Processor Security: Detecting Microarchitectural Attacks via Count-Min Sketches

Kerem Arıkan, Alessandro Palumbo, Luca Cassano*, Member, IEEE, Pedro Reviriego*, Senior Member, IEEE, Salvatore Pontarelli, Giuseppe Bianchi, Oğuz Ergin, and Marco Ottavi*, Senior Member, IEEE

Abstract—The continuous quest for performance pushed processors to incorporate elements such as multiple cores, caches, acceleration units, or speculative execution that make systems very complex. On the other hand, these features often expose unexpected vulnerabilities that pose new challenges. For example, the timing differences introduced by caches or speculative execution can be exploited to leak information or detect activity patterns. Protecting embedded systems from existing attacks is extremely challenging, and it is made even harder by the continuous rise of new microarchitectural attacks (e.g., the Spectre and Orchestration attacks). In this article, we present a new approach based on count-min sketches for detecting microarchitectural attacks in the microprocessors featured by embedded systems. The idea is to add to the system a security checking module (without modifying the microprocessor under protection) in charge of observing the fetched instructions and identifying and signaling possible suspicious activities without interfering with the nominal activity of the system. The proposed approach can be programmed at design time (and reprogrammed after deployment) in order to always keep updated the list of the attacks that the checker is able to identify. We integrated the proposed approach in a large RISC-V core, and we proved its effectiveness in detecting several versions of the Spectre, Orchestration, Rowhammer, and Flush + Reload attacks. In its best configuration, the proposed approach has been able to detect 100% of the attacks, with no false alarms and introducing about 10% area overhead, about 4% power increase, and without working frequency reduction.

Index Terms—Embedded systems, hardware security, microarchitectural attacks, microprocessors, RISC-V.

I. INTRODUCTION

S Ecurity requirements are nowadays vital in a huge range of digital systems, such as Internet of Things/edge computing [1], Industry 4.0 [2], or automotive [3]. Traditional security properties, e.g., confidentiality, integrity, and nonrepudiation, are achieved through strong cryptographic algorithms. Although, mathematically speaking, such algorithms are robust, their implementations may suffer from security flaws. In recent years, several cryptographic hardware accelerators demonstrated to be prone to a number of attacks, among which the side-channel analysis (SCA) [4]. As a result, the deployed systems may be vulnerable although featuring security-dedicated modules. SCA exploits unintended information leakage by analyzing timing information, power consumption, thermal footprint, or electromagnetic emanation of computing systems while executing security primitives to extract information about the processed data and then use them to infer sensitive information, e.g., secret keys or messages. Such attacks generally require the attacker to have physical access to the attacked system. Moreover, the attacker needs invasive equipment in order for the attack to be put in place [4].

In the last years, a new family of SCA attacks has been demonstrated to be effective on microprocessor-based systems, i.e., microarchitectural attacks [5]. These attacks do not require physical access to the attacked system and only rely on the observation of its timing behavior while running sensitive applications. The basic idea behind microarchitectural attacks is that, since computer architectures are optimized w.r.t. processing speed, there is a strong correlation between processed data, memory accesses, and execution times. As a consequence, such correlation represents an exploitable side channel in case the attacked process and the attacker share the same cache space [5]. As an example, a well-known microarchitectural attack is Spectre [6], where the attacker takes advantage of speculative execution to break address space isolation without exploiting any software bug. By exploiting Spectre, an attacker allows his/her own program to access the memory (and, thus, also secrets) of other programs and the operating system.

Several countermeasures against microarchitectural attacks working at all the abstraction levels have been proposed...
in the last few years [7]. Solutions specifically tailored to protect AES implementations have been proposed in [8] and [9]: in [8], bitslicing is exploited, while, in [9], the countermeasure relies on vector permutation. Compile-time solutions based on control-flow modifications have been proposed in [10] and [11]; these solutions have the huge drawback of significantly slowing down programs’ executions. In [12], an operating system-level countermeasure based on preventing cache sharing is presented, while, in [13] and [14], periodic cache flushing is used; again, the drawback is a significant slowdown of the system. Finally, several architecture-level countermeasures based on a modified use of the caches have been proposed. In [15] and [16], cache partitioning is exploited, and finally, in [17], cache accesses are randomized. Again, all these techniques significantly slow down the system due to the introduced modifications of the cache utilization.

In this article, we present a novel security checker (SC) for microprocessor-based embedded systems aimed at detecting the occurrence of microarchitectural attacks. The proposed checker relies on the count-min sketch (CMS) probabilistic data structures [18]. The basic idea of our proposal is to add to the systems under the protection of a security checking module (CM) between the instruction memory and the fetch unit within the microprocessor’s pipeline. Such CM is in charge of observing the fetched instructions and of identifying and signaling possible suspicious activities without interfering with the nominal activity of the microprocessor. On the one hand, the hardware architecture of the checker (size of the CMS and memory occupation) is configured at design time; on the other hand, the list of the considered attack models can be programmed at design time by the designer and then reprogrammed after deployment by the user in order to always keep it updated. Indeed, our proposal relies on the identification of the predefined and reproducible set of suspicious instruction patterns (and associated fetching frequency) belonging to the legitimate program or not, while indeed, in [20] and [21], the checker only verified if the fetched instructions belonged to the legitimate program or not, while the SC proposed in this article identifies complex instruction patterns (and associated fetching frequency) belonging to the considered microarchitectural attacks. By summarizing, the novelties and advantages of the proposed approach w.r.t. existing solutions are given as follows:

1) the generality w.r.t. the specific microprocessor and features, e.g., speculative and out-of-order execution;
2) the ability to protect the system irrespective of the executed application, i.e., it is not a protection mechanism specific for cryptography;
3) the programmability (and reprogrammability) that makes the solution effective for a vast range of attacks;
4) the limited and configurable false positive rate and the reduced detection time;
5) the transparency w.r.t. the nominal microprocessor functionality, i.e., no performance overhead is introduced;
6) the reduced area, power, and working frequency overhead.

This article is organized as follows. In Section II, we provide a general background. Section III presents the proposed solution and the design flow used for configuration and programming. In Section IV, we report the results of an experimental evaluation. Section V discusses the security-related advantages and limitations of the proposed checker. Section VI presents the related work. Finally, Section VII concludes this article.

II. BACKGROUND

In this section, microarchitectural attacks are reviewed and analyzed to find the existing common aspects, and the CMS probabilistic data structures are presented.

A. Microarchitectural Attacks

Multiple classifications have been proposed for microarchitectural attacks [4], [7]; two main families exist:

Time-Driven: The attacker measures the execution time of the executed operations to extract sensitive information [22], [23]. The rationale behind these attacks is that the execution time varies with the execution paths or cache hits/misses, which is often strongly related to the processed information. Therefore, the attacker can extract secret information, e.g., encryption keys, by controlling the content of the shared cache and measuring the running time of the victim program. However, as the time-driven attacks measure the whole execution time, they suffer from the noise introduced by the operating system and network. Thus, a large number of samples are needed for a statistical evaluation to extract secret information. The main advantage of these attacks is the wide applicability, which only requires execution time measurement.

Access-Driven: Here, the attacker monitors whether a specific component in the architecture is used or not. The monitored components may be the data cache [24], the instruction cache [25], and the branch prediction cache [26]. The information related to the use of these components is inferred by measuring the time required to access them. If a cache entry has been accessed by the victim program, the attacker program would observe a cache hit; otherwise, a cache miss.
The key difference between time- and access-driven attacks is that, in the former case, the attacker measures the victim process’ whole execution time, while, in the latter case, the attacker measures the execution time of a specific operation. This gives access-driven attacks higher fidelity.

1) Orchestration Attacks: Orchestration attacks [27] exploit the cache design choices used to manage read-after-write (RAW) hazards. This hazard can occur when two sequential instructions have a data dependency: the former instruction makes a write request, and the latter does a read request at the same address. To avoid this hazard, the pipeline is stalled. It must be noticed that current cache designs trigger the pipeline stall of possible RAW hazards if the write and read request share the same cache line, even if they are not accessing the same address. A simple example of an Orchestration attack is reported in the code snippet in Fig. 1. The snippet tries to create intentional RAW hazards to leak the content of the data stored in a protected address. Lines 1 and 2 are the initialization instructions. \texttt{x1} holds the protected address that represents the goal of the attack. \texttt{x2} holds the test address. We assume that \texttt{x1} is not accessible by the attacker, while \texttt{x2} is accessible. The attacker tries to guess the value of \texttt{x1} by iteratively increasing the content of \texttt{x2} by a “test_value” and by executing the subsequent instructions. In line 4, the test value is used as a memory address. Note that, since \texttt{x2} is accessible, this instruction does not cause an exception. This instruction represents the first instruction of the intentional RAW hazard. In line 5, the protected data stored in \texttt{x2} are used as a memory address. Since, in a pipelined system, the executed instructions do not trigger a memory boundary exception on the core until the writeback stage, we can use the content of \texttt{x2} as a memory address unless the data to be written into \texttt{x4} are fetched. In the time interval before the exception raising, the CPU executes line 6, which corresponds to the second instruction of the intentional RAW hazard. Now, if the address \texttt{x2} and the address \texttt{x4} (i.e., the content of \texttt{x2} that we are trying to discover) have the same higher bits, they point to the same cache line, thus triggering the pipeline stall delaying the execution of the snippet. Instead, if \texttt{x2} and \texttt{x4} have different higher bits values, the execution will be faster. After the attacker discovered the higher bits stored in \texttt{x2}, it will use a trial and error routine to guess the lower bits.

2) Spectre: The Spectre attack [6] has the same target as the Orchestration attack, i.e., try to discover the secret data stored in an address. The difference is that, in the Spectre attack, the attacker takes advantage of speculative execution employed by modern processors instead of the cache RAW hazards. In particular, Spectre exploits \texttt{misspeculation}, which is achieved by “training” the branch prediction mechanism by conditioning the victim branch with an index compared with the size of an accessible array. Running a code where there is a branch that is always taken will sooner or later induce the predictor to mark the victim branch as strongly taken. After looping through an accessible array, at the last iteration of the loop, the attacker tries to access the protected address. Due to the intentional misprediction activated by the loop, the code actually executes the read request. The data fetched by the misprediction will be removed from the CPU registers when the system detects the not allowed access. However, the data are kept in the cache hierarchy since all current CPU microarchitectures do not revert the effect in the caches due to the execution of miss-predicted instructions. Thus, the attacker can deposit into the cache the secret stored in a protected address and exploit timing information to discover it. In particular, as in the previous case, the attacker will use the secret (or a part of it) as a memory address and will insert this address in the cache. Finally, the attacker will check which addresses are stored in the cache by checking the cache access latency. The attack is exemplified in the code snippet reported in Fig. 2.

To elaborate on the reported code snippet, we can inspect the sequence in lines 3–5 as the misspeculated region. The branch that is going to be mis-predicted is at line 2 where the branch statement resides. As the first step, the attacker has to run the loop many times to misdirect the branch predictor to bias the predictor. Before the branch instruction at line 2, the attacker uses the test data at line 1 as register \texttt{x2}. The address pointed by \texttt{x2} is read and stored in the cache hierarchy. After multiple consecutive executions, \texttt{blt} is going to be predicted as strongly not taken and will not jump to the end tag at line 6. Thus, the misspeculated region is executed, and the data are stored in the cache. Now, the attacker can retrieve the data stored in the cache using common side-channel time instructions (not depicted in the snippet). The time to access \texttt{x5} can be used to indicate if \texttt{x5} has the same MSBs of \texttt{x2}.

3) Rowhammer: This is an exploit that takes advantage of a technological characteristic of DRAM memories as well as of a well-known side effect related to these memories. The technological characteristic consists of the need for periodically refreshing the content of all the memory cells because of the natural discharge of the employed capacitors and in the need for rewriting the content of the memory cells after any read and write operation because these operations cause a discharge of the accessed cells. The side effect of the DRAM technology is that contiguous memory cells electrically interact between themselves causing a charge leak. This unintended charge transfer may cause an unwanted change of the content of memory rows that are nearby the accessed row, but those were not actually addressed in the original memory access, also known as \textit{disturbance error} [28]. Such disturbance error may be exploited by an attacker to circumvent memory protection and isolation: indeed, disturbance error may represent
an unwanted “short-circuit” that the attacker may exploit. Extremely frequent accesses to a DRAM row may induce faster discharge in the capacitors belonging to the adjacent rows, which are called the victim rows. Therefore, the content of memory rows that should not be accessible to the attacker may be modified by accessing memory rows that belong to the memory space of the attacker. By exploiting this mechanism, the attacker may gain unrestricted access to the entire memory space of a system or gain unauthorized privileges.

The code snippet reported in Fig. 3 represents the basic Rowhammer attack: we assume the case where the contents of $x1$ and $x2$ are two memory addresses mapped in different memory rows but in the same memory bank. The code moves values ($x0$ and $x3$) into these addresses, and it then flushes the memory locations. By iteratively accessing and flushing (Hammering) those memory lines, the attacker will be able to modify the content of the adjacent lines.

1) **Flush+Reload**: This attack takes advantage of the fact that it is possible to know which operations the microprocessor is carrying out and which data it is processing by knowing the execution time of the instructions that the microprocessor is executing [29]. As an example, in the RSA cryptographic algorithm, sequences of square-reduce-multiply-reduce operations (dubbed SRMR-SEQs that take a long time) indicate an encrypted bit, while sequences of square-reduce operations (dubbed SR-SEQs that take a shorter time) indicate a plain text bit. The Flush+Reload attacks consist of the phases: 1) a memory line is flushed from the cache by the attacker; 2) the attacker waits a given time to allow the victim program to access the memory line; and 3) the attacker reloads the memory line, and he/she measures the time required to load it. If during phase two, the victim program did not access the previously flushed memory line, the reload operation in phase three will take a long time due to the fact that the data should be loaded from the main memory. On the opposite, the reload operation will take short in case the victim program accessed the memory line during phase two. As an example, in case the victim program is running RSA, the wait time at phase two may be set to the time required to execute SR-SEQs. If, at phase three, the attacker discovers that the reloaded data come from the cache, he/she may infer that the processed data was a chunk of plain text, while, if he/she discovers that the data come from the memory (because the reload came before SRMR-SEQs could be completed), the attacker may infer that the victim program was processing a chunk of encrypted data. More details about Flush+Reload may be found in [30].

### C. Count-Min Sketch

A CMS is a probabilistic data structure used to estimate the occurrence frequencies of a stream of events belonging to different types [18]. CMSs use hash functions to map events to frequencies, but, unlike hash tables, they use only sublinear space, at the cost of overcounting some events due to collisions. The goal of a CMS is to receive a stream of events, one at a time, and to estimate the frequency of the different

![Image](https://example.com/image.png)

**Fig. 3.** Code snippet representing the Rowhammer attack.

---

1. `mov (x1), %x0` : #read from address pointed by x1
2. `mov (x2), %x3` : #read from address pointed by x2
3. `cflush (x1)` : #flushing x1
4. `cflush (x2)` : #flushing x2

---

**B. Common Features of the Considered Attack Models**

We identified the following common aspects among the above analyzed attacks, and based on these, we then defined the working principles of the proposed methodology.

1) **Quick Repetitive Patterns**: All the previously presented attacks rely on the repetition of instruction patterns. Such repetitions are required to be back to back. Thus, the defender can inspect the set instructions’ signature and decide whether something dangerous is happening. As an example, the Orchestration attack relies on RAW hazards that exploit the data transaction’s latency to avoid boundary exceptions. Thus, the instructions belonging to the attack have to be sequentially executed within a small time interval to prevent a segmentation fault or a boundary trap. Similar considerations can be drawn for Spectre. Indeed, in both attacks, the access to the secret content must be followed by transient instructions. In the case of an Orchestration attack, the transient instructions are the ones that are executed by the induced RAW hazard; in the case of a Spectre attack, the transient instructions are the ones executed speculatively to avoid trapping before fetch.

2) **Critical Instructions**: Although different implementations of the same attack may exist, there are specific instructions that are essential for the attacker to find the protected data. Such key instructions cannot be replaced. Therefore, these are the instructions that any attack implementation must contain (and execute in a specific order) to actually carry out the attack itself; in other words, they represent an attack signature (that an SC may exploit to identify the attack). As an example, the code snippet in Fig. 2 reports the critical instructions for the Spectre attack: additional instructions to obfuscate the attack or to output intermediate data may be added, but the critical instructions must be present in the attack in that specific order.

3) **Register Patterns**: In order for an instruction sequence to be considered suspicious, it is not sufficient that critical instructions are included in the sequence. Indeed, it is necessary that the instructions in the sequence are arranged in specific patterns in terms of the executed instructions themselves but also in terms of the involved registers in order for the sequence to be an effective attack. As an example, let us consider the code snippet in Fig. 1: the fact that the source and destination registers of the `add` instruction at line 3 are the same ($x2$ in the specific case), and the very same register is used as the source register of the `sw` instruction at the subsequent line, as well as the fact that the destination register of the `lw` instruction at line 5 ($x4$ in the specific case) is the same as the source register of subsequent `lw` instruction makes this code snippet an Orchestration attack. On the other hand, if the instruction sequence would have been the same, but, for example, the `add` would have wrote data in $xu$, and the subsequent `sw` would have read data from $x3$, the code snippet would not have represented an Orchestration attack.
types of events in the stream. At any time, a CMS can be queried for the frequency of a particular event type \( x \) from a universe of event types \( U \). The CMS will return an estimate of this frequency that is within a certain distance from the exact frequency with a certain probability.

A CMS is generally composed of \( k \) arrays each with \( m \) counters that are initialized to zero. Moreover, a hash function is associated with each of the \( k \) arrays. This means that a CMS has a constant size regardless of the number of elements in the sets that it measures. A given element \( x \) is associated with \( k \) counters, one per array; in particular, for each specific array \( i \), the counter to which \( x \) is associated is identified by the value of the hash function \( h_i(x) \) (being \( h_i \) the hash function associated with the \( i \)th array). In other words, for each array of counters in a CMS, the output value of the hash function associated with the array is used as an address to identify the right counter to be accessed. It is worth mentioning that, for the sake of spatial efficiency, more than one element may be associated with the same counter. CMSs support two operations: \( \text{Update}(x) \) and \( \text{Estimate}(x) \). The \( \text{Update}(x) \) operation accesses all the counters associated with \( x \) and increments them. The \( \text{Estimate}(x) \) operation provides the CMS estimation of the frequency of \( x \). Such estimation is calculated by reading the \( k \) counters associated with \( x \) and returning the minimum value. Therefore, by construction, the CMS estimation is always equal to or larger than the real frequency of \( x \). The equality occurs when at least one of the \( k \) counters associated with \( x \) is not associated with other elements than \( x \). Otherwise, if for any given counter \( k_i \) associated with \( x \), at least another element \( y \) exists such that \( y \) is also associated with \( k_i \) (the so-called hash collision), the estimate will be larger than the actual number of times that \( x \) has appeared. In other words, this means that a CMS can either correctly predict (true positive and true negative conditions) or raise false alarms (false positive conditions). On the other hand, it is impossible by construction that a CMS falls into a false negative condition. It is worth noting that the larger \( k \) and \( m \) (and, thus, the larger the employed memory), the smaller the probability of overestimating when querying the CMS.

III. PROPOSED SECURITY SOLUTION

We propose a novel approach to detect and signal the occurrence of microarchitectural side-channel attacks (MSCAs) in microprocessor-based embedded systems. The proposed solution does not require any modification to the microprocessor under protection. Moreover, since it works directly at the circuit level, the proposed solution does not need either a multicore architecture or multithreading support from the operating system (such as several solutions based on machine learning). Therefore, we argue that our solution may be suitable both for high-performance computing and low-end embedded systems. Finally, we point out that the management of the warning signal produced after the detection of an attack does not fall into the scope of this work.

Our solution (as depicted in Fig. 4) relies on the insertion of an SC on the instruction bus, between the instruction memory and the microprocessor. Therefore, the SC is able to observe all the fetching activity performed by the microprocessor. Solely based on the observation of the fetched instructions and the frequency with which they are fetched, the SC determines whether an attack is going on or not. In other words, the proposed SC raises an alarm as soon as it recognizes the critical instructions composing the signature of an attack among the fetched instructions. On the other hand, no instruction execution is performed by the SC. More in detail, the SC module works on a time-window base: within a time window (whose duration can be programmed by the user), the SC observes and keeps track of all the instructions fetched by the microprocessor. Moreover, thanks to a CMS, the SC is able to estimate the occurrence frequency of a set of instruction sequences. When the programmed time window expires, the checker analyses the previously recorded fetching activity and compares it with a set of attack models that have been previously programmed by the user. As it will be presented in Section III-A, an attack model is described within the proposed technique in terms of a pattern of instructions that have to be fetched and a frequency threshold; this threshold describes the minimum number of times that the pattern has to be fetched in order to be considered suspicious. In case at least one of the MSCA attack models matches the fetching activity, i.e., the fetched instructions are the same as in the attack model, and they have been fetched more than the programmed threshold, a warning is signaled. Finally, it has to be mentioned that all the data used to program the SC, i.e., the duration of the time window and the attack models to be checked, come from a host PC through an AXI bus, as represented by the red arrow in Fig. 4.

The proposed SC module has a very limited overhead in terms of area occupation, power consumption, and working frequency reduction. Moreover, thanks to its programmability, it is possible to always keep updated the list of MSCA attacks detected by the proposed solution. Finally, please note that, since it is placed between the core and the instruction memory, thus solely observing the fetched instructions, the proposed SC can be employed irrespective of the specific microprocessor features, e.g., out-of-order or speculative execution. In the remainder of this section, we describe in detail the architecture of the SC module, and we then present the design flow that a

\footnote{It is worth mentioning here that the programmed attack models come from a previous security analysis that falls outside the scope of this work.}
user has to follow to properly instantiate and configure an SC within the actual system under protection.

A. Security Checker Architecture

The heart of the proposed security architecture is represented by the SC that, as we previously mentioned, works on a time-window base (a high-level representation of the SC is depicted in Fig. 5). Before running the system (or during system reconfiguration), the SC receives the programming data that are stored in the attack model description module (AMDM). During the working time window, while the monitored microprocessor is running, the SC reads every fetched instruction: this information is analyzed by the CM, which is in charge of monitoring the patterns of fetched instructions. The CM reads the attack models programmed by the user in the AMDM, and it tries to match them with the instructions actually fetched by the monitored microprocessor. Whenever at least one instruction pattern matching is found, the CM updates the CMS module (CMSM) to log the occurrence frequencies of the matched instruction patterns. When a timer within the AMDM expires (the duration of this timer can be programmed by the user), the CMSM is inspected. If suspicious instruction patterns have been fetched with a suspiciously high frequency (exceeding the programmed threshold), a warning is signaled. At the end of these operations, irrespective of attack detection, the CMSM is reset, and the next time window starts.

By summarizing, the possible conditions of the microprocessor’s fetching activity that may be verified by the checker at the end of a time window are given as follows.

1) No suspicious instruction pattern has been fetched; thus, no warning is signaled.

2) At least a suspicious instruction pattern has been fetched, but none exceeds the programmed threshold; thus, again, no warning is signaled.

3) At least a suspicious instruction pattern has been fetched, and at least one of them exceeds the programmed threshold; thus, a warning is signaled.

In the remainder of this section, we describe in detail the structure and functioning of the modules composing the SC.

1) Attack Model Description Module Architecture: The AMDM is the brain of the SC since it is in charge of storing the attack description models specified by the user. The AMDM is depicted in Fig. 6: it is a table containing the description of the attack models that the user wants to take into account plus a programmable timer. It is worth mentioning that the content of the attack description table (which is actually a memory) and the duration of the timer’s countdown are the sole components of the proposed architecture that needs to be programmed by the user (through an AXI bus connected to a host PC, as previously mentioned).

The timer determines the duration of the time window during which the fetching activity of the microprocessor is monitored. When the timer expires, the CM is triggered. Moreover, the timer is in charge of switching the operating mode of the CMSM between Update and Estimate through the u/e signal. While the time window countdown is on, u/e keeps the CMSM into Update mode so that it can monitor the fetched instruction sequences; when the timer expires, and the CM is triggered, u/e switches the CMSM into the Estimate mode, so that it can interact with the CM. When the CM ends its activity, the timer is reset so that a new countdown can start, and the CMSM is again switched into the Update mode.

The main component of the AMDM is the table (a memory) where the attack description data programmed by the user are stored. This table stores the attack patterns that represent signatures of the attacks the user wants to keep into account. In our model, an attack pattern is composed of a pattern ID and a set of prototype instructions characteristics of the attack itself. In turn, a prototype instruction is described by an opcode, a set of labels that represent the destination and source registers, and a frequency threshold. As an example, let us consider the following prototype instruction:

\[ \text{addi } x x - 10 \]

where addi is, of course, the opcode, the first \( x \) is the label for the destination register, the second \( x \) is the label for the destination register, and the third \( x \) is the label for the source register, and the number -10 is the frequency threshold.
first source register, the – means that no second source register is expected, and 10 is an example frequency. This example prototype instruction allows us to specify that suspicious instructions are `addi` with the same register as destination and first sources, without second source register and executed at least ten times within a single time window. On the other hand, `addi` instructions having different destination and source registers, specifying also the second source register or being executed less than ten times within a single time window, would not be considered suspicious. It should be noted that an attack pattern keeps into account only the instructions that are actually of interest for the attack itself and the order in which they have to be executed, leaving out all other instructions. In this way, provided the availability of an attack pattern, our checker is able to detect the activation of the attack irrespective of possible attack camouflaging techniques that the attacker could deploy, i.e., adding harmless instructions between attack instructions.

As a complete and real attack model description, let us refer to Table I, where both the previously discussed Orchestration attack (first block of rows) and Spectre attack (second block of rows) are modeled. Referring to the Orchestration attack reported in Fig. 1, the attack description states that the pattern (whose instructions have all ID 1 and frequency threshold 10) is composed of an `addi` (instruction 1) having the same register as both destination and source registers; then, the result of the `addi` is stored through an `sw` (instruction 2); finally, two consecutive `lw` instructions are executed (instructions 3 and 4) where the source register of the second `lw` is actually the same register as the destination register of the first `lw`. To be considered suspicious, these instructions have to be executed in the specified order (either one after the other or interleaved with other nonsuspicious instructions) at least ten times each in the same time window. Similar considerations can be drawn for Spectre described in the remaining rows of the table.

2) Count-Min Sketch Module Architecture: The CMSM is the eye of the SC since it is in charge of monitoring the activity of the microprocessor. The architecture of the CMSM is depicted in Fig. 7. As it has been previously mentioned, the CMSM has two modes of operations: Update and Estimate. The CMSM is in Update mode during the time window, while it is in Estimate mode when the time window expires, and the CM is triggered. The working mode of the CMSM is determined by the update/estimate input signal, which is indeed generated by the same timer that triggers the CM.

The core of the CMSM is the `k` hash functions that each used to generate the addresses to access the corresponding `k` array of counters. Each array features `m` counters. When in the Update mode, the CMSM receives the instructions fetched by the microprocessor and (after calculating the counter addresses through the hash functions), the corresponding counters are incremented. In other words, when in Update, the CMSM monitors the fetching activity of the microprocessor and keeps track of the occurring frequency of each instruction. On the other hand, when in the Estimate mode, the CMSM does not monitor the fetching activity of the microprocessor anymore, but it replies to frequency requests coming from the CM. In this functioning mode, the CMSM receives an instruction from the CM, and it calculates the `k` hash values associated with the instruction and reads the corresponding counters’ values. These `k` values are analyzed by the CMS analyzer that identifies the minimum value and returns it to the CM. In other words, when in the Estimate mode, the CMSM is used by the CM to estimate the occurring frequency of previously identified suspicious instructions. Finally, when the CM ends its analysis, before returning in the Update mode, the CMSM is reset so that all the counters restart from zero in the subsequent monitoring time window.

It is here worth mentioning that, when the CMSM is in the Update mode, it works transparently in parallel with the protected core, without interfering with the fetching activity. When the CMSM is in the Estimate mode, it completes its activity within one clock cycle; therefore, it does not interfere either with the protected microprocessor or with the subsequent Update phase. Indeed, as it will be discussed in the experimental section, the proposed SC has no impact on system performance.

3) Checking Module Architecture: The CM is the arm of the SC since it is in charge of detecting the suspicious activity of the microprocessor and, if necessary, raising warnings. The architecture of the CM is depicted in Fig. 8. It is composed of the activity monitor module (AMM), the pattern matching

---

**TABLE I**

**ATTACK MODEL DESCRIPTION FOR THE ORCH. AND SPECTRE ATTACKS**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Opcode</th>
<th>Dest</th>
<th>Source1</th>
<th>Source2</th>
<th>FreqThur</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>addi</td>
<td>A</td>
<td>A</td>
<td>–</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>B</td>
<td>A</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lw</td>
<td>D</td>
<td>C</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lw</td>
<td>E</td>
<td>D</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>li</td>
<td>A</td>
<td>B</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>blt</td>
<td>A</td>
<td>C</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bll</td>
<td>D</td>
<td>A</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add</td>
<td>E</td>
<td>F</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld</td>
<td>G</td>
<td>E</td>
<td>–</td>
<td></td>
</tr>
</tbody>
</table>

---

Footnote: Note that the two initial `li` instructions in the code snippet in Fig. 1 are merely initialization instructions, and they are not iteratively executed during the attack. For this reason, we are not considering them in the attack model.
module (PMM), and the frequency analysis module (FAM). The AMM is always active to receive the instructions fetched by the microprocessor and translate them into the corresponding prototype instructions (as described above). When a time window expires, the PMM is triggered: it loads the fetching activity of the microprocessor during the last time window from the AMM and the programmed attack models from the AMDM. Now, the PMM can check whether the patterns of the attack models programmed by the user occurred in the fetching activity that the microprocessor performed in the last time window. In case at least one instructions’ pattern is matched, the FAM is activated to check whether the suspicious instructions identified by the PMM have been executed more than the specified frequency threshold. Therefore, the FAM interacts with the CMSM by asking to estimate the frequency of a set of instructions (the ones identified by the PMM) and by getting back such frequency values. In case the frequency values of all the instruction prototypes of at least one of the patterns matched by the PMM are detected to exceed the threshold by the FAM, the FAM itself raises a warning. After this check, whether the warning has been raised or not, the FAM resets both the trigger within the AMDM and the CMSM so that a new monitoring time window can start.

B. Security Checker Design Flow

It is worth pointing out here that, by relying on the CMS data structure, our solution ensures a 100% detection probability of the programmed attacks (no false negatives). On the other hand, a theoretically possible vulnerability of our approach is related to a denial-of-service attack. More in detail, an attacker could enforce the proposed checker to signal nonexisting attack occurrences (false positives), thus making denial of service attacks hard, but also the larger the area occupation. $I$ could be chosen as the minimum amount of instructions that the attacker needs to execute in order to effectively carry out the attack. By having in mind the working frequency of the considered microprocessor and the calculated $I$, the designer can also identify the best duration for the time window. Finally, the threshold $t$ should be identified as the minimum number of times that the attack code should be executed in order for the attack to be successful. On the other hand, we point out that, in this work, $I$ and $t$ are considered as parameters coming from the security analyst that also provided the attack models fed in the proposed checker.

For our design flow, we borrow the point query approximation presented in [18]. The probability that the difference between a value estimated by a CMS and the corresponding expected value is larger than $(e \cdot I)/m$ is always smaller than $FP_p = e^{-k}$. In our case, the maximum error for the CMS not to cause a false positive is $t$; therefore,

\[
\frac{e \cdot I}{m} = t
\]

and, by inverting the formula,

\[
m = \frac{e \cdot I}{t}.
\]

Therefore, it is possible to calculate the value of $m$ as a function of $I$ and $t$. Fig. 9 reports the values of $m$ for $I$ from 500 up to 1000 and $t$ ranging from 20 up to 100.

Furthermore, provided that the value of $m$ has been set for the worst case value, the false positive probability would be
\[ \text{FP}_p = e^{-k}. \]

We can invert this formula and calculate
\[ k = \left\lceil \ln \frac{1}{\text{FP}_p} \right\rceil. \quad (3) \]

Therefore, it is possible to calculate the value of \( k \) as a function of the desired worst case false positive probability. Fig. 10 correlates the values of \( k \) and those of \( \text{FP}_p \). It is possible to notice that, for \( k = 4 \), the false positive probability already falls below \( 10^{-1} \) and for \( k = 6 \) below \( 10^{-2} \). It has to be pointed out that these values are worst case calculations that can be used by the designer in the very first phases of the design flow for a preliminary dimensioning of the proposed checker; the real \( \text{FP}_p \) values will be discussed in Section IV.

IV. EXPERIMENTAL RESULTS

A. Experimental Setup

For our experimental campaign, we used the RSD core [19], which is a 32-bit, speculative out-of-order, superscalar, two-fetch front-end, and five-issue back-end pipelines' RISC-V core with 16-kB instruction cache developed at The University of Tokyo.\(^3\) The synthesis and the implementation of the considered microprocessor have been performed on Vivado targeting a Virtex7 xc7z020clg484-1 FPGA. The obtained core counted 18 334 LUTs, 10 885 FFs, 4512 LUTRAM cells, and 17 BRAM cells and worked at 57 MHz with an estimated power consumption of about 0.926 W.

In order to assess both effectiveness and efficiency of our solution, we considered several implementations of the checker. In particular, we tried the number of hash functions \( k \) ranging from 1 up to 6, and for every value of \( k \), we considered values 32, 64, and 128 for \( m \) (the number of counters associated with each hash function). \( I \) has been fixed to 1000 and \( t \) to 100, and the size of the counters has been fixed to 8 bits.

\(^3\)It is worth mentioning that, in this article, we considered the RSD out-of-order core since it represents a quite complex and large case study; on the other hand, our approach could be applied to smaller and simpler in-order cores since it is independent of the execution order of the fetched instructions.

We considered three versions of the Spectre, Orchestration, Rowhammer, and Flush+Reload attacks, respectively, as case studies; Table II reports the total number of executed instructions and the detail about the number of executed loads, stores, and branches. Finally, the same information is reported in Table III for the four software benchmarks that we considered as case study programs being executed under attack.

B. Effectiveness Analysis

As a first experiment, we aimed at assessing the capability of the proposed checker in detecting the occurrence of the considered attacks. For each of the considered checker’s configurations and for each of the attacks, we ran 100 random simulations of each of the four benchmark programs where an attack was activated. In each simulation, we randomly chose...
the attack starting time and the program input. The result of this first set of experiments has been that, independent of the specific configuration, the proposed checker is always able to detect the activation of an attack as soon as the timer within the AMDM triggers the activation of the checker and to properly raise an alarm. This result should not surprise if we take into account that, as previously discussed, a CMS may either correctly predict (true positive and true negative) or raise false alarms (false positive), while it cannot fall into a false negative condition by construction. The average number of instructions that the attacker program was able to execute before being detected was 517 for Spectre, 527 for the Orchestration attack, 901 for Rowhammer, and 999 for Flush+Reload, which is about 0.3% of the instruction count for Orchestration, Spectre, and Flush+Reload and about 0.7% for Rowhammer (see the first column of Table II). Therefore, our solution is not only very effective in detecting the activation of an attack but also very fast.

The second set of experiments aimed at measuring the number of false positives raised by the checker in the various considered configurations. Indeed, as we have previously discussed, an attacker could aim at deploying a denial-of-service attack by inducing the checker at raising a huge number of false alarms, thus preventing the system to carry out its legitimate tasks. For this analysis, we performed a set of experiments similar to the previous one but instead of deploying the full attacks, i.e., executing the attack code more than \( t \) times, we executed it only a few times (less than the programmed threshold). In this way, we emulated an attacker that wakes up the CM (by executing at least once the attack code) and that then tries to cheat on the CMSM by causing hash collisions. The resulting false positive rates for the considered checker’s configurations when considering the Orchestration, Spectre, Rowhammer, and Flush+Reload attacks are shown in Figs. 11–14, respectively. As it can be observed from the shown figures, the false positive rate, except for \( k = 1 \) and \( k = 2 \), is always extremely low, and in most cases, it is zero. More important, such real false positive rates are always below the worst case values (which are represented in the figures by the black horizontal lines) calculated by following the design procedure described before, thus also validating that design flow. As a final note, we point out that the checker produces an answer (being it a warning or not) within one clock cycle; therefore, it does not interfere with the protected microprocessor activity.

C. Efficiency Analysis

In order to measure the overhead introduced by the proposed checker, we synthesized the unprotected core and the same core protected with the previously described checker configurations on a Virtex7 FPGA. Table IV reports the results of these experiments. For every configuration, the table reports

---

4Note that the SC configuration from \( (4, 64) \) up to \( (6, 128) \) is not reported in the figures for the sake of space and readability, considering that, in all benchmarks and checker’s configurations, the false positive rate is 0%.

5Note that labels referring to \( k = 5 \) and \( k = 6 \) are not reported on the lines in the graphs for the sake of space and readability.
TABLE IV
SYNTHESIS RESULTS: RESOURCE OCCUPATION, POWER CONSUMPTION, AND WORKING FREQUENCY

<table>
<thead>
<tr>
<th>#Checker configuration</th>
<th>#LUTs</th>
<th>#LUTRAMs</th>
<th>#FFs</th>
<th>#BRAMs</th>
<th>Power Consumption</th>
<th>Working Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>18334</td>
<td>4512</td>
<td>10885</td>
<td>17</td>
<td>0.926 W</td>
<td>57 MHz</td>
</tr>
<tr>
<td>1-32</td>
<td>18980</td>
<td>(+3.52%)</td>
<td>4520</td>
<td>(+0.18%)</td>
<td>11518 (+5.82%)</td>
<td>17</td>
</tr>
<tr>
<td>1-64</td>
<td>18981</td>
<td>(+3.53%)</td>
<td>4520</td>
<td>(+0.18%)</td>
<td>11518 (+5.82%)</td>
<td>17</td>
</tr>
<tr>
<td>1-128</td>
<td>18975</td>
<td>(+3.50%)</td>
<td>4512</td>
<td></td>
<td>11510 (+5.74%)</td>
<td>17 (+2.94%)</td>
</tr>
<tr>
<td>2-32</td>
<td>19024</td>
<td>(+3.76%)</td>
<td>4528</td>
<td>(+0.35%)</td>
<td>11535 (+5.97%)</td>
<td>17</td>
</tr>
<tr>
<td>2-64</td>
<td>19034</td>
<td>(+3.82%)</td>
<td>4528</td>
<td>(+0.35%)</td>
<td>11535 (+5.97%)</td>
<td>17</td>
</tr>
<tr>
<td>2-128</td>
<td>19024</td>
<td>(+3.76%)</td>
<td>4512</td>
<td></td>
<td>11519 (+5.82%)</td>
<td>18 (+5.88%)</td>
</tr>
<tr>
<td>3-32</td>
<td>19058</td>
<td>(+3.95%)</td>
<td>4536</td>
<td>(+0.53%)</td>
<td>11552 (+6.13%)</td>
<td>17</td>
</tr>
<tr>
<td>3-64</td>
<td>19063</td>
<td>(+3.98%)</td>
<td>4536</td>
<td>(+0.53%)</td>
<td>11552 (+6.13%)</td>
<td>17</td>
</tr>
<tr>
<td>3-128</td>
<td>19049</td>
<td>(+3.90%)</td>
<td>4512</td>
<td></td>
<td>11528 (+5.91%)</td>
<td>18.3 (+8.2%)</td>
</tr>
<tr>
<td>4-32</td>
<td>19082</td>
<td>(+4.08%)</td>
<td>4544</td>
<td>(+0.71%)</td>
<td>11569 (+6.28%)</td>
<td>17</td>
</tr>
<tr>
<td>4-64</td>
<td>19092</td>
<td>(+4.13%)</td>
<td>4544</td>
<td>(+0.71%)</td>
<td>11569 (+6.28%)</td>
<td>17</td>
</tr>
<tr>
<td>4-128</td>
<td>19066</td>
<td>(+3.99%)</td>
<td>4512</td>
<td></td>
<td>11537 (+5.99%)</td>
<td>19 (+11.76%)</td>
</tr>
<tr>
<td>5-32</td>
<td>19114</td>
<td>(+4.25%)</td>
<td>4552</td>
<td>(+0.89%)</td>
<td>11586 (+6.44%)</td>
<td>17</td>
</tr>
<tr>
<td>5-64</td>
<td>19124</td>
<td>(+4.31%)</td>
<td>4552</td>
<td>(+0.89%)</td>
<td>11586 (+6.44%)</td>
<td>17</td>
</tr>
<tr>
<td>5-128</td>
<td>19090</td>
<td>(+4.12%)</td>
<td>4512</td>
<td></td>
<td>11546 (+6.07%)</td>
<td>19.5 (+14.71%)</td>
</tr>
<tr>
<td>6-32</td>
<td>19198</td>
<td>(+4.71%)</td>
<td>4566</td>
<td>(+1.20%)</td>
<td>11591 (+6.49%)</td>
<td>17</td>
</tr>
<tr>
<td>6-64</td>
<td>19208</td>
<td>(+4.77%)</td>
<td>4566</td>
<td>(+1.20%)</td>
<td>11591 (+6.49%)</td>
<td>17</td>
</tr>
<tr>
<td>6-128</td>
<td>19116</td>
<td>(+4.27%)</td>
<td>4512</td>
<td></td>
<td>11555 (+6.16%)</td>
<td>20 (+17.65%)</td>
</tr>
</tbody>
</table>

D. Comparison

Finally, we compared the best configuration for the proposed solution, i.e., the one having a (3, 64) checker configuration, with four recent related works, namely, InvisiSpec [31], Jintide [32], Trust Guard [33], and Cyclone [34]. The summary of this analysis is reported in Table V, where the detection capability is reported for each solution, as well as the area and power overhead, and the introduced slowdown. Area and power overhead for InvisiSpec and power overhead for Cyclone are not reported because the authors conducted only a GEM5 simulation without providing any hardware implementation; on the other hand, the area overhead for Jintide is not provided because this is a system-level solution that requires a dedicated machine to be run, i.e., it is not an architecture-level solution.

From the numbers in the table, it clearly appears that all the considered solutions are actually able to detect about 100% of the attacks but with significantly larger overheads than the ones introduced by the methodology proposed in this article either in one or even in more than one of the three considered metrics. Only cyclone introduces a slightly smaller area overhead w.r.t. our solution. Indeed, our solutions demonstrated to be the most lightweight in terms of area and power consumption increase, and system slowdown while having 100% attack detection.

V. SECURITY ANALYSIS

The CM presented in this article is able to detect the activation of every microarchitectural attack having a specific fingerprint in terms of a recognizable pattern of instructions that has to be executed $n$ times in order for the attack to be effective. A corner case that the checker is also able to
manage is that of attacks where the instructions’ pattern has to be executed just once (it is sufficient to set the threshold \( t = 1 \)). On the other hand, the proposed checker is not able to detect attacks working below the microarchitectural level, e.g., gate level and the RTL level.

We want also to point out that, being our proposal a detection technique, our checker is able to signal the occurrence of an attack, but it is not able either to prevent it or to react/recover from its effects. We envision two possible scenarios, namely, a stealing one and a interference one, based on the goal of the attacker and the effect of the deployed attack on the system. In the stealing scenario, the attacker aims at stealing secret information, e.g., an encryption/decryption key, from the system. In the interference scenario, the attacker aims at either halting the functioning of the system or at modifying its behavior, e.g., gaining unauthorized privileges or executing unexpected programs. In both scenarios, after our checker raises a warning and before a recovery activity is carried out, the attacker has a not null time in which he/she can exploit the effect of the attack. To restore the security of the system, it could be necessary to change the encryption/decryption keys in the case of the stealing scenario or to restart the system in the case of the interference scenario. Again, we point out that the reaction activity carried out after attack detection falls outside of the scope of this article.

It should also be noted that our approach may be prone to denial-of-service attacks. Indeed, an attacker that wants to make unavailable the CPU protected by our method could enforce the proposed checker to signal nonexistent attack occurrences by exploiting the not null false positive rate of the adopted CMS architecture. However, as it has been discussed in this article, the adopted configuration of the SC can be defined based on the required worst case false positive rate, thus allowing to dramatically reduce the risk of denial-of-service attacks. Finally, if such a denial-of-service attack represents a particularly severe concern, the designer can select one of the larger checker’s configurations that bring the false positive rate to zero.

Finally, it should be noticed that the programming data stored in the AMDM are vital for the correct functioning of the proposed solution. If the attacker is able to modify such data, the entire protection mechanism would fail. Nevertheless, since our security proposal focuses on MSCAs (that are able at stealing secret information from the system but not at altering/modifying its functioning/configuration), we argue that such programming data tampering attack falls outside the scope of this article.

VI. RELATED WORK

Several countermeasures against microarchitectural attacks have been proposed in the last few years [7].

Since most microarchitectural attacks are based on the measurement of the variation of the execution time related to the processed data, a family of solutions, i.e., constant-time techniques, rely on making execution time constant via bitslicing [8], [35] or vector permutation [9]. Although being effective, these methods suffer from being extremely hard-to-implement and platform-dependent. Moreover, they significantly slow down the system.

Another family of countermeasures relies on compile-time modifications of the control flow of the program to be protected. In [10], a modified compiler has been proposed: it is able to automatically identify and remove those control flows that are highly dependent on the secret information. The method proposed in [11] generates, at compile time, a number of equivalent but different execution paths that are then randomly chosen at runtime. These solutions have limited applicability, and they significantly slow down execution.

A number of operating-system-level solutions have also been proposed. Osvik et al. [36] suggested to hide timing information by adding random delays or normalizing all timings to a fixed value, while, in [12], an operating system-level countermeasure based on preventing cache sharing is presented. Finally, in [13] and [14], periodic cache flushing is proposed to remove time variations that could be exploited by the attacker. Although being effective, again, all these solutions come at the cost of a high time overhead.

Several architecture-level solutions have been proposed where cache coloring is exploited. More in detail, the cache is divided into regions, and data are mapped into a cache region or another depending on the specific application. Percival [37] suggested avoiding cache sharing or selectively evicting cache based on the thread. Page [38] proposed cache partitioning to block cache-based side-channel attacks, while Wang and Lee [39] proposed the Partition Locked cache (PLcache) to dynamically lock cache lines. Finally, Yan et al. [31] proposed a strategy to defend against hardware speculation attacks in multiprocessors by making speculation invisible in the data cache hierarchy. All these techniques either limit or modified the use of caches, thus again introducing significant execution time overhead.

Finally, a large number of techniques that exploit machine learning, e.g., neural networks [40] and decision trees [41], [42], and the observation of hardware performance counters to detect microarchitectural attacks has been proposed in the very last years [43]. All the techniques proposed in these works achieve very high detection accuracy, but they all work at the software level; indeed, they rely on a specific process to execute the ML-based detection engine. Therefore, they either require multithreading support from the operating system or an additional (trusted and not attacked core) dedicated to the execution of the detection engine. As a consequence, these techniques can be applied to high-performance computing systems and high-end servers, but they are not suitable for low-end systems, e.g., smart cards or automotive embedded systems, where a single core is available, and the operating system is either not available or extremely simple.

VII. CONCLUSION

We have presented a methodology based on CMSs for protecting microprocessor-based embedded systems against microarchitectural attacks. We add a security CM in charge of observing the fetched instructions and of identifying and signaling possible suspicious activity. All this is carried out without interfering with the nominal activity of the microprocessor.
and without modifying it. The proposed approach can be programmed at design time and then reprogrammed after deployment to always keep updated the list of the attacks. We integrated the proposed approach in an RSD RISC-V core, and we proved its effectiveness in detecting the Spectre, Orchestration, Rowhammer, and Flush+Reload attacks. In its best configuration, the proposed approach has been able to detect 100% of the attacks within a very short time window, with no false alarms and introducing about 10% area overhead, about 4% power increase, and without working frequency reduction.

REFERENCES


Kerem Arıkan is currently working toward the bachelor’s degree in electrical and electronic engineering with a minor in computer engineering at the TOBB University of Economics and Technology, Ankara, Turkey.

He did an internship at Kasranga Bilişim Elektronik Ltd. şti., TOBB ETÜ Research Center, TOBB University of Economics and Technology, and an internship at the University of Rome Tor Vergata, Rome, Italy, as a Research Assistant, where he worked on fault tolerance and probabilistic data structures. His current research is focused on side-channel attack detection and mitigation techniques.

Alessandro Palumbo received the master’s degree in electronics engineering for telecommunications and multimedia from the University of Rome Tor Vergata, Rome, Italy, in 2019, where he is currently working toward the Ph.D. degree in electronics engineering.

He was a Researcher Assistant with CNIT, Italy, from April 2018 to October 2019. He participated in two EU projects: SESAMO and 5G-PICTURE.

His research interests include CPU microarchitectures, in particular machine learning techniques and probabilistic data structures for attacks’ detection.

Luca Cassano (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer engineering from the University of Pisa, Pisa, Italy, in 2006, 2009, and 2013, respectively.

He is currently an Assistant Professor with Politecnico di Milano, Milan, Italy. His research activity focuses on the definition of innovative techniques for fault simulation, testing, untestability analysis, diagnosis, and verification of fault-tolerant and secure digital circuits and systems.

Dr. Cassano won the European semifinals of the 2014 TTC’s E. J. McCluskey Doctoral Thesis Award with his Ph.D. thesis titled “Analysis and Test of the Effects of Single Event Upsets Affecting the Configuration Memory of SRAM-based FPGAs.”

Giuseppe Bianchi is currently a Full Professor of Networking with the University of Rome Tor Vergata, Rome, Italy. He has coordinated six large-scale EU projects. His research activity includes wireless networks (an area where he has carried out pioneering research work on WLAN modeling and assessment), programmable network devices, security monitoring and vulnerability assessment, and traffic modeling and control, and is documented in about 280 peer-reviewed international journal articles and conference papers, accounting for more than 20,000 citations.

Dr. Bianchi has been (or still is) an Editor for several journals in his field, including IEEE/ACM TRANSACTIONS ON NETWORKING, IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS, IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT, and Computer Communications (Elsevier).

Oğuz Ergin received the B.S. degree in electrical and electronics engineering from Middle East Technical University, Ankara, Turkey, in 2000, and the M.S. and Ph.D. degrees in computer science from the State University of New York at Binghamton, Binghamton, NY, USA, in 2003 and 2005, respectively.

He was a Senior Research Scientist with the Intel Barcelona Research Center, Spain, prior to joining the TOBB ETÜ Research Center, TOBB University of Economics and Technology, Ankara. He is currently a Professor with the Department of Computer Engineering, TOBB University of Economics and Technology. He is currently leading a research group at the TOBB ETÜ Research Center working on energy-efficient, reliable, and high-performance computer architectures.

Marco Ottavi (Senior Member, IEEE) was with AMD, USA, Sandia National Labs, USA, and the Electrical and Computer Engineering (ECE) Department, Northeastern University, Boston, MA, USA. He is currently an Associate Professor with the University of Twente, Enschede, The Netherlands, and the University of Rome Tor Vergata, Rome, Italy. His research interests include design issues in nanotechnology for emerging computing paradigms, computer architecture for dependable systems, reliability modeling, and fault-tolerant design techniques.

Dr. Ottavi received a prestigious “riento dei cervelli” Fellowship awarded by the Italian Ministry of University and Research in 2009. From 2011 to 2014, he was the Chair of COST Action IC1103 “Manufacturable and Dependable Multicore Architectures at Nanoscale.”