EDDIE: EM-Based Detection of Deviations in Program Execution

Alireza Nazari * Nader Sehatbakhsh† Monjur Alam‡ Alenka Zajic◇ Milos Prvulovic∓
Georgia Institute of Technology
Atlanta, GA, USA.
Email: {*anazari,†nader.sb,‡malam3}@gatech.edu
◇alenka.zajic@ece.gatech.edu ∓milos@cc.gatech.edu

ABSTRACT

This paper describes EM-Based Detection of Deviations in Program Execution (EDDIE), a new method for detecting anomalies in program execution, such as malware and other code injections, without introducing any overheads, adding any hardware support, changing any software, or using any resources on the monitored system itself. Monitoring with EDDIE involves receiving electromagnetic (EM) emanations that are emitted as a side effect of execution on the monitored system, and it relies on spikes in the EM spectrum that are produced as a result of periodic (e.g. loop) activity in the monitored execution. During training, EDDIE characterizes normal execution behavior in terms of peaks in the EM spectrum that are observed at various points in the program execution, but it does not need any characterization of the malware or other code that might later be injected. During monitoring, EDDIE identifies peaks in the observed EM spectrum, and compares these peaks to those learned during training. Since EDDIE requires no resources on the monitored machine and no changes to the monitored software, it is especially well suited for security monitoring of embedded and IoT devices. We evaluate EDDIE on a real IoT system and in a cycle-accurate simulator, and find that even relatively brief injected bursts of activity (a few milliseconds) are detected by EDDIE with high accuracy, and that it also accurately detects when even a few instructions are injected into an existing loop within the application.

CCS CONCEPTS

• **Security and privacy** → **Side-channel analysis and countermeasures**;

KEYWORDS

Hardware Security, EM Emanation, Malware Detection, Internet-of-Things

*†These two authors contributed to the paper similarly (author order does not reflect the extent of contribution).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISCA '17, June 24-28, 2017, Toronto, ON, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/XXXXX.XXXXXX

1 INTRODUCTION

As network-connected computing systems become a part of the infrastructure and are integrated into an increasing number of physical objects to form the Internet of Things (IoT), it is becoming increasingly important to detect malicious additions, removals, and changes to the software of the system [55]. Typically, an attack exploits a vulnerability in the software, e.g. a memory-related one [29], to take over (hijack) control-flow and execute its malicious code. Attacks are more valuable to their perpetrator if they remain undiscovered, so they typically introduce limited changes to the execution of the victim software, while leaving most of its functionality and code unaffected.

The most widely used form of defense against attacks are anti-viruses, which statically scan memory and/or files to detect patterns that indicate the presence of known viruses, Trojans, etc., [27]. To counteract this, some attacks use code mutation, encryption, and other approaches that make them hard to detect statically [84].

Dynamic detectors try to overcome this problem by monitoring the execution of the application, looking for suspicious activity. Some dynamic detectors look for dynamic behaviors that correspond to known types of attacks, but detection of previously unknown attacks typically relies on creating a model of its correct behavior and then looking for deviations from this model. Ideally, a detector would have a model that exactly specifies all possible correct behaviors and only correct behaviors, and then any deviation from this model can be reported as a problem. Unfortunately, most software is too complex to be fully specified in this way, and the checking of such detailed models would be too expensive, so most practical detectors track how some aspect of the execution evolves over time, build a model that approximates how that aspect of the execution behaves normally, then monitor that aspect of execution and report a problem when it deviates too much from the model.

The aspects of the execution that have been used for such anomaly detection include various software/hardware activities such as system calls, function calls, control flow, data flow, performance counters, etc. [2, 5, 9, 22, 25, 28, 53, 58, 61, 62, 66, 67, 76, 80, 82, 86].

However, information about these aspects of the execution is only available within the monitored system itself, so the detector must either execute on the monitored system or add dedicated software or hardware to the monitored system to convey the required information to another system. This creates both performance and resource (e.g. memory) overheads on the monitored system, and these overheads become very high if the monitoring uses an information-rich aspect of execution, such as tracking the basic-block sequence and verifying instruction words as they are executed. Furthermore, relying on the monitored system itself to actively participate in its own verification

allows the monitoring itself to be subverted in the same attack that takes over the whole system.

Recent work [13, 72] has shown that electromagnetic (EM) signal that are emanated as a side-effect of program execution can be successfully used for execution profiling, where the EM-based profiler attributes execution time by deciding which of the signal fragments observed in training is the best match for each fragment of the training-time signal. This work shows that EM signals may also be usable for anomaly detection. However, the decision in anomaly detection is not about selecting one of training-time signals. Instead, the anomaly detection decisions are about whether the signal *sufficiently deviates* from training-time signals, for some well-chosen metric of "deviates" and method of deciding about "sufficiently".

This paper describes an EM-based anomaly detection method, which we call *EM based Detection of Deviations in Program Execution (EDDIE)*. EDDIE identifies spikes in the EM spectrum of the received EM signal, which correspond to loops and other periodic activity. EDDIE's training consists of characterizing the the spectral spikes observed during training in each loop and other repetitive region of the program, as well as the statistical properties of these spikes. EDDIE's anomaly detection consists of assessing the probability that the spikes observed during monitoring belong to the same statistical distribution that was learned during training.

We believe that EDDIE would be particularly useful as a dedicated monitor for systems embedded in critical infrastructure, such as industrial, power plant, and other control systems, or for auditing the behavior in-body medical devices when a patient visits the doctor's office. In both scenarios, software changes and direct manipulation of devices is highly undesirable, while the cost of an EDDIE setup (see Section 5.1) is very low compared to the total cost of the monitored system(s).

Our evaluation focuses on how sensitive EDDIE is to the amount of injected execution, i.e. how many instructions can the injected code execute before being detected by EDDIE. We evaluate both a burst of injected execution, when the attack's injected work is performed all at once, and an injection into an existing loop, e.g. to improve stealth by spreading the injected work over time by interleaving it with the original work of the application's loop. We find that even relatively brief injected bursts of activity (a few milliseconds) are detected by EDDIE with high accuracy, and that it also accurately detects injections of even a few instructions into an existing loop body. One way to interpret these results is that, to avoid detection by EDDIE, the amount of injected execution per second must be very low, i.e. the injected code can avoid detection only if its activity utilizes a very small percentage of the monitored system's performance potential.

The main contributions of this paper are:

- Proposing EDDIE, a new approach to monitoring execution and detecting anomalies without any modification to or cooperation from the monitored system.
- A proof-of-concept implementation of EDDIE that demonstrates its potentials.
- A detailed characterization of EDDIE implementation in the context of code injection, showing that EDDIE can detect injected code even if a brief (a few milliseconds) burst of injected code is executed, and that it can detect injections of just a few instructions within a loop body.

The rest of this paper is organized as follows. Section 2 is a brief overview of the background material related to our approach, Section 3 provides an overview of the EDDIE approach, Section 4 discusses details of our proof-of-concept implementation of EDDIE, Section 5 presents the results of our experimental evaluation of EDDIE, Section 6 reviews related work, and Section 7 concludes our paper.

2 BACKGROUND

Electronic circuits within computers generate electromagnetic (EM) emanations [3, 38, 85] as a consequence of changes in current flow within a computing device. Since current flows in a system vary with program activity, these EM emanations often contain some information about program activity in the system. Most research work on EM emanations has focused on the risks they create as side-channels, i.e., as a way for attackers to extract sensitive data values (such as cryptographic keys) from the system [3, 30], and on countermeasures against such attacks, primarily for smart-cards used for authentication and payments [41, 45, 51, 64, 65, 73, 74, 77, 78].

More recently, power signals have been used to learn more about program behavior. For example, power fluctuations are used to identify web-pages during browsing [16] or find anomalies in software activity [4, 18]. EM emanations are used in [13, 72] to profile software activity. An observation is made in [72] that, when a loop executes on the profiled device, the emanated spectrum has spectral features that are repeatable and easy to observe. After detailed investigation, it was found that loops tend to have "peaks" in the spectrum that corresponds to per-iteration timing, so program execution can be efficiently profiled at the loop granularity by monitoring the emanated EM spectrum. It was also found that the strongest "peaks" are observed when an existing periodic signal (e.g. a clock signal with frequency F) is amplitude modulated by changes in processor activity.

To illustrate this concept, Figure 1 shows a spectrum of an AM modulated loop activity. In Figure 1 we can observe three peaks. The middle peak is at frequency $F_{clock} = 1.0079\ GHz$ and corresponds to the processor clock signal of the monitored device. The two peaks to the left and right of this peak (at frequencies $F_{1L} = 1.00536\ GHz$ and $F_{1R} = 1.01064\ GHz$, respectively) are the spectral components of a loop activity that is amplitude modulated by the clock (carrier) signal. The corresponding loop activity has per-iteration execution time of approximately $T = 379\ ns$, thus whenever this loop is active, it produces sidebands with frequency of $f = 1/T = 2.64\ MHz$ to the left and the right from the carrier signal.

While [72] has shown that the EM spectra can be used to deduce which loop in the program is currently active, this paper exploits EM spectra to characterize normal program behavior and then monitor program execution to identify deviations from normal behavior by using different statistical methods.

3 OVERVIEW OF EDDIE

The overall idea of EDDIE is to use the observed EM spectra over time as a surrogate for program behavior over time, gather training data about what the EM spectra should look like in each part of the

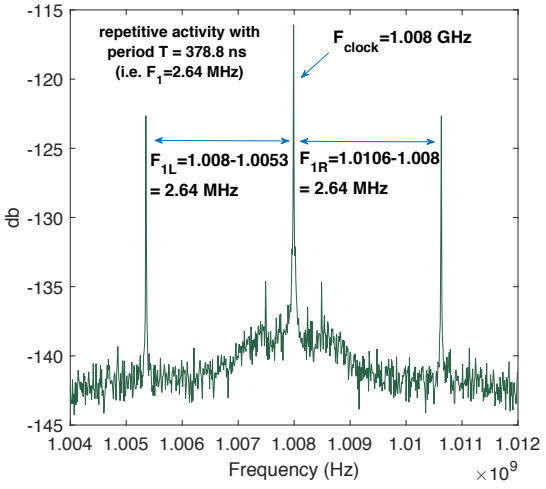

Figure 1: Spectrum of an AM modulated loop activity.

program during correct execution, and then monitor execution by looking for situations where the observed EM spectra statistically deviate from expected spectra, i.e., the observed spectra are unlikely to be outcome of a correct execution.

EDDIE obtains the EM signal from an antenna, uses the Short-Term Fourier Transform (STFT) to convert this continuous signal into a sequence of overlapping windows, and then converts the signal in each window into its spectrum, which we call Short-Term Spectrum (STS). All the actual training and monitoring in EDDIE is done on this sequence of Short-Term Spectra (STSs). Training in EDDIE consists of obtaining a number of STSs for each loop nest and each loop-to-loop transition that is possible during a valid execution in the program. During monitoring, EDDIE compares the observed STSs to those obtained during training, and reports a problem when the observed sequence of STSs is unlikely to have been produced by a valid execution.

Compared to directly using the corresponding time-domain signal values, use of STSs in EDDIE is advantageous both in terms of efficiency and in terms of accuracy. During monitoring, EDDIE needs to assess whether the difference between the monitoring-time signal and the training-time signal is larger than the difference that can be expected due to "usual" variation in the signal such as signal noise and measurement error, cache misses and other low-level events, etc. At any given time, the program is very likely to be executing a loop nest, so an STS is very likely to have a few prominent features (peaks) that are much stronger than the noise at that frequency, and whose position in the spectrum (frequency) is very resilient to "random" occurrences of low-level hardware events and completely unaffected by signal noise. This means that comparisons among STSs are usually very efficient because they involve checking only a few points (frequencies) in the spectrum, and it also means that STSs for the same region of code tend to be very similar to each other, so even small differences in STS peaks provide high confidence that something else is executing. In contrast, the time-domain signal for the same time window, typically consists of fluctuations that are relatively weak (compared to signal's noise level) and whose position in the time window shifts significantly (compared to the duration of each fluctuation) as a result of low-level hardware events. This means that comparisons would have to include most of the real-time samples collected in the time window, and that accuracy would suffer because relatively large differences among signals would have to be tolerated to avoid producing false positives.

4 IMPLEMENTATION OF EDDIE

This section describes our proof-of-concept implementation of EDDIE, using the *Susan* benchmark from the MiBench [36] suite to illustrate key details of the method. We first describe the training that is needed for EDDIE to build its model of correct execution of a program (Section 4.1). We then describe the three key aspects of EDDIE's monitoring implementation: using a statistical test on STSs to identify anomalous sequences of STSs (Section 4.2), how EDDIE controls the trade-off between accuracy and latency of its anomaly detection (Section 4.3), and how EDDIE decides when to actually report an anomaly to the user (Section 4.4).

4.1 Training in EDDIE

The main goal of the training phase is to (*i*) find the possible sequences in which loop and inter-loop regions may execute, and (*ii*) collect enough sample windows that correspond to each loop and inter-loop region of the program, along with information about which region each of the training window corresponds to.

The possible sequences of regions are represented as a loop-level state machine that is identified through compile-time analysis. This analysis begins with the traditional control flow graph (CFG) of the program. Each node in the CFG is a basic block, and an edge from some basic block A to some basic block B exists if execution of block A can immediately be followed by execution of block B. Note that the CFG defines a state machine that constrains the set of basic block sequences that may be observed in an execution of that program. To obtain the region-level state machine, for each loop nest we merge all the nodes in the CFG that belong to that loop nest into a single loop-region node, eliminating all edges between basic blocks inside that nest and also all edges that go from that nest directly to itself. We then eliminate each of the remaining basic-block nodes from the graph by connecting the sources of its incoming edges directly to the destinations of its successors, and finally we merge (into a single edge) those edges that have both the same source node and the same destination node. The result of this is a graph that represents the region-level state machine of the program: each state (graph node) represents a loop region and each edge represents an inter-loop region. Note that this region-level state machine is very compact compared to the traditional basic-block-level CFG, and that at runtime the state transitions in the region-level state machine occur much less often because each state represents execution of an entire loop nest.

During compilation for training runs, EDDIE adds instrumentation just before and just after each loop nest in the code that will be used in training runs. In each EDDIE training run, we record the signal while the instrumentation logs the region identifier, entry time, and exit time for each loop region that is executed. This allows us to map each part of the signal to the region that was executing at that time. This instrumentation is light-weight and requires very little

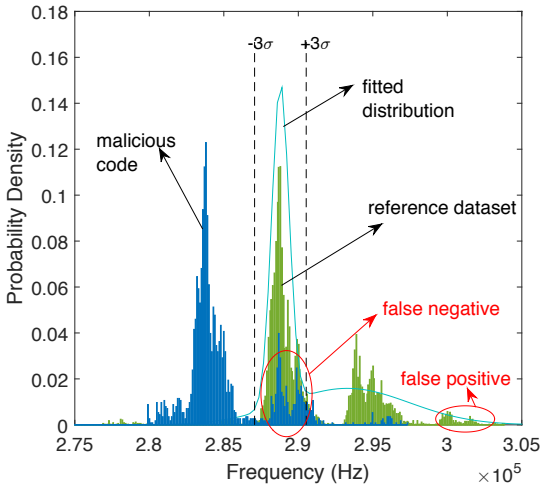

Figure 2: Normal (green) vs. Malicious (blue) activity. Parametric test can lead to inevitable false positives and false negatives.

memory to record its information. For example, only five loop nests are instrumented in the *Susan* benchmark.

EDDIE then runs the application multiple times, each time with different inputs. Multiple runs are needed to improve coverage of the regions, i.e., to obtain signals that correspond to regions that are only executed in some of the runs. Multiple runs also help gather a representative number of signal windows for regions that exhibit variation in behavior, e.g., due to control flow within the body of a loop. For example, our training for the *Susan* benchmark consists of 50 runs (each with different inputs).

The signals collected during training are then divided into sample windows. As discussed in Section 2, loops tend to produce spikes in the spectrum, so for each window EDDIE uses STFT to compute its spectrum, and then identifies the set of peak frequencies in that spectrum. A peak frequency in EDDIE is defined as a frequency at which at least 1% of the entire window's signal energy is concentrated. The number of peak frequencies can differ from window to window, especially if they correspond to different regions of the program. For example, for one loop nest in the *Susan* benchmark our training produces 1,200 windows, each with 15 peaks, while for another loop nest we have 750 windows, each with 7 peaks.

Finally, for each region in the program, EDDIE forms the set of sample windows that belong to that region and performs analysis to determine how many samples need to be jointly considered in statistical tests during the monitoring phase to achieve a desired level of reporting accuracy. This number depends on the statistical test that will be used and is explained in Section 4.2.

In summary, EDDIE's training obtains the application's region-level state machine and, for each region, a reference set of sample windows with their spikes already identified, and the knowledge of how many samples should be used in statistical tests for that region.

4.2 Statistical Test

A key aspect of EDDIE's decision-making is that it cannot be based on *whether* the observed (monitoring-time) STSs differ from the reference (training time) STSs. This is because STSs that belong to the same region of code are almost never *exactly* the same. Reasons for this include noise and external (e.g. radio) interference in the signal, and also "random" variation in program behavior (e.g. a specific path through the loop body may be taken slightly more or slightly less often depending on the window of time we are observing), micro-architectural events such as cache misses, branch mispredictions, etc.

Because of variation among STSs that belong to the same region of code, EDDIE's decision-making is based on statistical tests. This means that EDDIE views each code region as a generator of STSs that vary randomly according to some distribution that is specific to that region. During monitoring, EDDIE uses an appropriate statistical test to compute the probability that the region's reference distribution (the distribution of STSs obtained for that region during training) is the same distribution that has produced the STSs observed during monitoring. If the same-distribution probability is high enough, EDDIE considers the observed STSs to be "as expected" and takes no further action. Conversely, if the same-distribution probability is low enough, EDDIE considers the observed samples as anomalous and takes further action.

The simplest statistical tests are called *parametric* tests because they assume that the distribution belongs to a specific family, e.g. Gaussian (normal) distributions, and can be fully characterized using a relatively small set of parameters (e.g. mean and standard deviation for a normal distribution). However, we have found that many regions of code produce distributions that are poor matches for well-known distribution families. For example, Figure 2 uses green color to show how the frequency of the strongest spike is distributed among reference STSs that all belong to the same loop nest. This distribution is a poor fit for a normal distribution, so for illustration purposes we show (light blue line) the best-fitting multi-modal distribution that consists of two normal distributions. This bi-normal fitted distribution differs significantly from the actual (green) distribution, so statistical tests that assume a bi-normal distribution would report many false positives because the observed distribution of STSs *does* differ from the reference bi-normal distribution.

To overcome this problem, EDDIE uses a *nonparametric* test to compare the observed and reference STS distributions. A nonparametric test can take two groups of data and compute the probability that the two groups are both random samplings from the same population, without any a-priori assumptions about the nature of that population's underlying distribution. For EDDIE, this means that we can test the observed STSs against reference STSs without making any assumptions about which type of distribution the reference STSs belong to. Two best-known non-parametric tests are the *Wilcoxon-Mann-Whitney test* (*U-test*) [20] and the *Kolmogorov-Smirnov test* (K-S test) [54]. The U-test is sensitive to the differences in the median value in each of the two groups of data and the K-S test is sensitive to any difference between the two groups of distribution. We experimented with both tests and found that the K-S test shows better performance, so EDDIE uses that test.

In K-S test, we suppose that the reference data set has m elements with an empirical distribution function of $R(x)$, and that the data set observed during monitoring has n elements with an empirical distribution function of $M(x)$. The K-S test then computes

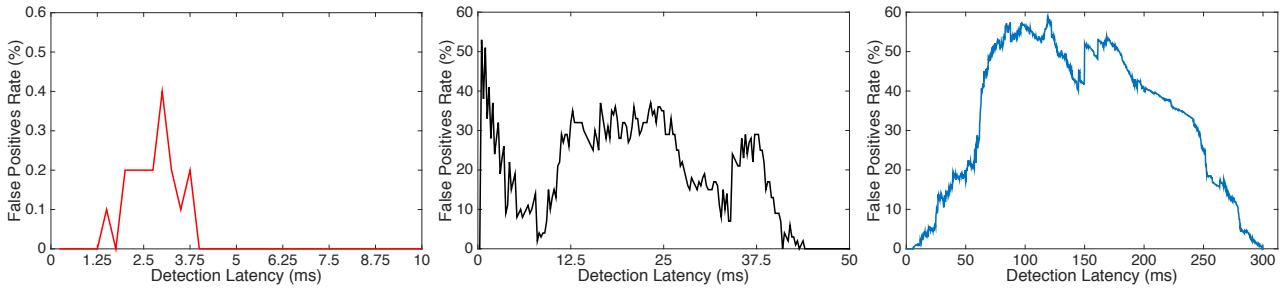

Figure 3: Buffer size selection for three loops, one whose spectrum has one sharp peak and its harmonics (left), one whose spectrum has several peaks and their harmonics (middle), and one whose spectrum has poorly defined peaks (right).

$D_{m,n} = \max_x \mid R(x) - M(x) \mid$, i.e., the largest difference between these two empirical distributions. The null hypothesis, H_0, is *the reference and monitoring-observed data sets were both drawn from the same population*. The test rejects this H_0 at significance level α if $D_{m,n} > D_{m,n,\alpha}$, where $D_{m,n,\alpha} = c(\alpha)\sqrt{\frac{m+n}{mn}}$ for sufficiently large m and n, and where $c(\alpha)$ is the reverse of Kolmogorov distribution at confidence level α. Intuitively, $D_{m,n,\alpha}$ is the magnitude of the difference that can be expected to exist between $R(x)$ and $M(x)$ even when the two data sets *are* drawn from the same population, and α is the fraction of test in which the tests is allowed to falsely reject H_0. Note that α cannot be zero because for any value of $D_{m,n}$ there exists a small possibility that it can occur even when H_0 is true.

A final consideration we need in order to use the K-S test in EDDIE is that the K-S test is a one-dimensional test, i.e., the data sets it can compare should have elements that are scalars (numbers), while in EDDIE a STSs are characterized by a set of peak frequencies (a vector of numbers). Thus we apply the K-S test to each dimension separately, i.e., one test compares the STSs according to their strongest peak, a second K-S test compares the STSs according to their second-strongest peak, etc. The results of these tests are then combined by counting the number of such tests that have rejected H_0. This is described in Section 4.4, together with other considerations EDDIE makes before it actually reports an anomaly to the user.

4.3 Managing the Trade-off Between Detection Accuracy and Latency

In the K-S test used in EDDIE (see Section 4.2), we can choose n, the number of monitoring-observed STSs that will be tested in each K-S test. With a small n, the K-S test uses only very recently observed STSs so EDDIE will have a low latency between when the anomalous execution begins and when it is detected. This *detection latency* can generally be expected to grow in proportion to the value of n. However, n also affects detection accuracy, and the relationship between accuracy and the value of n is not as straightforward. To illustrate this, Figure 3 shows the false rejection rate in EDDIE's K-S test, i.e., how often the K-S test fails in an injection-free run, as we increase n (which is shown in terms of detection latency).

When n is too small, for most regions the K-S test will rarely reject H_0 even when H_0 is actually false. This is not surprising because, intuitively, the test does not have enough monitoring-observed data points to reach a sufficiently high confidence level. As n increases, the rejection rate usually increases for both correct rejection (the STSs come from the injected part of the execution, not shown in Figure 3) and false rejection. At some point, the correct rejection rate becomes very high (most STS sets that include injections are rejected by the K-S test) but the false rejection rate is also relatively high. Further increase of n then results in virtually no change in correct rejections, but the false rejection rate drops and eventually becomes very close to zero.

We want EDDIE to be useful in practice, so its reports should have very few false positives. Although EDDIE's overall algorithm has additional considerations beyond the K-S test rejecting a group of STSs, it would be very hard to achieve a very low false positive rate if the K-S test labels a significant percentage of monitoring-time STS groups as anomalous.

Thus in EDDIE we should use the K-S test with the number of monitoring-observed STSs n that is large enough to provide a near-zero false rejection rate, but not much larger than that because that unnecessarily sacrifices detection latency. Unfortunately, as shown in Figure 3, this "sweet spot" value of n depends on the code region and differs quite a bit from region to region.

To accommodate this, during training EDDIE determines the desired value of n separately for each region. We select the desired n by applying the K-S test to training-time STSs for that region using different values of n, and selecting the smallest n that provides the minimum false rejection rate observed across the entire range of ns [2]. Note that all training runs are injection-free, so this method can only consider the false rejection rate, but we have found that this also results in near-optimum correct rejections.

4.4 Monitoring Algorithm

Algorithm 1 shows (with some simplifications) how EDDIE works and switches between regions and decides when to report an anomaly to the user.

The loop at Line 7 iterates over STSs observed during monitoring. The new STS is added to the set that will be used in the K-S test, and the oldest STS in the set is removed to maintain the set at the appropriate size for the current region. The K-S test is then used, one peak at a time, to compare the monitoring-observed set (*MonSet*, see Line 9) to the reference set for the current region. When the K-S test rejects the null hypothesis, EDDIE considers the possibility that

[2]For most regions the false rejection does reach zero at some value of n

the execution has progressed beyond the current region. This is also done using the K-S test, by comparing the appropriate number of monitoring-observed STSs to the reference STSs for each candidate region. Acceptances and rejections across all peaks are counted. After all candidates have been considered, if a next-region candidate had enough peaks accepted by the test, EDDIE's current region is changed to that candidate region. If multiple candidates have enough peaks accepted (this happens extremely rarely), EDDIE uses the candidate region with the most accepted peaks. If none of the next-region candidates is acceptable, EDDIE checks if the number of recent rejections in the K-S test is high enough (i.e. 3 in our implementation) to report an anomaly to the user. This allows EDDIE to tolerate a few K-S test rejections without reporting anything, which helps reduce the number of false reported anomalies that can occur in injection-free runs when, for example, interrupts and other system activity occasionally result in a "deviant" STS.

Algorithm 1 Malware detection algorithm

```
Regions: R_1...R_r
Current region number: c
Group size for K-S test: n_1...n_r
P(i, j) : jth peak in the the ith STS
pos ← 1; counter ← 0;
currRegion = R_1
while application is running do
    for p do=1..numPeaks(R_c)
        MonSet ← P(pos − n_c : pos, p)
        if test(RefSet_{c,k}, MonSet) = reject then
            for R_j ∈ successors of R_c do
                AltSet ← P(pos − n_j : pos, p)
                if test(RefSet_{j,k}, AltSet) = reject then
                    anomalyCnt ← anomalyCnt + 1
                else
                    changeCnt(j) ← changeCnt(j) + 1
                end if
            end for
        elseReset anomalyCnt and changeCnt
        end if
    end for
    if changeCnt > changeThreshold then
        j ← index of max(changeCnt)
        currRegion ← j
    end if
    if anomalycnt > reportThreshold then
        Report anomaly to user
    end if
    pos ← pos + 1
end while
```

5 RESULTS

In this section we first present our experimental setup (Section 5.1), then we present results of applying EDDIE to the EM emanations from a real IoT system (Section 5.2). We continue with a presentation of results of applying EDDIE to the power signal produced through architectural simulation (Section 5.3) and an investigation of which architectural features affect EDDIE's detection performance. Next, Section 5.4 provides an in-depth analysis of how EDDIE's detection performance changes as we change the percentage of loop iterations that are contaminated by code injection, and Section 5.5 investigates how EDDIE's detection performance changes when we change the dynamic footprint of the injection, i.e. the number of injected instructions that execute in each loop iteration, or in a burst outside loops. EDDIE's sensitivity to the confidence level used in the statistical test is discussed in Section 5.6 and, finally, Section 5.7 investigates the effect of changing the type of injected instructions.

5.1 Experimental Setup

We use two different experimental setups. One is a real IoT prototype system, a single-board Linux computer (A13-OLinuXino-MICRO [8]) with a 2-issue in-order ARM Cortex A8 processor with a 32kB L1 and a 256kB L2 cache, with a Debian Linux operating system. The EM signals emanated from this system are received by a commercial small electric antenna (PBS-E1 [1]) that is placed right above the device's processor, and the signal is recorded using a Keysight DSOS804A oscilloscope [43]. While this oscilloscope is relatively expensive (several tens of thousands US dollars), note that we use it mainly because of its built-in features for automated and calibrated measurements and ability for displaying the real-time signals. In additional experiments, we have observed similar EM spectra with less expensive (<$800) commercial software-defined radio receiver (USRP b200-mini) and we confirm that EDDIE can work efficiently on such lower-cost setups. While this cost is low enough for deploying EDDIE in some important scenarios (critical infrastructure, medical offices, etc.), for other scenarios we envision a custom design with a specialized receiver (ASIC block for STFT and peak finding, simple CPU for tests, and some flash for storing the model from training) attached to an antenna, with a <$100 total cost.

Our second setup is based on the SESC [69] cycle-accurate simulator, and uses the simulator-generated power signal for EDDIE's analysis. This setup is used to confirm that EDDIE is applicable across a wide range of systems, and to gain insight into which architectural features affect EDDIE's detection performance.

In our experiments, we use a total of 10 benchmarks from the MiBench [36] suite to test EDDIE algorithm. For the real IoT system, we execute each benchmark 25 times during training. The code for the training runs contains with our light-weight instrumentation, which is implemented as a Clang tool, and the code is also subjected to a separate analysis (which is not used to actually generate code) in LLVM [71] where we added a pass that statically finds the regions and the possible transitions between regions. For monitoring in this setup we use 25 runs per benchmark, without any instrumentation and with different inputs.

In simulation-based experiments we use fewer runs (10 training and 10 monitoring runs per benchmark) to reduce the overall simulation time.

5.2 EDDIE Results for Measured EM Emanations of a Real IoT Device

In this set of experiments, we inject code into different regions of each application. The injections are different for loop and inter-loop

Table 1: Accuracy for EDDIE monitoring of an actual IoT device

Benchmark	Detection Latency (ms)	False positives (%)	Accuracy (%)	Coverage (%)
Bitcount	42	0.99	100	99.9
Basicmath	25	1.8	99.9	99.9
Susan	32	1.39	92.1	95.9
Dijkstra	25	1.08	99.9	99.7
Patricia	28	0.98	92.3	95.2
GSM	24	0.9	96.2	57.1
FFT	17	0.76	93	99
Sha	11	1.9	97.2	98.9
Rijndael	12	0.56	99.9	97.1
Stringsearch	11	0.19	99.9	99.9

regions. Injections outside loops consists of invoking a shell and then, without doing anything else, returning back to the original application. This injection results in executing 476k injected instructions and adding about 3 ms to the execution time. When injection is made in a loop, we add an 8-instruction code that consists of 4 integer operations and 4 memory accesses. The rationale for the shellcode injection is that shellcode execution is often a fundamental step in many attacks, and our empty-shellcode injection results in less injected-code execution than any real shellcode-based attack where the attack's intended activity (payload) must either be executed or at least set up within the shellcode-invoked shell. The rationale for injecting only 8 instructions into a loop body is that an injection into a loop allows the injected code to be executed repeatedly, allowing the attacker to perform significant work over time but improve stealth by performing the work in small chunks.

The results for the IoT system are shown in Table 1. The first column shows the application, and the remaining columns report EDDIE's detection latency, false positive and accuracy percentages, and coverage. The results were obtained using $reportThreshold = 3$ in EDDIE's algorithm (see Section 4.4), i.e. EDDIE tolerates up to 3 consecutive K-S test rejections and only reports an anomaly for a rejection that is part of a 4-long (or longer) streak of test rejections. The average detection latency is measured as the average, among all injections that are reported, of the difference between when execution of injected code begins and the time when EDDIE reports it. This latency mainly reflects the number of STSs that are used in the K-S test (n in Section 4.2). False positives are the number of STS groups that are reported as anomalous but do not contain any injected execution, as a percentage of all STS groups. The average for false positives is $<1\%$ and the highest false positive percentage was only 1.9% (for the *Sha* benchmark). Accuracy is computed for each region as the total number of STS groups with a correct reporting outcome, i.e. those that contain injections and are reported by EDDIE plus those that contain no injections and are not reported, expressed as a percentage of all STS groups. The accuracy shown for each benchmark is the average of its per-region accuracy results. On average, EDDIE's accuracy is 95%. We observed that the bulk of the inaccuracies come from borders between two regions (i.e outside the loops), and further investigation has revealed two main causes for this: (i) non-loop code during some transitions creates poorly defined peaks, so better consideration of diffuse spectral features may improve EDDIE's accuracy, and (ii) the actual inter-loop transition is usually very brief and for different executions occurs at a different point in the window on which the STS was computed, so better identification of the boundaries of the actual inter-loop transition may help to create STSs that better represent the transition. Finally, we define coverage as the amount of time during which the STS is attributed to the region in the code that actually produced it. The main reason for imperfect coverage in our implementation is that some loops have no peaks in their STSs. For example, about 40% of the execution time in *GSM* is spent in one such loop, and this accounts for nearly all of its poor coverage.

5.3 Simulation Results and Sensitivity to Processor Architecture

To gain more confidence that EDDIE is a broadly applicable approach, and to get more insight into which aspects of the system's architecture have an effect on EDDIE's accuracy, we apply EDDIE to the power consumption signal generated by the SESC simulator with integrated CACTI [68] and WATTCH [12] power models for its cache and configurable pipeline. We first model a 1.8 GHz 4-issue out-of-order core with 32KB L1 and 64MB L2 caches, the power signal provided to EDDIE is sampled every 20 cycles, and EDDIE's STFT uses 0.1ms windows with 50% overlap. The code injection in these simulations is implemented by directly injecting dynamic instructions into the simulated instruction stream without changing the application's code or using any architectural registers. This maximizes the injection's stealth and is an idealized representative of an attack that uses only registers that are dead at the injection point in the original application.

EDDIE's results for these simulation-based experiments are shown in Table 2. False rejections occur on average in 0.7% STSs, an expected improvement over real-system experiments because the simulation has no signal noise, no interrupts or other system activity, etc. By comparing results from simulation and real-system experiments, we can also conclude that EDDIE's accuracy and detection latency are more affected by the applications itself (i.e. mostly the shape of the spectrum for the code regions) than by factors such as noise, interference, etc.

Intuitively, we expect EDDIE to perform better on systems whose architecture introduces less variation in executions of the same region of code. To get more insight into which architectural parameters have a significant impact on EDDIE's detection performance, we

Table 2: EDDIE's latency and accuracy when using a simulator-generated power signal

Benchmark	Average Latency	False Rejection	Accuracy	Coverage
Bitcount	7ms	0.8%	99.9%	99.9%
Basicmath	8ms	0.2%	99.9%	100%
Susan	5ms	0.7%	91.4%	96.6%
Dijkstra	10ms	0.3%	97.02%	99.9%
Patricia	13ms	0.4%	94.14%	98%
GSM	6ms	0%	100%	68.3%
FFT	5ms	0.4%	97.8%	99.1%
Sha	0.4ms	1.83%	100%	100%
Rijndael	0.6ms	0.24%	97.1%	97.2%
Stringsearch	0.2ms	0%	100%	100%

Figure 4: Detection latency of 15 different regions in in-order and out-of-order architecture.

configure the simulator to model an in-order processor with 3 different issue widths (1,2, and 4) and 2 different pipeline depths, and an out-of-order processor with 3 issue widths (1,2,4), 3 pipeline depths, and 5 ROB sizes, for a total of 51 configurations. We then simulate execution of 3 benchmarks (*Basicmath*, *Bitcounts*, and *Susan*) on each configuration and use N-way analysis of variance (ANOVA) to determine which factors have a significant impact on EDDIE's results.

We found that for out-of-order and in-order architectures EDDIE achieves similar false rejection and accuracy results, but its latency (i.e. number of STSs that need to be considered in the K-S test) is significantly higher for out-of-order architectures (see Figure 4) because an out-of-order core tends to produce more variation in its dynamically constructed instruction schedule, creating more variation among STSs and thus requiring more STSs to capture their distribution.

We also found that in in-order architectures pipeline depth and issue width have no statistically significant effect on EDDIE's results, and that in out-of-order architectures the ROB size and issue width also have no statistically significant impact on EDDIE's results. However, in out-of-order processors pipeline depth has a weak but statistically significant impact on detection latency. A closer look at the data reveals that in 27% of the code regions pipeline length increases detection delay, and that these affected regions are all loops

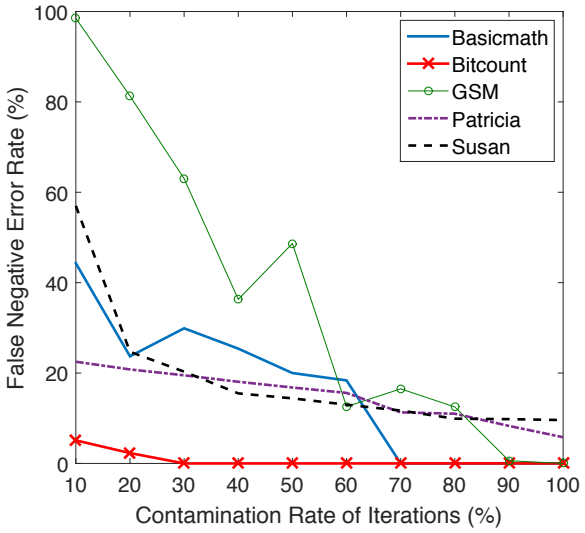

Figure 5: False negative rate of variable injection rates.

with control flow variation among iterations, so the likely explanation for increased detection delay in EDDIE is that a deeper pipeline results in more timing variation due to branch mispredictions, that in turn increases the size of the STS group that representatively captures this variation (and the n for the K-S test).

Finally, we repeated this analysis for different amounts of injected execution, and found that the impact of pipeline depth in out-of-order processors on EDDIE's results diminishes as the injection size increases, and for large-enough injections the pipeline depth no longer has a statistically significant impact of EDDIE's detection latency. This means that large amounts of injected activity can be detected quickly even when the processor's pipeline is deep, but for smaller injections longer pipelines result in longer detection latency.

5.4 Effect of the Execution Rate of Injected Code

An intuitive way to improve stealth is to further diffuse injected execution by injecting the code inside a loop body such that only some loop iterations execute (a small amount of) the injected code. To evaluate EDDIE in this context, we use our simulator-based setup and for the targeted loop region randomly choose the iterations that will be injected with 8 memory instructions and 8 integer operations. We use *contamination rate* to refer to the percentage of iterations that contain injected execution, and we repeat this set of experiments for contamination rates between 100% (where every iteration is injected) and 10% (where 90% of the iterations are injection-free).

Figure 5 shows the *false negatives*, i.e. the percentage of injection-containing STSs that are not reported by EDDIE, for different contamination rates. As expected, EDDIE's ability to detect the injection does diminish with the injection's contamination rate, but for most applications EDDIE still retains significant ability to detect injections even at low contamination rates. For example, for *Bitcount* EDDIE still detects >90% of injection-containing STSs even when only 10% of loop iterations actually contain injected execution. However, in *GSM* EDDIE detects only 5% of the STS at the 10% contamination rate. Note that this does not mean that EDDIE is inherently unable to

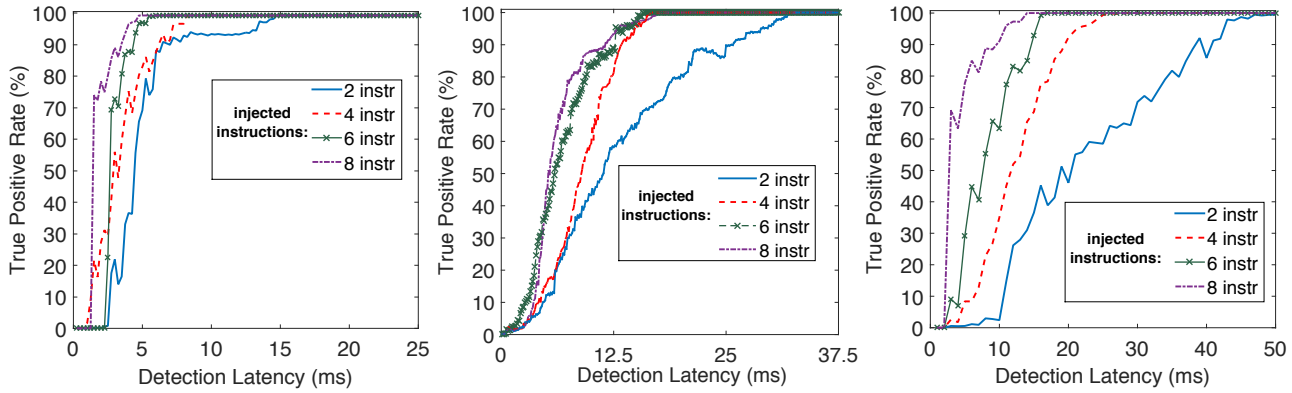

Figure 6: EDDIE's accuracy when changing the number of injected instructions inside loops.

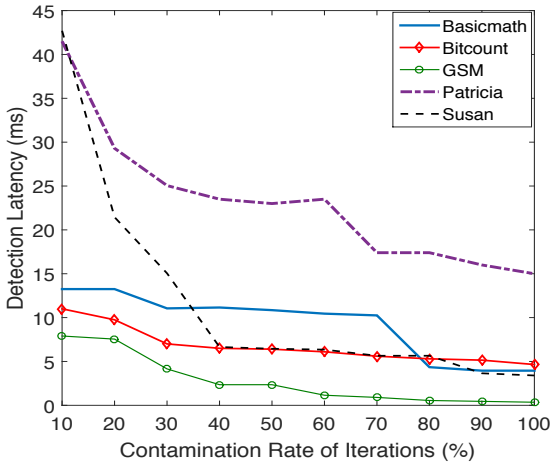

Figure 7: Detection latency of variable injection rates.

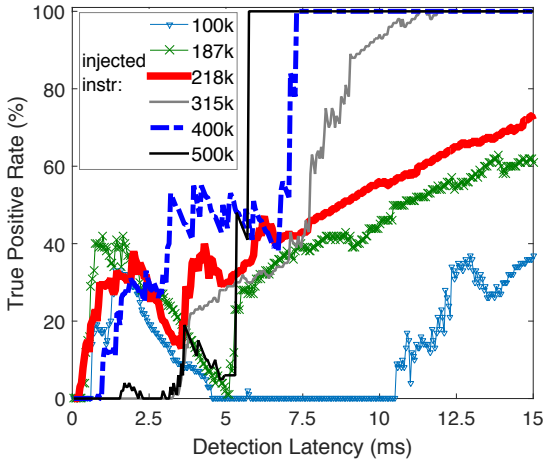

Figure 8: EDDIE's accuracy when changing the number of injected instructions outside loops.

detect injections that have low contamination rates. Indeed, Figure 7 shows the results in terms of detection latency (which is increased by increasing n in EDDIE's K-S test) that is needed to maintain EDDIE's accuracy. This indicates that EDDIE can very accurately detect even low-contamination-rate injections, but that detection of low-contamination-rate injections will have a longer latency.

5.5 Size of Injection

In this section, we analyze the impact of the number of injected instructions on EDDIE's detection accuracy.

Our analysis considers injection *inside* and *outside* of loops separately. This is because even a few instructions injected inside a loop can accomplish significant work for the attacker as the injected code is executed many times during the loop. Outside loops, however, a successful attack usually requires injection of many instructions (recall that even an empty-payload shellcode executes over 500,000 instructions).

Figure 6 shows how EDDIE's accuracy changes with the number of static instructions injected inside a loop. The smallest injection in this experiment consists of only two instructions: a store and an add. The remaining three injection sizes consists of 4, 6, and 8 instructions with the same instruction mix (equal number of stores and adds). To show how these results are affected by the loop's spectrum, Figure 6 shows the results for the same three loops used in Figure 3, i.e. a loop whose spectrum contains only one sharp peak and its harmonics, a loop with several less well defined peaks (and their harmonics), and a loop with a very diffuse peak that can be more accurately described as a hump. The figure shows that even two-instruction injections into the loop body can be detected by EDDIE with extremely high accuracy, but that smaller injections have longer detection latency (i.e. use of a larger n in EDDIE's K-S test).

Figure 8 shows the results for *outside* the loops. We use several different size of injections. In order to inject the code, we use an empty loop and put it between loop 2 and 3 in *Bitcount* application and change the number of iterations of this empty loop. As this number increased, number of injected instructions increased too.

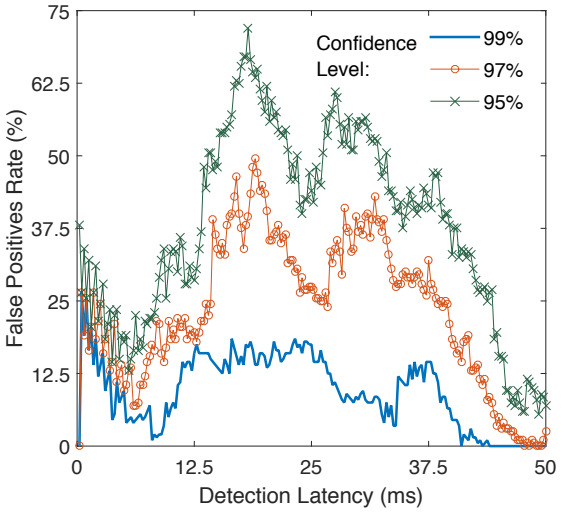

Figure 9: False positives in EDDIE for different K-S test confidence levels.

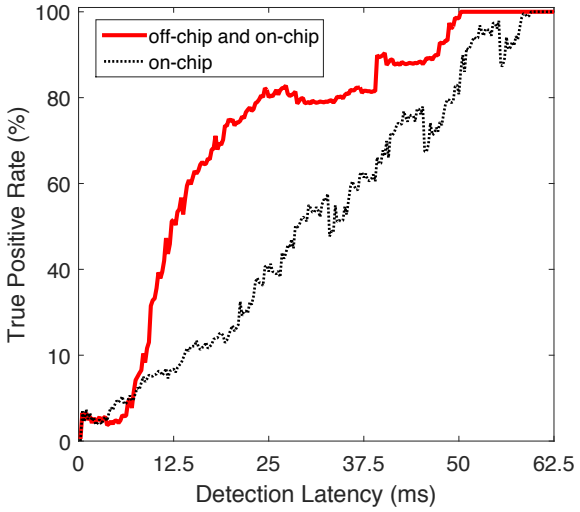

Figure 10: Effect of changing the type of injected instructions on latency and accuracy.

5.6 Effect of Changing the Confidence Level in the K-S Test

As mentioned earlier, we use KS-test to identify when the monitoring and the reference STSs differ too much. One important parameter in statistical tests is the confidence level, which introduces a fundamental trade-off between the test's false rejections (detecting an anomaly that does not really exist) and false acceptances (not detecting a real anomaly). Figure 9 shows three confidence levels in terms of how the false positive rate changes with detection latency. As we can see, the 99% confidence level (which we use in all our experiments except this one) results in fewer false positives and it practically eliminates the false positives with a reasonable latency. Lower confidence levels result in many false positives at low latencies, and even at high latencies may not reduce false positives to acceptable levels.

5.7 Effect of Changing Instruction

In order to show how different types of injected instructions can affect detection latency and accuracy, we perform experiments in which we inject two different instructions in a loop. In the first set, we inject eight add instructions, while in the second set we inject 4 add instructions and 4 store instructions that randomly access a relatively large array so they often experience a cache misses. Figure 10 shows the results for these two sets of experiments. These results indicate that instructions that result in off-chip activity tend to make the injection more visible and thus easier to detect quickly, but that even purely on-chip injections can be detected, albeit with an increased detection latency.

In additional experiments we used other on-chip instructions like MUL, DIV, etc. but the results were very similar to those of the ADD instruction.

6 RELATED WORK

Generally, detection methods can be divided into two major groups: *Signature-based* and *Anomaly-based* detections. Signature-based techniques (e.g. [26, 35, 40, 57]) can detect an attack if the signatures of the attack have been observed earlier. On the other hand, anomaly-based detections report any deviations from the reference model as anomalous, i.e., as a possible attack. Since our approach is anomaly-based, here we focus on related work in that domain.

Anomaly-based methods use different types of features for modeling a normal behavior of the system and then continuously monitor a system collect and examine features of interest. These methods can be categorized into *software* and *hardware* monitoring systems. Software techniques (e.g. [6, 14, 63]) usually suffer from high performance overhead, high false positive rate, and poor accuracy for more complicated malware.

Hardware and hardware-assisted techniques (e.g. [21, 24, 42, 48, 59, 60, 79]) have better accuracy and lower performance overhead, but have higher hardware complexity and power consumption. Examples of hardware-assisted techniques include function call monitoring via hardware [67], constraining control-flow [23, 86], hardware techniques for information flow tracking and tainting suspect data [75, 80], monitoring via hardware visualization [25, 62], branch history monitoring [82], hardware-based indirect branch tracking [61], hardware-assisted monitoring [9], binary trace validation [66], and hardware-based processing patterns validation [53].

Our proposed approach differs from hardware and software monitoring techniques because it has no hardware overhead and no extra power consumption and it also does not need any monitoring infrastructure (e.g. performance counters) on the host device.

Also related to our work is the large body of work on side channels. There are many papers that report analog side-channel attacks using EM emanations [3, 33], acoustics emanations [31, 37], power and timing variations [10, 11, 15, 19, 32, 34, 49, 50, 56, 70]. In addition to analog signals, many efforts have been made for extracting information using cache-based [7, 52, 83] side-channels and corresponding countermeasures [47, 81, 87]. This type of work is focused

on the ways for attacker to extract sensitive data values (such as cryptographic keys) from the system and on countermeasures against such attacks.

Very recently, program profiling through EM signals has been proposed [13, 72]. In particular, *ZOP* [13] attributes EM emanations to specific parts of the program by correlating EM signal samples at the granularity of cyclic paths. While the accuracy of matching in ZOP was above 95%, its computational complexity may prohibit practical use on real-world programs. On the other hand, *Spectral Profiling* [72] profiles the code by monitoring EM emanations and looking for spectral features produced by periodic program activity such as loops. While [72] has shown that the EM spectra can be used to deduce which loop in the program is currently active, this paper exploits EM spectra to characterize normal program behavior and then monitors program execution to identify deviations from normal behavior. This means that, in contrast to Spectral Profiling, EDDIE has to consider not only loop activity but also non-loop activity and loop-to-loop transitions, because malicious code injection can target these parts of the application. Furthermore, EDDIE has to not only match the spectral sample to some valid part of the application, but also assess the likelihood that the sample does *not* come from any valid part of the application.

Apart from EM signals, other analog signals such as power consumption have been used for anomaly detection. Examples are sensor node anomaly detection [44], electrical activity anomaly detection [39], malware detection on embedded medical devices [17, 18], building malware signature [46], etc. Compared to EDDIE, these techniques have higher false positive rates and increased detection latency. For example, *WattsUpDoc* [18] detects malware by monitoring system-wide power consumption, achieves only 80% accuracy with 1.7% false positive rate and with a latency that is measured in seconds.

7 CONCLUSIONS

This paper describes EM-Based Detection of Deviations in Program Execution (EDDIE), a new method for detecting anomalies in program execution, such as malware and other code injection, without introducing any overheads, adding any hardware support, changing any software, or using any resources on the monitored system itself. Monitoring with EDDIE involves receiving electromagnetic (EM) emanations that are emitted as a side effect of execution on the monitored system, and it relies on peaks in the EM spectrum that are produced as a result of periodic (e.g. loop) activity in the monitored execution. During training, EDDIE characterizes normal execution behavior in terms of peaks in the EM spectrum that are observed at various points in the program execution, but it does not need any characterization of the virus or other code that might later be injected. During monitoring, EDDIE identifies peaks in the observed EM spectrum, and compares these peaks to those learned during training. Since EDDIE requires no resources on the monitored machine and no changes to the monitored software, it is especially well suited for security monitoring of embedded and IoT devices. We evaluate EDDIE on a real IoT system and in a cycle-accurate simulator, and find that even relatively brief injected bursts of activity (a few milliseconds) are detected by EDDIE with high accuracy, and that it also accurately detects when even a few instructions are injected into an existing loop within the application.

ACKNOWLEDGMENT

This work has been supported, in part, by NSF grants 1563991 and 1318934, AFOSR grant FA9550-14-1-0223, and DARPA LADS contract FA8650-16-C-7620. The views and findings in this paper are those of the authors and do not necessarily reflect the views of NSF, AFOSR, and DARPA.

REFERENCES

[1] AARONIA. accessed April 6, 2016. Datasheet: RF Near Field Probe Set DC to 9GHz. http://www.aaronia.com/Datasheets/Antennas/RF-Near-Field-Probe-Set.pdf. (accessed April 6, 2016).

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 4.

[3] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. 2002. The EM side-channel(s). In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2002*. 29–45.

[4] CarlosR. Aguayo Gonzalez and JeffreyH. Reed. 2011. Power fingerprinting in SDR integrity assessment for security and regulatory compliance. *Analog Integrated Circuits and Signal Processing* 69, 2-3 (2011), 307–327. https://doi.org/10.1007/s10470-011-9777-4

[5] Mamoun Alazab, Sitalakshmi Venkatraman, Paul Watters, and Moutaz Alazab. 2011. Zero-day malware detection based on supervised learning algorithms of API call signatures. In *Proceedings of the Ninth Australasian Data Mining Conference-Volume 121*. Australian Computer Society, Inc., 171–182.

[6] Kevin Allix, Tegawendé F. Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. 2016. Empirical assessment of machine learning-based malware detectors for Android. *Empirical Software Engineering* 21, 1 (2016), 183–211.

[7] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 591–604.

[8] ARM. accessed April 3, 2016. ARM Cortex A8 Processor Manual. https://www.arm.com/products/processors/cortex-a/cortex-a8.php. (accessed April 3, 2016).

[9] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K Jha. 2005. Secure embedded processing through hardware-assisted run-time monitoring. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*. IEEE Computer Society, 178–183.

[10] A G Bayrak, F Regazzoni, P Brisk, F.-X. Standaert, and P Ienne. 2011. A first step towards automatic application of power analysis countermeasures. In *Proceedings of the 48th Design Automation Conference (DAC)*.

[11] Dan Boneh and David Brumley. 2003. Remote Timing Attacks are Practical. In *Proceedings of the USENIX Security Symposium*.

[12] David Brooks, Vivek Tiwari, and Margaret Martonosi. 2000. Wattch: A Framework for Architectural-level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. ACM, New York, NY, USA, 83–94. https://doi.org/10.1145/339647.339657

[13] Robert Callan, Farnaz Behrang, Alenka Zajic, Milos Prvulovic, and Alessandro Orso. 2016. Zero-overhead Profiling via EM Emanations. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA '16)*.

[14] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. 2012. A Quantitative Study of Accuracy in System Call-based Malware Detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. 122–132.

[15] S Chari, C S Jutla, J R Rao, and P Rohatgi. 1999. Towards sound countermeasures to counteract power-analysis attacks. In *Proceedings of CRYPTO'99, Springer, Lecture Notes in computer science*. 398–412.

[16] ShaneS. Clark, Hossen Mustafa, Benjamin Ransford, Jacob Sorber, Kevin Fu, and Wenyuan Xu. 2013. Current Events: Identifying Webpages by Tapping the Electrical Outlet. In *Computer Security- ESORICS 2013*. Lecture Notes in Computer Science, Vol. 8134. 700–717.

[17] Shane S. Clark, Benjamin Ransford, and Kevin Fu. 2012. Potentia Est Scientia: Security and Privacy Implications of Energy-Proportional Computing. In *7th USENIX Workshop on Hot Topics in Security, HotSec'12, Bellevue, WA, USA, August 7, 2012*.

[18] Shane S. Clark, Benjamin Ransford, Amir Rahmati, Shane Guineau, Jacob Sorber, Wenyuan Xu, and Kevin Fu. 2013. WattsUpDoc: Power Side Channels to Nonintrusively Discover Untargeted Malware on Embedded Medical Devices. In

Presented as part of the 2013 USENIX Workshop on Health Information Technologies.

[19] B Coppens, I Verbauwhede, K De Bosschere, and B De Sutter. 2009. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*. 45–60.

[20] G.W. Corder and D.I. Foreman. 2011. *Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach*. Wiley. https://books.google.com/books?id=T3qOqdpSz6YC

[21] Sanjeev Das, Yang Liu, Wei Zhang, and Mahinthan Chandramohan. 2016. Semantics-Based Online Malware Detection: Towards Efficient Real-Time Protection Against Malware. *IEEE Trans. Information Forensics and Security* 11, 2 (2016), 289–302.

[22] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. 2015. HAFIX: hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 74.

[23] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. 2014. Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation. In *Proceedings of the 51st Annual Design Automation Conference (DAC '14)*. 133:1–133:6.

[24] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore J. Stolfo. 2013. On the feasibility of online malware detection with performance counters. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*. 559–570.

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

[26] Ken Dunham. 2003. Evaluating Anti-Virus Software: Which Is Best? *Information Systems Security* 12, 3 (2003), 17–28.

[27] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2012. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)* 44, 2 (2012), 6.

[28] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 5.

[29] Aurélien Francillon and Claude Castelluccia. 2008. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 15–26.

[30] K Gandolfi, C Mourtel, and F Olivier. 2001. Electromagnetic analysis: concrete results. In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2001*. 251–261.

[31] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. 2015. Stealing Keys from PCs Using a Radio: Cheap Electromagnetic Attacks on Windowed Exponentiation. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*. 207–228.

[32] Daniel Genkin, Itamar Pipman, and Eran Tromer. 2014. Get Your Hands Off My Laptop: Physical Side-Channel Key-Extraction Attacks on PCs. In *Cryptographic Hardware and Embedded Systems Ű CHES 2014*, Lejla Batina and Matthew Robshaw (Eds.). Lecture Notes in Computer Science, Vol. 8731. Springer Berlin Heidelberg, 242–260. https://doi.org/10.1007/978-3-662-44709-3_14

[33] Daniel Genkin, Adi Shamir, and Eran Tromer. 2014. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*. 444–461.

[34] L Goubin and J Patarin. 1999. DES and Differential power analysis (the "duplication" method). In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 1999*. 158–172.

[35] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-cker Chiueh. 2009. Automatic Generation of String Signatures for Malware Detection. In *Recent Advances in Intrusion Detection, 12th International Symposium, RAID 2009, Saint-Malo, France, September 23-25, 2009. Proceedings*. 101–120.

[36] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 3–14.

[37] Y. Hayashi, N. Homma, T. Mizuki, H. Shimada, T. Aoki, H. Sone, L. Sauvage, , and J. L. Danger. 2013. Efficient evaluation of EM radiation associated with information leakage from cryptographic devices. *IEEE Transactions on Electromagnetic Compatibility* 55, 6 (2013), 555–563.

[38] H J Highland. 1986. Electromagnetic radiation revisited. *Computers and Security* (Dec. 1986), 85–93.

[39] T. D. Huang, Wen-Sheng Wang, and Kuo-Lung Lian. 2015. A New Power Signature for Nonintrusive Appliance Load Monitoring. *IEEE Trans. Smart Grid* 6, 4 (2015), 1994–1995.

[40] Kelly Hughes and Yanzhen Qu. 2014. Performance Measures of Behavior-Based Signatures: An Anti-malware Solution for Platforms with Limited Computing Resource. In *Ninth International Conference on Availability, Reliability and Security, ARES 2014, Fribourg, Switzerland, September 8-12, 2014*. 303–309.

[41] Yu ichi Hayashi, Naofumi Homma, Takaaki Mizuki, Haruki Shimada, Takafumi Aoki, Hideaki Sone, Laurent Sauvage, and Jean-Luc Danger. 2013. Efficient Evaluation of EM Radiation Associated With Information Leakage From Cryptographic Devices. *IEEE Transactions on Electromagnetic Compatibility* 55, 3 (2013), 555–563.

[42] Mikhail Kazdagli, Vijay Janapa Reddi, and Mohit Tiwari. 2016. Quantifying and Improving the Efficiency of Hardware-based Mobile Malware Detectorsis. In *Proceedings of the 49th International Symposium on Microarchitecture (MICRO-49)*.

[43] Keysight-Technologies. DSOS804A High-Definition Oscilloscope: 8 GHz, 4 Analog Channels. http://www.keysight.com/en/pdx-x202073-pn-DSOS804A/high-definition-oscilloscope-8-ghz-4-analog-channels?cc=US&lc=eng. (????).

[44] Mohammad Maifi Hasan Khan, Hieu Khac Le, Michael LeMay, Paria Moinzadeh, Lili Wang, Yong Yang, Dong Kun Noh, Tarek F. Abdelzaher, Carl A. Gunter, Jiawei Han, and Xin Jin. 2010. Diagnostic powertracing for sensor node failure analysis. In *Proceedings of the 9th International Conference on Information Processing in Sensor Networks, IPSN 2010, April 12-16, 2010, Stockholm, Sweden*. 117–128.

[45] M G Khun. 2003. Compromising emanations: eavesdropping risks of computer displays. *The complete unofficial TEMPEST web page: http://www.eskimo.com/˜joelm/tempest.html* (2003).

[46] Hahnsang Kim, Joshua Smith, and Kang G. Shin. 2008. Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services (MobiSys 2008), Breckenridge, CO, USA, June 17-20, 2008*. 239–252.

[47] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. 189–204.

[48] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. 361–372.

[49] P Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of CRYPTO'96, Springer, Lecture notes in computer science*. 104–113.

[50] P Kocher, J Jaffe, and B Jun. 1999. Differential power analysis: leaking secrets. In *Proceedings of CRYPTO'99, Springer, Lecture notes in computer science*. 388–397.

[51] M. G. Kuhn. 2013. Compromising emanations of LCD TV sets. *IEEE Transactions on Electromagnetic Compatibility* (2013), 564–570.

[52] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 605–622.

[53] Shufu Mao and Tilman Wolf. 2007. Hardware support for secure processing in embedded systems. In *Proceedings of the 44th annual Design Automation Conference*. ACM, 483–488.

[54] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951), 68–78.

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

[56] T S Messerges, E A Dabbish, and R H Sloan. 1999. Power analysis attacks of modular exponentiation in smart cards. In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 1999*. 144–157.

[57] Aziz Mohaisen and Omar Alrawi. 2014. AV-Meter: An Evaluation of Antivirus Scans and Labels. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 11th International Conference, DIMVA 2014, Egham, UK, July 10-11, 2014. Proceedings*. 112–131.

[58] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*.

[59] Meltem Ozsoy, Caleb Donovick, Iakov Gorelik, Nael B. Abu-Ghazaleh, and Dmitry V. Ponomarev. 2015. Malware-aware processors: A framework for efficient online malware detection. In *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*. 651–661.

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

[61] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. 2013. Transparent ROP exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. 447–462.

[62] Bryan D Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. 2008. Lares: An architecture for secure active monitoring using virtualization. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 233–247.

[63] Naser Peiravian and Xingquan Zhu. 2013. Machine Learning for Android Malware Detection Using Permission and API Calls. In *Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI '13)*. 300–305.

[64] T Plos, M Hutter, and C Herbst. 2008. Enhancing side-channel analysis with low-cost shielding techniques. In *Proceedings of Austrochip*.

[65] Francois Poucheret, Lyonel Barthe, Pascal Benoit, Lionel Torres, Philippe Maurine, and Michel Robert. 2010. Spatial EM jamming: A countermeasure against EM Analysis?. In *Proceedings of the 18th IEEE/IFIP VLSI System on Chip Conference (VLSI-SoC)*. 105–110.

[66] Roshan G Ragel and Sri Parameswaran. 2006. IMPRES: integrated monitoring for processor reliability and security. In *Proceedings of the 43rd annual Design Automation Conference*. ACM, 502–505.

[67] Mehryar Rahmatian, Hessam Kooti, Ian G Harris, and Elaheh Bozorgzadeh. 2012. Hardware-assisted detection of malicious software in embedded systems. *IEEE Embedded Systems Letters* 4, 4 (2012), 94–97.

[68] Glen Reinman and Norman P Jouppi. 2000. CACTI 2.0: An integrated cache timing and power model. *Western Research Lab Research Report* 7 (2000).

[69] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. 2005. SESC simulator. (January 2005). http://sesc.sourceforge.net.

[70] W Schindler. 2000. A timing attack against RSA with Chinese remainder theorem. In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2000*. 109–124.

[71] Colin Schmidt. 2014. Low Level Virtual Machine (LLVM). https://github.com/llvm-mirror/llvm. (2014).

[72] Nader Sehatbakhsh, Alireza Nazari, Alenka G. Zajic, and Milos Prvulovic. Spectral Profiling: Observer-Effect-Free Profiling by Monitoring EM Emanations. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016*.

[73] Hidenori Sekiguchi and Shinji Seto. 2009. Measurement of radiated computer RGB signals. *Progress in Electromagnetic Research C* (2009), 1–12.

[74] Hidenori Sekiguchi and Shinji Seto. 2013. Study on Maximum Receivable Distance for Radiated Emission of Information Technology Equipment Causing Information Leakage. *IEEE Transactions on Electromagnetic Compatibility* 55, 3 (2013), 547–554.

[75] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. 2016. HDFI: Hardware-Assisted Data-Flow Isolation. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. 1–17. https://doi.org/10.1109/SP.2016.9

[76] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/1024393.1024404

[77] Yasunao Suzuki and Yoshiharu Akiyama. 2010. Jamming technique to prevent information leakage caused by unintentional emissions of PC video signals. In *Electromagnetic Compatibility (EMC), 2010 IEEE International Symposium on*. 132–137.

[78] H. Tanaka. 2007. Information leakage via electromagnetic emanations and evaluation of Tempest countermeasures. In *Lecture notes in computer science, Springer*. 167–179.

[79] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. 2014. Unsupervised Anomaly-Based Malware Detection Using Hardware Features. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*. 109–129.

[80] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. 2008. Flexitaint: A programmable accelerator for dynamic taint propagation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 173–184.

[81] Zhenghong Wang and Ruby B. Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*. 494–505.

[82] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. 2012. CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (DSN '12)*.

[83] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 719–732.

[84] Ilsun You and Kangbin Yim. 2010. Malware Obfuscation Techniques: A Brief Survey.. In *BWCCA*. Citeseer, 297–300.

[85] Alenka Zajić and Milos Prvulovic. 2014. Experimental Demonstration of Electromagnetic Information Leakage From Modern Processor-Memory Systems. *IEEE Transactions on Electromagnetic Compatibility* 56, 4 (2014), 885–893.

[86] Mingwei Zhang and R Sekar. 2013. Control flow integrity for COTS binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. 337–352.

[87] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. 2016. A Software Approach to Defeating Side Channels in Last-Level Caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 871–882.