# PITEM: Permutations-based Instruction Tracking Via Electromagnetic Side-Channel Signal Analysis

Elvan Mert Ugurlu, *Student Member, IEEE,* Baki Berkay Yilmaz, *Student Member, IEEE,* Alenka Zajić, *Member, IEEE,* and Milos Prvulovic, *Member, IEEE*

**Abstract**—The emergence of cyber-physical systems (CPS) and internet of things (IoT) devices impose significant security and privacy concerns that necessitate robust monitoring and malware detection systems. This paper proposes PITEM, a framework for instruction-level monitoring and malware detection using electromagnetic (EM) side-channels. PITEM identifies *instruction types* with similar EM emanations using hierarchical clustering. To track all combinations of these *instruction types*, we generate EM signatures for all permutations of them. In testing, we predict the permutation class of testing traces by a matched-filter-like predictor. We test the performance on two devices (FPGA-based and ARM-based) with 50 MHz and 1 GHz clock frequencies. We achieve $95.67\%$ and $87.35\%$ accuracies for these devices for single execution of permutations. We note that the accuracy increases to $100\%$ when permutation blocks are repeated. Furthermore, we test the limits of the system by tracking permutations of instructions of the same type. With sufficient bandwidth and number of repetitions, individual instructions can be resolved with $87.5\%$ and $95.78\%$ accuracies for these devices. The performance is evaluated for different relative signal-to-noise ratio (SNR) levels and performance is stable for relative SNR values $> 15$ dB. Finally, we demonstrate PITEM's ability to detect *fine-grained* malware with 99.89% accuracy.

**Index Terms**—Electromagnetic Side-Channel, Instruction Tracking, Information Security, Security, Privacy.

## 1 Introduction

Cyber-physical systems (CPS) and internet of things (IoT) devices are becoming essential part of the human daily life. These systems and devices are used in many areas such as health-care, autonomous cars, agriculture, military, smart grids, smartphones, etc. [1], [2], [3], [4]. These systems promise many benefits including higher efficiency, sustainability and better connectivity. However, the operation of CPS and IoT systems heavily relies on the access to confidential and sensitive information. This leads to significant privacy and security concerns. In addition, any malicious attack to these systems imposes safety, malfunctioning, unauthorized actions, and other damage risks. Hence, it is compulsory to protect these systems with reliable monitoring and malware detection frameworks.

Recent studies demonstrate that CPS and IoT systems are vulnerable to side-channel attacks [5], [6]. Side-channels are unintentional and asynchronous channels that possibly leak confidential and sensitive information during the program activity [7]. These channels are created as a result of transistor switching and current flow changes during the program execution. [7], [8]. Several physical phenomena such as power consumption ([9], [10], [11]), temperature changes ([12], [13]), acoustic emanations ([14], [15]), and EM emanations ([8], [7], [16], [17]) have lead to many attacks on different systems. The significance of these attacks motivated researchers and experts to exploit these side-channels for defensive and protective applications such as monitoring code flow, reverse engineering, cryptanalysis and malware detection [18], [19], [20], [21], [22], [23].

Malware detection is one of the most commonly used applications of side-channel analysis. These malware detection systems utilize an external monitoring hardware that captures side-channel leakage and processes it for malware detection. Due to the separation of the monitoring and monitored systems, these systems are often called *external* monitoring systems and have the following desired properties: 1) they are *zero-overhead*, i.e., they do not use the resources of the monitored system, 2) the monitoring system is isolated from the monitored system that prevents the interference of the malware with the monitoring system, 3) these systems are *observer-effect free*. EDDIE and Syndrome are recently developed EM side-channel based malware detection frameworks [21], [24]. They are both based on Spectral Profiling [25] that profiles the program activity based on the per-iteration time of the periodic activities such as loops, and successive function calls. Although these frameworks report detection accuracies as high as $92\%$, these methods are *coarse-grain* malware detection systems that cannot detect small modifications. Also, since these methods are based on per-iteration time of the loops, an attack that makes instruction modifications while maintaining the total execution time cannot be detected. For example, if the order of instructions within a loop is changed, the per-iteration time remains the same, therefore, the modifications cannot be detected. Similarly, these methods can not locate the modifications precisely.

As another application of side-channel analysis, code-flow monitoring is performed on numerous levels: loops, paths, basic blocks, and instruction sequences [26], [25], [27],

*Elvan Mert Ugurlu, Baki Berkay Yilmaz and Alenka Zajic are with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, USA.*
*Milos Prvulovic is with the School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA.*

[28]. Main objective of these efforts is to verify that correct sequence of instructions has been executed by monitoring side-channel signals. These works report successful tracking of the executed instructions for devices with relatively low operating clock frequencies (1, 4 or 16 MHz) and simple processor architectures [18], [19], [29], [30]. Recently, [28] proposed a new technique to address devices with higher clock frequencies (50 MHz) and more complex processor architectures (6-stage-pipelined processor). They report promising results for tracking sequences of instructions but they do not investigate all possible sequences of instructions in a systematic way.

EM emanations during execution correlate with the program activity and they are indicator of the trace that the instruction goes through while being executed. Therefore, EM emanations are also called EM signatures. The instructions that go through similar traces generate similar EM emanations, whereas instructions with different trace characteristics generate distinct EM emanations. The key challenge for EM-side channel based instruction level tracking is the noise, which is stronger than the variation between the EM signatures of instructions in many cases.

To enable instruction-level tracking when noise is present, we propose a new framework: PITEM. This framework consists of two major steps: 1) identifying groups of instructions that are referred to as *instruction types* that have similar EM signatures, 2) tracking all possible orderings, i.e. permutations, of these *instruction types* and therefore, monitoring program flow at *instruction type* granularity. To make the definition of these terms more clear, we present a simple example. Assume that we investigate a processor that has only four instructions: ADD, SUB, MUL and STR. Let ADD and SUB operations have similar EM signatures, MUL operation have a different EM signature than ADD and SUB, and STR operation have a different EM signature than all the other three instructions. Based on the EM signatures, first part of our framework finds 3 *instruction types* that we reference with letters: *type* A (ADD and SUB), *type* B (MUL), and *type* C (STR). Second part of our framework detects which ordering of these instruction types has been executed. For example, a code block that has the instructions ADD-MUL-STR in order can be recognized as permutation "ABC", whereas a code block that executes the instructions MUL-STR-ADD can be detected as permutation "BCA". By generating all possible permutations of the *instruction types*, we generate instruction sequences systematically. Generating EM signatures for sequences rather than single instructions addresses the pipeline effect to a great extend as they represent the overall EM emanations for longer periods of time. Also, this method is not limited to devices with lower clock frequencies. Therefore, this framework is applicable in program activity tracking applications even for devices with complex processor architectures and higher clock frequencies.

Furthermore, this framework can be used in *finer-granularity* malware detection applications. By using the permutations as reference signals, modifications that are made at instruction level can be resolved. In applications where the program is expected to execute one of the allowable instruction sequences, this technique can determine any unexpected instruction executions and resolve at what point the modification has been made. Note that, unlike EDDIE [21] and Syndrome [24], this method does not rely on per-iteration time of loops and it can detect the changes when the ordering of the instructions is changed.

To show the feasibility of the proposed method, we perform testing on two devices with different architectures and operating clock frequencies. These devices are Intel's DE1 Altera FPGA Board with Altera NIOS-II (soft) processor [31], and A13-OLinuXino with ARM Cortex-A8 processor [32], operating at 50 MHz and 1 GHz clock frequencies, respectively. The results are reported for different experimental setups. We note that single execution of the permutations of the *instruction types* can be detected with as high as 95.67% accuracy. As the number of successive executions of the permutations increases, the detection accuracy increases to 100%. We also test the performance of the proposed method with different relative signal-to-noise ratio (SNR) levels. We note that the system performance is stable for relative SNR levels higher than 15 dB. Next, we test the limits of the system by tracking permutations of instructions from the same *instruction type*. As expected, the detection performance for single execution of the permutation is low, but it increases significantly (to as high as 87.5% and 95.78%) when the permutation is repeated. Finally, to demonstrate *fine-grained* malware detection capability of the proposed framework, we test the malware detection performance for single or a few instruction deviations from the expected code sequence. We observe that we can detect these changes with as high as 99.8% accuracy.

The remainder of the paper is organized as follows. Section 2 describes our procedure to identify *instruction types*, Section 3 explains the proposed permutation-tracking system, Section 4 presents experimental results, Section 5 includes robustness evaluation, and Section 6 draws conclusions.

## 2 Determining Instruction Types by using EM Side Channel

The proposed methodology is based on analyzing EM side-channel signals. As mentioned earlier, EM side-channels are created as a by-product of fast switching currents flowing through transistors during program execution. Therefore, execution of certain instructions generates distinct EM signatures. In this section, we describe the procedure to identify these instructions and call them *instruction types*. Since different architectures are implemented differently on micro-architecture level, these *instruction types* differ for different architectures. Fig. 1 presents an outline of the procedure step by step. These steps are explained in detail in the following sections.

### 2.1 Generating List of Instructions Under Investigation

This step includes examining the available instruction set for the given processor and selecting the desired and applicable instructions to investigate. The applicability of the instructions is based on the micro-architecture of the processor.

### 2.2 Generating Microbenchmark for Instructions

After instruction selection, we generate a microbenchmark for each instruction whose pseudo-code is given in Fig. 2. One should note that this work uses an *instrumented* measurement setup where an input/output (I/O) pin is set to

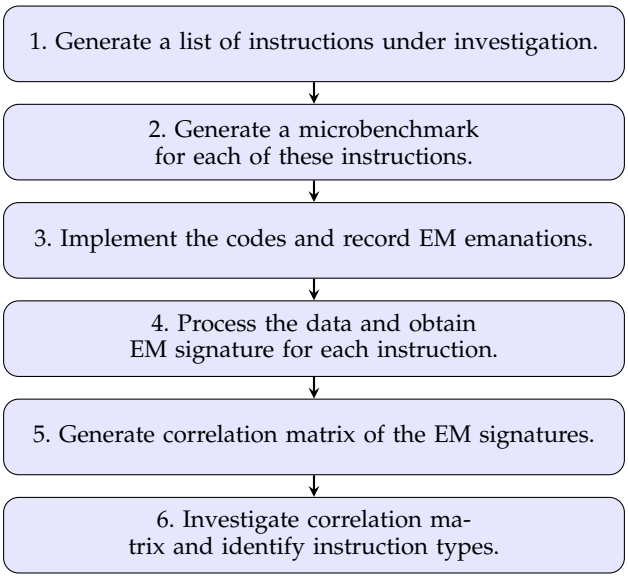



Fig. 1. Flowchart of determining instruction types.

```
while true
   for
      %empty for loop
   end

   % Set I/O Pin to High

   for N times
      %Instruction Under Interest
   end

   % Reset I/O Pin to Low

   for
      %empty for loop
   end
end
```

Fig. 2. Pseudo-code for Instruction Type Detection setup.

high voltage before the code under observation and reset to low voltage after the code under observation. Therefore, the input/output pin signal is used to find the starting and ending points of the region of interest.

For most processors, the EM signature difference caused by a single instruction does not generate a distinct variation [33], therefore, the instruction under interest is repeated $N$ times to magnify the EM signature. Note that this repetition is realistic because the pesudo-code structure is only used in determining *instruction types* step and it is not utilized in any testing scheme. The starting and ending markers are preceded and succeeded by two empty loops, respectively. This for-loop structure allows for a fair comparison since they make sure that the same instructions are pipelined before and after all instructions under interest.

### 2.3 Implementing the Codes and Recording EM Emanations

After microbenchmark implementation, EM emanation measurements are performed. The measurement includes two synchronized channels: 1) EM emanation signal, 2) I/O pin signal. For better localization, a near-field antenna with proper antenna gain and size should be utilized. The choice of the antenna size is based on their ability to capture relevant EM emanations from the processor and reject the interference from other parts of the device.

### 2.4 Data Processing to Obtain EM Signatures

In this step, the recorded EM emanation and I/O signals are processed to obtain EM signatures of each instruction. Fig. 3 presents an overview of data processing flow. First, the input I/O signal is filtered with a moving median filter to overcome overshooting. Then, the signal is normalized to account for possible DC offset. Next, the amplitude values are quantized to their binary representations by using a 3 dB threshold. The binary stream of data is smoothed by removing outliers and the starting and ending points are determined.

The input EM signal contains unintentional EM emanations that are amplitude modulated (AM) to the periodic signals present on the board [34]. Among these modulations, the one around the first harmonic of the operating clock frequency is the strongest and the most informative. Therefore, the input EM signal is firstly down-converted with clock frequency, and then, low-pass filtered to reduce the measurement noise at higher frequencies. However, interference due to non-program activity related periodic activities such as voltage regulators, memory refresh, etc. might still occur within the filtered bandwidth. Some of these interference are so strong that they dominate the EM signature. To eliminate them, we identify and store the frequencies at which these peaks occur from a recording that is collected during idle state of the board. Next, we remove these peaks in the "interference removing" step by using band-stop filters centered around these interference frequencies. Since the modulation around the clock frequency is not necessarily conjugate symmetric in the frequency domain, the resulting signal can be complex-valued. The amplitude of this complex-valued signal contains the shape information whereas the phase carries time shift information. Since our objective is to determine the shape, we proceed the data processing by only keeping the magnitude. Finally, the processed EM signal is cropped into chunks by using the cropping points obtained from input I/O signal, and the EM signatures are generated. These signatures represent the EM waveforms generated from the repetition of instructions $N$ times.

### 2.5 Generating Correlation Matrix

In this part, we generate a correlation matrix that represents the similarity between the generated EM signatures. The degree of (dis)similarity between two time domain signals can be measured in several ways: L1 norm, L2 norm, and cross-correlation etc. The power consumed by the devices fluctuates during run-time and this creates a DC offset in the measured EM emanations. Since L1 norm and L2 norm measure the sum of point-wise distances, it is an error-prone measure in the presence of DC offsets. Therefore, these distance measures are not suitable without normalization. On the other hand, cross-correlation measures the similarity of the waveforms, which is a suitable similarity metric for our purposes. The execution of some instructions takes longer than the others and the corresponding EM signatures are longer in length. To account for different lengths, while

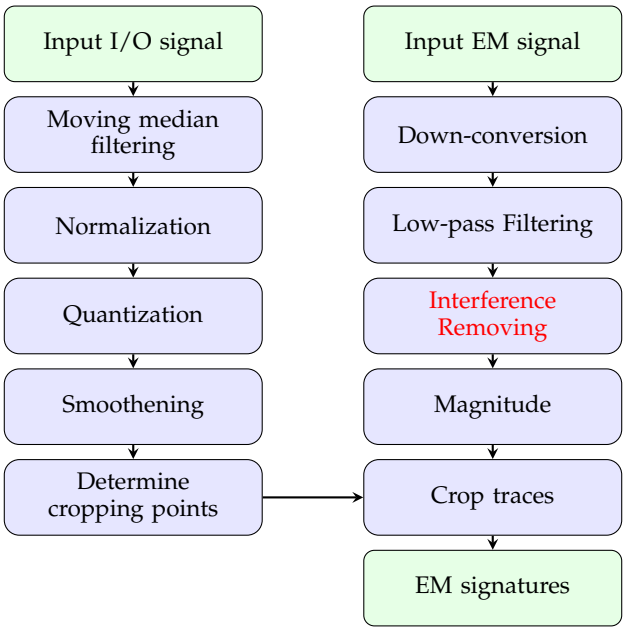



Fig. 3. Flowchart of data processing.

correlating two signals, the correlation is performed by sweeping the longer waveform with the shorter one and the highest correlation is denoted as the cross-correlation of these two waveforms. Cross-correlation is performed for all EM signature pairs to generate the correlation matrix.

### 2.6 Identifying Instruction Types

The objective of this step is to find the groups of instructions that have similar EM signatures. We refer to these distinct groups that have similar EM signatures as *instruction types*. The correlation matrix of the EM signatures shows how much the signatures are correlating with each other. A subjective method to find the *instruction types* from the correlation matrix is visual inspection. However, this approach is prone to misclassifications due to its subjective nature. As an objective method, we propose to utilize *hierarchical (agglomerative) clustering*. This clustering technique is a bottom-up algorithm that starts with treating each sample as a separate cluster and merges these clusters pair-wise until all samples are merged into a single cluster [35]. As the clusters are successively merged, a *cluster tree* (*dendrogram*), which is sequence of clusterings that partition the dataset, is generated [36]. Unlike other clustering algorithms such as *K-means*, this method does not require a random centroid initialization or a prior cluster number. This is especially useful in our application since we do not have an apriori knowledge about the number of *instruction types*. We can decide for the number of clusters by observing the *dendrogram*. The main disadvantage of this algorithm is its large time complexity ($O(N^2 \log(N))$), and space complexity ($O(N^2)$) [37], [38]. Since the number of instructions under investigation is typically not a large number, the dataset size is relatively small, therefore, the time and space costs are affordable. The output of the correlation matrix presents the similarity measure between the signatures. However, *hierarchical clustering* is based on the dissimilarities between samples. Therefore, by using [39], we convert the cross-correlation values, $\rho$, to distance values, $d$, as follows,

$$d = \sqrt{2(1-\rho)}. \tag{1}$$

## 3 Detecting Permutations of Instruction Types

This section explains a systematic way of tracking the execution of *instruction types*. In Section 2, *instruction types* are identified by repeating the same instruction for several times and clustering them based on the similarity of their EM signatures. Next step is to detect these *instruction types* in a testing scenario. The most straightforward way to do so is to find the EM signature of the *instruction types* when they are executed once (instead of $N$ times), and use these signatures as a dictionary while testing. However, this approach has two major drawbacks:

1) Most processors implement pipeline architecture where the execution of the instructions is divided into different stages. This allows for overlapping executions of consecutive instructions at different stages. Since all these stages utilize transistor switching activity during their operation, all stages behave as an EM emanation source. Therefore, the measured EM emanation by the antenna is a combination of the EM waves radiated from different stages. Due to the lack of a complex model to decouple these combinations, it is not possible to isolate the EM signature of a given instruction. Consequently, difference generated by a single-instruction becomes significantly smaller than the difference generated by instructions surrounding it [33].
2) Tracking a single instruction requires several samples per clock cycle and perfect cropping of the start and end points. Satisfying required sampling rate becomes costly for devices with high clock frequencies. Moreover, misalignment of the cropping points leads to significant loss in tracking performance.

Previous work in literature suggests to generate EM signatures for *instruction sequences* rather than *single instructions* and reports high self-correlation and low cross-correlation values for several instruction sequences to show the applicability of the proposed system [28]. However, the instruction sequences used in this work are relatively long and the choice of these instruction sequences is not systematic.

Instructions in a program appear in different orders. We propose to generate the sequences in a systematic way by generating all possible orderings of the *instruction types*. In other words, we propose to generate EM signatures for all permutations of the *instruction types* and track these permutations. Note that this approach addresses the aforementioned problems for the following reasons.

1) By generating EM signatures for the permutations, we observe the overall effect of the permutation block. Certain interactions caused by different orderings of these instructions and the impact of the pipeline are embedded into the EM signature. Although it is not possible to isolate the impact generated by each of these instructions, we obtain

```
while true
  for
   %empty for loop
  end

  % Set I/O Pin to High

  for N times
   -First instruction in permutation order,
   -Second instruction in permutation order,
   .
   .
   .
   -Kth instruction in permutation order.
  end

  % Reset I/O Pin to Low

  for
   %empty for loop
  end
 end
```

Fig. 4. Pseudo-code for Permuation Detection setup.

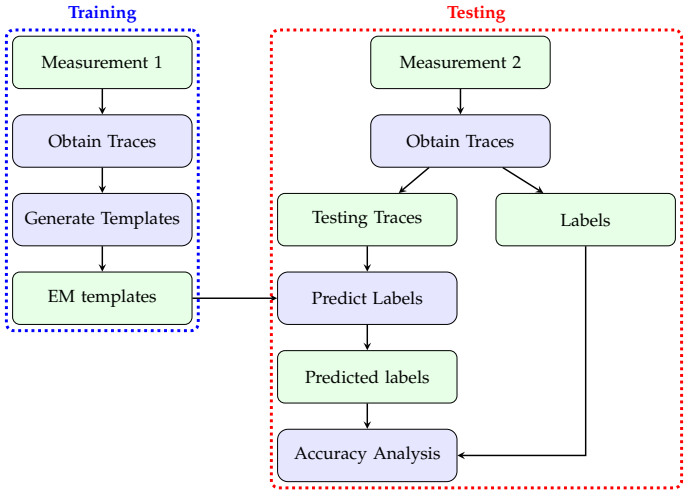



Fig. 5. Overview of Training and Testing.

an EM signature that covers their aggregate impact. Furthermore, by using permutations instead of single instructions, we have a longer waveform and the significance of the surrounding instructions becomes less dominant.

2) Having a sequence of instructions results in a longer EM waveform, therefore, we can afford to have less samples per clock cycle compared to the single instruction case. Furthermore, the impact of miscroppings becomes less evident since the ratio of miscropping length to the total EM signal length is smaller than the single instruction case.

One should keep in mind that inclusion of permutations is very helpful to address the pipeline effect but there are a few issues that cannot be addressed with this scheme:

- The EM signature still experiences the pipeline impact in the beginning and at the end due to the instructions that come before and after the permutation, respectively.
- For some processors, the pipeline length might be longer than the length of the permutation, and this limits the capability of the permutation to represent the emanation coming from all pipeline stages.

Please note that, in addition to pipeline structure, there are other computer architecture factors such as cache access, interrupts, compiler optimization, etc. that impact the EM emanations. Cache accesses and interrupts have much larger signatures [40] than single instructions and they need to be detected and removed as shown in [40] before single instruction tracking is applied. However, since our benchmark codes are relatively short, we do not experience these impacts and do not need to remove them. Furthermore, we keep all compiler optimization turned off to make sure that the compiled machine code matches the pseudo-codes.

Finally, note that the number of the permutations is given by $K!$, where $K$ is the number of clusters or *instruction types* that are obtained in Section 2.6. For large $K$, the number of permutations becomes a large number which leads to a costly measurement and large memory usage. Therefore, the choice of $K$ from the *cluster tree* should be made carefully.

The steps of detecting permutations of *instruction types* are explained step-by-step as follows.

### 3.1 Picking an Instruction to Represent Each Instruction Type

Since the instructions from the same *instruction type* have similar EM signatures, the most common instruction from each *type* is chosen to represent its *type*. *Instruction types* are labeled with the first $K$ capital letters of the alphabet.

### 3.2 Generating Microbenchmarks for Permutations

As mentioned earlier, with $K$ clusters, we need to generate $K!$ benchmarks that include all permutations. For example, if there are 4 identified *instruction types* (A, B, C and D), the permutations should include: $ABCD, ABDC, \ldots, DCBA$.

Execution of most programs and embedded systems go through loops during the operation. These loops include execution of the same instruction sequences several times. Furthermore, a program spends most of the execution time in functions that are called many times successively. Considering this repetition-based nature of the most programs, we propose to investigate the impact of repetition of instruction blocks on the tracking performance. In particular, we create EM signatures for different repetitions of the same instruction block. For ease of reference, we use the notation $(ABCD)_N$, where $(ABCD)$ is the investigated permutation block, and $N$ is the number of permutation block repetition. Note that the ultimate goal is to track permutation blocks with $N = 1$. However, due to the repetitive structure of code implementations, $N$ could be different than 1. For example, a certain permutation might appear within a loop that is repeated several times and this apriori knowledge can be used to improve tracking performance.

The microbenchmark structure for the permutations is very similar to the structure presented in Section 2.2 with a slight difference that the instruction under interest is swapped with the permutation block under interest. The pseudo-code for the new microbenchmark structure is shown in Fig. 4.

### 3.3 Implementing Code and Recording EM Emanations

After implementing the pseudo-codes, EM emanations from the device is measured as described in Section 2.3. Two measurements are taken for each microbenchmark at different times. The waveforms obtained in the first measurement are used for *training*, whereas the second measurement waveforms are used for *testing*.

### 3.4 Training: Generating Templates for Each Permutation

In this part, we generate templates for each permutation and these templates are used in the testing phase for prediction. A general overview of training can be found in the left hand part of Fig. 5. By using the same procedure that is described in Section 2.4, several EM signature traces are obtained for each permutation from *Measurement 1* recordings. These traces from the same permutation are aligned using cross-correlation. Then, they are cropped so that all of them have the same length. Finally, the EM template for the corresponding permutation is generated by using the point-wise average of the aligned and cropped EM signature traces.

### 3.5 Testing: Predicting Testing Measurements using Templates

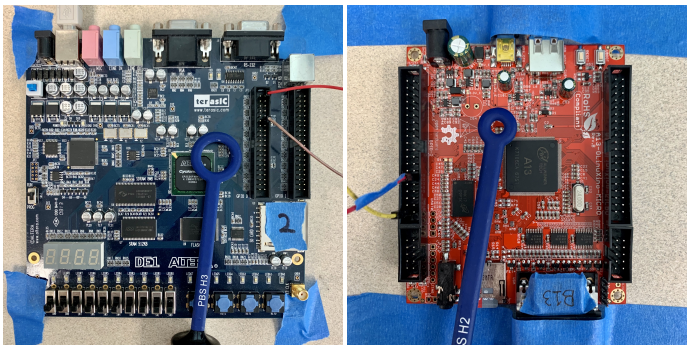
A general overview of testing can be found in the right hand part of Fig. 5. Testing traces are obtained by applying the procedure described in Section 2.4 to *Measurement 2* recordings. These testing traces are labeled with their permutation order. The prediction step implements a matched-filter-like structure where the filters are the normalized versions of EM templates generated in Section 3.4 [41]. We use such a structure because matched filter is the optimum receiver in terms of maximizing the signal-to-noise ratio (SNR) when the received signal is corrupted by additive random noise [42]. Another advantage of such a structure is that the matched filter finds the best offset between the received signal and the templates on its own without requiring synchronization. After correlating the testing trace with all filters, the permutation of the trace is predicted as the template whose corresponding filter gives the highest correlation.

## 4 Experimental Results

In this section, we explain our experimental setup and provide the results for *instruction type* determination and permutation tracking of *instruction types*.

### 4.1 Experimental Setup

To demonstrate the feasibility of the proposed methodology, we experiment on two devices. The first device is Intel's DE1 Altera FPGA Board that has Altera NIOS-II (soft) processor [31]. This processor is a general purpose RISC (reduced instruction set computer) processor that implements Nios-II architecture with 6 pipeline stages. The operating clock frequency is 50 MHz and this board does not have a present operating system. The second device is A13-OLinuXino, which is a low-cost embedded Linux mini-computer that has ARM Cortex A8 processor that operates at 1 GHz clock frequency [32]. The processor implements ARMv7-A architecture and is an in-order, dual-issue, superscalar



(a) DE1 Device. (b) A13 Device.

Fig. 6. Expertimental setups used for EM emanation recordings.

microprocessor with 13-stage main integer pipeline. For the rest of this paper, we refer to the first and second devices as the DE1 device, and the A13 device, respectively.

To record the EM emanations, we use Aaronia's H3 near-field magnetic probe for the DE1 device and H2 near-field magnetic probe for the A13 device [43]. These probes are chosen so that the resonance frequency of the probes are aligned with the operating clock frequencies of the devices. We locate the probes on the pin edges of the processors as shown in Fig. 6 using the results presented in [44]. Also note that two GPIO pins of these devices are utilized to record the marker signal. Measurements are obtained with Keysight's DSOS804A high-definition oscilloscope at 10 GHz sampling rate [45].

### 4.2 Instruction Type Determination Results

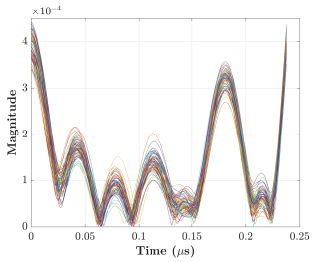
After inspecting the instruction set for both devices, we select 17 instructions per each device for investigation that are listed in Table 1 with their corresponding abbreviations. Note that these are frequently used instructions including mathematical operations such as addition and multiplication; logical operations such as AND, OR; and memory access instructions such as load and store, etc.

After selecting the instructions, we generate microbenchmarks for each of them that include $N = 10$ and $N = 100$ times repetitions for the DE1 and A13 devices, respectively. The reason for the higher $N$ value of the A13 device microbenchmarks is the higher operating clock frequency of the device. Next, we record the EM emanations and obtain the EM signatures for different instructions as described in Section 2.4. Fig. 7 and Fig. 8 present examples of the obtained EM signatures of different instructions for the DE1 and A13 devices, respectively. Note that each subfigure plots several EM emanation traces obtained for different executions of the given instruction. One can easily observe that these traces are highly aligned indicating that the EM signatures of the instruction sequences are relatively similar for successive executions.

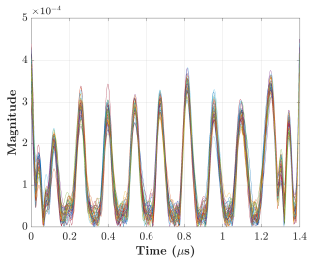
In Fig. 7, we observe that some instructions such as LDW and MULI have significantly different EM signatures that differ both in length and shape, whereas some instructions such as ADD and SUB have very similar EM signatures in both length and shape. Similar conclusions can be obtained for Fig. 8, as well. One should note that these conclusions validate our initial assumption that some instructions

TABLE 1
Investigated Instructions

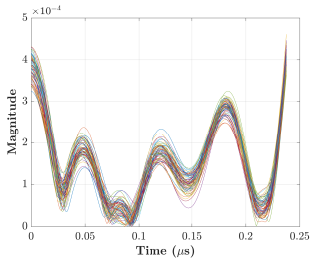
| DE1 Device | | | A13 Device | | |
|---|---|---|---|---|---|
| | Assembly Code | Description | | Assembly Code | Description |
| **ADDI** | `addi r20, r20,171` | Add immediate value to register | **ADDI** | `add r3, r3, #1` | Add immediate value to register |
| **ADD** | `add r20, r20, r16` | Add two register values | **ADD** | `add r3, r3, r3` | Add two register values |
| **SUBI** | `subi r20, r20,173` | Subtract immediate value from register value | **SUBI** | `sub r3, r3, #1` | Subtract immediate value from register value |
| **SUB** | `sub r20, r20, r16` | Subtract two register values | **SUB** | `sub r3, r3, r3` | Subtract two register value |
| **LDW** | `ldw r20, 0(r21)` | Load 32-bit word from memory to register | **LDR** | `ldr r3, [r1]` | Load a word from memory to register |
| **STW** | `stw r20, 0(r21)` | Store word from register to memory | **STR** | `str r3, [r1]` | Store a word from register to memory |
| **MUL** | `mul r20, r20, r16` | Multiply two register values | **MUL** | `mul r3, r3, r3` | Multiply two register values |
| **MULI** | `muli r20, r20, 173` | Multiply immediate value with register value | **SMULL** | `smull r3, r2, r3, r3` | Multiply two 32-bit signed values |
| **DIV** | `div r20, r20, r20` | Divide two register values | **UMULL** | `umull r3, r2, r3, r3` | Multiply two 32-bit unsigned values |
| **OR** | `or r20, r20, r16` | Bitwise logical OR operation of register values | **ORR** | `orr r3, r3, r2` | Bitwise logical OR operation of register values |
| **ORI** | `ori r20, r20, 173` | Bitwise logical OR operation of register and immediate values | **ORRI** | `orr r3, r3, #1` | Bitwise logical OR operation of register and immediate values |
| **MOVI** | `movi r20, 173` | Move immediate value to register | **MOVI** | `mov r3, #1` | Move immediate value to register |
| **MOV** | `movi r20, r16` | Move register value to register | **ANDI** | `and r3, r3, #1` | Bitwise logical AND operation of register and immediate values |
| **AND** | `and r20, r20, r20` | Bitwise logical AND operation of register values | **AND** | `and r3, r3, r2` | Bitwise logical AND operation of register values |
| **CMPEQ** | `cmpeq r20, r20, r16` | Compare if register values are equal | **CMP** | `cmp r3, r0` | Subtract register values to compare |
| **XOR** | `xor r20, r20, r16` | Bitwise logical XOR operation of register values | **CMPI** | `cmpi r3, #1` | Subtract immediate value from register value to compare |
| **XORI** | `xori r20, r20, 173` | Bitwise logical XOR operation of register and immediate values | **NOP** | `nop` | No operation |



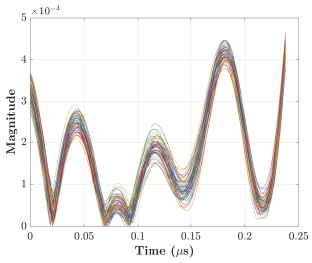

(a) LDW




(b) MULI

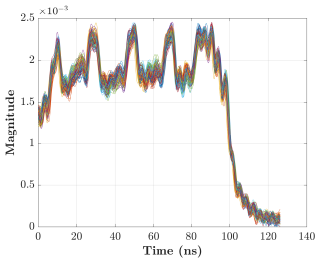



(c) ADD

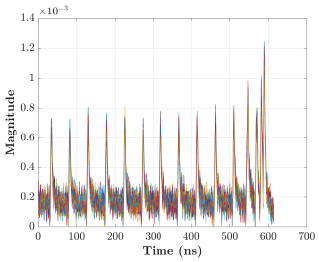



(d) SUB

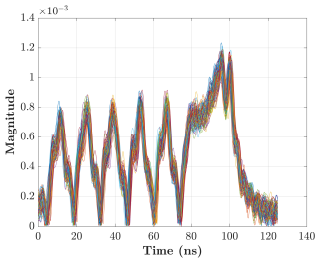
Fig. 7. Obtained EM signatures of several instructions for the DE1 device.



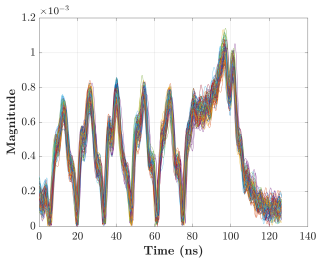

(a) LDR




(b) MUL




(c) ADDI



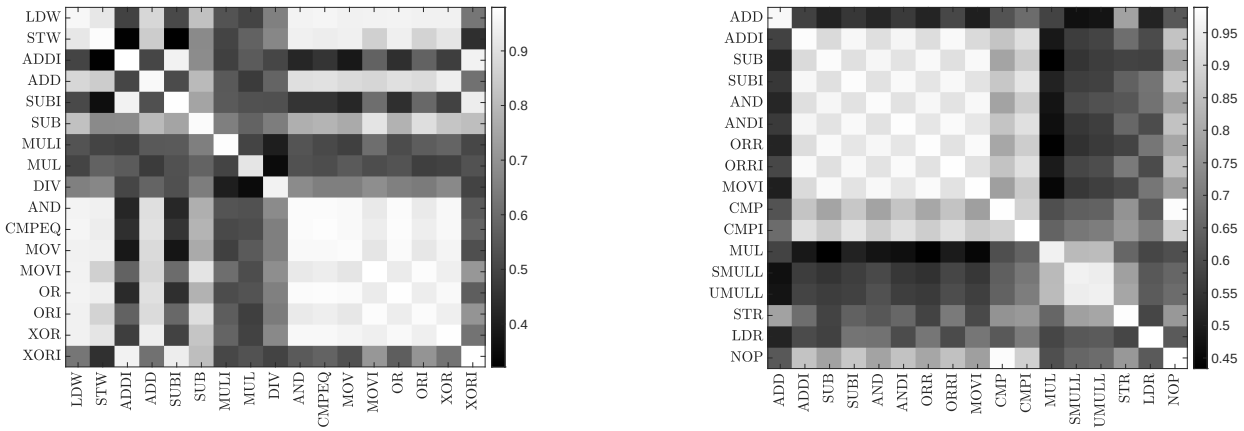

(d) SUBI

Fig. 8. Obtained EM signatures of several instructions for the A13 device.

have similar EM signatures while some have significantly different EM signatures. Therefore, our methodology also provides the capability of decreasing the entropy of the instructions. We also realize that the instructions with similar EM signatures are those that have similar physical implementations such as addition and subtraction.

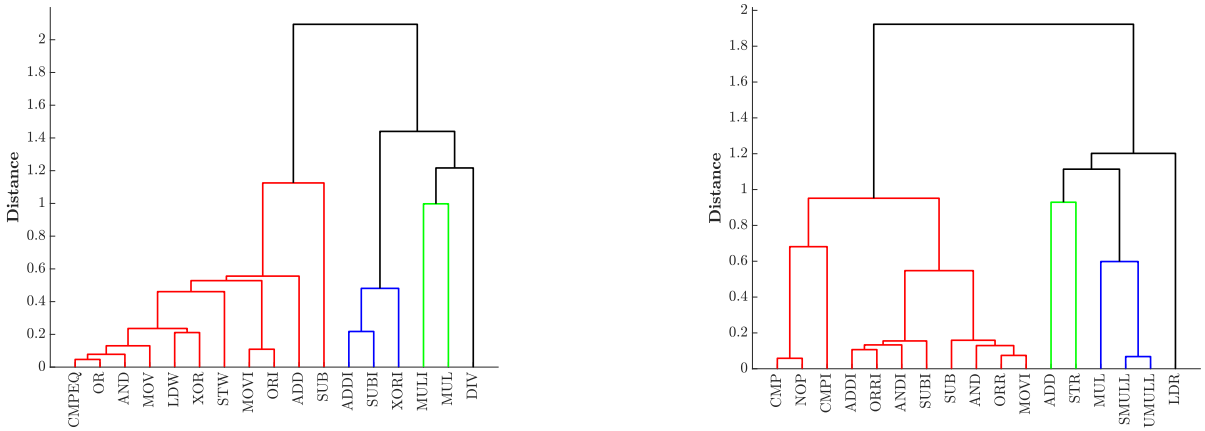
As can be seen in Fig. 8, the EM signature of LDW is significantly different than MUL, ADDI and SUBI. This difference reflects the significance of the additional memory system pipeline present in the A13 device for memory access instructions. Although the EM signatures are obtained by repeating the same instruction $N$ times, we are not observing the repetition of the same pattern $N$ times in the EM signature. We note that the beginning and ending parts of the signatures are significantly different than the middle parts, where we can observe the same kind of pattern repetition. This observation emphasizes the significance of the pipeline effect caused by the instructions that come before and after the instruction sequence. From these observations, we conclude that the EM signatures are indicative of the pipeline structure. These observations also prove that each stage of a pipeline emits EM signals while executing instructions.

After obtaining the EM signatures, we generate the correlation matrices for the DE1 and A13 devices as shown in Fig. 9. In Fig. 9, brighter colors indicate higher correlation, hence higher similarity. Then, we convert this correlation




(a) DE1 Device (b) A13 Device

Fig. 9. Correlation matrices for DE1 and A13 devices.




(a) DE1 Device (b) A13 Device

Fig. 10. Dendrogram of the instructions obtained with Hierarchical Clustering for DE1 and A13 devices. Different colors represent the clusters.

matrix to a distance matrix and cluster these instructions with *average-link hierarchical clustering* described in Section 2.6. The resulting *dendrograms* are presented in Fig. 10. The bottom part of the *dendrograms* starts with all instructions, and as it goes to the top, these instructions are merged pairwise so that they are in the same cluster until all instructions are merged at the top. By investigating the correlation matrices and *dendrograms*, we set the number of clusters, $K$, to be 4 in both cases as for both cases when $K$ is set to 4, resulting clusters include instructions that are similar in operation such as MULI and MUL. The clusters are indicated with different branch colors. Red, blue, green and black colors represent A, B, C and D type clusters, respectively.

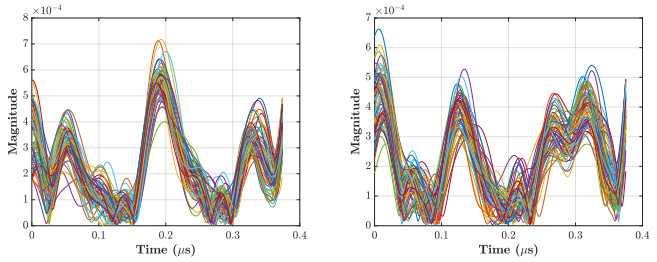
For DE1 device, A-type instructions are memory-access operations (LDW, STW) as well as the 1 clock-cycle arithmetic and logic operations that use register sources (ADD, SUB, AND, CMPEQ, MOV, OR, XOR), there are also two exceptions (MOVI, ORI) using immediate values. B-type instructions are the arithmetic and logic operations that use immediate values (ADDI, SUBI and XORI). C-type instructions are the multiplication operations (MULI and MUL) and D-type instructions are the division operation (DIV).

For A13 device, A-type instructions include all 1 clock-cycle arithmetic and logic operations that are either using register values or immediate values. One exception is the ADD instruction that uses register values, which is clustered as a C-type instruction along with the store instruction (STR). B-type instructions are the multiplication operations (MUL, SMULL, UMULL), and D-type instructions are the load operation (LDR).

As it has been discussed earlier, the clusters reflect the structure of the pipeline. For example, load and store operations include the memory address location calculations which are addition and subtraction operations. Therefore, these instructions have the same type as addition and subtraction for the DE1 device. However, A13 implements an external memory-access pipeline, which results in separate clusters for load and store operations. Similarly, we note that different variants of multiplication operations are clustered in the same group for both devices.

### 4.3 Permutation Tracking Results

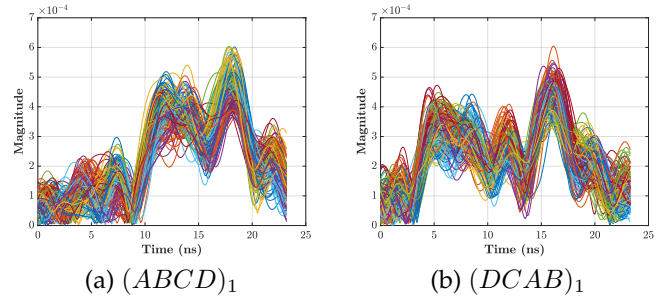
After determining the *instruction types*, to represent the entire type, we pick an instruction from each type that we encounter the most, but choosing them randomly from each




(a) $(ABCD)_1$ (b) $(DCBA)_1$

(c) $(ABCD)_{10}$ (d) $(DCBA)_{10}$

Fig. 11. Sample EM signatures of permutations with different $N$ values for the DE1 device.




(a) $(ABCD)_1$ (b) $(DCAB)_1$

(c) $(ABCD)_{10}$ (d) $(DCAB)_{10}$

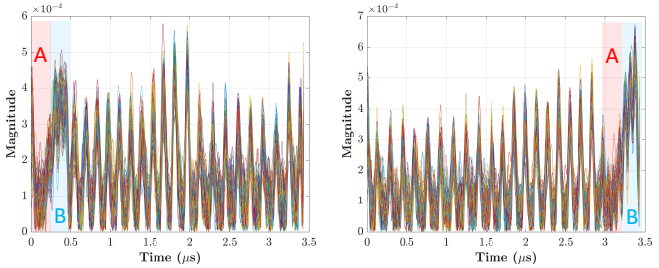
Fig. 12. Sample EM signatures of permutations with different $N$ values for the A13 device.

type leads to the same results as well. These instructions are LDW, ADDI, MULI and DIV for the DE1 device, and ADDI, MUL, STR and LDR for the A13 device representing A, B, C and D in the given order. Using these instructions, we generate microbenchmarks, record EM emanations and generate EM signatures for different $N$ values.

Fig. 11 and Fig. 12 present example EM signatures for different permutations and $N$ values obtained from the DE1 and A13 devices. For $N = 1$, the plotted EM signatures of the permutations $(ABCD)_1$ and $(DCBA)_1$ are visually different from each other for both devices. However, we note that, when $N = 1$, the EM signatures for the A13 device has a large variance among different executions, whereas this variance is much smaller for $N = 10$.

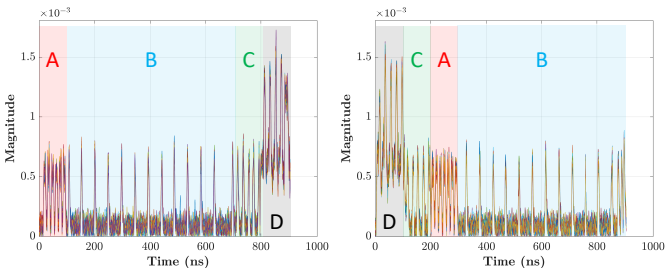
In Fig. 11, when $N = 10$, we can observe a pattern that is repeated 10 times, but this repetition is not present in Fig. 12 when $N = 10$. This difference can be explained by the difference of the pipeline lengths: DE1's pipeline length (6 stages) is shorter than A13's pipeline length (13 stages). Therefore, a permutation block of length 4 instructions is more capable including the contributions from the pipeline stages of the DE1 device and get the EM signature into a steady state, whereas the EM signature for A13 does not reach the steady state with 10 repetitions.

Note that for both devices, we cannot visually identify the locations of the A, B, C, and D *instruction types* from the EM signatures. As discussed earlier, this is because single execution of the instruction does not create significant variation in the signature and due to the pipeline, the variation generated by execution of single instruction is distributed to different stages of the pipeline. To test this, while executing the permutation only once $(N = 1)$, we repeat each instruction within the block 10 and 100 times for the DE1 and A13 devices, respectively. The EM signatures obtained with this repetition are shown in Fig. 13 and Fig. 14. In Fig. 13, we see that the instruction blocks A and B that appear in the beginning of ABCD permutation can be identified at the end of DCAB permutation. Note that identifying C and D precisely is still not possible. In Fig. 14, we can identify all instruction type blocks clearly as indicated in the figure. These results show that, although single execution of an instruction does not generate an identifiable waveform pattern, several consecutive executions can result in distinct waveforms. Please note that this is just an observation and cannot be directly used for testing purposes because enforcing the repetition of the same instruction within the blocks



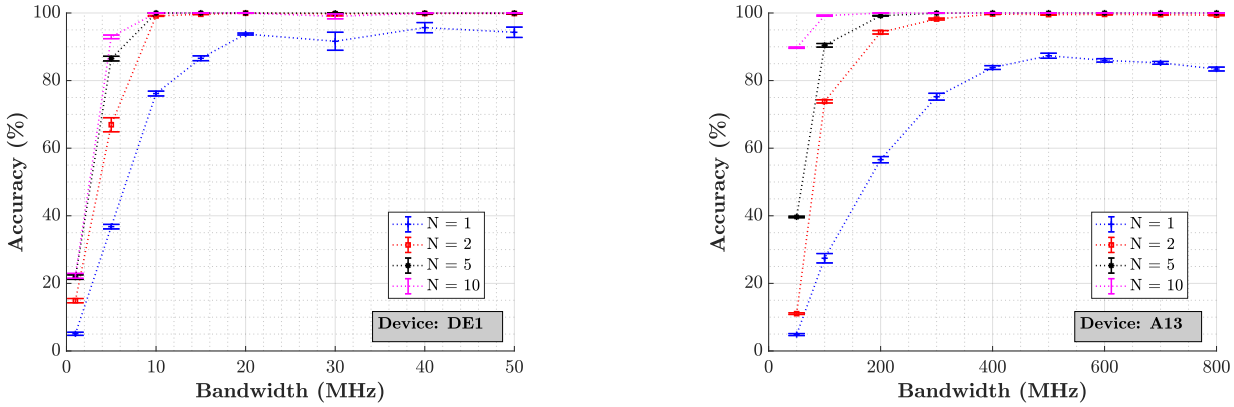

(a) $(A_{10}B_{10}C_{10}D_{10})_1$ (b) $(D_{10}C_{10}A_{10}B_{10})_1$

Fig. 13. Sample EM signatures when instruction types are repeated 10 times within the permutation block for the DE1 device.



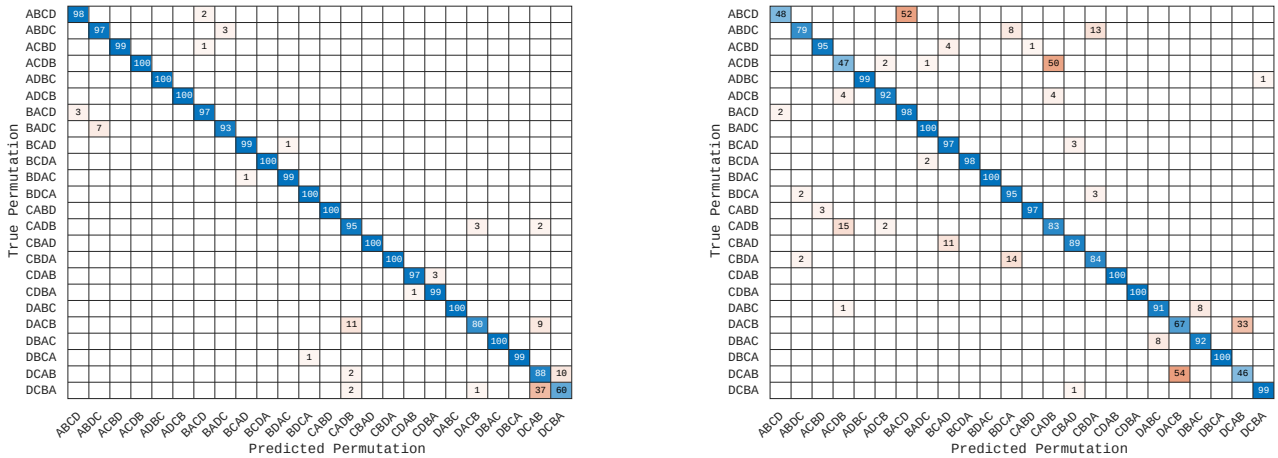

(a) $(A_{100}B_{100}C_{100}D_{10})_1$ (b) $(D_{100}C_{100}A_{100}B_{100})_1$

Fig. 14. Sample EM signatures when instruction types are repeated 100 times within the permutation block for the A13 device.




(a) DE1 Device. (b) A13 Device

Fig. 15. Impact of $N$ value (number of permutation block repetition) on accuracy.




(a) DE1 Device. (b) A13 Device

Fig. 16. Confusion charts for $N = 1$.

is not very realistic for many programs and, therefore, is not practically applicable.

We generate templates for each permutation using *Measurement 1*, and predict the permutation order of the snippets obtained from *Measurement* 2. To evaluate the deviation of our classification for different test sets, we perform 10 independent tests for each configuration. For each test, we use 100 snippets per each permutation, resulting in a total of 2400 testing snippets per test, and 24000 testing snippets for all tests.

"Bandwidth" is a *hyperparameter* in our application, which needs to be selected while training and before testing. One common approach to select *hyperparameter* values is *cross-validation*, where the training dataset is partitioned into training and validation sets, and the best *hyperparameter* values are selected based on the performance on the validation set. Note that the testing set is kept completely independent during this process. Although *cross-validation* is a valid approach for our scheme to set the "bandwidth" value, we aim to analyze the impact of the utilized low-pass filter bandwidth on the accuracy. Therefore, instead of setting "bandwidth" value using *cross-validation*, we report the accuracy for different bandwidth values as shown in Fig. 15. Accuracy is calculated as the ratio of number of the correctly classified permutations to the total number of testing traces.

The value for every point in Fig. 15 is obtained by averaging 10 independent test trials and the error bar around the marker indicates the standard deviation across these trials. Since the standard deviation for different trials is low, most of these points are concentrated around the average value.

In Fig. 15, we observe that the highest accuracies for $N = 1$ are 95.67% and 87.35% for DE1 and A13, respectively. For higher values of $N$, the accuracy significantly improves. We also note that for $N > 1$, accuracy increases for increasing bandwidth and converges to 100% beyond 10 MHz and 400 MHz for DE1 and A13 devices, respectively.

To investigate $N = 1$ case, we provide the confusion matrices for a single test in Fig. 16. Note that both matrices are mostly diagonally dominant. When we investigate the correlation matrix for the DE1 device, we realize that the permutations "DCAB" and "DCBA", which only differ by the order of A and B instructions, are the two classes that are mostly confused with each other. Similarly, for A13

device, "ABCD" is confused with "BACD", "ACDB" is confused with "CADB", "DACB" is confused with "DCAB". All these confusions are between two classes that differ by the ordering of only two instructions whereas the order and the location of the other two instructions are the same. This is expected as the EM signature of such classes that differ by only two instructions are similar to each other than EM signatures that differ by the ordering of all instructions.

A single increment in $N$ from 1 to 2 significantly improves the accuracy to 100%. This shows the trade-off between the number of repetitions and the accuracy. Including repeated versions of the permutation blocks is significantly increasing the detection accuracy in the expense of limiting the applicability of the proposed method. Therefore, if the system or program under investigation has repetitive nature, this method can be modified for better performance.

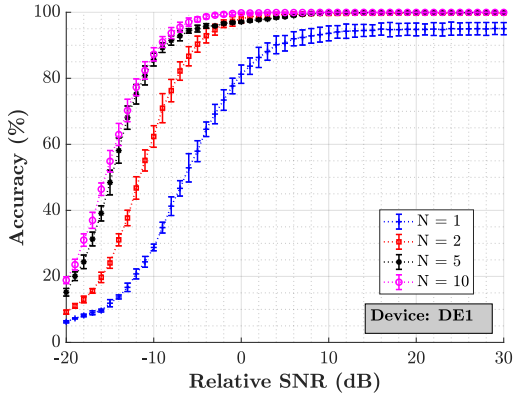
As discussed in Section 1, related works implement their framework on processors with low clock frequencies and simple pipeline architectures. [18] implements a Hidden Markov Models based recognition system using power side channels and achieves 70% recognition rate for a microcontroller with two pipeline stages and 1 MHz clock frequency. [19] uses power side channels and improves the recognition rate up to 100% for a microcontroller with 2 pipeline stages and 4 MHz clock frequency. [29] also uses power side channels and achieves 99% accuracy for a microcontroller with single-level pipelining and 16 MHz clock frequency. [30] utilizes EM-side channel and achieves 96.24% recognition rate for a microcontroller with 4 MHz clock frequency by decapsulating the integrated circuit with fuming nitric acid. Note that our framework achieves similar or better accuracies even for devices that have much higher clock frequencies and relatively sophisticated pipeline architectures, without any decapsulation procedure.

## 5 Further Evaluation of Robustness

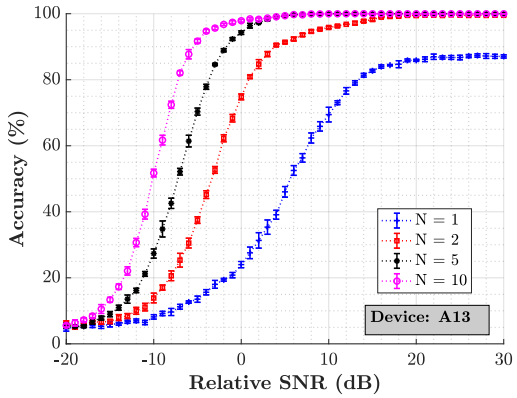
In this section, we test our methodology's robustness and extensions. Firstly, we evaluate the performance for different SNR levels. Next, we extend the tracking ability to *instructions* rather than *instruction types*. Finally, we demonstrate the *fine-grained* malware detection ability of the proposed framework.

### 5.1 Performance for Different SNR Levels

To evaluate the performance for different SNR levels, we assume that the measurement channel is corrupted by additive white Gaussian noise (AWGN). To estimate the SNR of the measured signal, we need to estimate the noise floor. One simple approach to measure the noise floor is to record the EM emanations during the idle state. However, this is not a valid approach because the power consumption during the idle state is lower than program execution due to the power optimizations. Lower power consumption leads to weaker EM emanations, which disqualifies this approach for SNR estimation. Therefore, we obtain measurements at locations with strongest EM emanations and use the power of the signals obtained from these measurements as the referenced signal power, which includes both the signal and the measurement noise. After obtaining these traces, we introduce additional additive white Gaussian noise with different noise powers to the testing traces. Since we are




(a) DE1 Device.




(b) A13 Device.

Fig. 17. Impact of relative SNR on accuracy for permutations of different instructions for different $N$ values.

introducing additional noise to the signal that is already corrupted by measurement noise, we refer to the ratio between the measured signal power and this additional noise power as *relative SNR*.

Let $\mathbf{x}$ be a vector representing a testing trace with a length of $L$ samples, and $x_i$ be the sample values of this trace such that $\mathbf{x} = [x_1 \quad x_2 \quad \dots \quad x_L]$. To add the noise, we first calculate $P_S$, the signal power of the vector, as

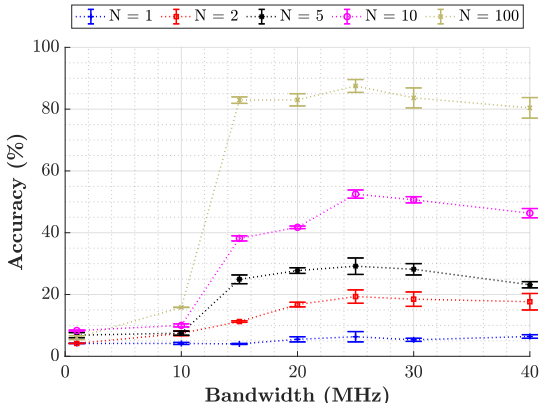
$$P_s = \frac{1}{L}\sum_{i=1}^{L} |x_i|^2. \tag{2}$$

Note that $P_s$ is the power of the measured signal and, therefore, it features contributions from both the signal itself and the measurement noise. Then, we add AWGN to each sample of $\mathbf{x}$ and obtain the new signal $\mathbf{y}$ that is corrupted by additional AWGN. The elements of $\mathbf{y}$, $y_i$, are obtained as

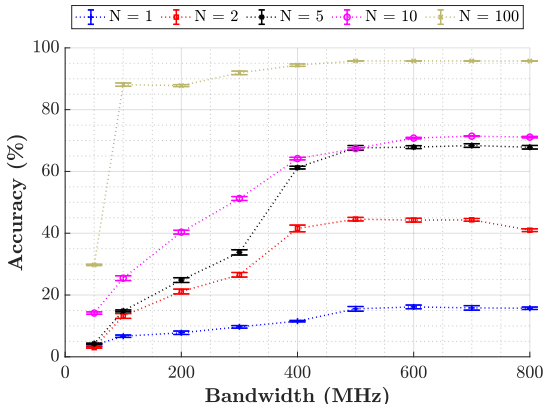
$$y_i = x_i + z_i \quad \text{for } 1 \le i \le L, \tag{3}$$

where $z_i \sim \mathcal{N}(0, \frac{P_s}{SNR_{lin}^{rel}})$ is independent and identically distributed realizations of a random variable that has a zero-mean Gaussian (normal) distribution with $\frac{P_s}{SNR_{lin}^{rel}}$ variance, and $SNR_{lin}^{rel}$ represents the relative SNR in linear scale.

One should note that we add AWGN to all testing traces, but the EM templates are kept as originals. Then, we perform the same testing procedure for different relative SNR values with 40 MHz (DE1) and 500 MHz (A13) low-pass




(a) DE1 Device.




(b) A13 Device.

Fig. 18. Impact of $N$ value on accuracy for permutations of instructions from the same instruction type.

filter bandwidths. The results are shown in Fig. 17, where we observe that the accuracy decreases for low relative SNR levels, as expected. Accuracy performance converges for relative SNR values higher than 15 dB. We observe that the permutations with larger $N$ are generally more resistant to AWGN. Finally, we note that DE1 device is more resistant to additional AWGN. We believe that the reasoning behind this is the relative simplicity of the DE1 device compared to A13, which implements a deeper pipeline and runs an operating system in the background.

### 5.2 Permutations of Instructions from the Same *Instruction Type*

In this section, our objective is to extend the applicability of the permutation tracking from tracking different *instruction types* to tracking *instructions* of the same *instruction type*. To do so, we pick 4 instructions from *instruction type* A for both devices. For the DE1, we pick ADD, AND, MOV and CMPEQ; for the A13, we pick ADD, AND, MOV and CMP. Note that these instructions use the same destination registers, register sources and immediate values, therefore we minimize the variation that might be caused by data values or register differences. Using these 4 instructions, we repeat the experiments and report the accuracies in Fig. 18. Note that, the accuracies for $N = 1$ are low because these instructions are from the same type and their permutations have very similar shapes. This low accuracy verifies that the instructions are clustered correctly.

If we can afford to repeat the permutation sequence for several times, i.e, increase $N$, accuracy successively increases. In fact, for $N = 100$, accuracy for the DE1 and A13 devices get as high as $87.5\%$ and $95.78\%$, which are relatively high accuracies considering the similarity of the instruction signatures. To be able to determine the differences between the instructions from the same cluster, we need more observations. This points to the trade-off between better instruction resolution and number of observations.

TABLE 2
Investigated Malware Scenarios for "ABCD" permutation

| Malware Type | Corresponding Instruction Sequences |
|---|---|
| Single Deletion | BCD, ACD, ABD, ABC |
| Double Deletion | CD, BD, BC, AD, AC, AB |
| One Insertion | AABCD, ABBCD, ABCCD, ABCDD |
| Two Insertions | AAABCD, ABBBCD, ABCCCD, ABCDDD |
| Three Insertions | AAAABCD, ABBBBCD, ABCCCCD, ABCDDDD |
| One substitution | BBCD, CBCD, DBCD, AACD, ACCD, ADCD, ABAD, ABBD, ABDD, ABCA, ABCB, ABCC, |

### 5.3 Demonstrating Finer-Granularity Malware Detection Capability of PITEM

In this section, we demonstrate *fine-grained* malware detection capability of our framework for several scenarios. By *fine-granularity*, we refer to only one or a few instruction changes in the machine code such as deletion, insertion and substitution. For simplicity, we consider a scenario where the expected code in the program execution is the permutation "ABCD". We would like to get notified if the processor executes a different instruction sequence such as a single deletion like "ABC", a single insertion like "ABCCD", a single substitution like "ABCA", etc.

To do so, we follow a similar procedure as in Section 3 with a few modifications so that the task is a binary classification (malware detected or not). We generate EM signatures of all possible permutations in the training step. Additionally, we utilize a validation dataset, which is an independent set of measurements to determine a confidence threshold. We predict the permutation labels of the snippets in the validation set using the testing structure shown in Section 3.5 and store the highest correlation values of the matched filters. Using these highest correlation values, we determine the confidence threshold as the minimum correlation value that corresponds to a true classification. In the testing part, we collect EM signatures for modified versions of the "ABCD" permutation that are listed in Table 2. Next, we predict the permutation corresponding to these testing snippets. Finally, we alert malware if 1) the predicted permutation is other than "ABCD", or 2) the highest correlation for the snippet is below the confidence threshold.

We provide experimental results for the described procedure on DE1 device. From the validation set, we determine the confidence threshold to be 0.9560. For the malware scenarios listed in Table 2, a true classification means malware is alerted. We alert malware with 99.89% accuracy when one of the modified instructions listed in Table 2 is executed. In addition, we classify that there is no malware with 100% accuracy when "ABCD" is executed. These results demonstrate that PITEM can indeed be used for *fine-grained* malware detection applications.

## 6 Conclusion

This paper proposed PITEM, a new approach for instruction-level tracking using EM side channel. PITEM is a tool that can be used for *fine-grain* malware detection with an ability to locate the malware injection precisely. It first identifies groups of instructions, *instruction types*, that have similar EM signatures using hierarchical clustering. After identifying *instruction types*, during training phase, it generates templates for all possible permutations of these *instruction types*. In the testing phase, we obtain testing traces and predict the best matching template with a correlation based, matched-filter-like predictor. The proposed methodology is tested using two different devices, one FPGA-based processor and one ARM-based IoT device. The results are reported for different repetitions of the permutation blocks. For single execution of the permutation block, we obtain 95.67% and 87.35% accuracies for the two different testing devices. We observe that with only two repetitions of the permutation blocks, these accuracies significantly increase to 100%. Next, we evaluate the performance of the detection system under AWGN. We note that the performance of the system is stable for $> 15$ dB relative SNR and the performance gradually decreases for lower values of relative SNR. We also note that repeated permutation blocks are more resilient to AWGN. Then, we perform detection of the permutations from the same *instruction type*. Although the detection accuracy is low for single execution of the instructions within the block, the accuracy increases significantly when the instructions are repeated. Finally, we illustrate that PITEM can detect single or a few instruction deviations from the expected permutation sequence with 99.89% accuracy, which shows its suitability for *fine-grained* malware detection applications.

## Acknowledgments

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

## References

[1] E. A. Lee, "Cyber physical systems: Design challenges," in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2008, pp. 363–369.

[2] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *Design Automation Conference*, 2010, pp. 731–736.

[3] R. Baheti and H. Gill, "Cyber-physical systems," *The impact of control technology*, vol. 12, no. 1, pp. 161–166, 2011.

[4] D. Bandyopadhyay and J. Sen, "Internet of things: Applications and challenges in technology and standardization," *Wireless personal communications*, vol. 58, no. 1, pp. 49–69, 2011.

[5] H. Vincent, L. Wells, P. Tarazaga, and J. Camelio, "Trojan detection and side-channel analyses for cyber-security in cyber-physical manufacturing systems," *Procedia Manufacturing*, vol. 1, pp. 77–85, 2015.

[6] S. R. Chhetri, A. Canedo, and M. A. A. Faruque, "Confidentiality breach through acoustic side-channel in cyber-physical additive manufacturing systems," *ACM Transactions on Cyber-Physical Systems*, vol. 2, no. 1, pp. 1–25, 2017.

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

[8] A. Zajic and M. Prvulovic, "Experimental demonstration of electromagnetic information leakage from modern processor-memory systems," *IEEE Transactions on Electromagnetic Compatibility*, vol. 56, no. 4, pp. 885–893, Aug 2014.

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

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

[11] J. Balasch, B. Gierlichs, R. Verdult, L. Batina, and I. Verbauwhede, "Power analysis of atmel cryptomemory–recovering keys from secure eeproms," in *Cryptographers' Track at the RSA Conference*. Springer, 2012, pp. 19–34.

[12] J. Brouchier, T. Kean, C. Marsh, and D. Naccache, "Temperature attacks," *Security Privacy, IEEE*, vol. 7, no. 2, pp. 79–82, March 2009.

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

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

[15] D. Genkin, A. Shamir, and E. Tromer, "RSA key extraction via low-bandwidth acoustic cryptanalysis," in *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, 2014, pp. 444–461.

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

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

[18] T. Eisenbarth, C. Paar, and B. Weghenkel, *Building a Side Channel Based Disassembler*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 78–99. [Online]. Available: https://doi.org/10.1007/978-3-642-17499-5_4

[19] M. Msgna, K. Markantonakis, and K. Mayes, "Precise instruction-level side channel profiling of embedded processors," in *Information Security Practice and Experience*, X. Huang and J. Zhou, Eds. Cham: Springer International Publishing, 2014, pp. 129–143.

[20] L. Xiao, Y. Li, X. Huang, and X. Du, "Cloud-based malware detection game for mobile devices with offloading," *IEEE Transactions on Mobile Computing*, vol. 16, no. 10, pp. 2742–2750, 2017.

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

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

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

[24] N. Sehatbakhsh, H. Hong, B. Lazar, B. Johnson-Smith, O. Yilmaz, M. Alam, A. Nazari, A. Zajic, and M. Prvulovic, "Syndrome: Spec-

tral analysis for anomaly detection on medical iot and embedded devicesexperimental demonstration," in *Hardware Demo at IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2017.
[25] N. Sehatbakhsh, A. Nazari, A. Zajic, and M. Prvulovic, "Spectral profiling: Observer-effect-free profiling by monitoring em emanations," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 59.
[26] R. Callan, F. Behrang, A. Zajic, M. Prvulovic, and A. Orso, "Zero-overhead profiling via em emanations," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 401–412.
[27] R. Rutledge, S. Park, H. Khan, A. Orso, M. Prvulovic, and A. Zajic, "Zero-overhead path prediction with progressive symbolic execution," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 234–245.
[28] B. B. Yilmaz, E. M. Ugurlu, A. Zajic, and M. Prvulovic, "Instruction level program tracking using electromagnetic emanations," in *Cyber Sensing 2019*, vol. 11011. International Society for Optics and Photonics, 2019, p. 110110H.
[29] J. Park, X. Xu, Y. Jin, D. Forte, and M. Tehranipoor, "Power-based side-channel instruction-level disassembler," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.
[30] D. Strobel, F. Bache, D. Oswald, F. Schellenberg, and C. Paar, "Scandalee: a side-channel-based disassembler using local electromagnetic emanations," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 139–144.
[31] May 2019. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/solutions/partners/partner-profile/terasic-inc-/board/altera-de1-board.html
[32] OlinuXino A13, https://www.olimex.com/Products/OLinuXino/A13/A13-OLinuXino/open-source-hardware.
[33] R. Callan, N. Popovic, A. Zajic, and M. Prvulovic, "A new approach for measuring electromagnetic side-channel energy available to the attacker in modern processor-memory systems," in *2015 9th European Conference on Antennas and Propagation (EuCAP)*, April 2015, pp. 1–5.
[34] M. Prvulovic, A. Zajić, R. L. Callan, and C. J. Wang, "A method for finding frequency-modulated and amplitude-modulated electromagnetic emanations in computer systems," *IEEE Transactions on Electromagnetic Compatibility*, vol. 59, no. 1, pp. 34–42, 2016.
[35] M. Omran, A. Engelbrecht, and A. Salman, "An overview of clustering methods," *Intell. Data Anal.*, vol. 11, pp. 583–605, 11 2007.
[36] Yee Leung, Jiang-She Zhang, and Zong-Ben Xu, "Clustering by scale-space filtering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 12, pp. 1396–1410, 2000.
[37] T. Kurita, "An efficient agglomerative clustering algorithm using a heap," *Pattern Recognition*, vol. 24, no. 3, pp. 205 – 209, 1991. [Online]. Available: http://www.sciencedirect.com/science/article/pii/003132039190062A
[38] R. H. Turi, *Clustering-based colour image segmentation*. Monash University PhD thesis, 2001.
[39] S. van Dongen and A. J. Enright, "Metric distances derived from cosine similarity and pearson and spearman correlations," 2012.
[40] M. Dey, A. Nazari, A. Zajić, and M. Prvulovic, "Emprof: Memory profiling via em-emanation in iot and hand-held devices," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 881–893.
[41] G. Turin, "An introduction to matched filters," *IRE Transactions on Information Theory*, vol. 6, no. 3, pp. 311–329, 1960.
[42] J. G. Proakis and M. Salehi, *Fundamentals of communication systems*. Pearson Education India, 2007.
[43] "Rf and near-field probes for emc measurements." [Online]. Available: https://aaronia.com/antennas/rf-and-near-field-probes
[44] F. Werner, D. A. Chu, A. R. Djordjević, D. I. Olćan, M. Prvulovic, and A. Zajić, "A method for efficient localization of magnetic field sources excited by execution of instructions in a processor," *IEEE Transactions on Electromagnetic Compatibility*, vol. 60, no. 3, pp. 613–622, 2017.
[45] "Dsos804a high-definition oscilloscope: 8 ghz, 4 analog channels." [Online]. Available: https://www.keysight.com/en/pdx-x202073-pn-DSOS804A/high-definition-oscilloscope-8-ghz-4-analog-channels?cc=US&lc=eng

**Elvan M. Ugurlu** (S'19) received the B.Sc. degree in electrical and electronics engineering from Bilkent University, Ankara, Turkey in 2018. He is currently pursuing his Ph.D. in the School of Electrical and Computer Engineering at Georgia Institute of Technology, Atlanta, Georgia, USA. He is a graduate research assistant in Electromagnetic Measurements in Communications and Computing (EMC2) Lab at the Georgia Institute of Technology focusing on electromagnetic side-channel analysis. His research interests include digital signal processing, machine learning and electromagnetics.

**Baki B. Yilmaz** (S'16) received the B.Sc. and M.Sc. degrees in Electrical and Electronics Engineering from Koc University, Turkey in 2013 and 2015 respectively. He joined Georgia Institute of Technology in Fall 2016 and he is currently pursuing his PhD in School of Electrical and Computer Engineering, focusing on quantifying covert/side-channel information leakage and capacity. Previously, he worked on channel equalization and sparse reconstruction. His research interests span areas of electromagnetic, signal processing and information theory.

**Alenka Zajic** (S'99-M'09-SM'13) received the B.Sc. and M.Sc. degrees form the School of Electrical Engineering, University of Belgrade, in 2001 and 2003, respectively. She received her Ph.D. degree in Electrical and Computer Engineering from the Georgia Institute of Technology in 2008. Currently, she is an Associate Professor in the School of Electrical and Computer Engineering at Georgia Institute of Technology. Prior to that, she was a visiting faculty member in the School of Computer Science at Georgia Institute of Technology, a post-doctoral fellow in the Naval Research Laboratory, and a design engineer at Skyworks Solutions Inc. Her research interests span areas of electromagnetic, wireless communications, signal processing, and computer engineering.

Dr. Zajić was the recipient of the 2017 NSF CAREER award, 2012 Neal Shepherd Memorial Best Propagation Paper Award, the Best Student Paper Award at the IEEE International Conference on Communications and Electronics 2014, the Best Paper Award at the International Conference on Telecommunications 2008, the Best Student Paper Award at the 2007 Wireless Communications and Networking Conference, and the Dan Noble Fellowship in 2004, which was awarded by Motorola Inc. and the IEEE Vehicular Technology Society for quality impact in the area of vehicular technology. Currently, she is an editor for IEEE Transactions on Wireless Communications.

**Milos Prvulovic** (S'97-M'03-SM'09) received the B.Sc. degree in electrical engineering from the University of Belgrade in 1998, and the M.Sc. and Ph.D. degrees in computer science from the University of Illinois at Urbana-Champaign in 2001 and 2003, respectively. He is a Professor in the School of Computer Science at the Georgia Institute of Technology, where he joined in 2003. His research interests are in computer architecture, especially hardware support for software monitoring, debugging, and security.

He is a past recipient of the NSF CAREER award, and a senior member of the ACM, the IEEE, and the IEEE Computer Society.