frac{x-\mu}{\sigma}$ is called a standard normal variate. The probability distribution of z is called Standard Normal Distribution, and is defined by the probability density function (pdf), $p(z) = (1/\sqrt{2\pi})e^{-z^2/2}$, where $-\infty < z < +\infty$.

An important result is if $\overline{x}$ denotes the mean of a random sample of size n, drawn from a normal population with mean μ and standard deviation (s.d.) σ, then, $z = \frac{\overline{x}-\mu}{\sigma/\sqrt{n}}$ follows standard normal distribution.

Likewise, a random variable is said to follow Student's t-distribution, or simply t-distribution, if its pdf is of the form: $f(t) = K(1 + \frac{t^2}{n})^{-(n+1)/1}$, where K is a constant and $-\infty < t < +\infty$. The parameter n is called the number of degrees of freedom (df.).

In statistics, the number of degrees of freedom is the number of values in the final calculation of a statistic that are free to vary. Estimates of statistical parameters can be based upon different amounts of information or data. The number of independent pieces of information that go into the estimate of a parameter are called the degrees of freedom. In general, the degrees of freedom of an estimate of a parameter are equal to the number of independent scores that go into the estimate minus the number of parameters used as intermediate steps in the estimation of the parameter itself (i.e. the sample variance has N-1 degrees of freedom, since it is computed from N random scores minus the only 1 parameter estimated as intermediate step, which is the sample mean).

4.3 Estimation and Test of Significance

The objective of sampling is to infer the features of the population on the basis of sample observations. Statistical inference has two different ways: 1) Point Estimation, and 2) Interval Estimation. In point estimation, the estimated value is given by a single quantity, which is a function of the given observations. In interval estimation, an interval within which the parameter is expected to lie is given by using two quantities based on sample values. This is known as Confidence Interval, and the two quantities which are used to specify the interval, are known as Confidence Limits.

Let $x_1, x_2, \cdots, x_n$ be a random sample from a population of a known mathematical form which involves an unknown parameter θ. The confidence intervals specify two functions t_1 and t_2 based on sample observations such that the probability of θ being included in the interval (t_1, t_2) has a given value, say c: ie. $P(t_1 \leq \theta \leq t_2) = c$. The probability c with which the confidence interval will include the true value of the parameter is known as Confidence Coefficient of the interval.

Let, us illustrate this using an example. Let us consider a random sample of size n from a Normal population $N(\mu, \sigma^2)$, where the variance σ^2 is known. It is required to find confidence intervals for the mean, μ. We know that the sample mean $\overline{x}$ follows approximately a normal distribution with mean μ and variance σ^2/n. Thus, the statistic $z = (\overline{x} - \mu)/(\sigma/\sqrt{n})$ has a standard normal distribution. From the properties of the standard normal curve, 95% of the area under the standard normal curve lies between the ordinates at $z = \pm 1.96$. Thus, we have:

$$P[1.96 \leq (\overline{x} - \mu)/(\sigma/\sqrt{n}) \leq 1.96] \quad = \quad 0.95 \tag{2}$$

Thus arranging terms, the interval $(\overline{x} - 1.96\frac{\sigma}{\sqrt{n}}, \overline{x} + 1.96\frac{\sigma}{\sqrt{n}})$ is known as the 95% confidence interval for μ. For almost sure limits, we replace the value 1.96 by the value 3.

In some cases, the population may not be truly a normal distribution, but the sample distributions of statistics based on large samples are approximately normal.

4.4 Test of Significance: Statistical Hypothesis Testing

Statistical tests often require to make decisions about a statistical population on the basis of sample observations. For example, given a random sample, it may be required to decide whether the population from which the sample has been obtained, is a normal distribution with a specific mean and standard deviation. Any statement or assertion about a statistical population or its parameters is called a Statistical Hypothesis. The procedure which enables us to

decide whether a certain hypothesis is true or not is called Test of Significance or Statistical Hypothesis Testing.

A statistical hypothesis which is set up (ie. assumed) and whose validity is tested for possible rejection on the basis of sample observations is called Null Hypothesis. It is denoted as H_0 and tested for acceptance or rejection. On the otherhand, an Alternative Hypothesis is a statistical hypothesis which differs from the null hypothesis, and is denoted as H_1. This hypothesis is not tested, its acceptance (or rejection) depends on the rejection (or acceptance) of that of the null hypothesis. For example, the null hypothesis may be that the population mean is 40, denoted as $H_0(\mu = 40)$. The alternative hypothesis could be $H_1(\mu \neq 40)$.

The sample is then analyzed to decide whether to reject or accept the null hypothesis. For this purpose, a suitable statistic, called Test Statistic is chosen. Its sampling distribution is determined, assuming that the null hypothesis is true. The observed value of the statistic would be in general different from the expected value because of sampling fluctuations. However if the difference is very large then the null hypothesis is rejected, Whereas, if the differences is less than a tolerable limit then H_0 is not rejected. Thus it is necessary to formally determine these limits.

Assuming the null hypothesis to be true, the probability of obtaining a difference equal to or greater than the observed difference is computed. If this probability is found to be small, say less than 0.05, the conclusion is that the observed value of the statistic is rather unusual, and has arisen because the underlying assumption, ie. the null hypothesis is not true. We say that the observed difference is significant at 5 per cent level of significance, and hence the null hypothesis is rejected at 5 per cent level of significance. The level of significance, say α also corresponds to a $(1 - \alpha)$ level of confidence. If however this probability is not very small, say more than 0.05, the observed difference cannot be considered unusual and is attributed to sampling fluctuations only. The difference, now is not significant at 5 per cent level of significance.

The probability is inferred from the sampling distribution of the statistic. We find from the sampling distribution of the statistic the maximum difference which is exceeded say 5 per cent of cases. If the observed difference is larger than this value, the null hypothesis is rejected. If it is less, there is no reason to reject the null hypothesis.

Let us illustrate this with an example. Suppose, the sampling distribution of the statistic is a normal distribution. Since, the area under the normal curve under the ordinates at mean ± 1.96 (standard deviation) is only 5 per cent, the probability that the observed value of the statistic differs from the expected value by 1.96 times or more the standard error of the statistic (which is the standard deviation of the sampling distribution of the statistic) is 0.05. The probability of a larger difference will be still smaller.

If, therefore the statistic $z = \frac{\text{(Observed Value)–(Expected Value)}}{\text{Standard Error (SE)}}$ is either greater than 1.96 or less than -1.96, the null hypothesis H_0 is rejected at 5 per cent level of significance. The set of values $z \geq 1.96$, or $z \leq -1.96$ constitutes what is called the Critical Region of the test.

Thus the steps in Test of Significance can be summarized as follows:

(1) Set up the Null Hypothesis H_0 and the Alternative Hypothesis, H_1. The null hypothesis usually specifies some parameter of the population: $H_0(\theta = \theta_0)$.
(2) State the appropriate test statistic T and also its sampling distribution, when the null hypothesis is true. In large sample tests, the statistic $z = (T - \theta_0)/SE(T)$, which approximately follows standard Normal distribution is often used. In small sample tests, the population is assumed to be normal and various test statistics are used which follow Standard Normal, Chi-Square, t distribution.
(3) Select the level of significance, α of the test, or equivalently $(1 - \alpha)$ as the level of confidence.
(4) Determine the critical region of the test for the chosen level of significance.
(5) Compute the value of the test statistic z on the basis of sample data and the null hypothesis.
(6) Check whether the computed value of the test statistic lies in the critical region. If it lies, then reject H_0, else H_0 is not rejected.

With this background, we eventually arrive at the proposed T-test based detection test, which is essentially a test of equality of two moments drawn independently and randomly from two populations. The starting point is the first moment, where equality of two means from the two samples are tested for equality. Thus, the statistic $T = \overline{x_1} - \overline{x_2}$, and thus the statistic $z = \frac{\overline{x_1} - \overline{x_2}}{SE(\overline{x_1} - \overline{x_2})}$ is chosen for testing the null hypothesis: $H_0(\mu_1 = \mu_2)$, where μ_1 and μ_2 are the two means for the two independent samples. As discussed, the standard error of the difference of means $\overline{x_1} - \overline{x_2}$ is $SE(\overline{x_1} - \overline{x_2}) = \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}$.

For a large distribution, the test statistic $z = \frac{\overline{x_1} - \overline{x_2}}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$ follows standard normal distribution. However, for tests with any sample sizes, a more exact sampling distribution for z is the t-distribution, and this gives rise to the Welch's t-test. The statistic z then follows the t-distribution with degrees of freedom calculated according to Welch-Satterthwaite, as $v = \frac{SE(\overline{x_1} - \overline{x_2})}{\frac{(\sigma_1^2/n_1)}{n_1 - 1} + \frac{(\sigma_2^2/n_2)}{n_2 - 1}}$. The null hypothesis of two equal means is rejected when the test statistic $|z|$ exceeds a threshold of 4.5, which ensures with degrees of freedom > 1000, $P[|z| > 4.5] < 0.00001$, this threshold leads to a confidence of 0.99999.

Let Q_0 and Q_1 indicate two sets which are under the test. let also μ_0 (resp. μ_1) sand s_0^2 (resp. s_1^2) stand for the sample mean and sample variance of the sets Q_0 (resp. Q_1), and n_0 and n_1 the cardinality of each set. The t-test statistic is computed as:

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}} \tag{3}$$

4.5 Online Computation of the Proposed T-Test

We end this section with a short note on how that the T-test computation between two distributions Q_0 and Q_1, with n_0 and n_1 data points respectively, can be done online, without the need

for storing the distributions under test. This method is, in particular, suitable for computing the T-test for large distributions on resource-constrained devices, where expending significant memory for storing these distributions is not possible. The idea is to maintain the sum of points S_0 and S_1, and the corresponding to the squared sum of points $\hat{S}_0$ and $\hat{S}_1$, for either distribution under test, and to update these on the fly for each new observation. Let the m^{th} observation for Q_0 and Q_1 be $x_{0,m}$ and $x_{1,m}$ respectively. Then, the updation is done as follows:

$$S_0 = S_0 + x_{0,m} \quad , \quad S_1 = S_1 + x_{1,m}$$
$$\hat{S}_0 = \hat{S}_0 + x_{0,m}^2 \quad , \quad \hat{S}_1 = \hat{S}_1 + x_{1,m}^2$$

Following the online computation phase, the sample mean and variance of either distribution may be computed as:

$$\mu_0 = S_0/n_0 \quad , \quad \mu_1 = S_1/n_1$$
$$s_0^2 = \frac{\hat{S}_0 - \frac{S_0}{n_0}}{n_0 - 1} \quad , \quad s_1^2 = \frac{\hat{S}_1 - \frac{S_1}{n_1}}{n_1 - 1}$$

The t-statistic between the two distributions may now be computed as per Equation 3 described above.

4.6 Hardware Performance Counters

Hardware Performance Counters (HPCs) contain rich source of information of the internal activities of the processor. The performance optimizing tools monitor these HPCs and provides these information to processes or users in the aim to improve the performance of the processors. Events such as cpu-cycles, cache-misses, branch-instructions, branch-misses, page-faults, context-switches are some of the many events whose counts are provided by the HPCs. On occurrence of such hardware or software events the counter registers are incremented by one.

A handle on these HPCs can be easily obtained using the commonly used PC profiling tool “Perf" in Linux for statistical profiling of event “branch-miss” from HPCs. In 2009, ‘perf’ subsystem was added to the Linux kernel, and this makes user access to performance counters less clumsy, without kernel patches or re-compiles. The merge of perf event source with the main Linux kernel source tree has provided an easy access to the Linux users to hardware counters for the first time [**?**]. It is to be noted, these perf tools present a simple to use interface with commands like perf stat (obtain event counts), perf record, perf report, perf annotate and perf top for profiling various events such as branch prediction, branch misses, instruction and data cache hits and misses, cpu cycles etc. In our experiments we claim to observe the HPC events like cycles, instructions, cache-references , cache-misses, branches, branch-misses, by running the command ‘perf stat -e branch-misses executable-name’, where the executable is an operating system standard utility, like `ls, netstat`, etc.

In literature [**?**] data from performance counters have been used to develop a malware detector in hardware using machine learning techniques. A new Virtual Machine Monitor (VMM) named NumChecker is proposed in [**?**], which exploits HPCs to detect kernel root-kits in a guest Virtual Machine.

This paper uses low-level HPCs in conjunction with high-level library utilities to develop distinguishers to detect the presence of malware executables. Furthermore, it is a major concern whether a detection mechanism can be bypassed. In this paper, we hence integrate an attestation mechanism which ties all eligible programs to the underlying hardware, by using a DRAM based PUF. The PUF that we employ uses the phenomenon of row-hammers, which which we describe in the next subsection.

5 OVERALL ARCHITECTURE

In this section, we discuss the overall architecture of the proposed scheme, *Lambda*, which consists of the Lightweight Online Malware Detection Unit. The scheme is based on a dual core platform, where one of the cores is used to perform as a watch-dog, while the other core is sanitized against malware. Fig. 2, depicts the overall methodology, where Core 1 is being sanitized, by the Watchdog Core (Core 2). When a new program arrives for execution on the platform, it is tested its behavior (benign or malicious) behavior on the watchdog core through the Malware Detection Methodology described in the next section. Only when it is attested by the watchdog core as benign program, it is allowed to run on the sanitized core. In the following section, we describe the methodology of the malware detection which is being implemented in Watchdog Core (Core 2).

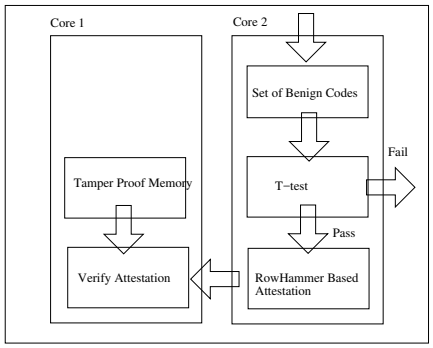

Figure 2: The Lambda Architecture

5.1 The Malware Detection Methodology

Malware programs are nevertheless software programs that still exhibit phase behavior: which means they perform a sequence of activities. It has been shown that these phases correspond to patterns in micro-architectural events. Another vital property which helps in using these low-level micro-architectural events for detecting the presence of malware, is that the events differ radically between programs. Moreover, since the correlation between the program and the micro-architectural events are hard to formalize, it makes it very difficult for malware writers to rewrite the programs, which does the same functionality but have different low-level event trace.

The set of malware can be infinite, however they are expected to perform an expected set of objectives. For instance, a malware would potentially affect the filesystem, try to hide some files, affect the network, may create additional processes, it may affect the run-levels, or change the present working directories. The intuition of our detection strategy is to create a set of benign operating system library executables, called *references*, like `ls, netstat, ps, who, pwd`, and which would be run-time monitored by observing a set of low-level hardware events, called *observers*, like

`cycles, instructions, cache-references, cache-misses, branches, branch-misses`.

Embedded platforms have a set of application programs that are supposed to run on it. For eg. MP3 and video player programs are expected to run on an iPod, and applications such as internal guidance system and GPS are expected to run on an avionics embedded system designed for airplanes. We call such application programs as *benign* application programs. The idea is based on the hypothesis that the performance events would vary significantly when a malicious code runs in the system, compared to when a benign application program runs on it. So, we try to quantify the functional distance, defined by some suitable metric, of the program under test from a set of benign application programs. A smaller distance can be interpreted as closeness in functionality to the benign programs, and as a result, the benign nature of the program under test. Similarly, a larger distance can be interpreted as major deviation between the functionalities of program under test and the set of benign programs, and thus, indicating possible malicious nature of program under test. Thus, the initial pre-processing would be to create templates of benign environment, by observing the HPC observers, by monitoring the set of references. When the malicious code executes, it is expected that the statistics generated by the observers by the references are significantly different.

There are two important advantages of such a detection strategy:

- Since we are characterizing the benign environment, the detection method would be able to identify the presence of unknown malware codes too.
- The detection method is based on the functional objective of a malware and not its specific construct. This enables to develop a strategy, which is dynamic not reliant on static properties of malware codes.

The detection methodology consists of two phases, the training phase and the detection phase. A prerequisite step for both the phases is perf stats collection. We first describe the perf stats collection for a program under test and then proceed with formalizing both the aforementioned phases, in the following section.

5.1.1 Perf Stats Collection. : As discussed in section 4.6, Perf tool allows to get a handle for Hardware Performance Counters. Let the set of indicators or references be denoted as $\mathcal{R} = \{r_1, i_2, \cdots, r_m\}$ and the observers (HPCs) as $\mathcal{H} = \{h_1, h_2, \cdots, h_n\}$. The combination of HPCs and indicators are called tuples. We have $p \times q$ tuples. For program under test, t, when the indicator r_i is run in its presence, we observe the hardware performance counter h_j using the Perf tool. This is done for s trials. Let the value of hardware performance counter h_j for k^{th} trial when indicator r_i is run in presence of t be denoted as $\lambda^k_{i,j}(t)$. Let $\theta_{i,j}(t) = \lambda^1_{i,j}(t), \lambda^2_{i,j}(t), \cdots, \lambda^s_{i,j}(t)$. When done for all $p \times q$ tuples, we obtain a distribution matrix, for the program under test, t, as $\mathcal{D}(t) = \{\theta_{i,j}(t) | i \in \{1, 2, \cdots, m\}, j \in \{1, 2, \cdots, n\}\}$.

5.1.2 Training Phase. : For the training set, we collect two sets of program; one which contains malwares and other which contains benign application programs. Let the malware set be denoted as $\mathcal{M} = \{m_1, m_2, \cdots, m_x\}$ and the benign application programs as $\mathcal{B} = \{b_1, b_2, \cdots, b_y\}$. After the perf stats collection for both the sets, we will obtain two sets of distributions. We denote the distribution for the set of malwares as $\{\mathcal{D}(m_1), \mathcal{D}(m_2), \cdots, \mathcal{D}(m_x)\}$ and for the set of benign application programs as $\{\mathcal{D}(b_1), \mathcal{D}(b_2), \mathcal{D}(b_y)\}$. We say a malware m_α is detected against a benign application program b_β by the tuple (r_i, h_j), if the distance, defined some suitable metric Δ, between the values corresponding to that tuple in the two distributions is greater than a threshold. Thus, the malware is detected if $\Delta(\theta_{i,j}(b_\beta), \theta_{i,j}(m_\alpha) >$ t, for a threshold t.

The objective of the on-line detection mechanism should be to be able to perform a lightweight evaluation, by evaluating the distance using some well-founded statistical technique. We propose the computation of the Welch's t-test, which accepts (or rejects) with a confidence measure whether two distributions are alike based on degrees of freedom and a predefined significance level. Therefore, the distance metric, Δ, can be taken as the t-values obtained by applying Welch's t-test on the distributions. Intuitively, the indicator-HPC tuple that is able to detect more malwares in the training phase should have higher influence or weight during the actual detection (testing) phase. A basic strategy to assign weights to tuples would be through the count of correct classification of a malware by the tuple. If a tuple (r_i, h_j) correctly identifies a malware, m_α against a benign program b_β, then we increase its count. Thus the value of count will range from 0 to the number of malware-benign program pairs, that is $x \times y$. When done for all $p \times q$ tuples, we obtain a count matrix with p columns and q rows. These counts can be normalized to obtain a weight matrix of the tuples. An illustrtation of weight matrix is shown in table 1.

Table 1: Weight Matrix of HPC-Indicator tuple

HPC	I_1	I_2	I_3	I_4	I_5
H_1	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}
H_2	w_{21}	w_{22}	w_{23}	w_{24}	w_{25}
H_3	w_{31}	w_{32}	w_{33}	w_{34}	w_{35}
H_4	w_{41}	w_{42}	w_{43}	w_{44}	w_{45}
H_5	w_{51}	w_{52}	w_{53}	w_{54}	w_{55}
H_6	w_{61}	w_{62}	w_{63}	w_{64}	w_{65}

5.1.3 Testing phase (Detection). : For a new program under test, t whose behavior (benign or malicious) is unknown, the perf stats are collected as described previously. We obtain the distribution $\mathcal{D}(t) = \{\theta_{i,j}(t) | i \in \{1, 2, \cdots, p\}, j \in \{1, 2, \cdots, q\}\}$. Now this distribution is compared against all the benign programs using t-test from the Training phase for each indicator-HPC tuple. We get a count of the number of times t was classified as malware (or benign) by that tuple based on whether the t-value was greater (or lesser) than the t-critical value. After repeating this for all $p \times q$ tuples we get a count matrix similar to the weight matrix obtained in the Training phase. After normalizing the count for each tuple and multiplying it weight obtained in the Training phase we obtain a score for each tuple. After summing over all the $p \times q$ scores we obtain an overall score, S. This S can be interpreted as an overall distance of the program t from the profile benign program. If S is greater than some threshold value, t is classified as malware, otherwise it is classified as benign.

We elaborate the implementation details and the choice of the parameters in the experimental section.

6 EXPERIMENTAL RESULTS

In this section, first we elaborate on the setup and the bench marks we chose to validate our proposed methodology. Later we perform some key experiments to support our choice of indicators, hardware events we monitored and the effectiveness of proposed t-tests for runtime malware detection. Finally we present a comparative study of our approach with some other machine learning based approaches for runtime and observe the overhead of PUF based attestation.

6.1 Experimental Setup and choice of bench marks

We chose an ALTERA SoCKit, model VEEK-MT to validate our proposed methodology. A snapshot of our model is shown in Fig.**??**. This VEEK-MT-SoCKit board consists of a dual-core ARM Cortex-A9 processor with an Altera 28-nm Cyclone V FPGA. As mentioned earlier we allowed unknown programs to run on one of the core C0 (say), simultaneously run the monitoring code on core C1 (watchdog core), thus utilizing both the available cores. We setup a VMware environment to run Ubuntu 32 bit and 64 bit virtual machines, on which our malware dataset are initially characterized and efficiency of our method is verified. The kernel version 2.6.35 is chosen such that most of the malware could successfully run. Since the availability of malicious programs that can run on x86 architecture is plenty, this step is performed. Later we used another set of programs which can run on ARM platform (both malicious and benign) to test the effectiveness of our proposed method on the ARM platform.

In this work we restricted ourselves to detection of linux based malware. We have got the latest malware release from **??**, **??**. We classified these malware into different families using the classification interface provided by **??** as shown in Table. 3. Overall we collected 58 malware samples that could execute on x86 architecture and 10 samples that could run on ARM platform. For benign programs use standard linux benchmark programs, CHStone and UnixBench.

6.2 Analysing Indicator programs, Hardware Performance Counters

In our methodology we have adopted a set of indicator programs which are used to track the behaviour of unknown program. They are ls, pwd, who, netstat, ps. These indicator programs cover various resources of operating systems (OS), namely network, file system, process, memory. Any malicious behaviour by the unknown program can be tracked by observing the above listed OS resources.

As mentioned previously, we characterized the behaviour of the above mentioned indicator programs using a set of hardware performance counters (HPCs) which are available in the Technical Reference Manual of Cortex A9 **??**. These are termed as perf counters. We chose six counters namely cycles, instructions, cache-references, cache-misses, branches, branch-misses from the available set of perf-counters. These are chosen in such a way that the effects of unknown programs on indicator programs can be easily highlighted. The key modules operating for the HPCs are the digital counters which increment the status after completion of each event and the event selectors which determine the type of events to monitor. These HPCs often provide an indication of run time behaviour of a particular program, hence allowing necessary debugging and tuning to be carried out. The key advantage with using HPC are they do not require source code instrumentation.

Table 2: List of Malware Data Set Chosen for Experiments

Malware Family	Train	Test	Validation	Total
Agent	14	4	2	20
Bo	3	1	1	5
Bodoor	1	1	0	2
Boost	1	0	0	1
Dancer	1	0	0	1
Divine	1	0	0	1
Explodoor	2	1	0	3
fpath	1	0	0	1
Ircshell	1	0	0	1
Lala	1	0	0	1
Netbus	1	0	0	1
Phobi	5	2	1	8
Rooter	2	1	0	3
Sitc	1	0	0	1
Small	7	3	1	11
SSH	1	1	0	2
Suffer	1	0	0	1
Unfstealth	1	0	0	1
total	45	14	5	64

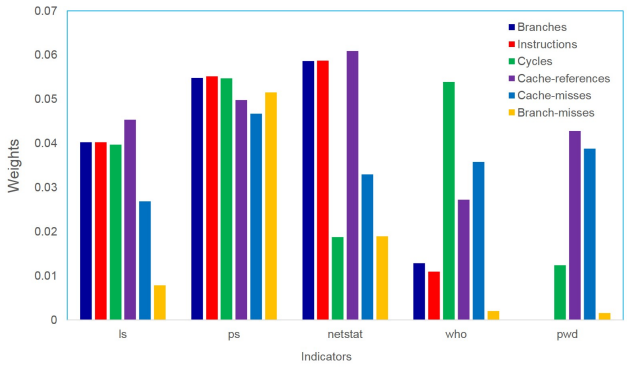

Figure 3: Performed a statistical test for choosing the indicator-tuple combination that can yield less false positive rate.

6.3 Evaluating the t-test based detection mechanism

As described in Section 4.4, we have carried the univariate t-test with the following details. For 5 indicators namely *ps, ls, pwd, netstat, who*, the behavior the six HPCs namely *branches, instructions, cycles, cache-references, cache-misses, branch-misses* are collected. In order to avoid the affect of CPU noise []karri, we performed the t-test over the data 250 consecutive runs of both indicator in presence and absence of the unknown program. The univariate t-test is performed to ensure that there are no false negatives (malware classified as benign) in the detection flow.

Table 3: CPU Performance Comparison

Method	CPU Performance	Hardware Used	detection coverage	Performance Counters
Numchecker	14	4	95	yes
MAP	3	1	93	yes
GuardOL	1	1	97	no
LAMBDA	1	0	99.9	yes

As stated in Section 6.1, we have considered 50 benign bench marks and 100 malware for validating our experiments. The profiles of all the five indicator programs run in the presence of benign programs are stored. In a similar fashion the behavior of indicators in presence of malwares is also stored. Figure.4 shows the behavior of the predefined five indicator programs run in presence of benign programs like adpcm, aes, sha, jpeg.

6.4 Comparative study

Our online t-test mechanism requires minimal storage space, hence comparatively light weight than corresponding machine learning based applications. [(by Manaar): Different machine learning models need to optimize different sets of parameters during training phase, which could get more complex for a large number of datasets. The t-test approach proposed here doesn't require any parameters to be updated (taking the only parameter level of significance as an input), thereby taking much lesser time during training phase.] Machine learning With similar data set both for training and testing cases we formulated a multi layer perceptron (MLP) based approach and verified its accuracy. Other classification approaches like Random forest, decision tree, knn were also tested with the same data using WEKA classification tool. The receiver operating characteristics curve (ROC) is plotted and shown in Fig. **??**.

We compare our propose methodology with three major works in the same field bone by Karri etal., Ozosoy etal. and Sanjeev etal. Since we focused on developing lightweight detection mechanism, we use CPU performance and amount of hardware needed as metrics of comparison. The results of our comparative study are shown in Table. 2.

-

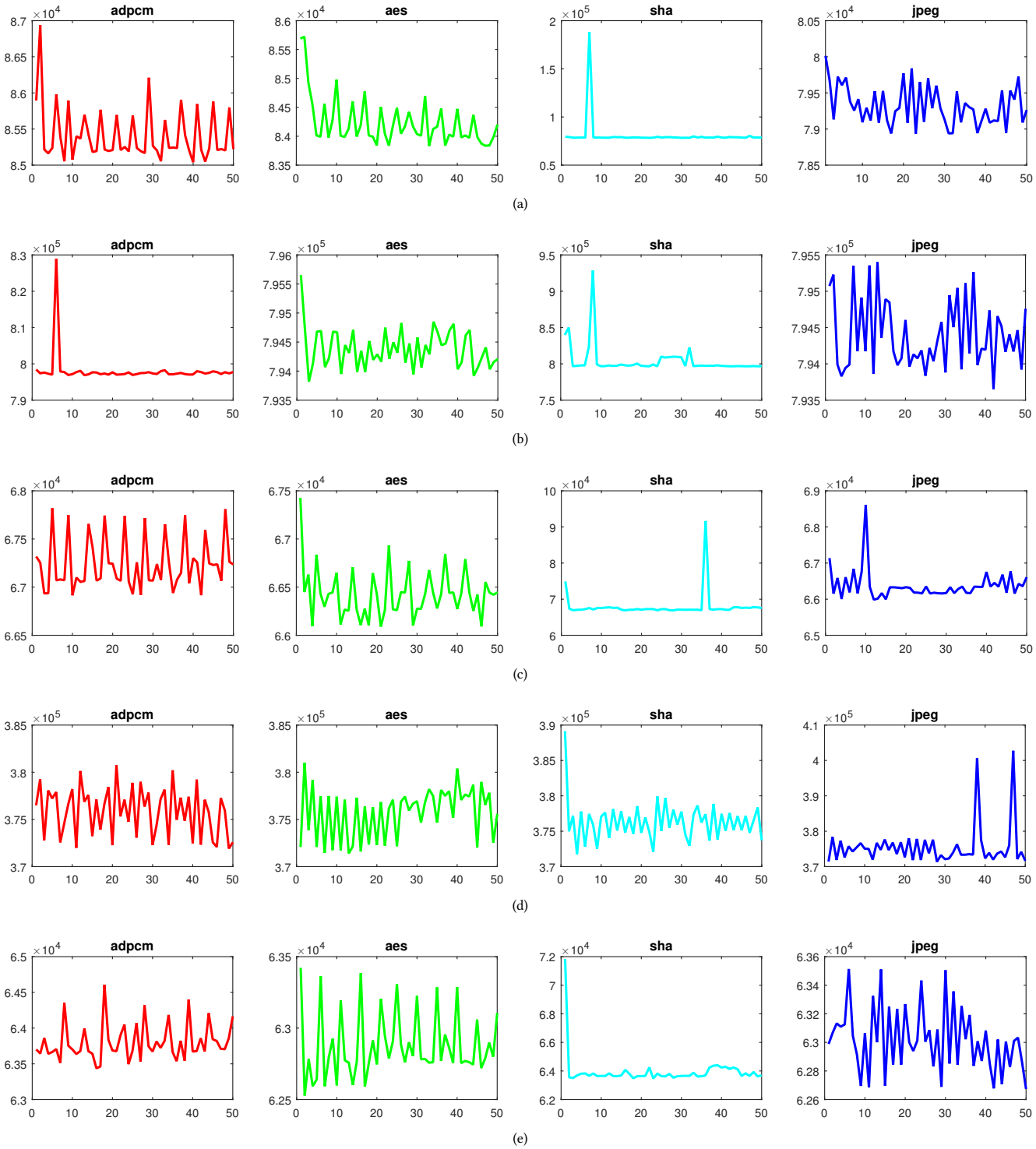

Figure 4: Behavior of the HPC(cycles) for various indicator programs (a) ls (b) ps (c) who (d) netstat (e) pwd, when run in presence of a standard benign benchmarks (adpcm, aes, sha, jpeg)

Figure 5: **The above 'VEEK-MT-SoCKit' board, a product from Altera is used to validate our runtime anomaly detection approach.**