Securing IoT Devices using Dynamic Power Management : Machine Learning Approach

Nikhil Chawla, *Student Member, IEEE,* Arvind Singh, *Student Member, IEEE,* Harshit Kumar, *Student Member, IEEE,* Monodeep Kar, *Member, IEEE,* and Saibal Mukhopadhyay, *Fellow, IEEE*

***Abstract*—The shift in paradigm from cloud computing towards edge has resulted in faster response times, more secure and energy efficient edge. IoT devices form a vital part of edge, but despite legions of benefits it offers, increasing vulnerabilities and escalation in malware generation have rendered them insecure. Software based approaches are prominent in malware detection, but they fail to meet the requirements for IoT devices. Dynamic Power Management(DPM) is an architecture agnostic and inherently pervasive component existing in all low-power IoT devices. In this paper, we demonstrate Dynamic Voltage and Frequency Scaling (DVFS) states form a signature pertinent to an application, and its run-time variations comprises of features essential for securing IoT devices against malware attacks. We have demonstrated this proof of concept by performing experimental analysis on Snapdragon 820 mobile processor, hosting Android operating system (OS). We developed a supervised machine learning model for application classification and malware identification by extracting features from the DVFS states time-series. The experimental results show $>$0.7 F1 score in classifying different android benchmarks and $>$0.88 in classifying benign and malware applications, when evaluated across different DVFS governors. We also performed power measurements under different governors to evaluate power-security aware governor. We have observed higher detection accuracy and lower power dissipation under settings of ondemand governor.**

***Index Terms*—Android OS, Application Inference, CPUFreq, Dynamic Voltage and Frequency Scaling, Energy Efficiency, Malware Detection, Machine Learning, Side-Channel,Snapdragon**

I. Introduction

Internet of Things (IoT) form an interconnected network of devices that finds usage in wide spectrum of applications like industry, healthcare, home, autonomous cars and smart grid [1] [2]. These devices generate, process, and communicate sensitive confidential data which is susceptible to attacks that invade users' privacy. Also, these devices are pervasive, possess weaker defenses and have widespread connectivity which has created a vast attack surface for adversary exploitation. For instance, a potentially malicious application can exploit vulnerabilities to gain access to the device and steal private information or infer activities through eavesdropping software events by hiding in the background.

Manuscript received January 16, 2020; revised May 19, 2020; revised July 20, 2020; accepted August 20, 2020. This work is supported by Semiconductor Research Corporation through Texas Analog Center of Excellence (#2810.002)

N. Chawla, H. Kumar and S. Mukhopadhyay are with the Department of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 30332 USA e-mail:(nchawla6@gatech.edu; hkumar64@gatech.edu; saibal.mukhopadhyay@ece.gatech.edu)

A. Singh is with the Rambus Inc, Sunnyvale, CA, 94089, USA (e-mail: arsingh@rambus.com)

M. Kar is with IBM T. J. Watson Research Center, New York, NY, USA (e-mail:monodeepkar@gmail.com)

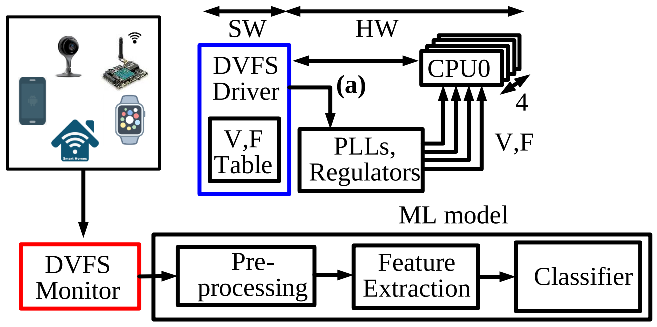


Fig. 1. (a) Dynamic voltage frequency scaling, hardware-software co-optimization approach for power management (b) A machine learning based malware detector using DVFS states features

The software-based malware detectors, particularly AntiVirus (AV) are vulnerable to software exploits and require consistent updates. They can get compromised as result of software exploits. Moreover, many AV software are power hungry and hence less suitable for energy constrained IoT devices. Alternatively, hardware-based malware detectors(HMD) have gained interest [3]–[5]. HMD can be implemented in hardware, remain isolated and cannot be easily tampered. [5]. They are also difficult to evade because malware leaves imprints as signatures in hardware, even though they remain undetectable in software.

HMD based on features derived from hardware events like performance counter have been shown to detect and contain malware in IoT devices [6]. HMD based on performance counters approaches have comparable detection accuracy with software based approaches, but these performance monitors vary from one kind of architecture to another [7] due to the variation in hardware organizations thereby requiring knowledge of the architecture. Moreover, limited number of HPCs are available on IoT based platforms which requires selection of relevant HPCs for efficient implementations [6]. More recently, there are demonstrations of malware detection by monitoring electromagnetic emission (EM) from IoT and embedded devices [8] [9]. But, detector based on EM emissions requires external side-channel measurements. To perform malware detection on IoT devices, it is beneficial to have a detector that is agnostic to hardware architecture and does not require external measurements.

In this paper, we explore power based hardware malware detector(P-HMD) for IoT devices. The power management is an essential hardware feature which is architecture agnostic and pervasive component common to broad spectrum of IoT devices, ranging from gateways, edge to resource constraint devices [10]–[12].In particular, we focus on power management signatures like, Dynamic Voltage and Frequency Scaling (DVFS) states for malware identification. DVFS is one of the drivers of energy management that dynamically configures the voltage and frequency in accordance to processor's workload. It is a hardware-software co-optimization technique implemented at CPU core level as shown in Fig.1(a). For instance, in linux based platforms, DVFS states are accessible through CPU frequency scaling framework.

A malware detector can be formed by generating a template of all legitimate applications based on their DVFS states variations during their execution and any application that deviates from set of legitimate templates can be labeled as malware or unknown. DVFS states contain coarse-grained information of multiple threads, executing in conjunction with profiled application. The background activity of these threads is random in nature and forms uncorrelated noise, which is inconvenient to model using template. Moreover, templates are effective approaches for well-understood devices and its complexity increases for modern system on chips [13]. To resolve this problem, ML techniques are an attractive alternative. Given a complex dataset, ML algorithms can learn targeted application specific DVFS features, given numerous training examples of application under different background noise conditions. In contrast to forming templates, machine learning based attacks are promising in black box settings, with only limited understanding of target implementation [14].

We demonstrate that ML models can extract application specific features from DVFS states variations in the presence of uncorrelated noise. Therefore, it can effectively characterize an application's behavior during its execution. We propose supervised machine learning models Support Vector Mchines (SVM), Random Forest (RF) and K-Nearest Neighbors (KNN) to learn features derived from DVFS states time-series of different applications. We further extend to demonstrate feasibility of a malware detector for IoT devices as shown in Fig.1(b). DVFS states of a processor are accessible through cpufreq framework in linux kernel. The decision to modulate the DVFS state of a CPU core is determined by cpufreq governors. Since, these governors show differences in power behavior of applications, therefore, we demonstrate that there exists a power-security aware design space to meet energy efficiency as well as provide security against malware. This paper makes the following key contributions

- We develop supervised machine learning (ML) based classification models to exploit the relationship between DVFS state variation with an application's behavior, thereby used for identifying applications running on a processor.
- We show that the developed models can be used to classify known applications and detect unknown applications
- We performed Welsh t-test analysis on DVFS signatures of different applications and showed distinguishable signatures.
- We demonstrate the extension of the proposed machine learning methodology in identifying a malware from benign application, showed comparison of P-HMD with prior works on HMDs, and present a framework for integration of P-HMD within an IoT device.
- We show that the optimal DVFS governor for high malware detection accuracy can reduce energy-efficiency during normal operations, creating an inherent trade-off between energy and security.
- We discuss the challenges on applicability and scalability of the DVFS based detector in real-world

In this paper, we demonstrated the proposed approach on Snapdragon 820 mobile platform. The experiments are performed on the Intrinsyc Development Kit Open Q 820 SOM, hosting Android OS, with underlying linux kernel.We used android benchmark dataset [15] for application inferencing and malware samples were collected from Drebin dataset [16]. These commercial platforms have different cpufreq governors, as part of linux kernel. Some of the prevalent cpufreq governors are ondemand, conservative and interactive. We have evaluated the application classification and malware detection accuracy using features derived from these cpufreq governors. A large majority of devices in IoT ecosystem support linux OS. According to the IoT developers survey 2018 and 2019, linux dominates as the choice of operating system for both gateways and edge devices [17] [18]. Therefore, we believe observations in this work will be applicable to a significant portion of IoT ecosystem.

The rest of the paper is organized as follows. Section II discusses background on dynamic power management in processors and related work on its interaction with security,Section III presents the characterization of DVFS traces, Section IV presents the proposed methodology and results for application classification and inferencing, Section V presents application of the proposed methodology to malware detection, Section VI presents the comparisons to related work and future directions and Section VII concludes the paper.

II. Background

A. Dynamic Voltage and Frequency Scaling (DVFS)

IoT/mobile processors incorporate various dynamic power management mechanisms. In particular, dynamic voltage and frequency scaling (DVFS) has become an integral part of modern processors to improve energy efficiency, increase battery life, and manage thermal effects. With DVFS, the supply voltage and operating frequency are scaled with respect to the varying workloads of the target system. Modern systems have multiple DVFS governors, that directs the rules for DVFS state of a core. A user can monitor the voltage-frequency point of a core (without requiring any privileged permission) and change the DVFS governor (superuser access). The DVFS governors have tunables like workload sampling interval, frequency upscaling, and downscaling factors etc. The governors assess CPU utilization by evaluating busy time of CPU to down-scale or up-scale frequency, which in turn is application dependent. Thus, frequency states of all cores are continuously updated by cpufreq driver in Linux kernel and no special privileges are required for user access. These DVFS states creates a software-based path to a side-channel information leakage.

B. CPUFreq Governors

The CPUFreq module in linux kernel is a combination of three layers: cpufreq core, scaling governors and scaling drivers. The core provides framework for all platforms that support cpufreq. The scaling governors implement algorithms to decide on DVFS state by estimating CPU capacity. The scaling drivers communicate with the hardware and are responsible for actual change in the DVFS state [19]. They access the platform-specific hardware interface to change the DVFS state. There are multiple cpufreq governors accessible through linux sysfs filesystem [20].

1) Ondemand: The governor estimates the CPU load using its own worker routine and computes the fraction of time CPU was not idle. The ratio of non-idle to the total CPU time is estimated as load. In multi-core platforms, governor is attached to multiple CPUs. Therefore, load is estimated for all of CPU's and highest is considered for load estimate. The CPU frequency values are proportional to the estimated load, therefore maximum load corresponds to highest frequency [21].

2) Conservative: This governor estimates the load similar to the ondemand governor described above. It does not change frequency proportionately as ondemand governor, but instead it modulates frequency, once the load exceeds the defined higher or lower threshold of CPU load.

3) Interactive: This governor also estimates the load similar to ondemand and conservative governor. But, it aggressively scales the CPU frequency under intensive activities. To accomplish this, it configures a timer and if the CPU is very busy after coming out of idle, during the timer interval, it immediately ramps the frequency to the highest value. If the CPU is not sufficiently busy, then the governor evaluates the load and changes the CPU frequency.

C. big.LITTLE Processors

The Linux Scheduler prioritizes scheduling of tasks by computing a nice value of all tasks in the run queue. In addition to this, there is load balancer for multi-core systems. Completely Fair Scheduler (CFS) in linux platforms uses per-entity load tracking (PELT) and in recent linux versions (window-based load tracking scheme) to evaluate task load. In addition to per task load, it keeps track of aggregate demand on a CPU, by accumulating demand from all the tasks. Snapdragon based platforms have big.LITTLE processors to maximize the energy efficiency. Heterogeneous multi processors(HMP) scheduler is aware of the difference in performance levels and the energy efficiencies of the big and the little cores, and therefore, is assigns the core based on performance demands of the process. In addition to this, HMP scheduler provides guidance to the cpufreq governors for changing the frequency,by tracking CPU utilization of all the cores [22].

D. Related Work

In recent years, there is a growing interest in using ML techniques and low-level hardware features for learning malicious behaviors. Authors have shown an application's behavior can be characterized using EM-emissions, to identify anomalous executions paths during program runtime [9] [23]. Hardware Performance Counters (HPCs) have been leveraged to characterize behavior using microarchitectural events like, caches misses, branch predictors, IPC etc. [3] [4]. J. Demme et.al., showed supervised machine learning methods to learn HPCs stats of different applications for classifying malware applications [3]. There is a growing interest in understanding the interactions of DVFS and security. The past studies have shown that DVFS can been used to create security threat. For example, Tang et. al. presented a CLKSCREW methodology, which performs unconstrained undervolting/ overclocking to inject faults to perform differential fault attack (DFA) for recovering key from Trustzone on ARM processors [24]. N Chawla et. al. demonstrated the feasibility of performing application inference using DVFS states [25]. On the other hand, there have been studies exploiting DVFS for enhancing security. For example, Yang et. al. studied the use of DVFS as a countermeasure to power side channel attack on encryption engines [26]. A. Singh et. al. has demonstrated that fast DVFS enabled by on-chip regulator and adaptive clocking helps inhibit power/EM-based side-channel attack and differential fault analysis attack [27].

Both profiling based and ML based techniques have been utilized for application inferencing. R. Spritzer et.al. demonstrated that selective information from process logs form a strong correlation with an application that is used to form templates and utilized dynamic time warping (DTW) for application inference with upto 96% accuracy in identifying 100 android applications [28]. In addition to inferring applications, software exposed information sources from linux file system have be shown to mount inference attacks. Using /proc/ <pid> /statm along with number of context switches, is shown to infer a visited webpage by a user [29] Similarly, size of the memory footprint of specific applications in /proc/ <pid> /statm is used to infer the user interface. More recently, activity transitions of an application are inferred using runtime memory statistics. [30] Moreover, identifying which application is running can help in launching specific attacks. For instance, an app running in the background to identify application which require login credentials, can execute phishing-based attacks [30] to steal login credentials.

III. Characterization of DVFS Traces

In this section, we discuss characteristics of DVFS traces that are used to develop the DVFS based application analysis framework. A subset of android benchmarks are used for analysis.

A. Measurement Setup

In order to demonstrate the proof of concept showcasing unique correlation between time varying DVFS state of different application, we generated an automated DVFS state acquisition environment. It comprises of Intrinsyc APQ 8096 Development Kit, which has an embedded Snapdragon 820 system on module(SoM) quad-core processor. It has 2 little and 2 big cores, with highest operating frequency of 2.2 GHz. It has 32 GB of internal storage and 3GB of RAM [Fig. 2]. The

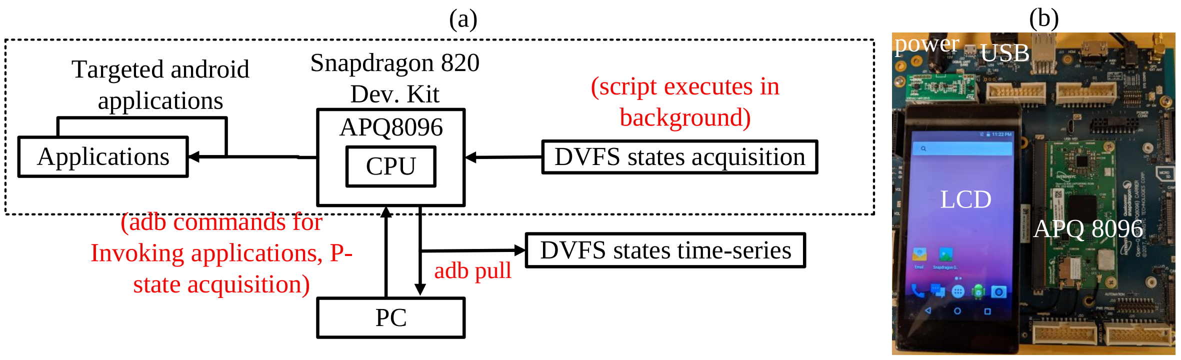


Fig. 2. Data acquisition: (a) A methodology for DVFS states trace acquisition. (b) Measurement Setup for DVFS states acquisition from Snapdragon 820 Intrinsyc APQ8096 Development Kit

Snapdragon host android Nougat 7.0 operating system. The development kit has LCD display screen for user interface. The setup only comprises methodology for DVFS state acquisition of different android applications. But, machine learning model training and validation is performed offline.

1) Evaluated Applications: The proof of concept is demonstrated by measuring DVFS states of android applications. Although evaluated android applications are benchmark applications [31], similar approach is generalized to any android application. The benchmark applications, namely whetstone, dhrystone, linpack and loops are CPU bound benchmarks. For instance, linpack benchmark performs floating point operation like addition and multiplication, busspeed and randmemi are memory bound benchmarks. Randmemi tests for data transfer speed from caches and memory. We also used some multi-threaded CPU performance benchmark in the dataset.

2) DVFS State Data Collection: The DVFS states information of a core is derived from cpufreq module in linux /sysfs file system. We generated a compiled C binary that reads the CPU operating frequency state, logs the timestamp and save these values to a file. This file runs in the background along with profiled application as shown in Fig. 2(a). The ARMv8 processor has 4 cores, two little and two big cores. The two little cores have shared power domain, and vice-versa for the other. The scheduler can assign the process to any core and the tasks can migrate between individual cores. Therefore, frequency states of both the cores are captured, when application executes on the processor. The scripts are programmed and executed simultaneously to capture the CPU frequency state of individual core.

3) Trigger, Log and Fetch: There is a separate script that executes on desktop. It manages the invocation of DVFS state acquisition script, initialization of android application activity, pseudo-random inputs for interacting with application, and storing trace files on mobile device as shown in Fig. 2. This is achieved through android debug (adb) interface shell commands. Each profiling android application's package name and activity name is extracted from apk dump and saved to this script running on PC. The pseudo random touch events for interaction with the application are simulated using monkey tool [32]. We set the percentage of system keys inputs to zero and gave 500ms delay between every input. The DVFS state trace collection time is set for 10 seconds for experiments..

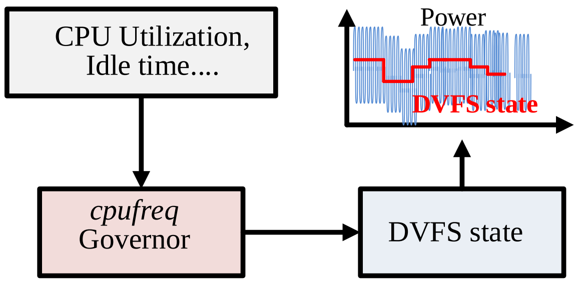


Fig. 3. Correlation between DVFS states and power dissipation

It can be tuned for different settings. Finally, the DVFS state trace is saved in the internal storage of mobile device.

B. Power Phases and DVFS Traces of Applications

The power behavior of an application is dependent on workload of applications executing on CPU core, contributing to CPU utilization. An application's workload behavior comprises of both low-power (low-utilization) and high-power (high-utilization) phases. The DVFS states of core are decided by cpufreq governors by monitoring core utilization over time interval. For example, a high activity on the core (due to increased workload) leads to a higher frequency DVFS state which would eventually increase power dissipation (Fig. 3). Therefore, any changes to utilization of a CPU core, which leads to dynamic variation in power patterns, is also reflected in the DVFS states of the core. In other words, the measured DVFS signature of an application reflects the fine-grain variations in the power phases of that application. To illustrate the inherent relation between power phases and DVFS traces, we have plotted the DVFS traces of various benign applications with ondemand governor configuration for 10s execution time (Fig. 4). The y-axis represents current operating frequency (GHz) of the CPU core and x-axis represents the time. We would like to highlight that each DVFS state transition or sequence of state transitions reflects different power behavior of an application during its execution and each application's overall DVFS signature comprises of both low-power and high-power phases as noted in the figure. We note that, the

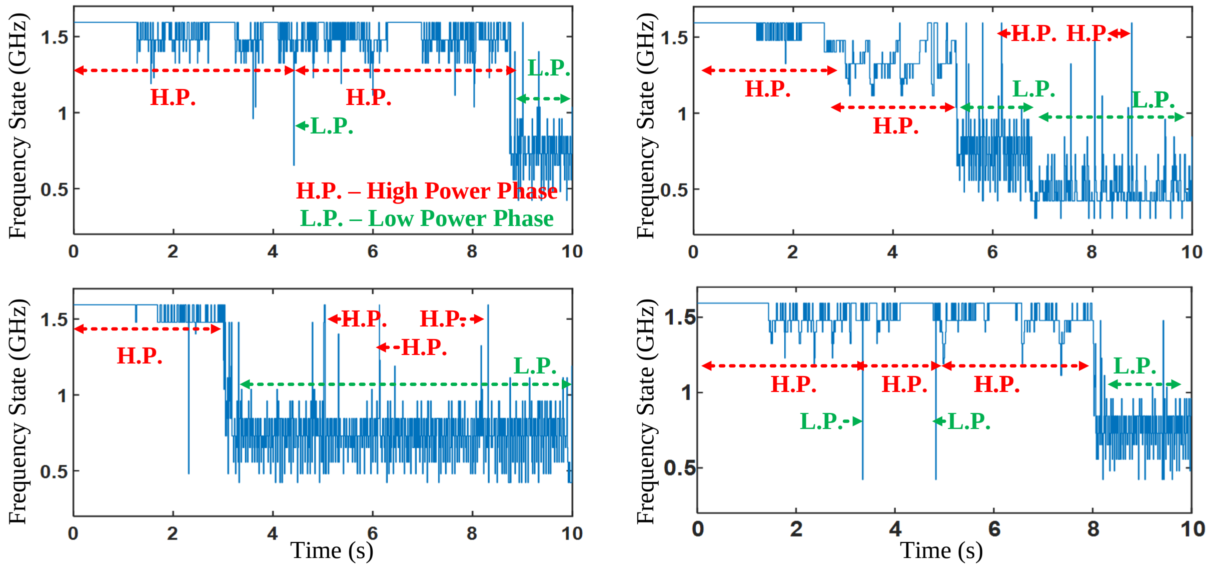


Fig. 4. Low Power (L.P.) and High Power (H.P.) phases of application execution highlighted in DVFS signatures of benchmark applications

notion of "power phase" can be related to average DVFS frequency state within a time-duration. However, the DVFS traces reflect fine-grain changes in power dissipations even within a phase thereby providing a more detailed signature for application inferencing and malware detections.

C. Uniqueness of DVFS Signatures

The workload characteristics of applications have unique correlation with DVFS states. We applied Welsh t-test to establish uniqueness in DVFS signatures of different applications.Welsh's t-test is a statistical measure to quantify similarity between samples drawn from two populations based on their means.Welsh t-test has applications in Test Vector Leakage Assessment (TVLA) analysis to quantify leakage of proposed countermeasure against power side-channel attacks [33]. In this particular case, the null hypothesis is "DVFS signatures of applications selected from different classes have similar means", therefore they are not distinguishable.If t-statistic $>$ 4.5 or t-statistic $<$ -4.5, then the null hypothesis is rejected with confidence score of 99.999% and p-val $<$ 0.00001 [34]

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}$$

Here, $\bar{X}$ is sample mean, s is standard deviation and n denotes sample size. We study the evidence of unique distinguishable features of different applications by obtaining Welsh t-test scores among all possible pairs of an application's raw DVFS signatures.

First, we perform pairwise t-test on DVFS signatures of different applications to establish that signatures are unique and distinguishable. We selected ondemand governor and corresponding DVFS signatures of benchmark applications. Altogether, we perform $^{14}C_2 - 14$ unique t-tests. We take multiple observations for DVFS traces of each application (100 traces) and perform a 2-sample t-test at each time point and obtain variation of t-statistic over time samples.We reject the null hypothesis if t-statistic at any time point crosses the threshold. The maximum t-statistic score computed over time samples for all possible combinations of t-test is summarized in Table I. The t-test results for all possible pairs of applications except (7,14) have maximum t-statistic greater than 4.5 or smaller than –4.5. This observation shows DVFS signatures of evaluated applications are distinguishable since they have distributions with different means

Second, we perform the t-test on DVFS signatures of same application collected over multiple iterations to establish they are non-distinguishable.We equally split the collected traces for an application into two groups. The traces belonging to a sub-group are selected randomly and size of each sub-group is 50. We again perform a 2-sample t-test at each time-point and observe the variation of t-statistic over time. We reject the null hypothesis if t-statistic at any time point crosses the threshold. The maximum t-statistic score computed over time samples for 14 possible t-test combinations is summarized in Table II. We can confirm from observations in Table II, that t-statistic is below the threshold and therefore it doesn't disproves the null hypothesis, implying the two populations have been selected from similar distributions with equal means.

The t-test analysis establishes that DVFS signatures of different applications are distinguishable. However, the t-test itself may not provide a direct approach for detection of individual applications. In following section, we will present feature extraction methods and machine learning (ML) techniques to detect applications based on DVFS signatures.

D. Impact of Governors

We study the leakage behavior of different DVFS governors. To perform this study, we characterized an application's behavior by fixing one of the possible DVFS state and observed the likelihood of the DVFS state at every time point. We took

TABLE I
MAXIMUM T-STATISTIC BETWEEN DVFS SIGNATURES FOR POSSIBLE PAIRS OF APPLICATIONS

App	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	N/A	-10.6	5.85	-12.7	-10.6	46.3	-10.2	19.3	59.4	54.6	-13.7	-10.9	110	-11.9
2		N/A	5.78	9.1	9.0	47.7	9.86	19.3	60.2	57.8	-10.2	8.33	111	10.1
3			N/A	9.18	7.9	14.1	9.4	19.3	19.1	13.2	-6.4	8.3	22.3	10.1
4				N/A	-8.8	46.4	9.49	19.3	61.3	57.4	-9.1	-8.5	108	10.1
5					N/A	45.7	4.98	14.7	58.7	53.5	-11.9	-5.93	109	6.04
6						N/A	-45.2	-44.1	18.8	12.9	-53.6	-48.8	87.8	-46.3
7							N/A	11.1	57.6	54.3	-9.8	-4.58	101	4.21
8								N/A	56	52	-19.3	-11.3	104	-10.8
9									N/A	-15.3	-66	-60	77.9	-60
10										N/A	-61.3	-57	81.5	-54.9
11											N/A	8.46	130	10.1
12												N/A	112	4.66
13													N/A	-109
14														N/A

TABLE II
MAXIMUM T-STATISTIC BETWEEN DVFS SIGNATURES OF SAME APPLICATION

App	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	3.36	3.25	2.62	3.39	3.59	3.51	4.33	3.24	4.82	4.19	3.79	4.37	3.74	3.80

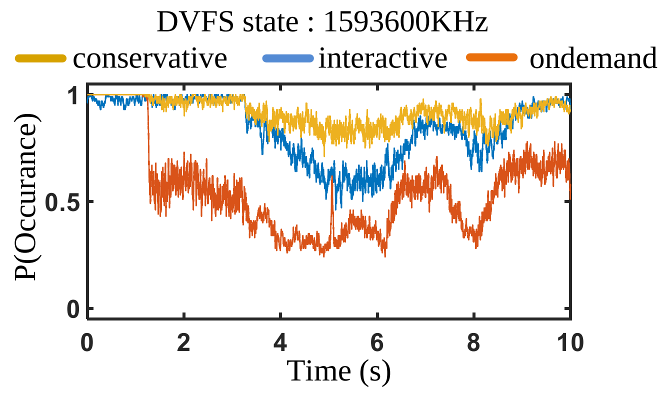


Fig. 5. Likelihood occurance of states 1593600KHz at every time sample for a benchmark application with all governors

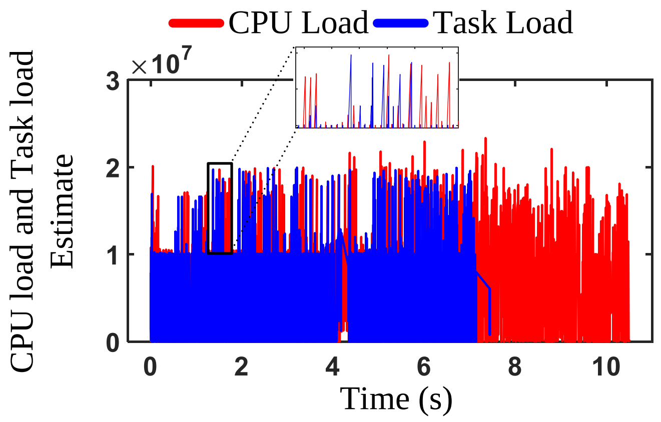


Fig. 6. CPU load and Task Load Estimate generated using ftrace tool to demonstrate a high correlation between CPU load and foreground process

100 instances of DVFS state time-series of the benchmark application, 'LinpackSP2' and calculated occurrence probability of that DVFS state [P(occurrence)] over 100 instances of the same trace. Fig. 5 shows the likelihood of occurrence of frequency state (1593 MHz) for an application with DVFS state signature generated from different CPUFreq governors. A higher likelihood of particular states across time, indicates less likelihood of its switching to different states. A value near 0.5 indicates, other states might have higher chances at those time-point. It can be concluded from Fig. 5, governors have different leakage behavior for the same application.

E. Foreground vs Background Applications

In smartphones, foreground activity of application depends on user inputs which dynamically varies CPU utilization of the task contributing to the CPU load. In response to the CPU load, cpufreq governor scales voltage and frequency states of the CPU. In smartphones, there are numerous system level threads and user applications simultaneously contending for resources. But maximum CPU load is contributed by application running in the foreground, which the user is currently interacting with, whereas the background user level applications may not be actively executing on CPU core. Moreover, smartphones prioritize performance of applications to enhance Quality of Service(QoS). To understand the impact of background processes on DVFS state signature of targeted application, we evaluated correlation between CPU load and targeted application task load. A strong correlation between them implies reduced influence of background tasks in DVFS state signature. This is demonstrated through a motivating example detailed below.

The CPU load and task load are monitored using HMP scheduler events using ftrace framework in Linux [35]. It has multiple trace point events for kernel functions. The sched_update_task_ravg is an HMP scheduler trace event which tracks updates on currently running tasks [22]. It has various task related observable parameters which provide task

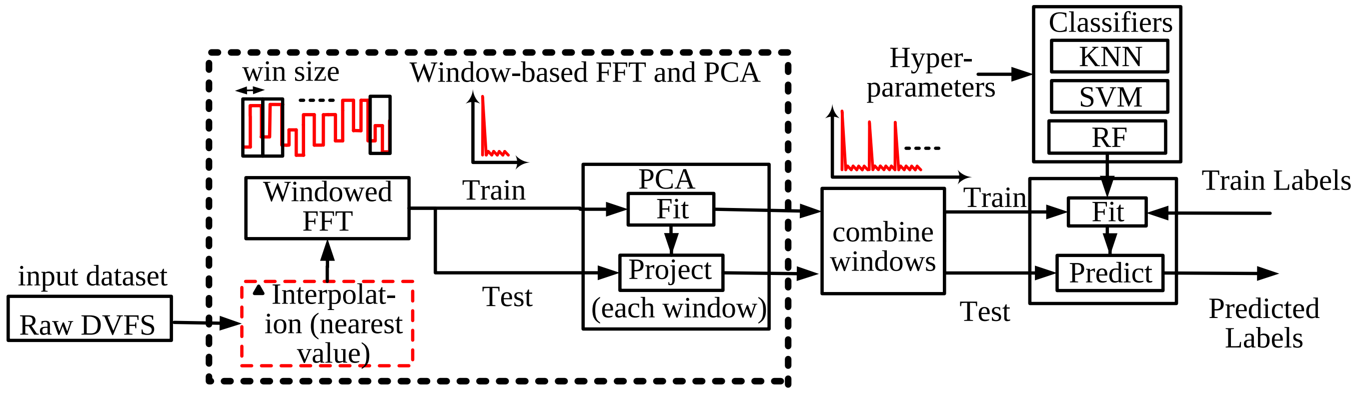


Fig. 7. The proposed methodology for application classification. Pre-processing and feature extraction performed on short sequences of DVFS states time-series. The feature vector is generated by combining windows and subsequently passed to classifier

specific information like PID, current CPU its running on(cpu), cpu frequency(cpu_freq), cpu utilization of task in current window (sum) and cumulative CPU demand of all the tasks in current window (curr_runnable_sum). For experimental evaluation, we initialized the targeted application, an android benchmark, 'LinpackSP2' [15] that executes for duration of approx. 7.5s in presence of system level threads running in the background and collected the sched_update_task_ravg trace events for duration of 10s. It comprises of all currently running processes, and we observe the task load(sum) for targeted application and cumulative CPU demand (curr_runnable_sum). Fig. 6 shows the trend of these two variables for a particular CPU core, on which targeted application was executing for majority time. The plot depicts a high correlation between task load and cumulative CPU load, suggesting that maximum load is contributed by currently executing application.

IV. Application Classification Methodology

In this section, we discuss the detailed methodology used in DVFS based application classification. Fig. 7 shows an overview of the proposed classification approach.

A. *Pre-processing and Feature Extraction*

The input dataset comprises of current operating frequency time-series of both the CPU cores during runtime of the android application. Instead of choosing the exact frequency states, we chose frequency indexes (0 to 29) to avoid need for normalization of data as some machine learning algorithms (e.g., support vector machines) do not perform well if the input data has large values. The entire time-series length is partitioned into non-overlapping smaller sequences of finite length, and Fast Fourier Transform (FFT) is performed on each of short sequences. The amplitude of the Fourier transform is used as features. This is followed by dimensionality reduction on FFT of individual short sequence using principal component analysis (PCA). The variance in the dataset is evaluated across multiple instances of same application under background noise conditions as well as different applications. We selected the components that meet 99% variance in individual short sequence window. The number of principal eigen vectors will vary with different governors. Finally, the reduced sequences are concatenated to form feature vector. The proposed methodology is shown in Fig. 7.

B. *Machine Learning Algorithms*

To learn program runtime characteristics from DVFS states, we chose supervised machine learning algorithms suitable for varying type of datasets. Both our training and testing datasets vary a lot in terms of dimensionality, number of observations and noise characteristics. The noise in turn depends on the selected DVFS governor, application that is being executed, the sampling speed at which the DVFS monitor is capturing the frequency states and essential system and user applications running in the background. Before, selecting ML algorithm for fitting the features derived from different governors, we first examined linear separability of features of different classes (applications). To evaluate linear separability, we selected a linear SVM kernel, and set regularization parameter C to very high value thereby forcing the optimizer to make 0 error in classification. We fit the training dataset and tested the accuracy on the same data after fitting. A 100% accuracy indicates zero training error and linearly separable features of different classes. We performed this analysis on features derived from different governors and obtained 100% accuracy on training dataset. The regularization parameter is chosen appropriately for actual classification task. We explored both linear and simple non-linear ML models for application inference and classification task.

Nearest Neighbors (KNN): KNN algorithm can classify datasets with linear or non-linear distributions. It selects k neighbors from the N-dimensional feature space. The neighbors are assigned weights based on distance metric (here Euclidean). K-NN performs well in low-dimensional feature space, with large number of observations which is the case with DVFS dataset.

Support Vector Machines (SVM): Like KNN, SVM can also classify linearly or non-linearly distributed datasets with a proper kernel (linear, poly, radial basis function RBF) It constructs a hyperplane to divide the feature space. The hyperplane is chosen such that it maximizes the margin between the features. For noisy as well as high dimensional data, SVM tends to outperform other ML algorithms. But drawback is increased training time.

Random Forest (RF): Random forest is an ensemble learning method, that uses results from multiple decision trees. Using ensemble of decision tress avoids overfitting problem. These commonly applied to multi-class classification problem. We haven't performed detailed hyper-parameter optimization but tested out classifiers performance by tuning the parameters

C. Classifier Performance Evaluation Metrics

The performance of machine learning model for classifying applications can be evaluated using different metrics like Precision, Recall, True Positives (TP). For, the purpose of application classification task, we have used a generic F1 score metric and receiver operating characteristic (ROC) curves. F1 scores is the harmonic mean of the precision and recall. The classification metric F1 score takes both False positives (FP) and False Negatives (FN). Precision gives the proportion of datapoints that were actual relevant, whereas recall gives the accuracy in classifying all relevant instances in dataset.

$$F1\ score = \frac{2 * Precision * Recall}{Precision + Recall}$$

$$Precision = \frac{TruePositives(TP)}{TruePositives + FalsePositives(FP)}$$

$$Recall = \frac{TruePositives(TP)}{TruePositives + FalseNegatives(FN)}$$

A receiver operating characteristic curve (ROC) gives a trend of True positive rate (TPR) with false positive rates (FPR). Higher TPR and low false alarming rate are desired for good performance of classifier. Area under curve (AUC) is indicative of performance of the classifiers. Higher area indicates high true positive rate at lower false positive rates.

We also evaluated generalization of the model using learning curves which evaluates bias and variance error. The correct predictions of model to new examples is achieved by reducing both these errors. A High bias indicates poor relation between features and target output(underfitting). High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs(overfitting). A low bias is indicated with lower training error and low variance is estimated with smaller gap between training and validation curve with increasing number of training examples.

D. Classification Results

We used several supervised machine learning algorithms to learn features derived from DVFS states time-series of different applications as discussed in Section III, and respective applications as their labels, thereby forming a classifier.

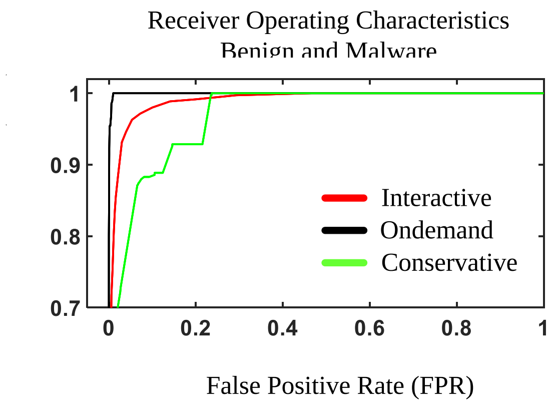


Fig. 8. ROC curves generated for benchmark application classification with RF inference model evaluated across all governors

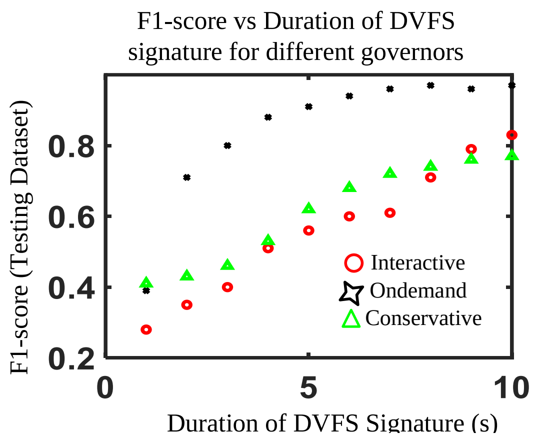


Fig. 9. F1-score vs Duration of DVFS signature for different governors generated using RF model

1) Training: The dataset comprises of 14 android benchmark applications. During training, each android application of the dataset is executed for a duration of 10 seconds in presence of default activities executing in the system. There are no constraints imposed on background scenarios during training. The training environment is reflective of true android mobile phone scenario created by end user. As described in Section III, user can start multiple android applications, but at particular time, user is interacting with one application in the foreground. The DVFS signature is reflective of cumulative activity of CPU core, but its dominated by foreground application. User interface (UI) is an essential component of android applications, and in order to collect DVFS states of android application, we need to provide interactive user inputs. To clarify, "user interaction" do not imply a physical interaction between user and a device. The interaction can be virtual, aided by specific tools [32]. For instance, fuzzer tools like monkey can simulate such pseudo-random UI events. Also, user input interactions are limited to smartphones and do not extend to most of IoT devices. Moreover, the threads can be scheduled on either core and as well might migrate between cores. Therefore, it is essential to capture frequency states from both cores. In addition to this, multiple instances of DVFS

TABLE III
F1 SCORES FOR APPLICATION INFERENCE WITH DIFFERENT GOVERNORS

CPUFeq Governor	KNN	SVM	RF
Interactive	0.7	0.79	0.83
Ondemand	0.84	0.95	0.97
Conservative	0.64	0.64	0.7

state traces are acquired for every profiled application. This ensures that variations arising due to dissimilar background conditions are considered. We fixed the number of training instances per application as 75 in our experiments. The number of samples in DVFS state time-series is 20k for individual core. The feature vectors generated after pre-processing DVFS state time-series of individual core as described in Section III are concatenated. There are 14 labels for each application.

2) Testing: We collected multiple DVFS state time-series traces of all the application learned by the model under different background conditions every time. We tested each application with 25 instances. The feature vector is generated by computing FFT on windowed sequence of time-series trace and projected to corresponding eigen vectors to form the reduced representation. Reduced feature representation of windowed sequence is concatenated to form feature vector. Finally, performance of the classifier is evaluated on the testing dataset.

3) Accuracy Analysis: Table III shows F1 scores for application classification using different supervised machine learning models and DVFS states features extracted from different cpufreq governors. The F1 scores are highest for random forest classifiers in comparison to SVM and KNN. On comparing classification accuracies with different ML models, KNN does not perform optimal fitting. It can be concluded it is no best algorithm when data is noisy, or features are not consistent. The trend in F1 scores across different cpufreq governors clearly shows the algorithmic differences of these governors. The F1-scores shows highest value for features extracted from ondemand governor. We also obtained ROC curves for RF inference model as shown in Fig. 8. A higher AUC (higher TPR for lower FPR) is observed for features derived with ondemand governor compared with interactive and conservative. These results indicates DVFS state variation of ondemand governor has more distinguishing features followed by interactive and conservative. The trends are in consensus with their algorithmic behavior as described in Section II.

The high detection accuracy on application classification summarized in Table III shows that ML model has extracted unique features of different applications and subsequently distinguish them with high confidence scores. The results reconfirms the evidence of unique distinguishable features of different applications as shown via Welsh t-test scores on raw DVFS signatures in Section III.C.

E. Discussions on Time-Series Length

In above application classification task, we kept the duration of DVFS signature fixed for 10s. It is essential to understand how much data is actually required to attain the F1-scores as shown in Table III. In general, longer duration of DVFS

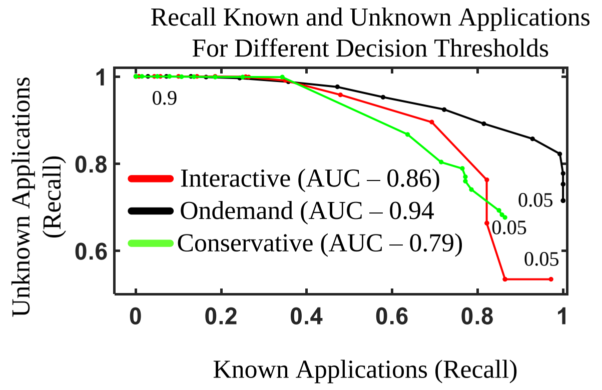


Fig. 10. Recall for known and Unknown application generated using trained RF model evaluated across different decision thresholds

signature will increase data collection, training and inference time. But, it will also improve the accuracy. Therefore, it is ultimately a trade-off between model accuracy and training time. To study the effect of time-series duration on model accuracy, we trained the model on different duration of time-series and observe the F1-score on testing dataset. The plot for depicting this trend is shown in Fig. 9. As expected, we can observe that test accuracy increases with longer time-duration of the DVFS signature. Interestingly, we observe that Certainly, features derived from interactive and conservative governor require more data for improving the classification score. On the other hand, features derived from on-demand governor can classify these 14 applications with 0.97 F1-score with 8s time-samples. Therefore, applications can be classified with 10s of data collection with on-demand governor; however, for higher accuracy using other governors we need to collect longer duration of data.

F. Identifying Unknown Applications

We evaluated the machine learning model on a dataset of applications whose features are not learnt by model. These are referred as unknown applications. We demonstrate that this trained model can be used to flag a potentially malicious or application not known to user. The trained model with known applications dataset is applied to a test dataset of some of the unknown applications. We used applications in MiBench suite [36] for the testing and measured 40 instances of each program. To detect the unknown program, pre-processing and feature extraction is performed, and then prediction probabilities are obtained from the machine learning model. We define a decision threshold based on probability to classify the unseen application in the unknown program class only when probability is less than decision threshold. We varied this decision threshold that subsequently introduced False negatives (FN) in identifying known applications. Therefore, there exists this trade-off between recall for known and unknown applications at different thresholds. A plot demonstrating this trend for different governors is shown in Fig. 10. The model generates higher probability of an example learned by model in comparison to application from unknown application. Therefore, at lower threshold, higher recall is observed for known

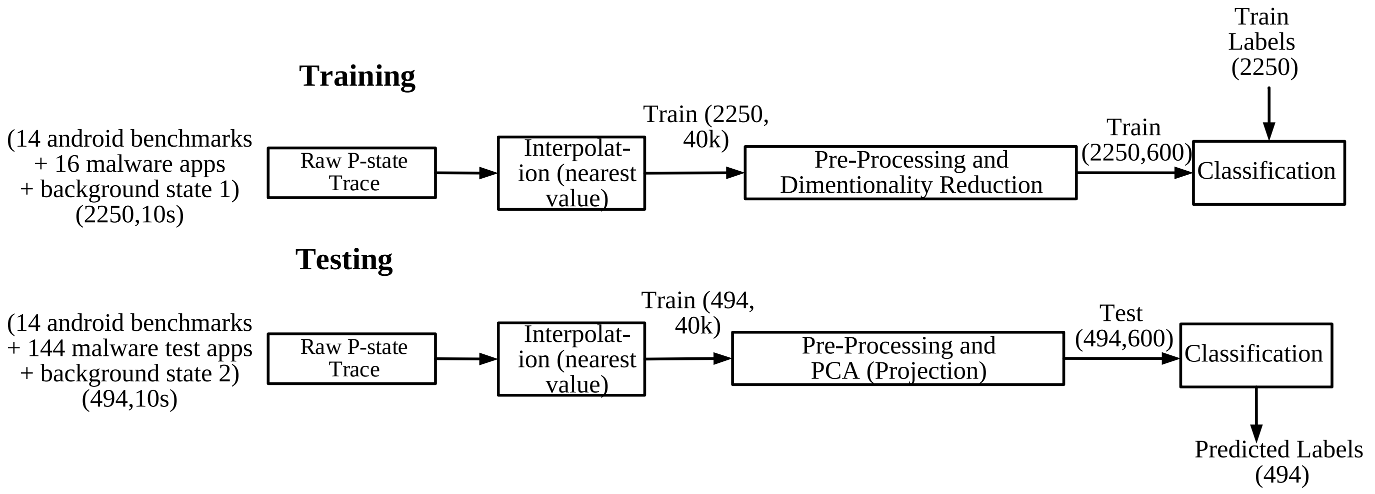

Training
(14 android benchmarks + 16 malware apps + background state 1) (2250,10s)
Raw P-state Trace
Interpolation (nearest value)
Train (2250, 40k)
Pre-Processing and Dimentionality Reduction
Train (2250,600)
Train Labels (2250)
Classification
Testing
(14 android benchmarks + 144 malware test apps + background state 2) (494,10s)
Raw P-state Trace
Interpolation (nearest value)
Train (494, 40k)
Pre-Processing and PCA (Projection)
Test (494,600)
Classification
Predicted Labels (494)

Fig. 11. Benign and Malware Classification training and testing approach derived from application inference methodology. The distinction of background states indicates different background conditions during training and testing.

applications and vice-versa. Higher AUC for training model derived from ondemand governor indicates more distinguishability between features of known and unknown application (Fig. 10).

V. Application to Malware Detection

There is growing need to protect edge devices against malwares. The existing software-based malware detectors performs detail malware analysis, but they are not scalable to edge devices. The static signature-based malware detectors match patterns of new malware samples within the existing databases for detection. These can easily be evaded by carefully crafting the malware code. In contrast, dynamic malware detectors analyze behavior of software and hardware parameters during execution of a malware. It overcomes the drawbacks of static approaches but can have high implementation overheads [37]. We extend application inference methodology detailed above to form a detector for red-flagging malicious applications. In our malware detector model, subsequent malware analysis is delegated to cloud-servers.

The surge of AI has diversified its applications in multiple domains including malware detection. There is tremendous research in utilizing machine learning to model malware based on features and even understanding their behavior. Malware behavior can be characterized using features at software and hardware abstraction. The sequence of system calls, application programming interface (APIs), network traffic in software and microarchitectural specific features like branch misses, cache hits etc. derived from system software are few examples [3] [4]. Side-channel leakage derived from power and Electromagnetic emissions (EM) manifests a program's behavior at instruction level granularity. For instance, authors in [9] have shown an application's behavior can be characterized using EM-emissions, to identify anomalous executions paths during programs runtime.

A. Machine Learning Model Generation and Validation

We extend the machine learning model used for application inference to malware identification. The DVFS state time series of a processor is correlated to the application(s) scheduled on the core. Hence, DVFS state transitions can capture differences between benign and malware applications. The experimental setup used for this analysis is described in Section III.A. The dataset comprises of packaged android malware application and set of benign applications

1) Dataset: The benign dataset comprises of android benchmark applications as described in Section III. It comprises 14 android apps. The malware samples are collected from Drebin dataset [16]. It comprises of 179 different malware families. Although benign applications are benchmarks, the methodology is generally applicable to android applications. In order to create a near balanced dataset, we selected only 16 android malware applications, each belonging to different family.

2) Training: Each malware application composition has benign as well as malware proportion. Training is not performed on malicious portion separately, but instead classifier sees both benign and malware codes. Therefore, the entire DVFS state time-series is labeled as malware. Different malwares have their own activation mechanism and payloads as described in [38] and It is well understood that their power behavior of these applications will be unique, but we are interested in runtime interactions of android applications with DVFS states of a processor. The dataset comprises of 14 benign benchmark applications and 16 malware applications. During training, each android application of the dataset is executed for a duration of 10 seconds in presence of default applications running in the system. As described earlier, there are no constraints imposed on background scenarios during training. In addition to this, multiple instances of DVFS state traces are acquired for every benign and malware application in the dataset. This ensures that variations arising due to dissimilar background conditions are considered. We fixed the number of training instances per application as 75 in our experiments. The

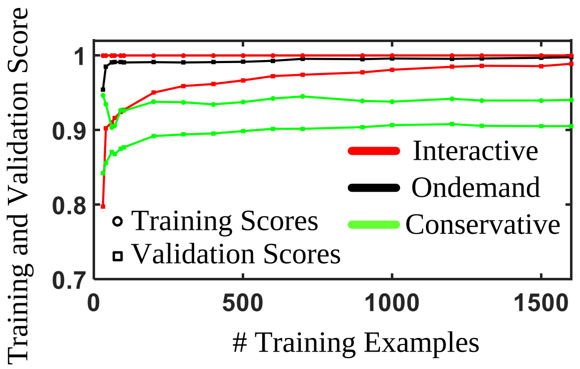


Fig. 12. Training and Validation Score against number of training examples (Learning Curves) generated with RF model for features dervied from different governors.

TABLE IV
F1 SCORES FOR BENIGN AND MALWARE CLASSIFICATION WITH DIFFERENT GOVERNORS

cpufreq governor	KNN	SVM	RF
Interactive	0.88	0.78	0.78
Ondemand	1	0.72	0.99
Conservative	0.57	0.98	0.97

training methodology for benign and malware classification is shown in Fig. 11.

3) Testing: The testing dataset comprises of new samples of android malware, not trained on the model. Since, it is imperative to correctly predict malwares compared to benign. Therefore, goal is to obtain low false negatives (FN) and comparatively lower False Positives (FP). The trained supervised machine learning model is tested on dataset comprising of 144 different malware applications which are variants of malware applications used in training. These belong to the same malware family, on which the model is trained, but it's a variant of those malware applications. It comprises of 14 android benchmarks applications measured under different background conditions, for 10s duration. We measured 25 instances each of these benign applications. The testing methodology for benign and malware classification is shown in Fig. 11.

TABLE V
CONFUSION MATRIX GENERATED FROM RANDOM FOREST CLASSIFIER FOR DIFFERENT GOVERNORS

cpufreq governor	# TP	# FP	# TN	# FN
Interactive	41	6	344	103
Ondemand	142	0	350	2
Conservative	144	15	335	0

B. Classification Results

Since RF classifier shows relatively higher F1 score, we obtained the learning curves for RF trained model with features derived from different governors as shown in Fig. 12. The training score attains highest value at lower number of training examples for ondemand and interactive governor compared to conservative. It indicates a low bias in the trained

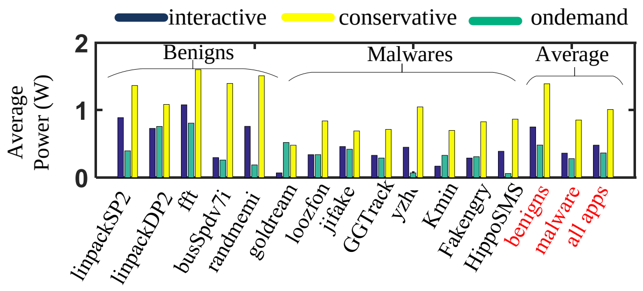


Fig. 13. Average Power for benign and malware applications with different cpufreq governors

model. Similarly, an increase in validation scores with number of training examples and attaining close to training score indicates low-variance and a good generalizability of model for new examples in case of ondemand and interactive governor. In contrast, validation score does not improve with increasing number of training examples for conservative governor, it also shows comparitely higher variance, and poor generalizability of model.

Table IV shows the F1-scores on testing dataset evaluated with different classifiers and DVFS states features generated from different governors. The F1 scores are highest for RF classifiers in comparison to SVM and KNN. On comparing classification accuracies with different ML models, we can conclude, that KNN does not perform optimal fitting. These results indicate the model is generalizable to variants of existing malware samples learned during training. The performance of classifier across features derived from different cpufreq governors is evaluated using #False Positives and #False Negative. Table V shows the confusion matrix values with RF classifier. FP indicates the number of benign samples being labeled as benign and vice-versa for FN. The combined FP and FN scores for ondemand is lowest followed by conservative and ondemand. These results also indicate features derived from ondemand governor have more distinguishability compared to other governors. It also indicates more information harnessed from ondemand governor.

We also analyzed the effect of time-series length on testing accuracy for malware detection task as described in Section IV.E We observed that 10s of data had sufficient features for benign against malware classification.

C. Power and Security

The power consumption of android applications changes with different cpufreq governors and consequently malware identification accuracy as shown in Table IV and Table V. The tradeoffs between average power consumption of different applications and malware identification accuracy are evaluated for different governors. We measured average power consumption of android benchmark and subset of malware applications (Fig. 13). Power is measured for application runtime duration of 10s by computing current drawn (averaged over 10 measurements) from 3.8V buck converter supplying current to the Snapdragon processor using a 5m on-board resistor. The average power consumption accumulated across all benign

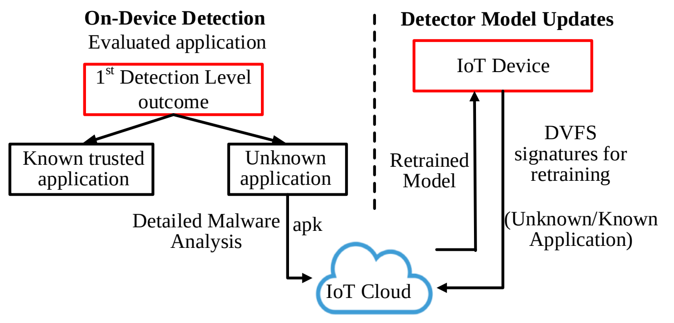

On-Device Detection
Evaluated application
1st Detection Level outcome
Known trusted application
Unknown application
Detailed Malware Analysis
apk
IoT Cloud
Detector Model Updates
IoT Device
Retrained Model
DVFS signatures for retraining
(Unknown/Known Application)

Fig. 14. On-Device DVFS based detection for red-flagging malware and detailed malware analysis and model updates on IoT cloud servers

apps is higher compared to malware apps for all governors. Enabling conservative governor dissipates highest power for benign and malware applications. The power consumption of applications under ondemand governor is least when averaged across all applications. Moreover, features extracted from ondemand governor also shows higher application identification scores (0.97) and lower number of False Positives and False Negatives for malware identification. In summary, applications dissipate relatively less power and shows higher identification accuracy with ondemand governor compared to interactive and conservative.

D. Secure Detector System

The realization of detector encompasses multiple design choices depending on software or hardware. The hardware based implementations may include machine learning inference engines as accelerators, or dedicated core etc. In software, machine learning inference can be executed at kernel or flashed in firmware. Allocating higher privileges to detector creates more secure detection framework. First, it will not only be useful for monitoring malicious activity at application level, but they can potentially detect malware exploits at kernel level. A hardware centric approach to detect rootkits using hardware performance counters is demonstrated in [39]. Second, the detector won't fail if system level software gets compromised. More recently, more secure feature of trusted execution environments (TEE), like Trustzone are extended to IoT based platforms. Implementing such detector will ensure that underlying detector code and data is protected from tampering or adversarial attacks. The specific tasks performed by proposed detector are described below

1) Data Acquisition: The data acquisition step involves polling values of DVFS states. In linux based platforms, DVFS states can be accessed through CPU Frequency scaling infrastructure. The current frequency state of all CPU cores are sampled periodically for a specified duration. In our proposed DVFS based detector data is collected for 10s duration.The polling of DVFS states from cpufreq module introduces noise in the profiled application's DVFS signature. We have taken this into consideration during training by collecting multiple instances of same application

2) Data Analysis: The DVFS signature of application is analyzed by detector (ML inference model) as shown in Fig. 14. The detector determines if DVFS signature belongs to set of trusted applications (known) else it is detected as unknown. A new application(malware or benign) will be first be detected as unknown application. A detected unknown application will be classified as "malicious" by default and flagged for further analysis at the cloud servers to determine whether the application can be trusted.

3) Threat Response: Once the application is redflagged, it is deleted from the system and more detailed malware analysis is performed to scrutinize its behavior. Detailed malware analysis is both time and resource consuming task and it cannot be efficiently performed on-device. It can be analyzed on cloud servers by sending detected application. If the application is identified as threat after detailed analysis, IoT devices should be notified about updating ML model with DVFS signatures of the detected "malware" application.

4) Model Updates: ML Model updates are required under multiple scenarios. An application's power signature can change on software updates. Also, the proposed detector is trained on limited malware and trusted applications. Therefore, model would also require updates about new malware applications and new trusted applications to be included in the existing model. Considering our current approach, an update would involve retraining the machine learning model with pre-existing data and new data and finding the best fit model using hyper-parameter optimization. In our current approach, any update or a new application (trusted or un-trusted) will first be detected as an unknown application. For a software update or a new trusted application, the detected application can be mapped as "known". This involves steps like, generate a trigger to enable collection of the DVFS signatures, send the collected DVFS signature to the cloud (or gateway) for re-building of the model, and receive the new ML model from cloud once re-training is performed.

We believe that the retraining is better performed at cloud or at the gateway depending on the available processing power.The primary overhead for the edge device will be data collection and data transfer associated with the model creation.The upload data volume for unknown application is estimated around 23MB [Samples per trace (40K) * Training Examples (75) * size(float64)]. The download data volume from cloud servers comprising of retrained model is estimated around 10MB.[Apps (31) * Training Examples (75) * Feature Vector Length (600) * size(float64)]. The model fitting and optimization will be performed at the cloud, and new ML models can essentially be deployed to all edge devices that are based on same OS. In some sense, this will be similar to "security update" of the operating system or kernel that runs the DVFS-based detector.

VI. Discussion

A. Dealing with Software Update

In this section we discuss the impacts of software updates on our ML model for application classification and how to deal with such updates using the detection system discussed in Section V.D. We study an example by analyzing three versions of "SubwaySurfers" android application (v.2.2.0, v.2.1.2 and

TABLE VI
RECALL KNOWN VS UNKNOWN VERSIONS OF SUBWAYSURFERS APPLICATION AT DIFFERENT DECISION THRESHOLDS

Recall - RF model	Thresh=0.75	Thresh=0.9
SubwaySurfers v2.2.0 (known)	1	1
SubwaySurfers v2.1.2 (unknown - minor update)	0	0.47
SubwaySurfers v1.116.1 (unknown - major update)	1	1

v.1.116.1). The version v.2.2.0 is minor update from v.2.1.2 and it is a major update from v.1.116.1. We followed DVFS signature data collection process described in Section IV. We used same sequence of pseudo-random inputs for interacting with all three versions of the applications. The DVFS signatures are analyzed in segments of 10s. The RF classifier model and ondemand governor are used for analysis. We first trained the ver. 2.2.0 in our existing model as a trusted application and use other versions to represent application updates. We followed the methodology described in Section IV.F for obtaining recall for unknown applications (v.2.1.0 and v.1.116.1). We compare the confidence score with pre-defined decision threshold and compute the accuracy of known (v.2.2.0) and unknown (v.2.1.2 and v.1.116.1) for different values of decision thresholds. The Recall for known and unknown applications at different decision thresholds is tabulated in Table VI. As expected, the version v.2.2.0, which was part of training dataset, is classified as a known application with a recall score of '1' even for very high decision thresholds. We next discuss the cases for minor and major software updates.

1) Impact of a Minor Software Update: . It is important to note that that every application update may not result in appreciable change in DVFS signatures. To demonstrate this, we considered classifying the version v.2.1.2 as an "unknown" application using a RF model trained with v.2.2.0. However, v.2.1.2 shows a low recall for classifying as an "unknown" application even at high decision threshold of 0.9. This is consistent with the observation that as v.2.2.0 is minor update to v.2.1.2 (executable binary size differs by 1MB only), the DVFS signatures may not be significantly different. In this example, the ML model may not need to be updated (or retrained).

2) Impact of a Major Software Update: . We now consider the case of a major software update. We start with the original model trained with version v.2.2.0 of "SubwaySurfers" and test with the version v.1.116.1. The version v.1.116.1 shows a recall score of '1' for being detected as an "unknown application" as the confidence score of v.1.116.1 being detected as "SubwaySurfer" (version v2.2.0) is much lower than 0.5. This is consistent with our observation that v.1.116.1 to V2.2.0 was a major update (37MB). Once the version v.1.116.1 is detected as an "unknown" application, the next step is to update the ML model by re-training with new DVFS signatures of updated version (v1.116.1) of the application "SubwaySurfer". We followed the training methodology described in Section IV. After retraining, we obtain the F1-scores of 100% for correctly classifying the updated version as "SubwaySurfer". The analysis shows that a major software update can be incorporated by re-training the ML model.

B. Comparison with Related Works

There have been prior works that leverages machine learning methods using features derived from software, hardware or combined for malware detection [3] [6] [8] [4]. But, no prior methods have focused on feasibility of DVFS based power signatures malware analysis and subsequent detection. We compare the proposed DVFS based detector against hardware feature-based machine learning techniques, like performance counters, power and EM side-channel etc. To show a fair comparison, we selected literature that explores signature-based approach for malware detection. The summary of the comparison is described in Table VII. We compare the detectors based on machine learning performance metrics, like precision, recall, AUC etc, and their feature complexity to access time to detection (inference time).

The performance of DVFS based detection is comparable to other hardware-based detection techniques as tabulated in Table VII. Note, datasets used for different detectors are not identical. For instance, J.Demme et.al. have evaluated the accuracy on 37 malware families [3]. S.M.P Dinakarrao et.al. have considered malwares belonging to 4 categories, namely, backdoors, trojan, virus, rootkits in [6]. S.S. Clark et.al. targeted specific malwares for medical devices [8]. The complexity of detector and time to detection can be evaluated using feature dimensions. J.Demme et.al. have demonstrated detector with 368M performance counter samples of malware and non-malware in ML model [3]. S.M.P Dinakarrao et.al. have shown reduced feature complexity of OneR classifier detection model for IoT devices using 2 HPC [6]. This would have similarities to DVFS based detector which relies on DVFS states from 2 cores. The exact comparison on feature complexity also depends on length of data collection, which indeed is different. These key insights derived from these comparisons are DVFS based malware identification is much simpler and has smaller footprint due to its low feature complexity.

C. Challenges

Practical implementations of the proposed detector requires addressing challenges like, scalability of the detector with increasing data, number of applications and type of platforms. Some of these challenges on applicability and scalability of the detector in real-world scenarios are discussed here.

1) Extension to other platforms: The experimental analysis is performed on an Android platform, but we have only used CPU Frequency scaling feature of the Linux OS(no Android-specific functions). Linux is a prevalent OS in many devices in the IoT ecosystem (gateway, edge, and resource constraint devices). According to the IoT developers survey 2018, linux dominates in the choice of operating system for both gateways and resource constraint devices [17]. In IoT developers survey 2019, linux is still dominant OS for gateways and edge nodes. But, FreeRTOS has gained more popularity for resource constraint devices [18]. These surveys suggest Linux is still a very important platform for IoT devices.

TABLE VII
COMPARISON OF PROPOSED WORK WITH EXISTING HARDWARE BASED MALWARE DETECTION APPROACHES

Reference	Information	Platform	Performance	Features	Detection Algorithm
[3]	Performance Counters (HPC)	OMAP4460, ARM Cortex-A9, Android 4.1	83% AUC for 10% False Positive Rate (FPR)	368M samples	Decision Trees, ANN
[8]	Power Signal (AC)	Embedded Medical Device	Detects known malware with 94% accuracy and unknown with 85%	8 time-domain features (statistical), spectral	RF, 3-NN, Perceptron
[6]	Performance Counters (HPC)	Intel ATLASEDGE, ARM processor	93% detection accuracy	2, 4, 8 HPCs	MLP, OneR, JRip
Proposed	DVFS	Snapdragon 820, ARMv8 processor, Android 7.0	99% detection accuracy (malwares variants)	DVFS states (2 CPU cores)	SVM, RF, K-NN

The platforms based on linux kernel have CPU frequency scaling infrastructure like cpufreq, cpuidle. In this paper, we collect DVFS states from cpufreq module in linux kernel. Therefore, we believe the observations made in this work will be applicable to a significant portion of IoT ecosystem. The power management is essentially a hardware feature which is common to most of the resource-constraint SoCs/processors [10]–[12]; although, the exact mechanism to access the power management signatures can vary across operating system. In other words, the basic observations made in the paper is applicable to most of the modern processors/SoCs.

2) ***Algorithm Scalability****:* A new application (benign or malware) would have a unique DVFS behavior and it would eventually lead to unique features. At the best case, if features of new applications exhibit similarity with existing benign or malware class, then the current model may perform well. If the signature of the new application is different, the hyperparamaters of the current model must be re-tuned (as discussed above) to generate a new version of the current ML model. At the extreme case, as the numbers of applications increase the non-linearity in the feature space is likely to grow as well making it difficult to classify the applications using simpler ML models discussed in the paper; for example, an SVM model with linear kernel won't be able to find hyperplane that separate classes. In such cases, a non-linear machine learning algorithm like RF, or ultimately, deep learning based approach, would be more scalable than linear algorithms like SVM. The future work on this topic would consider use of such complex non-linear ML algorithms like deep neural networks.

3) ***Time-Series Length****:* The time-duration of the data collection during training and inference will impact classification/detection accuracy. In principle, the time-series of DVFS states of individual application should be collected over long enough duration to ensure DVFS states are monitored for different phases of application execution. We expect a longer time-series of DVFS data will increase overall detection accuracy. On the other hand, in a practical detector, a longer data collection will increase training and inference time. Therefore, it is ultimately a trade-off between model accuracy and training time. Also, it is important to note that length of the time-series must be longer than run-time of the application. We have shown in Section IV.D that 10s of data had sufficient features for malwares and benign classification. From above observations, we can say that 10s of data was sufficient for classification task in our example using on-demand governor. However, longer duration of data collection will generally improve the accuracy and a single application should be exhaustively trained for its different execution phases.

4) ***Combining Features****:* The robustness of malware detection can be improved by data fusion with other sources of leakage in hardware, like performance counters. Combining features from lower-level stacks (hardware, firmware) and implementing detector in hardware gives higher privileges and anti-tampering advantages. Future work will also explore feasibility of power based detectors in learning common features of different malware families.

VII. CONCLUSION

This work experimentally demonstrated identification of applications executing on Android platform by deriving features from power signatures like DVFS states of a processor's core and using supervised machine learning model to establish their subsequent correlation. The F1 scores of application classification is > 0.7 evaluated across different machine learning models and cpufreq governors. We also showed identification of unknown applications using the learned model based on known applications. The results show > 0.79 AUC on recall curves for known and unknown applications evaluated with RF model and all cpufreq governors. We showed an extension of the proposed machine learning methodology of application identification to malware identification. We obtained > 0.88 F1 score using RF based classification between benign and malware applications evaluated across all governors. We explored power secure awareness of selecting cpufreq governor by evaluating power consumption of different benchmark and malware applications. We observe applications dissipate less power on an average with ondemand governor configuration and we obtain higher identification accuracy.We demonstrated a framework for practical malware detection system based on power signatures like DVFS states with on-device identification and cloud-based software updates features.We discussed pitfalls and challenges associated with proposed detection system.

REFERENCES

[1] A. Zanella *et al.*, "Internet of things for smart cities," *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 22–32, Feb 2014.

[2] F. Conti *et al.*, "An iot endpoint system-on-chip for secure and energy-efficient near-sensor analytics," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 9, pp. 2481–2494, Sep. 2017.

[3] J. Demme *et al.*, “On the feasibility of online malware detection with performance counters,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA ’13. New York, NY, USA: ACM, 2013, pp. 559–570. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485970

[4] K. N. Khasawneh *et al.*, “Ensemble learning for low-level hardware-supported malware detection,” in *Research in Attacks, Intrusions, and Defenses*, H. Bos *et al.*, Eds., Cham, 2015, pp. 3–25.

[5] M. Ozsoy *et al.*, “Malware-aware processors: A framework for efficient online malware detection,” in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 651–661.

[6] S. M. Pudukotai Dinakarrao *et al.*, “Lightweight node-level malware detection and network-level malware confinement in iot networks,” in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 776–781.

[7] N. Patel *et al.*, “Analyzing hardware based malware detectors,” in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017, pp. 1–6.

[8] S. S. Clark *et al.*, “Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices,” in *Proceedings of the 2013 USENIX Conference on Safety, Security, Privacy and Interoperability of Health Information Technologies*, ser. HealthTech’13. Berkeley, CA, USA: USENIX Association, 2013, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2696523.2696532

[9] A. Nazari *et al.*, “Eddie: Em-based detection of deviations in program execution,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA ’17, 2017, pp. 333–346.

[10] Intel® edison development platform. [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/edison/sb/edison_pb_331179002.pdf

[11] Pelion iot platform. [Online]. Available: https://www.arm.com/products/iot/pelion-iot-platform

[12] G. Halfacree *et al.*, *Raspberry Pi User Guide*, 1st ed. Wiley Publishing, 2012.

[13] L. Lerman *et al.*, “Template attacks vs. machine learning revisited and the curse of dimensionality in side-channel analysis,” in *Revised Selected Papers of the 6th International Workshop on Constructive Side-Channel Analysis and Secure Design - Volume 9064*, ser. COSADE 2015. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 20–33.

[14] S. Picek *et al.*, “Side-channel analysis and machine learning: A practical perspective,” in *2017 International Joint Conference on Neural Networks (IJCNN)*, May 2017, pp. 4095–4102.

[15] R. Longbottom, “Updated android benchmarks for 32 bit and 64 bit cpus from arm and intel contents,” 03 2018.

[16] D. Arp *et al.*, “Drebin: Effective and explainable detection of android malware in your pocket,” in *NDSS*, 2014.

[17] Iot developer survey 2018. [Online]. Available: https://www.slideshare.net/Eclipse-IoT/iot-developer-survey-2018-130998751

[18] Iot developer survey 2019. [Online]. Available: https://outreach.eclipse.foundation/download-the-eclipse-iot-developer-survey-results

[19] “MS Windows NT kernel description,” https://www.kernel.org/doc/html/v4.15/admin-guide/pm/cpufreq.html, accessed: 2010-09-30.

[20] Linux cpufreq governors. [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

[21] V. Pallipadi *et al.*, “The ondemand governor: past, present and future,” in *Proceedings of Linux Symposium, vol. 2, pp. 223-238*, 2006.

[22] Scheduler extensions. [Online]. Available: https://android.googlesource.com/kernel/msm/+/android-msm-marlin-3.18-nougat-dr1/Documentation/scheduler/sched-zone.txt

[23] Y. Han *et al.*, “Watch me, but don’t touch me! contactless control flow monitoring via electromagnetic emanations,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’17, 2017, pp. 1095–1108.

[24] A. Tang *et al.*, “Clkscrew: Exposing the perils of security-oblivious energy management,” in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC’17, 2017, pp. 1057–1074.

[25] N. Chawla *et al.*, “Application inference using machine learning based side channel analysis,” in *2019 International Joint Conference on Neural Networks (IJCNN)*, July 2019, pp. 1–8.

[26] S. Yang *et al.*, “Power attack resistant cryptosystem design: a dynamic voltage and frequency switching approach,” in *Design, Automation and Test in Europe*, March 2005, pp. 64–69 Vol. 3.

[27] A. Singh *et al.*, “Exploiting on-chip power management for side-channel security,” in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 401–406.

[28] R. Spreitzer *et al.*, “Procharvester: Fully automated analysis of procfs side-channel leaks on android,” in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ser. ASIACCS ’18. New York, NY, USA: ACM, 2018, pp. 749–763. [Online]. Available: http://doi.acm.org/10.1145/3196494.3196510

[29] S. Jana *et al.*, “Memento: Learning secrets from process footprints,” in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 143–157.

[30] Q. A. Chen *et al.*, “Peeking into your app without actually seeing it: Ui state inference and novel android attacks,” in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC’14. Berkeley, CA, USA: USENIX Association, 2014, pp. 1037–1052. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671291

[31] R. Callan *et al.*, “A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 242–254.

[32] Ui/application exerciser monkey. [Online]. Available: https://developer.android.com/studio/test/monkey

[33] G. Goodwill *et al.*, “A testing methodology for side channel resistance,” 2011.

[34] T. Schneider *et al.*, “Leakage assessment methodology,” in *Cryptographic Hardware and Embedded Systems – CHES 2015*, T. Güneysu *et al.*, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 495–513.

[35] S. Rostedt. (2008) ftrace - function tracer. [Online]. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[36] M. R. Guthaus *et al.*, “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)*, Dec 2001, pp. 3–14.

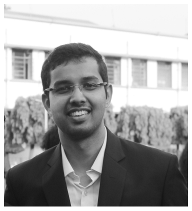
[37] C. Kolbitsch *et al.*, “Effective and efficient malware detection at the end host,” in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM’09, 2009, pp. 351–366.

[38] Y. Zhou *et al.*, “Dissecting android malware: Characterization and evolution,” in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 95–109.

[39] B. Singh *et al.*, “On the detection of kernel-level rootkits using hardware performance counters,” in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS ’17, 2017, p. 483–493. [Online]. Available: https://doi.org/10.1145/3052973.3052999

Nikhil Chawla received the B.S. degree in Electronics and Communication Engineering from National Institute of Technology, Kurukshetra, India, in 2016, and the M.S. degree in Electrical and Computer Engineering from Georgia Institute of Technology, Atlanta, GA, USA, in 2017, where he is currently pursuing Ph.D. degree under the supervision of Prof. S. Mukhopadhyay. He was Security Researcher Intern with Intel Corporation, Hillsoboro, OR, USA, in 2020. His current research interests include cryptography hardware design, hardware security, machine learning for security, power management in modern SoCs.

Arvind Singh received his Ph.D. degree in Electrical and Computer Engineering from Georgia Institute of Technology, Atlanta, USA, in 2019. His doctoral research was focussed on energy-efficient and side channel secure cryptographic hardware for IoTs. Previously, he received his B.S. and M.S. degrees in electrical engineering from IIT Kanpur, Kanpur, India, in 2010. From 2010 to 2014, he was with the ASIC Design Team, NVIDIA, Bengaluru, India, where he was involved in tapeouts of multiple generations of Tegra and Graphics processors. He was an Intern with the Circuits Research Labs, Intel, Hillsoboro, OR, USA, and the ASIC/VLSI Research Group, Qualcomm, San Diego, CA, USA, in the summer of 2016 and 2017, respectively. After graduation, he joined Rambus Cryptography Research in San Francisco, USA, in the role of Sr. ASIC Design Engineer where he is involved with research and development of public key encryption accelerators and CryptoManager Root-of-Trust products. His current research interests include public key encryption algorithms and hardware and secure and energy-efficient architectures.



Harshit Kumar received the B.S. and M.S. degree in Electronics and Electrical Communication Engineering, with a specialization in VLSI and Microelectronics, from IIT Kharagpur, West Bengal, India, in 2019. He is currently working towards the Ph.D. degree in Electrical and Computer Engineering with Georgia Institute of Technology, Atlanta, Georgia, under the guidance of Prof. S. Mukhopadhyay. His current research interests include hardware security, understanding the interaction between malware and power management in modern SoCs, and malware analysis in general

Monodeep Kar received the B.Tech. degree in Electronics and Electrical Communication Engineering from Indian Institute of Technology Kharagpur, India, in 2012, and the M.S. and Ph.D. degrees in Electrical and Computer Engineering from Georgia Institute of Technology, Atlanta, GA, USA, in 2014 and 2017, respectively. He is currently a Research Staff Member with IBM Research, Yorktown Heights, NY, USA. He has authored over 40 conference/journal articles. His current research interests include domain specific hardware acceleration, power management and hardware security.

Saibal Mukhopadhyay received the B.E. degree in electronics and telecommunication engineering from Jadavpur University, Kolkata, India, in 2000, and the Ph.D. degree in electrical and computer engineering from Purdue University, West Lafayette, IN, USA, in 2006. He is currently the Joseph M. Pettit Professor with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, USA. He has authored or coauthored more than 200 articles in refereed journals and conferences. He holds five U.S. patents. His current research interests include the design of energy-efficient, intelligent, and secure systems. His research explores a cross-cutting approach to design spanning algorithm, architecture, circuits, and emerging technologies. He was a recipient of the Office of Naval Research Young Investigator Award in 2012, the National Science Foundation CAREER Award in 2011, the IBM Faculty Partnership Award in 2009 and 2010, the SRC Inventor Recognition Award in 2008, the SRC Technical Excellence Award in 2005, the IBM Ph.D. Fellowship Award from 2004 to 2005, the IEEE Transactions on VLSI Best Paper Award in 2014, the IEEE TRANSACTIONS ON COMPONENT, PACKAGING, and MANUFACTURING TECHNOLOGY Best Paper Award in 2014, and multiple best paper awards in International Symposium on Low-power Electronics and Design in 2014, 2015, and 2016.