SideGuard: Non-Invasive On-Chip Malware Detection in Heterogeneous IoT Systems by Leveraging Side-Channels

Fatemeh Arkannezhad, Pooya Aghanoury, Justin Feng, Hossein Khalili, Nader Sehatbakhsh
SsysArch Lab, ECE Department, University of California, Los Angeles, CA, USA

2024 IEEE Security and Privacy Workshops (SPW) | 979-8-3503-5487-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/SPW63631.2024.00030

***Abstract*—As heterogeneous systems become more common and diverse in IoT and CPS settings, securing these systems against malware has become a daunting task. To combat this, real-time hardware and/or (hardware-)software malware detection has gained popularity. *Hardware* malware detectors are effective but often require invasive changes to the CPU, hence limiting their usefulness in diverse settings. *Software* methods are non-invasive but often come with large performance overheads and/or disruptions to the main functionality of the device.**

This study proposes SideGuard, a new, non-invasive approach for detecting malware by analyzing the system's internal power consumption. With a tailored power sensor, our method utilizes this measured power consumption signal as a stand-in for program behavior. It collects training data, understanding how signals should appear in different program sections during proper execution. It then monitors execution, identifying instances where the observed signal deviates from the expected ones. For monitoring, the crucial idea is to *indirectly* measure power using customized sensors on an embedded FPGA or co-processor common in modern heterogeneous IoT systems. Notably, the monitoring unit (e.g., embedded FPGA) doesn't need a direct CPU connection but simply shares the power source, offering a *key advantage*: the malware detection unit requires no CPU changes, resulting in zero performance, power, and area overhead for the main CPU. Implementing this idea requires addressing *several new challenges* compared to prior work. Specifically, we introduce a new software-signal processing *co-design* approach. Results show that our approach can achieve >95% accuracy in detecting real-world malware. As heterogeneous IoT systems become more common, we believe our method is a strong contender for securing future hardware systems.

I. Introduction

Computing systems, especially embedded and smart IoT systems, are increasingly targeted by malware [1]. There are various methods to detect malware. Among them, hardware malware detectors (HMD) have gained attention in recent years [2], [3], [4]. They are favorable because they don't require complex software support or cause significant slowdowns in the system.

HMDs typically function by incorporating two key elements into the system: *monitoring logic* that gathers hardware-level information (e.g., performance counters) about the application, and a *classifier* to identify potential malware and anomalies. While highly effective, the main drawback of HMDs is the need for invasive changes to the device's CPU and/or its underlying system. Despite being feasible for many systems, it is not suitable or even possible for others, especially those already in use, older systems, and custom heterogeneous systems in which the system integrator lacks control over the internal design of each hardware component and can only add/remove components. Given the shift toward more *heterogeneity*, where various components including CPUs, FPGAs, and sensors are integrated on the same device/chip, there is a *strong demand* for techniques that can match the capabilities and performance of HMDs without invasive modifications.

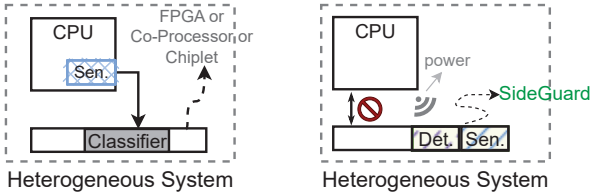

Fig. 1. Conventional hardware malware detectors (left) vs. our proposed method (right). Instead of collecting performance counters from the CPU and using a classifier, our method indirectly measures power consumption. No connection between the monitor and CPU is needed.

To address this demand, we propose a new on-device *non-invasive* malware detection method called SideGuard. The key insight is leveraging on-chip power side-channel as a means to *indirectly* monitor the system, and using a new detection algorithm that utilizes this data to detect anomalies. As shown in Figure 1, instead of collecting hardware-related features from the CPU, SideGuard indirectly measures the power consumption (i.e., a side-channel signal) of the CPU using customized sensors ("Sen") implemented as a *separate* component. This data is then used by our detection algorithm ("Det.") also implemented on the same unit, collectively creating an on-device malware detection unit.

The ***key advantage*** of this method is that it doesn't require any hardware support from the CPU or any connection to it, unlike current HMDs. This feature broadens its suitability for various systems with SoCs. In such setups, the malware detection unit can reside in a distinct component (an FPGA) or as a separate IP (in an SoC). Additionally, as the detection module is physically separated from the CPU, it creates an "air gap," further enhancing robustness.

Designing and implementing SideGuard involves ***several new contributions***. First, although the utilization of on-chip power sensors (e.g., ring oscillators) has been employed previously for side-channel attacks and hardware Trojan detection [5], [6], [7], [8], [9], this study stands out as the ***first*** to

DOI 10.1109/SPW63631.2024.00030

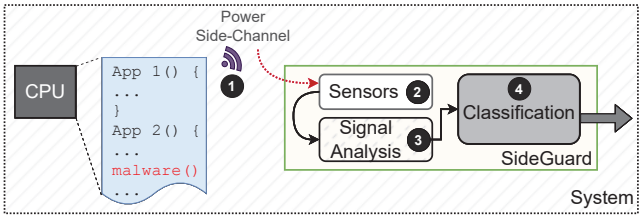

Fig. 2. Software running on a CPU creates unintentional power consumption fluctuations (i.e., power side-channels). SIDEGUARD captures this information using an array of on-chip power sensors and leverages a signal analysis and detection algorithm to find anomalies.

apply on-chip power consumption for *dynamic program monitoring* and consequently for detecting malware. Unlike earlier studies, dynamic program monitoring *introduces entirely new research inquiries*.

Second, creating SIDEGUARD involves tackling *two new research challenges*. The initial one is devising and executing the malware detection algorithm. The fundamental distinction between our approach and current classifiers for malware detection lies in the *nature of our features*. Traditional HMDs utilize hardware performance counters, which offer a naturally discrete feature space. In contrast, our problem deals with a continuous time-series data feature space. Also, in *contrast to* previous methods that used on-chip power monitoring for detecting Trojans, our task (dynamic program monitoring) demands a significantly more intricate detection algorithm. The difficulty stems from the need to monitor a *diverse range of software applications and types of malware*. This is a departure from the monitoring of just a *single* application (such as cryptographic cores) and a *single* malicious behavior, as seen in Trojan detection techniques.

The next challenge involves the system and hardware design aspects of SIDEGUARD. Two main questions need addressing here. Firstly, how to design the sensors and manage continuous data effectively to reduce storage usage? Secondly, considering the power and storage constraints on the device (especially for embedded and IoT devices) and the requirement for an advanced signal processing method to handle time-series data, how can a detection algorithm be implemented efficiently in terms of both area and power usage?

We systematically analyze the effectiveness of our detection framework using various malware on a real SoC system, a DE1-SoC board. In short, the key contributions of this paper are as follows.

- We propose a new non-invasive hardware malware detection mechanism suitable for heterogeneous IoT systems.
- We design and implement a new detection algorithm for time-series power side-channel data collected by our custom sensors.
- We evaluate our design and implementation using different malware families and standard benchmark applications.
- Our framework is implemented on two real-world SoC systems as a proof-of-concept.

II. SYSTEM DESIGN OVERVIEW

Threat Model and Assumption. We focus on malware detection for *heterogeneous* "smart" embedded/IoT devices such as robotic devices, medical devices, and smart home systems. We target devices equipped with a system-on-chip (SoC) and/or heterogenous 2.5D systems, comprising various IPs and/or chips/chiplets. These components include sensors, actuators, and processing elements where one or multiple cores are controlled by an operating system. We assume that our detection framework is implemented on the system using a hardware component such as an embedded FPGA (eFPGA) and/or a co-processor implemented as a separate IP and/or chip. It's worth noting that comparable assumptions were made in previous hardware malware detection frameworks, utilizing the eFPGA/co-processor to implement the classifier [3], [10], [2], [4]. Therefore, SIDEGUARD doesn't introduce new hardware; instead, it suggests a method to *repurpose* the existing hardware.

We assume that the system is initially secure. The system, however, can get compromised as it starts executing various applications. Once it is compromised, the adversary controls the entire CPU and kernel OS. Furthermore, we assume that SIDEGUARD and its underlying hardware (i.e., eFPGA) is part of the root-of-trust (RoT) and can only be re-programmed through a secure update. Further, the RoT is additionally protected from an adversary since the monitoring framework is physically separate from the CPU and not controlled by the OS (i.e., *air-gapped*). Providing this air-gap eliminates the possibility of the monitor being infected by the same attack vectors that have compromised the host system.

For detecting malicious activities, SIDEGUARD doesn't possess a priori knowledge about the type of attack or its power signatures and detection solely depends on the signals gathered by the sensors during monitoring. Further, SIDEGUARD always maintains accurate reference models for malware-free signatures. These models are stored internally and remain uncompromised. The models, however, can be updated through a secure update, if needed. Moreover, we assume that the adversary is familiar with the system and program(s), including any existing vulnerabilities, and can manipulate the system by sending random inputs.

System Overview. The high level design of SIDEGUARD is shown in Figure 2. Internally, SIDEGUARD consists of three main components: a sensor array, a signal analysis unit, and a classifier. We briefly explain each in the following.

The first component is the *on-chip power sensors* (❷). An essential feature of SIDEGUARD is its complete non-invasiveness. Therefore, the sensors (power or other types) should not have direct connections to the CPU. Instead, their design should allow indirect monitoring of the CPU (when running different applications). To achieve this, we utilize a time-to-digital converter (TDC) primitive [8]. These sensors are capable of tracking alterations in power usage within the shared power distribution network (PDN) by sensing changes in the delay of a propagating signal through a chain of buffers or other logic, thereby capturing the behavior of various applications operating on the host CPU (❶). A primary challenge in our design is the balance between sensor circuit size and precision. While more sensors provide more accurate data, they also occupy additional space and consume more power.

We opt for a design with 32-bit granularity. In Section IV-B, we'll delve into our design specifics and share our accuracy detection results.

The second element is a signal analysis module. As detailed in Section I, the fundamental distinction between our problem and current HMD solutions is our analysis involving a continuous time-domain signal. Consequently, we present a novel signal analysis algorithm (❸ and ❹). In our design, a crucial strategy is adopting a co-design approach. This means tailoring our signal analysis strategy to match the observed behavior of the target software. Further specifics are provided in the next section.

III. Malware Detection Approach

Problem Overview. Utilizing the sensors detailed in Section II, a continuous stream of sampled data is fed to the Signal Analysis module (refer to ❶-❸ in Figure 2). With this time-series data, our goal is to pinpoint the initiation of any malicious operations. To achieve that, we view the gathered time series data, which we call *signal*, as a substitute for program behavior across time. Accordingly, we compile training data outlining the expected appearance of the collected data in various segments of the program. We then track execution, identifying instances where the observed data deviate (using a later-described metric) from the anticipated pattern—indicating that the observed data are unlikely to result from correct execution.

There are numerous methods to analyze time series. Examples include autoregressive integrated moving average (ARIMA), seasonal decomposition of time series (STL), or machine learning models like Long Short-Term Memory (LSTM) networks. Specific to our malware detection, however, the desired approach should be:

1) Capable of withstanding variations in program execution, including slight changes in power consumption caused by microarchitectural variations (such as cache misses), diverse execution paths (due to branches), and others.
2) Resilient to temporal and process variations. This includes variations due to the physical differences between one device and another, potentially leading to slight changes in power consumption when running the same program. It also considers temporal fluctuations within the same device, like variations in device temperature that might also result in changes in power consumption.
3) Able to handle various programs with different behaviors while accurately pinpointing unusual behaviors caused by malware.

Detection Algorithm. To address these needs, SideGuard takes the following approach: Rather than directly processing the time-domain signal, SideGuard first breaks down the signal into fixed-size pre-determined sequences (which we call *frames*) and then employs the Short-Term Fourier Transform (STFT) to transform each frame to a frequency-domain spectrum (which we call frequency frame or FF). Each FF is represented as a vector of size m, where m denotes the number of frequency bins. The features utilized for analysis within SideGuard consist of the *amplitudes* associated with each

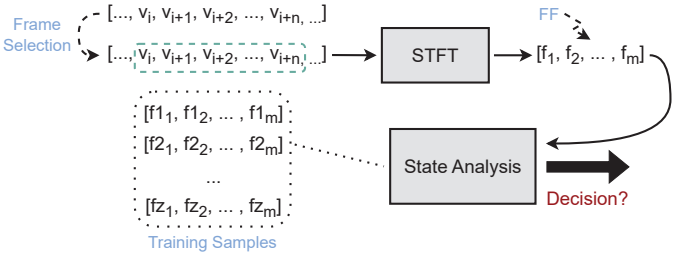

Fig. 3. SideGuard analyzes the collected power signals (a time series) by grouping them into overlapping *frames* and taking a short-time Fourier transform (STFT) to create frequency features (FF). These frequency-domain features are then used for analysis. The main advantage of this approach is its robustness to program and device variations.

frequency bin. All training and monitoring activities within SideGuard are conducted on this sequence of features. These steps are also shown in Figure 3.

To monitor a program using FFs, SideGuard first identifies unique program phases called *regions*. We define "region" as a sequence of consecutive FFs where the distance between any pair of FFs is less than TH. More formally:

$$\{FF_i\}_{i=t}^{t+L} \text{ is a region (of size L) if:}$$
$$\forall i \in \{t \leq i < t+l\} : Dist(FF_i, FF_{i+1}) < TH, \quad (1)$$

where $Dist(,)$ is defined as:

$$Dist(FF_i, FF_{i+1}) = \sum_{j=1}^{K} (FF_{i_{sortA}}[j] - FF_{i+1_{sortA}}[j])^2 / j. \quad (2)$$

To identify a region, SideGuard takes the top K elements of an FF (i.e., by sorting an FF using the amplitude of each bin in ascending order and storing the *index* in $FF_{i_{sortA}}$) and computes the distance (between indices). SideGuard then defines a "new" region once the distance between consecutive FFs exceeds the threshold. As indicated by Equation 1, the comparison of indices corresponds to the squared distance between the two. Moreover, this comparison is weighted by dividing it by the iterator, giving greater importance to more prominent features (note that we sort FF by its amplitude first, thus the first index is the most prominent element).

To clarify why we've chosen this method, consider the following ***crucial observations***: it is well-known that most programs tend to have distinct phases and, more importantly, they tend to *repeat* these phase behaviors [11], [12], [13], [14]. In simpler terms, programs often repeat actions, like executing repetitive code with loops (e.g., FOR or WHILE loops). This repetition means running a similar set of instructions over and over. So, the power consumption should follow a repetitive pattern, with the signal's period matching the time for each repetition. In our method, for specific lengths, frames in SideGuard become periodic signals. Their frequency-domain transformations (FFs) also show this periodic nature. Signal processing basics tell us that a periodic signal in time has clear "spikes" in the frequency domain. So, certain frequency bins in each FF should have significant magnitudes, forming spikes. As the program goes through phases, its

```
1  // The (simplified) main loop
2  while(1){
3
4  // Phase 1: read sensors
5  [speed, position] = ReadSpeed();
6
7  // Phase 2: read commands
8  cmd = ReadCommand();
9
10 // Phase 3: update the position
11 status = UpdateRobot(speed, position, cmd);
12
13 // Phase 4: status/output update
14  UpdateStatus(status, cmd);
```

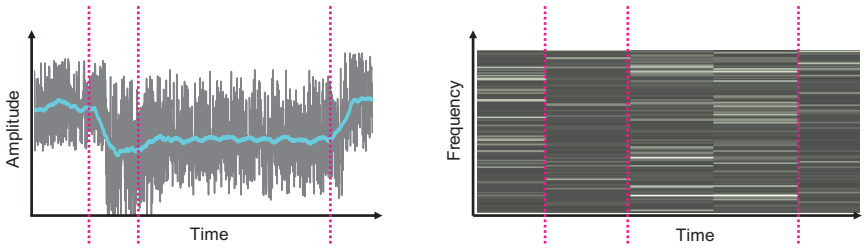

Fig. 4. A sample code for a robotic arm is shown on the left. This program has four main phases. The time-domain signal is shown in the middle. The bright blue signal shows the moving average. The signal exhibits periodic patterns with each phase. The frequency-domain signal (waterfall plot) is shown on the right. The signal shows clear periodic patterns for each phase, allowing SIDEGUARD to track the code using these frequency features (each bright line is a feature).

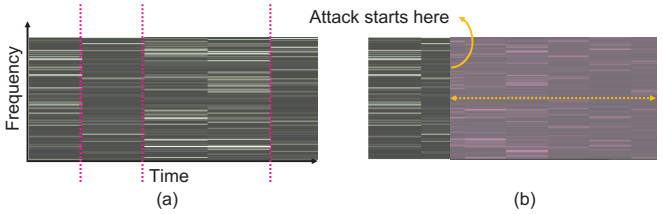

Fig. 5. The difference between frequency-domain (FFs) signal in a normal program vs. malware (SYN flood with a buffer overflow).

periodic behavior changes, showing up as different sets of spikes in the frequency bins. The key idea is that these magnitudes in frequency bins are a distinct and robust feature, making it effective to classify devices.

Using the concept of regions, our algorithm reports anomaly if ***(i)*** the current identified region is not part of observed regions during the training time (i.e., the algorithm identifies a "new" region that was never observed before), and/or ***(ii)*** the transition between regions does not match with a "valid" transition observed during the training time.

We opt for this method because it stems from a ***crucial insight***: malware exhibits fundamentally different behavior compared to regular programs [13], [15]. In our system, this translates to anticipating distinct and predictable phases in a program's execution. However, when potential malware disrupts the program unexpectedly, the execution shifts, causing significant changes in the monitored features, such as FFs and the magnitude of each bin.

Finally, during the training phase, we execute a standard application without malware to capture the power signal. Subsequently, we identify distinct regions using the specified metric. SIDEGUARD records an instance for each region, noting the observed sequence of these regions. This procedure is repeated for each application requiring profiling. An important aspect of training is to ensure satisfactory coverage. Since SIDEGUARD depends on observing valid regions during training, it's the user's responsibility to execute the target application with diverse inputs, ensuring comprehensive code coverage. Methods for achieving this coverage are beyond the scope of this paper, but examples include using techniques like fuzzing and/or symbolic execution.

To showcase the details of the algorithm better, we use a simple yet realistic example for a robotic arm application. The code for this example is shown in Figure 4. This figure also shows the time-domain and frequency-domain signals when running this application on an SoC (a DE1-SoC board. Details of the setup will be provided in Section IV-A).

Figure 4 shows that the program consists of four clear phases: checking the sensors, executing commands, moving the arm, and updating the status. To keep things simple, we haven't displayed the specifics of each step, but each phase involves a series of actions. Crucially, the signals shown in Figure 4 in both time and frequency domains suggest that these phases have different patterns. In particular, looking at this figure, it's evident that FFs for each phase share very similar patterns, noticeable as bright lines in the right figure. Therefore, arranging each FF in ascending order and comparing the top K elements (the brighter lines in the figure) gives a reliable estimate of the current program region.

To check if this also works with malware, we compare the signals while the robotic arm application is running to when an attack occurs randomly during execution. In this test, we carry out a buffer overflow followed by introducing a Mirai malware payload [1] as an example of malware (you can find more details about benchmarks in Section IV). The results are displayed in Figure 5. As can be seen from the figure, the attack starts somewhere during the second phase (when the program is reading commands). It is evident from this figure that running malware generates distinct signatures in the frequency domain. Keep in mind that SIDEGUARD doesn't require any information about the malware or its signatures. It solely detects the deviation of signatures (i.e., FFs) from the anticipated behavior. For example, in this scenario, SIDEGUARD anticipates signatures akin to the second region or a "new" region resembling what was observed during training (i.e., the third region) but the generated samples from malware do not look similar to any of them.

Robustness and Key Insights. A critical next step for SIDEGUARD involves enhancing its robustness. Remember that our algorithm needs to be resilient against variations in program, temporal, and devices. The key to achieving this lies in the concept of FFs. However, our further analysis reveals that while FFs effectively handle program variations due to their inherent averaging behavior, additional optimizations are necessary to make SIDEGUARD robust against device and temporal variations.

To address this, we implement the following improvements in our feature selection and distance metric:

- Instead of defining a new region when two consecutive FFs don't match, we apply a smoothing filter where the criterion for a new region becomes five samples. This allows us to tolerate temporal noise when one or a few FFs behave differently.
- To accommodate device variation, we modify $Dist$ to calculate the difference between *adjusted* samples instead of absolute samples. More formally, we define $idx_i = argmax\{FF_i\}$ and then create $FF_{i_{sortA}}$ by storing $|j - idx_i|$ for each sample instead of storing j (i.e., the index).

The two optimizations allow SIDEGUARD to tolerate temporal and device-level variations. More results are provided in Section IV-C.

IV. EVALUATIONS

A. Setup

Hardware. To implement SIDEGUARD, we use a Terassic DE1-SoC development board. It has a dual-core cortex-A9 ARM core that supports Linux OS, an Intel Cyclone V FPGA, SDRAM on both the core and FPGA, and additional peripherals including SD card, JTAG, etc. This board is representative of common SoCs that are used in the market that have an embedded FPGA along with multiple hard cores [9]. Applications (and malware) are executed on one of the ARM cores while SIDEGUARD is implemented on the FPGA. The cores and the FPGA can communicate (through memory-mapping) via a series of connection *bridges* and AXI buses (FPGA-to-Hard Processor, f2h, and vice versa, h2f). Such connection is not needed during the monitoring but can be used for other purposes such as secure update, synchronization, and/or alerting the CPU if/when malware is detected.

To assess SIDEGUARD's resilience to device variations, we employ an extra SoC board, specifically a Zynq™ UltraScale+™ MPSoC. This board, akin to the DE1-SoC, features ARM core(s) and an FPGA. Additionally, we replicate our experiments on a second DE1-SoC board identical to the original. We present the primary results, including accuracy, area, and power, based on the DE1-SoC board. The other two boards serve to scrutinize SIDEGUARD's robustness. Refer to Section IV-C for more detailed information.

Benchmarks and Mawlare. SIDEGUARD are evaluated using five representative applications from the MiBench standard benchmark suite [16]. Specifically, we use the following applications: {*bitcount, basicmath, qsort, susan, fft*}. These applications mirror typical behaviors of embedded systems like security, telecom, and networks. This list is carefully selected since each has unique features that are important for thoroughly evaluating SIDEGUARD. For example, we specifically chose "bitcount" and "basicmath" because they serve as solid representations of applications with various distinct regions and diverse activities, including nested loops, recursive functions, and interactions with memory. "susan" and "fft" were picked because they effectively embody common and popular activities in the embedded system domain, such as image processing and telecommunication.

To evaluate how well SIDEGUARD can detect malware, we implement two classes of malware: Ransomware and a DDoS attack by using a (Mirai) botnet. Each malware is executed by exploiting (an intentional) software vulnerability (buffer overflow) on each of the benchmark applications.

For the DDoS attack, we port the command and control (C&C) and the bots from the Mirai's open-source code to run on our device. The DDoS payload execution begins right after the buffer overflow exploit where a shell is invoked, and ends after sending 100 SYN packets. The application then resumes its normal activity. As another payload, we also implement a simple Ransomware prototype payload that uses AES-128 with CBC mode to encrypt data. This encryption represents the bulk of the execution activity created by Ransomware. We believe that choosing these malware families offers a practical assessment for SIDEGUARD and aligns with previous research [3].

B. Implementation and Results

Implementation of SIDEGUARD. The SIDEGUARD is implemented on the FPGA using Verilog HDL and Quartus. We use default frequencies for the CPU and FPGA, i.e., 25 MHz and 50 MHz, respectively. For power monitoring, we implement a time-to-digital converter (TDC) sensor network based on the method proposed by Ma *et al.* [8]. In short, to transform voltage fluctuations into a digital number, we utilize a carry chain (CARRY) primitive, and flip-flops store this value. By iterating this primitive, we establish a 32-bit voltage sensor. Our circuit can attain approximately eight samples per clock sampling rate.

Realizing SIDEGUARD detection logic involves leveraging an FFT core and implementing a finite-state machine (FSM) for FF analysis. A 64-point pipelined FFT engine is used. Furthermore, SIDEGUARD requires internal memory for storing the training data. We set K (see Equ. 1) to five hence storing each region in the training data is a $5 \times log(64)$-bit data. Applications used in our setup average around five regions per application. As a result, the total storage required is less than 100 bytes. For a real-world setup with 10-20 possible applications, this number will be still below 1 KB.

Overall, the complete design of SIDEGUARD, encompassing the sensors, FFT engine, detection logic, and memory storage, occupies less than 35% of the entire FPGA. For a larger size FPGA (e.g., Virtex FPGAs in UltraScale MPSoC), this overhead is well below 20%.

On the CPU, SIDEGUARD imposes *zero* overhead since the system is used as is, without requiring any additional changes (software or hardware). Regarding power, SIDEGUARD consumes approximately 47 mW at its peak. This amount constitutes about 3% of the total power of the system.

Results. To gauge how well SIDEGUARD detects malware, we present the detection accuracy while using the five applications described earlier. Specifically, we use five runs for training (to capture temporal and program-level variations) and then use 1000 runs without malware and 500 runs with each malware (i.e., DDos and Ransomware) for all five applications. Results are shown in Table I.

TABLE I
RESULTS FOR DETECTING MALWARE IN FIVE REPRESENTATIVE APPLICATIONS FROM THE MIBENCH BENCHMARK SUITE. TWO MALWARE EXAMPLES WERE USED: RANSOMWARE AND MIRAI BOTNET. FALSE POSITIVES (FPS) DENOTE NORMAL RUNS INCORRECTLY LABELED AS MALICIOUS, WHILE FALSE NEGATIVES (FNS) REFER TO SCENARIOS WHERE MALWARE WAS EXECUTED BUT NOT DETECTED.

Bitcount		Susan		FFT		Basicmath		qsort	
FP	FN	FP	FN	FP	FN	FP	FN	FP	FN
1.7%	.1%	4.1%	<.1%	3.5%	<0.1%	2.9%	<0.1%	3.1%	<.1%

TABLE II
RESULTS ASSESSING ROBUSTNESS AGAINST SCENARIOS INVOLVING THE SAME DEVICE AND DIFFERENT DEVICES.

Experiment	Average Change in Accuracy	
	FP	FN
Same-Type Board (DE1)	+2.7%	0%
Different Board (Zynq)	+0.3%	0%

As shown in the table, SIDEGUARD can successfully detect almost all the instances of the attack (i.e., >99.9% true positive rate). The key reason for this is using Frequency Features (FF) allows us to observe very different signatures from the malware compared to the signature created by normal activities. Table I also reports the false positive rates (FP). As can be seen, in all instances, SIDEGUARD maintains a very low FP (3.06% on average). This indicates that SIDEGUARD is robust against different sources of program variations. In Section III, we described various techniques used to enable such robustness including the usage of FFs.

C. Further Robustness Analysis

To examine how well SIDEGUARD extends to other devices, we repeat our experiments using two other setups. First, we examine how well SIDEGUARD performs when another instance of a same-type device is used (i.e., another DE1-SoC board). Specifically, we train SIDEGUARD in one device and then test on the other (using the same set of applications and malware families explained in Section IV-B). The results are shown in Table II.

The results indicate that SIDEGUARD is highly transferable to other device types. As shown in Table II, our tests on a different board (Xilinx Zynq) show very similar accuracy (false positive and negative). For same-type devices, i.e., when training on one and testing on the other, we see an average of 2.7% increase in false positive rates. However, the results are still below 5% on average showing that SIDEGUARD is still accurate when considering devices and/or temporal variations.

D. Comparison with State-of-the-Art

To conclude this section, we will briefly compare SIDEGUARD with existing methods. Two closely related groups include hardware malware detectors (HMDs) [3] and Trojan detection methods using on-chip power consumption to identify Trojans [8]. In comparison to the former, SIDEGUARD boasts a distinct advantage—its non-invasiveness. Our malware detection unit can function independently as a standalone module within the system. This positions it as an excellent choice for heterogeneous IoT systems, where the system integrator is trusted but lacks access to individual components, especially the CPU, to implement additional security measures.

In contrast to the latter, SIDEGUARD tackles a notably more intricate challenge: identifying dynamic malware within a regular application. The key distinction lies in the multitude of potential applications operating on a device, making naive solutions or those reliant solely on representation and/or machine learning less accurate. Instead, SIDEGUARD introduces a co-design approach, combining software and signal analysis. By leveraging program behavior insights, it devises a precise yet efficient detection mechanism.

V. CONCLUSIONS

This study introduced a novel non-invasive technique for identifying malware by examining the power consumption on the chip. The main concept was to indirectly gauge power using tailored power sensors integrated into an embedded FPGA, a component commonly found in modern heterogeneous systems. A significant advantage of our approach is that the malware detection doesn't necessitate CPU modifications, resulting in zero impact on the main CPU's performance, power, and area. The results indicated that our method can achieve >95% accuracy in detecting real-world malware. With the increasing adoption of heterogeneous IoT systems, our method emerges as a resilient choice for the evolving landscape of future hardware systems. Its adaptability and efficiency make it a promising solution amidst the growing prevalence of IoT and CPS systems and their security vulnerabilities.

REFERENCES

[1] M. Antonakakis *et al.*, "Understanding the mirai botnet," in *26th {USENIX} security symposium*, 2017, pp. 1093–1110.

[2] J. Demme *et al.*, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH computer architecture news*, vol. 41, no. 3, pp. 559–570, 2013.

[3] K. Basu *et al.*, "Preempt: Preempting malware by examining embedded processor traces," in *DAC, 2019*, 2019, pp. 1–6.

[4] K. N. Khasawneh *et al.*, "Rhmd: Evasion-resilient hardware malware detectors," in *MICRO, 2017*, 2017, pp. 315–327.

[5] I. Giechaskiel *et al.*, "C 3 apsule: Cross-fpga covert-channel attacks through power supply unit leakage," in *S&P, 2020*. IEEE, 2020, pp. 1728–1741.

[6] A. Boutros *et al.*, "Neighbors from hell: Voltage attacks against deep learning accelerators on multi-tenant fpgas," in *ICFPT, 2020*. IEEE, 2020, pp. 103–111.

[7] Z. Xie, S. Li, M. Ma, C.-C. Chang, J. Pan, Y. Chen, and J. Hu, "Deep: Developing extremely efficient runtime on-chip power meters," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.

[8] H. Ma *et al.*, "On-chip trust evaluation utilizing tdc-based parameter-adjustable security primitive," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 10, pp. 1985–1994, 2020.

[9] M. Zhao and G. E. Suh, "Fpga-based remote power side-channel attacks," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 229–244.
[10] C. Konstantinou *et al.*, "Hpc-based malware detectors actually work: Transition to practice after a decade of research," *IEEE Design & Test*, vol. 39, no. 4, pp. 23–32, 2022.
[11] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGPLAN Notices*, vol. 37, no. 10, pp. 45–57, 2002.
[12] N. Sehatbakhsh, A. Nazari, A. Zajic, and M. Prvulovic, "Spectral profiling: Observer-effect-free profiling by monitoring em emanations," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–11.
[13] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic, "Eddie: Em-based detection of deviations in program execution," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 333–346.
[14] J. Feng, T. Zhao, S. Sarkar, D. Konrad, T. Jacques, D. Cabric, and N. Sehatbakhsh, "Fingerprinting iot devices using latent physical side-channels," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 7, no. 2, pp. 1–26, 2023.
[15] N. Sehatbakhsh, A. Nazari, M. Alam, F. Werner, Y. Zhu, A. Zajic, and M. Prvulovic, "Remote: Robust external malware detection framework by using electromagnetic signals," *IEEE Transactions on Computers*, vol. 69, no. 3, pp. 312–326, 2019.
[16] M. R. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC-4, 2001*. IEEE, 2001, pp. 3–14.