Program Profiling Based on Markov Models and EM Emanations

Baki Berkay Yilmaz[a*], Elvan Mert Ugurlu[a*], Frank Werner[a], Milos Prvulovic[b], and Alenka Zajic[a]

[a]The School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA
[b]The School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, USA

ABSTRACT

As one of the fundamental approaches for code optimization and performance analysis, profiling software activities can provide information on the existence of malware, code execution problems, etc. In this paper, we propose a methodology to profile a system with no overhead. The approach leverages electromagnetic (EM) emanations while executing a program, and exploits its flow diagram by constructing a Markov model. The states of the model are considered as the heavily executed blocks (called *hot paths*) of the program, and the transition between any two states is possible only if there exists a branching operation which enables execution of corresponding states without any intermediate state. To identify the state of the program, we utilize a supervised learning method. To do so, we first collect signals for each state, extract features, and generate a dictionary. The features are considered as the activated frequencies when the program is executed. The assumption here is that there exists at least one unique frequency component that is only active for one unique state. Moreover, to degrade the effect of interruptions and other signals emanated from other parts of the device, and to obtain signals with high Signal-to-Noise Ratio (SNR), we average the output of Short-Time Fourier Transform (STFT). After extracting features, we apply Principle Component Analysis (PCA) for dimension reduction which helps monitoring systems in real time. Finally, we describe experimental setup and show results to demonstrate that the proposed methodology can detect malware activity with high accuracy.

Keywords: Spectral Profiling, EM Emanations, Side Channels, Malware Detection.

1. INTRODUCTION

Program profiling is a technique used by developers to analyze the performance of a given system and improve the performance by optimization procedures.[1–4] The main objective of profiling is to diagnose the parts of a code that are frequently executed. These heavily executed parts of the code are called *hot path*s or *hot region*s, and they are generally comprised of loops, functions and code blocks that are called several times during program execution. For optimization purposes, developers focus on these *hot region*s since the program spends most of the time in these regions.[5] Another useful application of profiling is *malware detection*.[6,7] Since malware usually changes the execution order of the program, monitoring the paths that are taken during program execution can reveal information about the existence of malware.

Profiling is typically implemented using *instrumentation*,[8–11] where a specific code is introduced before and/or after the *hot region*s. These specific codes enable the profiler to know the execution order of the *hot region*s as well as the time spent in each of them. However, insertion of these specific codes poses two significant disadvantages: 1) instrumentation code is executed during the program execution and causes over-head, which is especially problematic for Internet of things (IoT) devices with limited resources, and 2) insertion of the special code causes major changes in the behavior of the program.[12]

Profiling can also be realized with an *external* monitoring system that does not require *instrumentation*.[12–16] The main advantages of such systems are: 1) the profiling system does not use the resources of the monitored system, and hence has *zero-overhead*, 2) the profiling system is seperate from the monitored system that results

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

in *observer-effect-free* monitoring, and 3) profiling system is physically isolated from the monitored system, and this is especially useful for malware detection since the malware to be detected cannot interfere with the profiling system. On the other hand, unlike *instrumented* profiling, an *external* monitoring system requires an extra system that performs data acquisition and data processing.

The proposed profiling scheme in this work utilizes electromagnetic (EM) side channels. These channels are unintentionally generated EM emanations that are by-products of the changes in the current flows and transistor switches.[17,18] Earlier research has shown that these emanations are correlated with the program activity and can be leveraged to infer further information about the program.[17–22] A large body of research in EM side channels are focused on applications in cryptography.[16,18,23,24] EM side channels are also used to track program activity at different levels such as loops, code blocks, and instructions.[13,16,25,26] Another application is finding anomalies in software activity.[12–16] In addition to the EM side-channels, other pysical side channels such as acoustic emanations,[27,28] power analysis,[29–32] temperature changes,[33,34]etc. are also utilized for similar purposes.

Most programs and embedded systems maintain periodic activities during their operations. In particular, program spends most of the time in loops or functions that are called many times. These periodic operations result in unintentional EM emanations whose spectrum contains peaks at corresponding frequencies. Spectral Profiling shows[12] these peaks occur at frequencies that are the inverse of the per-iteration time of the periodic activities. In other words, if the per-iteration time of a periodic activity is T seconds, the corresponding spectrum has a peak around the fundamental frequency $f = 1/T$ Hertz. In addition to the fundamental frequency, peaks at the harmonics of the fundamental frequency ($2f, 3f, 4f$, etc.) are also observed in the spectrum.

As shown in FASE,[35] EM emanations are generally amplitude-modulated on harmonics of periodic signals such as operating clock signal, voltage regulators, etc. Among these periodic signals, we observe the strongest modulation around the first harmonic of the clock frequency. As a matter of fact, when a periodic activity with per-iteration time T and corresponding fundamental frequency $f = 1/T$ gets modulated on operating clock frequency, f_c, the spectrum contains spikes at frequencies $f_c \pm f, f_c \pm 2f, f_c \pm 3f$, etc.

Based on these observations, EDDIE[14] and SYNDROME[15] propose malware detection frameworks by monitoring the spectrum of the operating devices and comparing them with malware-free spectrum. In particular, EDDIE compares observed short-time spectrum (STS) with the training STSs and makes decisions based on a statistical test. SYNDROME utilizes similar techniques and extends the applicability of the framework to a variety of devices, particularly real-world medical devices. Both EDDIE and SYNDROME perform well with high Signal-to-Noise Ratio (SNR) levels, however, they fail under low SNR conditions. Also, in these work, STSs of observed signals are compared with training signals based on pre-determined frequencies that might result in low accuracy when the clock frequency shifts. ZOP[13] presents a zero-overhead profiling framework that uses *signature matching* and reports accuracy as high as 95%, but the proposed framework is fairly complicated, which makes implementation on computers with limited resources relatively difficult.

We propose a new monitoring model that is based on Markov Models. As mentioned earlier, during execution, a program goes through *hot path*s such as loops and functions whose per-iteration time and spectrum vary. We exploit these changes in the spectrum and use these *hot path*s as the Markov states of the proposed framework. Possible transitions from these *hot path*s are assumed to be known apriori. This is a valid assumption since training phase enables access to the source code. Hence, these *hot path*s and possible transitions can be extracted through a deep-investigation of possible branching operations or by means of compile-time analysis. Once these transitions are determined, training phase only requires a signal flow that contains single execution of each *hot path*.

In training phase, we extract features by using Short-Time Fourier Transform (STFT), and identify important frequencies. Next, we apply principal component analysis (PCA)[36] on the training signals to generate a model that significantly reduces data size by keeping distinctive features. This reduction in the data size is especially important for real-time monitoring system with high sampling rates and it significantly simplifies the computation complexity. After extracting the features, we monitor the program flow with a Markov Model whose parameters are also trained during a training phase.

We test the proposed framework on a single-board Linux computer and an Android phone by using different applications. The results indicate that the state of the program can be successfully determined for both devices and all applications. Furthermore, any *injection-based attack* that is larger than the sensitivity of the trained model can also be successfully detected.

The rest of this paper is organized as follows. Section 2 explains data size reduction and feature extraction techniques, Section 3 describes the Markov Model-based monitoring system, Section 4 presents the experimental setup and discusses the results, while Section 5 provides a summary and draws conclusions.

2. EXTRACTING FEATURES FOR MARKOV MODEL USING PCA

In this section, we describe how to extract features from raw recorded EM emanation signals. As mentioned earlier, EM emanations are usually modulated on strong periodic signals that are present on the board. In this context, raw recorded electromagnetic emanation signals refer to the EM side channel content around the first harmonic of clock frequency. In other words, raw signals are the time domain recordings whose frequency spectrum is centered around the operating clock frequency.

The feature extraction described in this section contains three main steps: 1) identifying major frequencies and reducing the dimension of the data size based on the identified major frequencies, 2) applying PCA for further data size reduction in an optimal manner, and 3) combining above-mentioned steps to extract features to be used by the Markov-based monitoring model. These steps are explained in more detail as follows.

2.1 Primary Dimension Reduction by Identifying Important Frequencies

As mentioned earlier, during the execution of each *hot region*, the spectrum contains peaks at corresponding fundamental frequencies. In addition to these frequencies, additional peaks occur at different frequencies due to interrupts and external noise. Since these additional peaks do not reveal any further information about the *hot regions*, they are *undesired* peaks and should be omitted. In this section, we explain our methodology to identify the relevant frequencies and disregard the *undesired* ones. This identification serves as the primary step of input data size reduction for the overall framework and is based on STFT averaging in.[37] Before STFT averaging, we divide the measured raw training signal, $\mathbf{\Theta}$, into smaller subsections, $\mathbf{\Theta}_i$, each of length I_S samples. Setting the number of non-overlapping samples between consecutive STFT operations as O_S, the number of STFT operations for each $\mathbf{\Theta}_i$ is Ψ where

$$\Psi = \texttt{floor}\left(\left(I_S - \mathcal{F}\right)/O_S + 1\right),$$

and $\mathcal{F}$ is the FFT-size of the STFT window. These STFT operations are averaged and major frequency components are determined with an adaptive threshold. Note that Ψ is the number of STFT instances to be averaged and is determined by the length of $\mathbf{\Theta}_i$, or equivalently, I_S. Therefore, the *averaged spectrum* $\mathbf{m}_i$ for the i^{th} subsection is written as

$$\mathbf{m}_i[k] = 20\log_{10}\left\{\frac{1}{\Psi}\sum_{n=1}^{\Psi}\left|X_n^i[k]\right|\right\}, \tag{1}$$

where $k \in \{0, 1, \cdots, \mathcal{F}-1\}$, and

$$X_n^i[k] = \sum_{\xi=0}^{\mathcal{F}-1}\mathbf{\Theta}_i\left[\xi + (n-1)O_S\right]\exp\left(-j2\pi k\xi/\mathcal{F}\right). \tag{2}$$

Note that Equation 1 performs a log-transformation to reduce the skewness of the STFTs, $X_n^i[k]$, in Equation 2. Furthermore, experiments show that the modulating clock frequency slightly varies during program execution and this results in small shifts of the fundamental frequencies in the spectrum. To account for this shifting phenomena, we downsample $\mathbf{m}_i[k]$ with a max-pooling filter of length N_m. Consequenyly, the size of $\mathbf{m}_i[k]$ reduces by a factor of N_m and this process compensates for possible frequency shifts. Next, we obtain an adaptive threshold $\mathbf{t}_i[k]$ for $\mathbf{m}_i[k]$ by means of moving filters and identify the major frequency peaks that fall above this threshold. Figure 1 demonstrates a sample *averaged spectrum* and adaptive threshold for a sample subsection where FFT-size is 4096 and downsampling factor N_m is 4. Note that any frequency index whose value

falls above the threshold qualifies as a major frequency for the given subsection. The spectrum averaging and major frequency identification process is repeated for all subsection $\mathbf{\Theta}_i$'s and the history of major frequencies are kept to generate *Major Frequency Identifier* model. Given an STFT operation of FFT-size $\mathcal{F}$, *Major Frequency Identifier* filters out the irrelevant frequencies and outputs a vector of size N_{MF}. Note that N_{MF} is dependent on application and the history of major frequencies.

2.2 Secondary Dimension Reduction by PCA

This section descibes how data input size is further reduced in an optimal manner by PCA. During training, the *hot regions*, or in other words, the states are known. Therefore, we can obtain a data matrix $\mathbf{M}$ that consists of *averaged spectra* corresponding to each state. Let N_S be the number of states that are present in the monitored system, and N_A be the number of *averaged spectra* that $\mathbf{M}$ should contain per each state. In a similar manner as in Equation 1, we can extract N_A *averaged spectra* for each state from N_A different subsections of that state and generate $\mathbf{M}$ by filling its columns with these *averaged spectra*. Note that $\mathbf{M}$ is a matrix of size $\mathcal{F} \times (N_A \times N_S)$ where $\mathcal{F}$ is the FFT-size. First N_A columns of $\mathbf{M}$ contain *averaged spectra* for state 1, second N_A columns contain *averaged spectra* for state 2, and last N_A columns contain *averaged spectra* for state N_S. By using *Major Frequency Identifier* described in Section 2.1, we identify the major frequency indices and decrease the size of $\mathbf{M}$ to $N_{MF} \times (N_A \times N_S)$ and denote the resulting matrix as $\mathbf{A}$.

To reduce the dimension of $\mathbf{A}$ in a way that keeps the distinctive features among its columns as much as possible, we propose to apply PCA.[36] To do so, we first need to subtract the mean of the *averaged spectra* from all columns of $\mathbf{A}$ and generate a new *zero-mean* matrix $\tilde{\mathbf{A}}$ whose columns are given by

$$\tilde{\mathbf{a}}_n = \mathbf{a}_n - \boldsymbol{\mu}, \tag{3}$$

where $\mathbf{a}_n$ are the columns of $\mathbf{A}$, and $\boldsymbol{\mu}$ is the mean of the columns of $\mathbf{A}$ given by

$$\boldsymbol{\mu} = \frac{1}{N_A \times N_S} \sum_{n=1}^{N_A \times N_S} \mathbf{a}_n. \tag{4}$$

Note that $\tilde{\mathbf{A}}$ is also of size $N_{MF} \times (N_A \times N_S)$. Typically, $N_{MF} < (N_A \times N_S)$ and $\tilde{\mathbf{A}}$ is full-rank, meaning that rank of $\tilde{\mathbf{A}}$, denoted by R, is equal to N_{MF}. Therefore, by computing the singular value decomposition (SVD) of $\tilde{\mathbf{A}}$, we get $\tilde{\mathbf{A}} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ where

- $\mathbf{U}$ is a $N_{MF} \times R$ matrix whose columns $\mathbf{u}_n \in \mathbb{R}^{N_{MF}}$ form an orthobasis for the range space of $\tilde{\mathbf{A}}$,
- $\mathbf{V}$ is a $(N_A \times N_S) \times R$ matrix whose columns $\mathbf{v}_n \in \mathbb{R}^{(N_A \times N_S)}$ form an orthobasis for the range space of $\tilde{\mathbf{A}}^T$, and

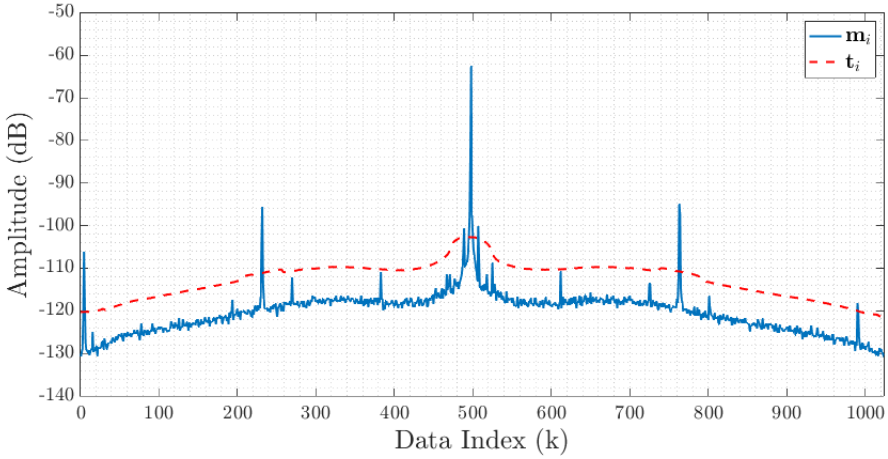


Figure 1. Downsampled frequency spectrum for a sample subsection ($\mathbf{m}_i$ and $\mathbf{t}_i$ denote the sample *averaged spectrum* and corresponding adaptive threshold, respectively.)

- $\mathbf{\Sigma}$ is a $R \times R$ diagonal matrix given by

$$\mathbf{\Sigma} = \begin{pmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_R \end{pmatrix}, \tag{5}$$

  where σ_r are the singular values of $\tilde{\mathbf{A}}$ ordered as $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_R > 0$.

To project $\tilde{\mathbf{A}}$ from a N_{MF}-dimensional space to a lower dimensional space with dimension K, we define

- $\mathbf{U}_K = [\mathbf{u}_1\ \mathbf{u}_2\ \dots\ \mathbf{u}_K]$, a $N_{MF} \times K$ *projection matrix* whose columns $\mathbf{u}_{K,n} \in \mathbb{R}^{N_{MF}}$ are the first K columns of $\mathbf{U}$ and form an basis for the K-dimensional subspace
- $\mathbf{\Sigma}_K^{-1}$, a diagonal $K \times K$ *normalization matrix* that contains the inverse of the largest K singular values of $\tilde{\mathbf{A}}$ in its diagonal entries as

$$\mathbf{\Sigma}_K^{-1} = \begin{pmatrix} \frac{1}{\sigma_1} & 0 & \dots & 0 \\ 0 & \frac{1}{\sigma_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{\sigma_K} \end{pmatrix}. \tag{6}$$

Using the *projection* and *normalization* matrices, $\mathbf{U}_K$ and $\mathbf{\Sigma}_K^{-1}$, we propose two projection schemes to obtain the matrix $\mathcal{A}$ that is the projection of $\tilde{\mathbf{A}}$ onto K-dimensional subspace.

1. **Non-Normalized Projection**: $\mathcal{A} = \mathbf{U}_K^T \tilde{\mathbf{A}}$, where the dimensions corresponding to larger singular values have larger impact because larger singular values correspond to larger variation.
2. **Normalized Projection**: $\mathcal{A} = \mathbf{\Sigma}_K^{-1} \mathbf{U}_K^T \tilde{\mathbf{A}}$, where contribution of each singular value is equalized so that each dimension has the same weight.

The decision on which of these projection schemes should be chosen is based on the application. Also note that if the largest K singular values have similar values, the normalized projection converges to its non-normalized counterpart. In either case, $\mathcal{A}$ is a $K \times (N_A \times N_S)$ matrix. As a result of two-staged dimension reduction, the dimension of a single *averaged spectrum* is reduced from $\mathcal{F}$ to K. Considering that typical values for $\mathcal{F}$ and K are 4096 and less than 10, respectively, this is a significant data size reduction. Also, one should note that this reduction is performed in an optimal way that keeps the distinctive features among different *averaged spectra* as much as possible.

As a result of secondary data size reduction, we have learned the parameters for *Subspace Projector* ($\boldsymbol{\mu}$, $\mathbf{\Sigma}_K^{-1}$ and $\mathbf{U}_K$) as well as the *Comparison Model* ($\mathcal{A}$ and labels denoting the state of each column of $\mathcal{A}$). One should note that as much as computing the SVD of a matrix might be costly, this computation is only performed in the training phase and only one time. Once the system is trained and the parameters are learned, during monitoring, the projection is performed by means of basic matrix operations such as subtraction, multiplication, etc.

2.3 Feature Extraction for Markov Model

This section describes how aforementioned data size reduction models, *Major Frequency Identifier* and *Subspace Projector*, are utilized to extract features for Markov model during monitoring phase. Similar to the training, during monitoring, we swipe through the recorded raw signal, apply STFT operations and compute *averaged spectra* as shown in Equation 1.

Each computed *averaged spectrum*, denoted by $\mathbf{m} \in \mathbb{R}^{\mathcal{F}}$, corresponds to one *observation* that represents the current state of the program. To reduce the size of $\mathbf{m}$ and extract features for Markov model, we perform the following procedure:

1. Obtain the major frequencies, $\boldsymbol{\alpha} \in \mathbb{R}^{N_{MF}}$, from $\mathbf{m}$ by using *Major Frequency Identifier*.

2. Subtract $\boldsymbol{\mu}$ (learned mean parameter) from $\boldsymbol{\alpha}$ to obtain $\tilde{\boldsymbol{\alpha}} = \boldsymbol{\alpha} - \boldsymbol{\mu}$.

3. Project $\tilde{\boldsymbol{\alpha}}$ onto K-dimensional subspace using $\boldsymbol{\Sigma}_K^{-1}$ and $\mathbf{U}_K$ to obtain $\hat{\boldsymbol{\alpha}}$. One should remark that this projection can be either

   (a) **Non-Normalized Projection**: $\hat{\boldsymbol{\alpha}} = \mathbf{U}_K^T \tilde{\boldsymbol{\alpha}}$, or,

   (b) **Normalized Projection**: $\hat{\boldsymbol{\alpha}} = \boldsymbol{\Sigma}_K^{-1} \mathbf{U}_K^T \tilde{\boldsymbol{\alpha}}$.

   Note that the projection at this stage should be the same as the projection performed during the training phase. After the projection, $\hat{\boldsymbol{\alpha}} \in \mathbb{R}^K$ is a representation of the *averaged spectrum*, $\mathbf{m}$, in K-dimensional subspace.

4. Calculate the distance vector $\mathbf{d} \in \mathbb{R}^{N_A}$ whose elements are the distances of the observation to corresponding states. This distance is calculated by evaluating the ℓ_2-norm between $\hat{\boldsymbol{\alpha}}$ and the columns of *Comparison Matrix* ($\mathcal{A}$). Since the labels (the states that columns of $\mathcal{A}$ belong to) are known, we average the evaluated ℓ_2-norms for each state to obtain an average distance to each state, and fill in the corresponding elements of $\mathbf{d}$ with the average distances.

Consequently, the extracted feature for a given observation is the distance vector $\mathbf{d} \in \mathbb{R}^{N_A}$. One straightforward approach to detect the current state is to pick the state with the minimum distance. During execution, the program might exit the main state for short duration of time as a result of interrupts and inter-loop stages that are not necessarily malware. If the observation is made during these short out-of-state stages, the minimum distance might favor incorrect states that are not possible to transition to. In such a situation, selecting the state with minimum distance from a single observation would alert malware even if the system is malware-free. Therefore, this approach results in very high false positives (alerting malware when there is no malware). To decrease the false positive rates and increase the accuracy of malware detection, in the next section, we propose a Markov-based monitoring model that uses the distance vector $\mathbf{d}$ as its input.

3. MARKOV MODEL-BASED DETECTION AND MONITORING

This section presents the algorithm to monitor the program and detect malware based on a Markov model. For ease of explanation, we first present the algorithm used for testing in Section 3.1, and in Section 3.2, we present how the parameters are determined during thetraining phase. Note that in real implementation, these are performed in reverse order, i.e., the parameters are learned in the training phase before testing starts.

3.1 Markov Model-based Monitoring Algorithm

A slightly simplified version of the algorithm to monitor the program is presented in Algorithm 1. The model contains three parameters: 1) $\mathbf{t}_S$, an N_A-dimensional vector containing minimum amount of time that program needs spend in each state, 2) t_L, threshold for announcing that the tested signal belongs to a cluster (i.e., state), and 3) m_g, maximum allowable *glitch* occurrence before announcing malware. A *glitch* refers to an observation for which the predicted state is not an allowable state based on the previous states. There are several factors that can cause a *glitch*: malware, interrupts, and inter-loop stages, etc. As mentioned earlier, announcing malware after a single *glitch* results in high false positive rates. To compansate for that, we announce malware if m_g consecutive glitches occur. One should note that all of these parameters are pre-determined in training phase as described in Section 3.2.

In addition to three aforementioned parameters, the model also takes a state transitions dictionary as an input. This dictionary contains allowable transitions between states and possible *exit* states (allowable states from which the Markov chain can be exited).

Profiling starts by obtaining the distance vector $\mathbf{d}$ for a given observation as described in Section 2. In the beginning of a program, we monitor until the distance to the beginning state is below the threshold t_L, once this is satisfied, monitoring starts. As a design specification to reduce false positives, we prioritize staying in the

Algorithm 1: Profiling Procedure

Data: $\boldsymbol{\mu}$, $\mathbf{U}_K$, $\boldsymbol{\Sigma}_K^{-1}$, $\mathcal{A}$

// $\mathbf{t}_S$: Vector containing minumum amount of time for execution of each state.
// t_L: Threshold for announcing the tested signal belongs to a cluster.
// m_g: Maximum time for glitch occurrence before announcing malware.
// *dict*: Dictionary that stores state transitions and exit states.
// $\mathbf{d}$: Vector containing the distance values to each state.
not_detected = 0
malware_detected = # of states + 1
current = *not_detected*
`transitions` = []
started = 0 // tracks whether the first state has occurred.
c_state = 0 // state counter.
c_glitch = 0 // glitch counter.

while *true* **do**
  Apply the procedure in Section 2 to obtain the distance vector $\mathbf{d}$.
  if *started* **then**
    if $\mathbf{d}$[*current*] *is smaller than* t_L **then**
      c_state += 1
      c_glitch = 0
    else
      if *c_state is larger than* $\mathbf{t}_S$[*current*] **then**
        candidates = find candidate states from *dict* based on *current*
        best_candidate = get the candidate with lowest $\mathbf{d}$[*candidates*] value
        if $\mathbf{d}$[*best_candidate*] *is smaller than* t_L **then**
          current = *best_candidate* // state transition occurs.
          c_state = 0
          c_glitch = 0
        else
          c_glitch += 1
        end
      else
        c_glitch += 1 // minimum time from state transition has not passed yet.
      end
    end
  else
    next = *1* // start state.
    if $\mathbf{d}$[*next*] *is smaller than* t_L **then**
      current = *next*
      started = 1
    end
  end
  Append *current* to `transitions`
  if *c_glitch is larger than* m_g **then**
    if *isExitState(current)* **then**
      Display message: "Markov State Exited."
      current = 0
    else
      Display message: "MALWARE DETECTED!"
      current =*malware_detected*
    end
    Set last m_g elements of `transitions` to *current*
    started = 0
    current = *not_detected*
    c_state = 0
    c_glitch = 0
  end
end

Result: `transitions`

same state over transitioning to another state, in other words, the previous state is favored over the other states. To realize that, we first check whether the distance to the previous state is less than t_L, in which case the current state is predicted as the previous state. Otherwise, we check whether the program has stayed in the current state longer than the minimum time that needs to be passed before leaving that state ($\mathbf{t}_S[current]$) . If this minimum time has not passed, transition is not possible, hence we note this observation as a *glitch* and the *glitch counter* gets incremented. However, if the minimum time has passed, transition to another state is possible. In this case, we use the dictionary to find the possible *candidate* states that can be transitioned to. Among these *candidate*s, the candidate with the minimum distance is selected as the *best candidate*. If the distance to *best candidate* is less than t_L, the current state is predicted as this *best candidate* state. Otherwise, the observation qualifies as a *glitch* that increments the *glitch counter*.

Next, the predicted state is appended to the transitions array. Then, we check whether *glitch counter* is larger than m_g. If *glitch counter* is not larger than m_g, we proceed to the next observation by calculating the new distance vector $\mathbf{d}$ and following the same procedure. However, if the *glitch counter* is larger than m_g, there is either a malware or Markov chain is exited through an *exit* state. To determine that, we check the dictionary to see if the current state is an *exit* state. Based on the this, we display a message that denotes the existence of malware or the exit from the Markov State.

3.2 Learning Parameters for Markov Model

This section describes how the parameters t_L , $\mathbf{t}_S$, and m_g are determined in training phase. To determine these parameters, we only need one path that contains a typical length of observations for each state. Let N_O be the total number of observations corresponding to the training signal, and $\mathbf{d}_i$ be the distance vector corresponding to the i^{th} observation where $i \in \{1, 2, \ldots, N_O\}$. The threshold t_L is determined as

$$t_L = \mu_L + 2.5 \times \sigma_L, \tag{7}$$

where

$$\mu_L = \frac{1}{N_O} \sum_{i=1}^{N_O} min(\mathbf{d}_i), \tag{8}$$

$$\sigma_L = \sqrt{\frac{1}{N_O} \sum_{i=1}^{N_O} \left(min(\mathbf{d}_i) - \mu_L\right)^2}, \tag{9}$$

and $min(\mathbf{d}_i)$ denotes the minimum element of the i^{th} distance vector $\mathbf{d}_i$.

Training signal is collected from a malware-free program execution, therefore, we know that monitoring the training signal should not alert malware. Motivated by this idea, in the training phase, we use an altered version of Algoritm 1 to determine $\mathbf{t}_S$ and m_g with the following modifications:

1. m_g and $\mathbf{t}_S$ are initialized as *infinty* and unit vector (a $N_S \times 1$ vector whose all elements are 1), respectively. By setting m_g to *infinty*, we ensure that the entire training signal is monitored without *glitch counter* exceeding m_g and alerting malware. Also, setting $\mathbf{t}_S$ to unit vector makes transitions to allowable states possible regardless of the time spent in the current state. These two assignments are valid for monitoring a malware-free training signal.

2. A new empty vector $\mathbf{h}_g$ (*glitch history*) is introduced, and the value of *glitch counter* is appended to $\mathbf{h}_g$ before every operation where *glitch counter* is reset to 0. Note that $\mathbf{h}_g$ keeps track of the number of consecutive glitches that occur during malware-free training signal, and it also represents the length of *glitch*es that occur due to interrupts and inter-loop stages.

After all observations of the training signal are evaluated by Algorithm 1, we obtain a vector $\mathbf{h}_g$ of size $N_g \times 1$, where N_g represents how many times the *glitch counter* was reset to 0. The elements of $\mathbf{h}_g$ are given by $\mathbf{h}_{g,i}$ for

$i \in \{1, 2, \ldots, N_g\}$, i.e., $\mathbf{h}_g = [\mathbf{h}_{g,1}\ \mathbf{h}_{g,2}\ \ldots\ \mathbf{h}_{g,N_g}]^T$. By using $\mathbf{h}_g$, m_g is determined as

$$m_g = round\left(max(\mathbf{h}_g) + 3 \times \sqrt{\frac{1}{N_g}\sum_{i=1}^{N_g}(\mathbf{h}_{g,i} - \mu_g)^2} \right), \tag{10}$$

where

$$\mu_g = \frac{1}{N_g}\sum_{i=1}^{N_g}\mathbf{h}_{g,i}, \tag{11}$$

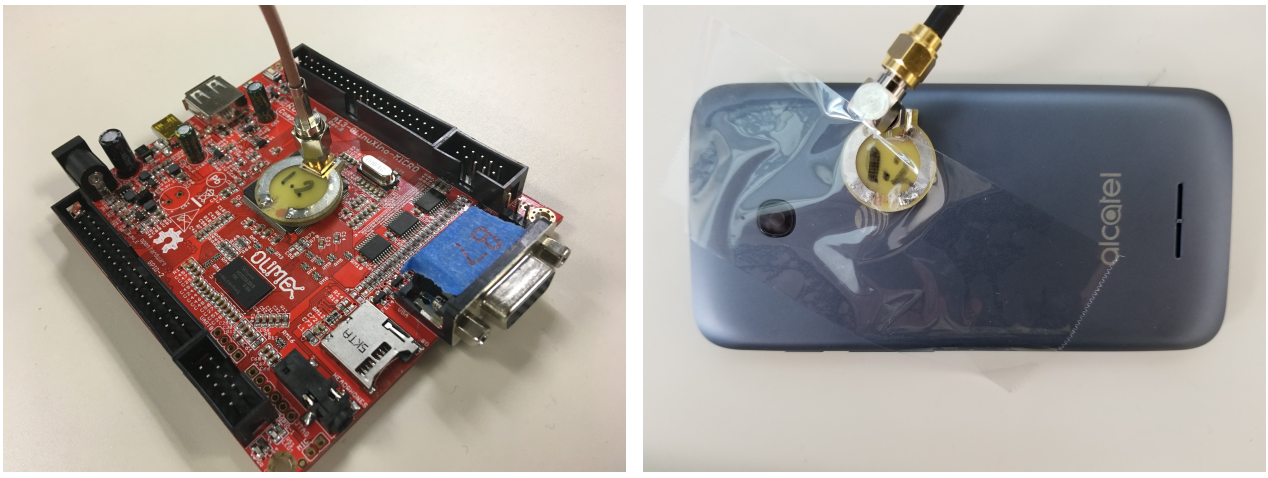
and $max(\mathbf{h}_g)$ denotes the maximum element of $\mathbf{h}_g$.

Assuming that the training signal is a typical flow of the program, we use the times spent in each state of the training signal to calculate $\mathbf{t}_S$. The altered version of Algorithm 1 outputs the corresponding states of the observation stream. As a rule of thumb, we determine the minimum time that needs to be spend in each state as 5% of the time spent for the training signal for each corresponding state. Note that, this percentage might vary for different programs and also different states of the same program. Therefore, the specific characteristics of the program and different states should be taken into account in the training phase while designing a monitoring system for a new program.

4. EXPERIMENTAL SETUP AND RESULTS

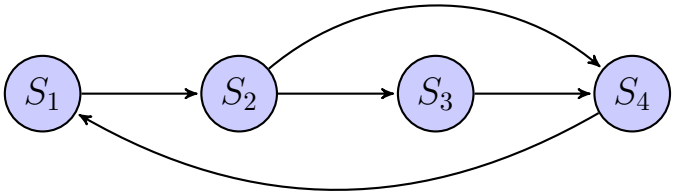
In this section, we present the experimental setup that is used to test the proposed monitoring framework and we also present the experimental results. The experiments were conducted on two different devices: 1) OLuniXino A13,[38] a single-board Linux computer with a dual-issue, in-order ARM Cortex processor, and 2) Alcatel Ideal,[39] an Android phone with 1.1 GHz Qualcomm Snapdragon 210 processor. To capture the EM emanations, we place a custom-made near-field probe above the processor of these devices as shown in Figure 2. The data is recorded by a spectrum analyzer (Keysight N9020A MXA[40]) with a sampling period of 260 nanoseconds and 3 MHz bandwidth. The reason behind using such an advanced device is the built-in features of the device such as real-time signal display and calibrated measurements. However, any other measurement device such as commercial software-defined radio receivers also provide similar results without a significant performance loss.

In our experiments, we use a program that is created using SAVAT[17] as well as two applications (*Basicmath* and *Bitcount*) from the MiBench[41] suite. The state transition diagrams for these programs are given in Figure 3. Note that these programs consist of different number of states and different possible state transitions.

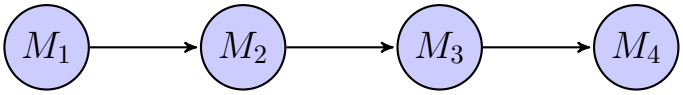


(a) OLinuXino A13.[38] (b) Alcatel Ideal.[39]

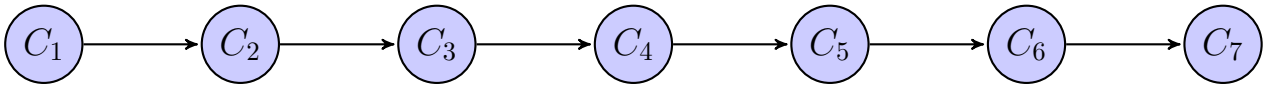
Figure 2. Experimental setups for the measurements.



(a) The state transition diagram of *SAVAT*[17]-based program.



(b) The state transition diagram for *Basicmath* from MiBench.[41]



(c) The state transition diagram for *Bitcount* from MiBench.[41]

Figure 3. State Transition Diagram for Tested Programs

First, we train Markov Model-based monitoring systems for each of these programs, and then we monitor for program profiling and malware detection. To provide the reader with rule-of-thumb choices for the parameters that are mentioned in Sections 2 and 3, in Table 1, we list the parameter values that are used for training the models. Also, in all models, the projections on lower dimensional subspaces are performed as *non-normalized projection*s.

Table 1. Parameter Values used for Training the Models in the Experiments.

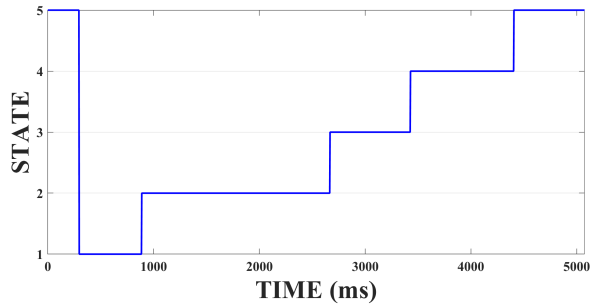
Parameter	**Chosen Value**
Ψ	20
$\mathcal{F}$	4096
O_S	2048
N_m	4
N_A	40
K	3

We present the results of the program profiling and malware detection capabilities of the proposed framework in two sections: 1) program profiling of the malware-free SAVAT-based program and determining taken *hot path*s, and 2) detecting malware that are injected into MiBench applications.

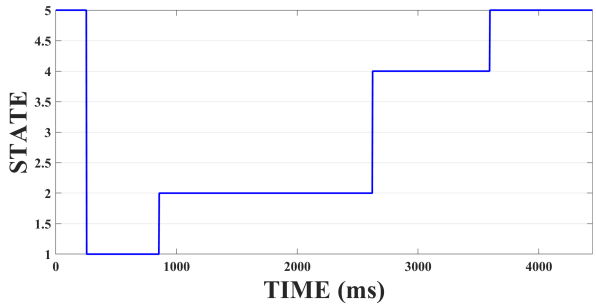
4.1 Program Profiling of Malware-free SAVAT-based Program

The SAVAT-based program whose state transition diagram is shown in Figure 3(a) is a proof-of-concept implementation of a program that has multiple transitions between different states. The main objective to test such a system is to show the profiling capability of the proposed framework on programs with relatively complicated state transitions.

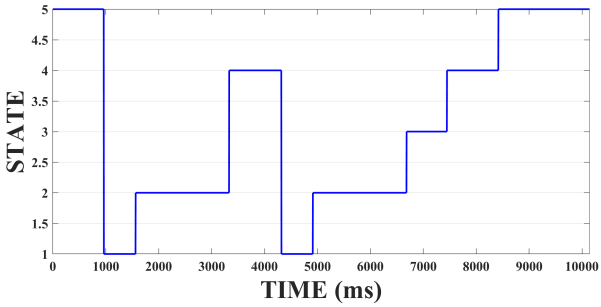
The monitoring results are shown in Figure 4. One should note that the SAVAT-based program has 4 states, and State 5 in the figures represents the *idle* state, where program is not in any of the *hot regions*. Figure 4 shows that the proposed framework can successfully profile the given program with different transition possibilities and determine the current state of the program at a given time.



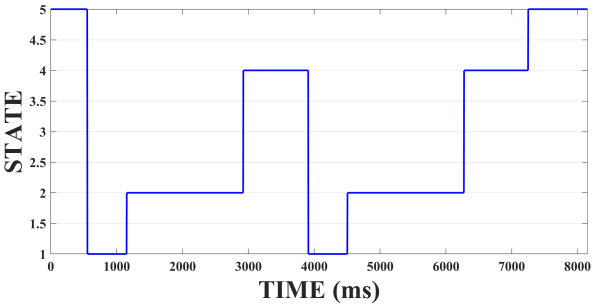
(a) State transition when program execution follows the path 1-2-3-4.



(b) State transition when program execution follows the path 1-2-4.



(c) State transition when program execution follows the path 1-2-4-1-2-3-4.



(d) State transition when program execution follows the path 1-2-4-1-2-4.

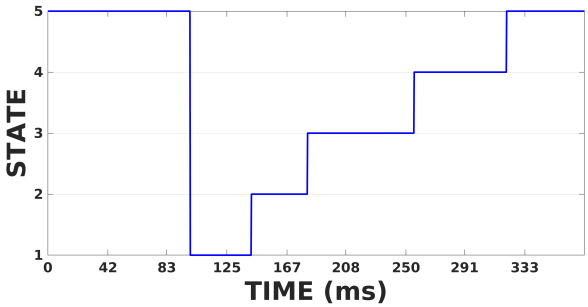
Figure 4. State transitions for a program with the state diagram in Figure 3(a).

4.2 Detecting Malwares Injected into MiBench Applications

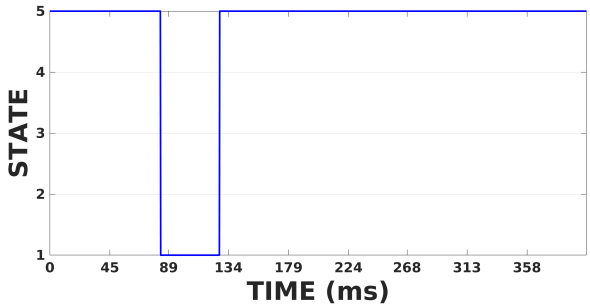
In this section, we provide the results of malware detection performance of the proposed framework using MiBench applications. The tested malware in our experiments are *injection-based attacks*. Additional malicious code segments can be injected inside or outside of the loops. Injections within the loops cause the per-iteration time of the loops to increase, which results in a decreased fundamental frequency, and hence a change in the spectrum. On the other hand, attacks outside the loop result in an increased inter-loop time. Note that injections within the loops are generally easier to detect because they change the spectrum of the entire state, whereas injections outside the loops only change the spectrum of the inter-loop state. Therefore, the minimum injection size that can be detected is generally larger for outside loop injections. However, this is not a very restricting limitation because attackers need to inject many instructions to inter-loop stages whereas only few instructions injected inside loops can help them to achieve their goals. Note that the sensitivity of the malware detection depends on the parameter m_g, and injections that take less time than m_g generally can not be detected.

Figure 5 and 6 represent predicted state transitions under different scenarios for *Basicmath*, and *Bitcount*, respectively. Figures 5(a) and 6(a) show the state transitions when the monitored programs are malware-free. Note that State 5 and State 8 represent the *idle* states for *Basicmath*, and *Bitcount*, respectively. The other figures in Figure 5 and 6 presents the state transitions with three different types of malware. In all these scenarios, the malware is successfully detected.

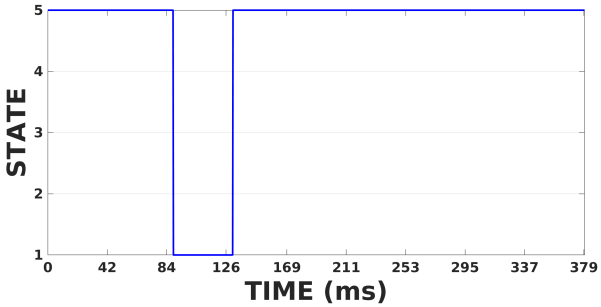
The results in Section 4.1 and 4.2 demonstrate that the proposed framework can both determine the current state of the program for profiling, and detect injected malware.



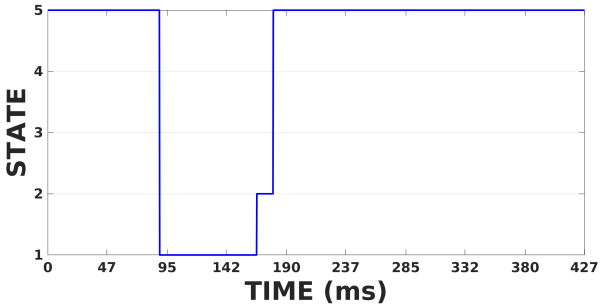
(a) State transitions for the Basicmath program when it is working properly.



(b) State transitions for the Basicmath program when it has malware Type-1.

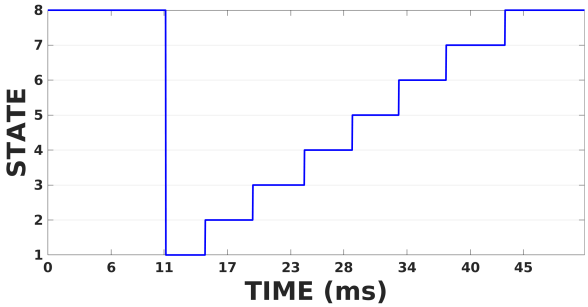


(c) State transitions for the Basicmath program when it has malware Type-2.

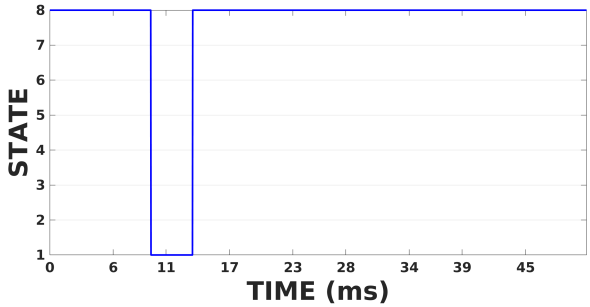


(d) State transitions for the Basicmath program when it has malware Type-3.

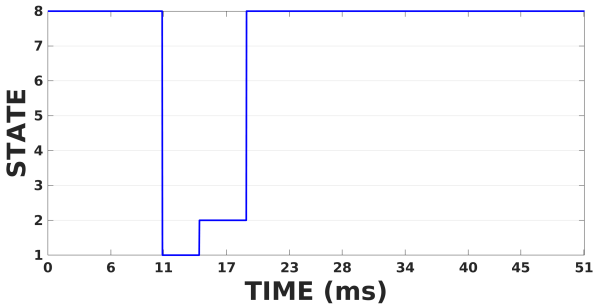
Figure 5. State transitions for the Basicmath program with the state diagram in Figure 3(b).



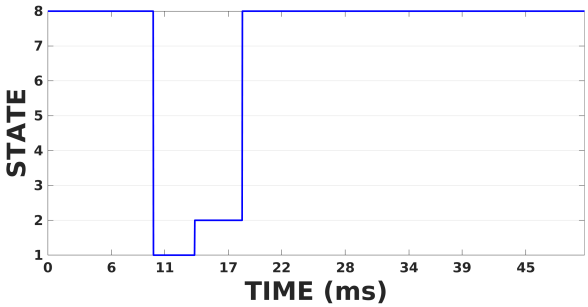
(a) State transitions for the Bitcount program when it is working properly.



(b) State transitions for the Bitcount program when it has malware Type-1.



(c) State transitions for the Bitcount program when it has malware Type-2.



(d) State transitions for the Bitcount program when it has malware Type-3.

Figure 6. State transitions for the Bitcount program with the state diagram in Figure 3(c).

5. CONCLUSIONS

Program profiling is a commonly used technique for analyzing and optimizing the performance of the code. This paper presents a framework to externally profile systems and detect malware with zero-overhead by leveraging EM side emanations from the system. Monitoring is based on a Markov-model whose states are the heavily executed code sections, i.e., *hot paths*. Possible transitions between these states are assumed to be known apriori. Markov-model makes the state predictions based on the features extracted by the proposed *Feature Extraction Procedure* that consists of averaging Short-Time Fourier Transforms, identifying major frequencies, applying principle component analysis and projecting on lower dimensional subspaces. The parameters for the Markov-model and *Feature Extraction Procedure* are learned through a supervised learning that requires only one execution of each state. To evaluate the performance of the proposed framework we test the monitoring model on two different devices and three different programs and show that the proposed methodology can monitor the current state of the programs and detect all malware activities within the sensitivity of the trained model.

ACKNOWLEDGMENTS

This work has been supported, in part, by NSF grant 1563991 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 and DARPA.

REFERENCES

[1] Demme, J. and Sethumadhavan, S., "Rapid identification of architectural bottlenecks via precise event counting," in [*2011 38th Annual International Symposium on Computer Architecture (ISCA)*], 353–364, IEEE (2011).

[2] Moseley, T., Connors, D. A., Grunwald, D., and Peri, R., "Identifying potential parallelism via loop-centric profiling," in [*Proceedings of the 4th international conference on Computing frontiers*], 143–152 (2007).

[3] Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S.-T. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A., and Weihl, W. E., "Continuous profiling: Where have all the cycles gone?," *ACM Transactions on Computer Systems (TOCS)* **15**(4), 357–390 (1997).

[4] Graham, S. L., Kessler, P. B., and Mckusick, M. K., "Gprof: A call graph execution profiler," *ACM Sigplan Notices* **17**(6), 120–126 (1982).

[5] Debray, S. and Evans, W., "Profile-guided code compression," *ACM SIGPLAN Notices* **37**(5), 95–105 (2002).

[6] Han, W., Xue, J., Wang, Y., Liu, Z., and Kong, Z., "Malinsight: A systematic profiling based malware detection framework," *Journal of Network and Computer Applications* **125**, 236–250 (2019).

[7] Elish, K. O., Shu, X., Yao, D. D., Ryder, B. G., and Jiang, X., "Profiling user-trigger dependence for android malware detection," *Computers & Security* **49**, 255–273 (2015).

[8] Ball, T. and Larus, J. R., "Efficient path profiling," in [*Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*], 46–57, IEEE (1996).

[9] Joshi, R., Bond, M. D., and Zilles, C., "Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems," in [*Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*], 239, IEEE Computer Society (2004).

[10] Zhao, Q., Cutcutache, I., and Wong, W.-F., "Pipa: Pipelined profiling and analysis on multicore systems," *ACM Transactions on Architecture and Code Optimization (TACO)* **7**(3), 1–29 (2010).

[11] Wallace, S. and Hazelwood, K., "Superpin: Parallelizing dynamic instrumentation for real-time performance," in [*International Symposium on Code Generation and Optimization (CGO'07)*], 209–220, IEEE (2007).

[12] Sehatbakhsh, N., Nazari, A., Zajic, A., and Prvulovic, M., "Spectral profiling: Observer-effect-free profiling by monitoring em emanations," in [*The 49th Annual IEEE/ACM International Symposium on Microarchitecture*], 59, IEEE Press (2016).

[13] Callan, R., Behrang, F., Zajic, A., Prvulovic, M., and Orso, A., "Zero-overhead profiling via em emanations," in [*Proceedings of the 25th International Symposium on Software Testing and Analysis*], 401–412, ACM (2016).

[14] Nazari, A., Sehatbakhsh, N., Alam, M., Zajic, A., and Prvulovic, M., "Eddie: Em-based detection of deviations in program execution," in [*2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*], 333–346, IEEE (2017).

[15] Sehatbakhsh, N., Hong, H., Lazar, B., Johnson-Smith, B., Yilmaz, O., Alam, M., Nazari, A., Zajic, A., and Prvulovic, M., "Syndrome: Spectral analysis for anomaly detection on medical iot and embedded devicesexperimental demonstration," in [*Hardware Demo at IEEE International Symposium on Hardware Oriented Security and Trust (HOST*], (2017).

[16] Khan, H. A., Alam, M., Zajic, A., and Prvulovic, M., "Detailed tracking of program control flow using analog side-channel signals: a promise for iot malware detection and a threat for many cryptographic implementations," in [*Cyber Sensing 2018*], **10630**, 1063005, International Society for Optics and Photonics (2018).

[17] Zajic, A. and Prvulovic, M., "Experimental demonstration of electromagnetic information leakage from modern processor-memory systems," *IEEE Transactions on Electromagnetic Compatibility* **56**, 885–893 (Aug 2014).

[18] Agrawal, D., Archambeault, B., Rao, J. R., and Rohatgi, P., "The EM side-channel(s)," in [*Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*], 29–45 (2002).

[19] Gandolfi, K., Mourtel, C., and Olivier, F., "Electromagnetic analysis: Concrete results," in [*International workshop on cryptographic hardware and embedded systems*], 251–261, Springer (2001).

[20] Yilmaz, B. B., Sehatbakhsh, N., Prvulovic, M., and Zajic, A., "Communication model and capacity limits of covert channels created by software activities," *IEEE Transactions on Information Forensics and Security* **15**, 776–789 (2019).

[21] Yilmaz, B. B., Callan, R., Prvulovic, M., and Zajic, A., "Quantifying information leakage in a processor caused by the execution of instructions," in [*MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*], 255–260, IEEE (2017).

[22] Yilmaz, B. B., Callan, R., Zajic, A., and Prvulovic, M., "Capacity of the em covert/side-channel created by the execution of instructions in a processor," *IEEE Transactions on Information Forensics and Security* **13**(3), 605–620 (2018).

[23] Alam, M., Khan, H. A., Dey, M., Sinha, N., Callan, R. L., Zajic, A. G., and Prvulovic, M., "One&done: A single-decryption em-based attack on openssl's constant-time blinded RSA," in [*27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*], 585–602 (2018).

[24] Hayashi, Y.-i., Homma, N., Mizuki, T., Shimada, H., Aoki, T., Sone, H., Sauvage, L., and Danger, J.-L., "Efficient evaluation of em radiation associated with information leakage from cryptographic devices," *IEEE Transactions on Electromagnetic Compatibility* **55**(3), 555–563 (2012).

[25] Yilmaz, B. B., Ugurlu, E. M., Zajic, A., and Prvulovic, M., "Instruction level program tracking using electromagnetic emanations," in [*Cyber Sensing 2019*], **11011**, 110110H, International Society for Optics and Photonics (2019).

[26] Rutledge, R., Park, S., Khan, H., Orso, A., Prvulovic, M., and Zajic, A., "Zero-overhead path prediction with progressive symbolic execution," in [*Proceedings of the 41st International Conference on Software Engineering*], 234–245, IEEE Press (2019).

[27] Backes, M., Dürmuth, M., Gerling, S., Pinkal, M., and Sporleder, C., "Acoustic side-channel attacks on printers," in [*19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*], 307–322 (2010).

[28] Genkin, D., Shamir, A., and Tromer, E., "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 (2014).

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

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

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

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

[33] Hutter, M. and Schmidt, J.-M., "The temperature side channel and heating fault attacks," in [*Smart Card Research and Advanced Applications*], Francillon, A. and Rohatgi, P., eds., *Lecture Notes in Computer Science* **8419**, 219–235, Springer International Publishing (2014).

[34] Brouchier, J., Kean, T., Marsh, C., and Naccache, D., "Temperature attacks," *Security Privacy, IEEE* **7**, 79–82 (March 2009).

[35] Callan, R., Zajić, A., and Prvulovic, M., "Fase: finding amplitude-modulated side-channel emanations," in [*2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*], 592–603, IEEE (2015).

[36] Wold, S., Esbensen, K., and Geladi, P., "Principal component analysis," *Chemometrics and intelligent laboratory systems* **2**(1-3), 37–52 (1987).

[37] Yilmaz, B. B., Ugurlu, E. M., Prvulovic, M., and Zajić, A., "Detecting cellphone camera status at the distance by exploiting electromagnetic emanations," in [*Military Communications Conference (MILCOM), MILCOM 2019-2019 IEEE*], IEEE (2019).

[38] OlinuXino A13. https://www.olimex.com/Products/OLinuXino/A13/A13-OLinuXino/open-source-hardware.

[39] Alcatel Ideal. https://www.phonescoop.com/phones/phone.php?p=5097.

[40] Keysight Signal Analyzer. https://www.keysight.com/en/pdx-x202266-pn-N9020A/mxa-signal-analyzer-10-hz-to-265-ghz?pm=spc&nid=-32508.1150426&cc=US&lc=eng.

[41] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B., "Mibench: A free, commercially representative embedded benchmark suite," in [*Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538)*], 3–14, IEEE (2001).