Processor Security: Detecting Microarchitectural Attacks via Count-Min Sketches

Kerem Arikan, Alessandro Palumbo, Luca Cassano, Pedro Reviriego, Salvatore Pontarelli, Giuseppe Bianchi, Oguz Ergin, Marco Ottavi

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, Marco Ottavi, *Senior Member, IEEE*

***Abstract*—The continuous quest for performance pushed processors to incorporate elements like 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 paper 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 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.**

Kerem Arıkan is with the Department of Electrical and Electronic Engineering and Oğuz Ergin is with the Department of Computer Engineering, TOBB University of Economics and Technology, Ankara, Turkey Email: karikan@etu.edu.tr, oergin@etu.edu.tr

Alessandro Palumbo is with Dipartimento di Elettronica, Ingegneria dell'Informazione, Università degli Studi di Roma Tor Vergata, Italy. Email: alessandro.palumbo@uniroma2.it

Giuseppe Bianchi is with CNIT (Consorzio Nazionale Iteruniversitario per le Telecomunicazioni), Dipartimento di Elettronica, Ingegneria dell'Informazione, Università degli Studi di Roma Tor Vergata, Italy. Email: giuseppe.bianchi@uniroma2.it"

Luca Cassano is with the Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy. Email: luca.cassano@polimi.it

Pedro Reviriego is with the Departamento de Ingeniería Telemática, Universidad Carlos III de Madrid, Leganés 28911, Madrid, Spain. Email: revirieg@it.uc3m.es

Salvatore Pontarelli is with the Dipartimento di Informatica, Università di Roma La Sapienza, Italy. Email: salvatore.pontarelli@uniroma1.it

Marco Ottavi is with Università degli Studi di Roma Tor Vergata, Italy and also with the University of Twente, The Netherlands. Email: m.ottavi@utwente.nl

Kerem Arıkan and Alessandro Palumbo contributed equally to this manuscript; therefore, they have to be considered both as first author.

I. Introduction

Security 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*, *non-repudiation*, 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 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 analysing 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 one 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 of 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], [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], [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], [16] cache partitioning is exploited and finally, in [17] cache accesses are randomized. Again, all these techniques significantly slowdown the system due to the introduced modifications of the cache utilization.

In this paper we present a novel security checker for microprocessor-based embedded systems aimed at detecting the occurrence of microarchitectural attacks. The proposed checker relies on the Count-min Sketch probabilistic data structures [18]. The basic idea of our proposal is to add to the systems under protection a security checking module between the instruction memory and the fetch unit within the microprocessor's pipeline. Such checking module 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 Count-min Sketch, 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 pre-defined and reproducible set of suspicious instruction patterns (and associated frequency within a given time window) that are representative of specific microarchitectural attacks. These instruction patterns, one for each attack of interest, are carefully identified by the security engineer at design and then used to program the security checker. Therefore, our proposal is able to check several microarchitectural attacks in parallel thus representing a flexible and scalable security solution. It is worth mentioning that the proposed solution does not need any modification of the microprocessor under protection. We integrated the proposed approach in the large RSD core [19] and we proved its effectiveness in detecting several versions of the Spectre, Orchestration, Rowhammer and Flush+Reload attacks. The solution demonstrated to be able to detect 100% of the attacks in an small time window while causing an extremely reduced (and configurable at design-time) amount of false alarms and introducing about 10% area overhead, about 4% power increase and without working frequency reduction.

Solutions where, like in the current paper, probabilistic data structures are used for hardware security purposes have been proposed in [20], [21]. In both papers the considered attacks were hardware Trojan horses modifying the fetching activity of the system and infesting the main memory (in [20]) and the microprocessor (in [21]). The solutions proposed in those papers are much simpler than the one here proposed: indeed, in [20], [21] the checker only verified if the fetched instructions belonged to the legitimate program or not while the security checker proposed in the current paper 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:

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

The paper 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 and finally, Section VII concludes the paper.

II. Background

In this section microarchitectural attacks are reviewed and analyzed to find the existing common aspects and the Count-Min sketch probabilistic data structures are presented.

A. Microarchitectural Attacks

Multiple classifications have been proposed for microarchitectural attacks [7], [22]; two main families exist:

Time-driven, where the attacker measures the execution time of the executed operations to extract sensitive information [23], [24]. 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, where the attacker monitors whether a specific component in the architecture is used or not. The monitored components may be the data cache [25], the instruction cache [26], and the branch prediction cache [27]. 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-driven 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 li x1, %protected_addr      #load protected_addr in x1
2 li x2, %accessible_addr     #load accessible_addr in x2
3 addi x2, x2, %test_value    #add test_value to x2
4 sw x3, 0(x2)                #store x3 in the address pointed by x2
5 lw x4, 0(x1)                #load in x4 from the address pointed by x1
6 lw x5, 0(x4)                #load in x5 from the address pointed by x4
```

Fig. 1. A code snippet representing an orchestration attack

```
1 lw x1, 0(x2)          #load in x1 from the address pointed by x2
2 blt x1, x3, end       #loop branch that induces transient instructions
3 slli x4, x1, 2        #logical left shift for offset
4 add x5, x6, x4        #add x6 and x4 and store it in x5
5 lw x7, 0(x5)          #load in x7 from the address pointed by x5
```

Fig. 2. A code snippet representing the Spectre attack

1) Orchestration Attacks: Orchestration attacks [28] 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, and 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 to the same address. A simple example of orchestration attack is reported in the code snippet in Figure 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. `x1` holds the protected address that represents the goal of the attack. `x2` holds the test address. We assume that `x1` is not accessible by the attacker, while `x2` is accessible. The attacker tries to guess the value of `x1` by iteratively increasing the content of `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 `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 `x2` is 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 `x2` as a memory address unless the data to be written into `x4` is 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 x2 and the address x4 (i.e. the content of 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 x2 and x4 have different higher bits values, the execution will be faster. After the attacker discovered the higher bits stored in `x2`, it will use a trial and error routine to guess the lower bits.

2) Spectre: The Spectre attack [6] has the same target of 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 *misspeculation*, which is achieved by "training" the branch prediction mechanism by conditioning the victim branch with an index comparison 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 to the protected address. Due to the intentional misspeculation activated by the loop, the code actually executes the read request. The data fetched by the missprediction 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. So, the attacker can deposit into the cache the secret stored in a protected address and exploits 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. After will check which addresses are in the cache checking the cache access latency. The attack is exemplified in the code snippet reported in Figure 2.

To elaborate on the reported code snippet, we can inspect the sequence at lines 3-5 as the miss-speculated region. The branch that is going to be miss-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 `x2`. The address pointed by `x2` is read and stored in the cache hierarchy. After multiple consecutive executions, `blt` is going to be predicted as strongly not taken and will not jump to the end tag at line 6. Thus the miss-speculated region is executed and the data is 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 to `x5` can be used to indicate if x5 has the same MSBs of `x2`.

3) Rowhammer: This is an exploit that takes advantage of a technological characteristic of DRAM memories as well as of a well-know side effect related to these memories. The technological characteristic consists in 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 re-writing 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 that were not actually addressed in the original memory access, also known as *disturbance error* [29]. 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

```
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
```

Fig. 3. A code snippet representing the Rowhammer attack

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 Figure 3 represents the basic Rowhammer attack: we assume the case where the content 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.

4) 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 the microprocessor is executing [30]. As an example, in the RSA cryptographic algorithm sequences of square-reduce-multiply-reduce operations (dubbed SRMR-SEQs, that take a long time) indicate a 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 consists of the phases: i) a memory line is flushed from the cache by the attacker, ii) the attacker waits a given time to allow the victim program to access the memory line, and iii) 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 at phase three will take a long time due to the fact that the data should be loaded from the main memory. At 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 a 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 [31].

B. Common Features of the considered Attack Models

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

Quick Repetitive Patterns: All the previously presented attacks rely on the repetition of instruction patterns. Such repetitions are required to be back to back. So 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. So 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 case of an Orchestration Attack, the transient instructions are the ones that are executed by the induced RAW hazard; in case of a Spectre Attack, the transient instructions are the ones executed speculatively to avoid trapping before fetch.

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 a attack signature (that a security checker may exploit to identify the attack). As an example, the code snippet in Figure 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.

Register Patterns: In order for an instruction sequence to be considered as 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 the sequence to be an effective attack. As an example, let consider the code snippet in Figure 1: the fact that the source and destination registers of the `addi` instruction at line 3 are the same (`x2` in the specific case) and that 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 `addi` would have wrote data in `xu` but the subsequent `sw` would have read data from `x3`, the code snippet would not have represented an Orchestration attack.

C. Count-Min Sketch

A *count–min sketch (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 sub-linear 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 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 $\mathcal{U}$. 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 which are initialized to zero. Moreover, a hash

function is associated to each of the k arrays. This means that a CMS has a constant size regardless of the number of elements in the sets it measures. A given element x is associated to 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 to the $i^{th} array$). In other words, for each array of counters in a CMS, the output value of the hash function associated to 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: `Update(x)` and `Estimate(x)`. The `Update(x)` operation accesses all the counters associated with x and increments them. The `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 exist 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 condition). 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. The Proposed Security Solution

We propose a novel approach to detect and signal the occurrence of microarchitectural side channel attacks (MSCAs) into 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 (like several solutions based on machine learning do). Therefore, we argue that our solution may be suitable both for high performance computing and for 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 (depicted in Figure 4) relies on the insertion of a *Security Checker (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 of 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.

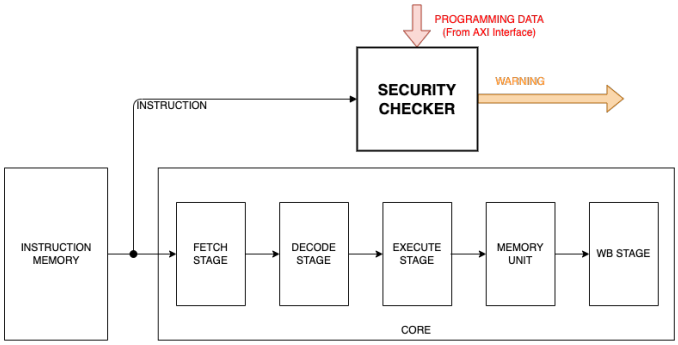

Fig. 4. The architecture of the secured system including the proposed checker.

On the other hand, no instruction execution is performed by the SC. More in details, 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 Count-Min Sketch, 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[1]. As it will be presented in the next subsection, 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 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 signalled. 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 Figure 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 security checker 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 details the architecture of the SC module and we then present the design flow that a user has to follow to properly instantiate and configure a SC within the actual system under protection.

A. The security checker architecture

The *heart* of the proposed security architecture is represented by the *Security Checker (SC)*, that, as we previously

[1]It is worth mentioning here that the programmed attack models come from a previous security analysis that falls outside the scope of this work.

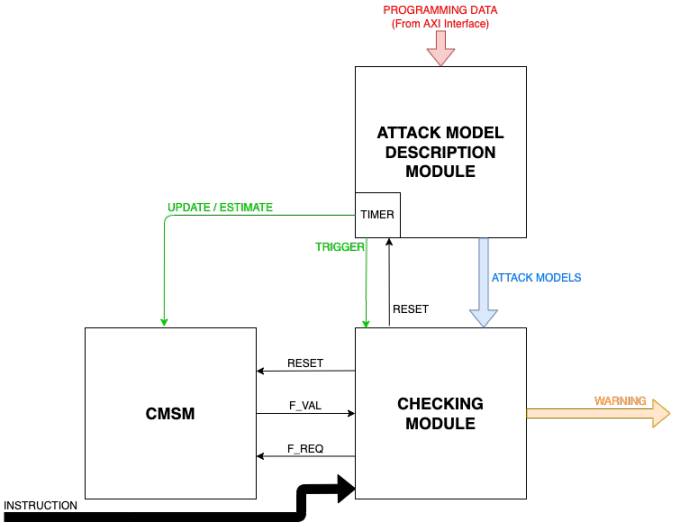

Fig. 5. The internal structure of the proposed security checker.

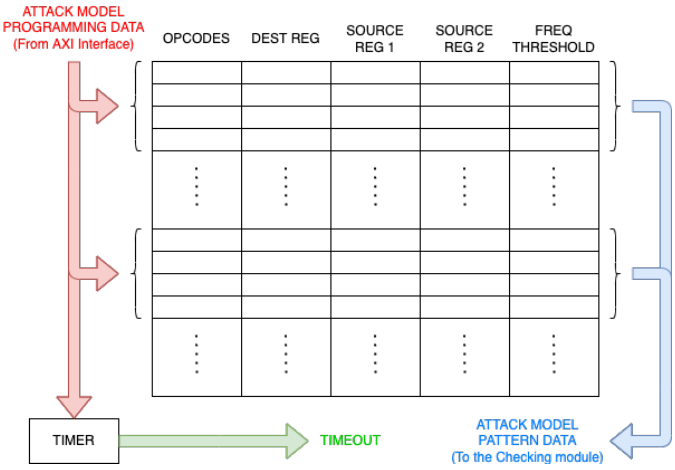

Fig. 6. The architecture of the Attack Model Description Module.

mentioned, works on a time-window base (a high-level representation of the SC is depicted in Figure 5). Before running the system (or during system reconfiguration) the SC receives the *Programming Data* which 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 analysed by the *Checking Module (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 *Count-Min Sketch 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 signalled. At the end of these operations, irrespective of an 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:

- No suspicious instruction pattern has been fetched, thus, no warning is signalled,
- At least a suspicious instruction pattern has been fetched but none exceeds the programmed threshold, thus again, no warning is signalled, and
- At least a suspicious instruction pattern has been fetched and at least one of them exceeds the programmed threshold, thus a warning is signalled.

In the remainder of this subsection we describe in details the structure and functioning of the modules composing the SC.

1) The Attack Model Description Module architecture: The *Attack Model Description Module (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 Figure 6: it is a table containing the description of the attack models 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 need 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* which 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* characteristic 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:

```
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 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 source, without second source register and executed at least ten times within a single time window. On the

TABLE I
ATTACK MODEL DESCRIPTION FOR THE ORCH. AND SPECTRE ATTACKS

Pattern	Opcode	Dest	Source1	Source2	FreqThr
1	addi	A	A	–	10
	sw	B	A	–	
	lw	D	C	–	
	lw	E	D	–	
2	ld	A	B	–	8
	blt	A	C		
	sll	D	A	Y	
	add	E	F	D	
	ld	G	E	–	

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 as 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 modelled. Referring to the Orchestration attack reported in Figure 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 register; 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[2]. To be considered suspicious, these instructions have to be executed in the specified order (either one after the other or interleaved with other non suspicious 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) The Count Min Sketch Module architecture: The *Count-Min Sketch Module (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 Figure 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 are k hash functions each used to generate the addresses to access the corresponding k array of counters. Each array features m counters. When in 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 Estimate mode, the CMSM does not monitor the fetching activity of the microprocessor any more but it replies to *frequency requests* coming from the CM. In this functioning mode the CMSM receives an instruction from the CM, it calculates the k hash values associated with the instruction and reads the corresponding counters' values. These k values are analysed by the *CMS Analyzer* that identifies the minimum value and returns it to the CM. In other words, when in 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 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 security checker has no impact on system performance.

[2]Note that the two initial li instructions in the code snippet in Figure 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.

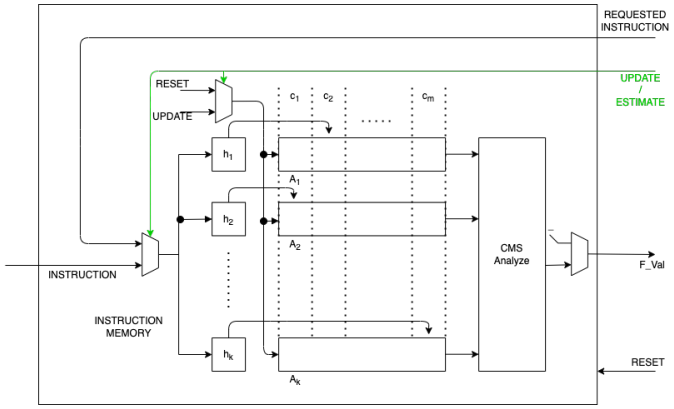

Fig. 7. The architecture of the Count-Min Sketch module.

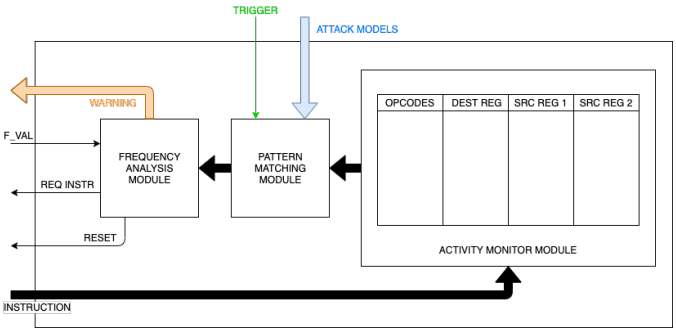

Fig. 8. The architecture of the Checking module.

3) The Checking Module architecture: The *Checking Module (CM)* is the *arm* of the SC since it is in charge of detecting suspicious activity of the microprocessor and, if necessary, raising warnings. The architecture of the CM is depicted in Figure 8. It is composed of: the *Activity Monitor Module (AMM)*, the *Pattern Matching Module (PMM)* and the *Frequency Analysis Module (FAM)*. The AMM is always active to receive the instructions fetched by the microprocessor and to translate them in 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. The security checker design flow

It is worth pointing out here that, by relying on the Count-min Sketch 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 details, an attacker could enforce the proposed checker to signal non existing attack occurrences (false positives), thus making the system continuously fail. Indeed, the attacker could make the CPU execute an instruction sequence belonging to an attack for at least once, thus not exceeding the frequency threshold but still making the CM triggering the CMSM. Then, the attacker could forge specific instruction sequences that cause hash collisions inducing the CMSM to increase the counters associated with the *real* attack sequence. If such hash collisions can be induced a number of times sufficient to cause a threshold violation the CMSM will signal a false positive.

Fortunately, given a specific setting of the checker in terms of number of hash functions k and number of counters per hash function m, the worst case false positive rate can be calculated at design time. More in details, provided the number I of instructions executed within a time window and the threshold t representing the number of times an instruction pattern has to be executed within a time window to be considered suspicious it is possible to calculate the values for k and m so to get a desired worst case false positive probability FP_p. In other words, it is always possible to identify values for k and m to find an acceptable trade-off between security and cost. Indeed, the larger k and m the smaller the false positive rate (thus making denial of service attacks hard) but also the larger the area occupation. I could be chosen as the minimum amount of instructions 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 the attack code should be executed in order for the attack to be successful. On the other hand, we point out that in the current work I and t are considered as parameters coming from the security analyst that also provided the attack models fed in the proposed checker.

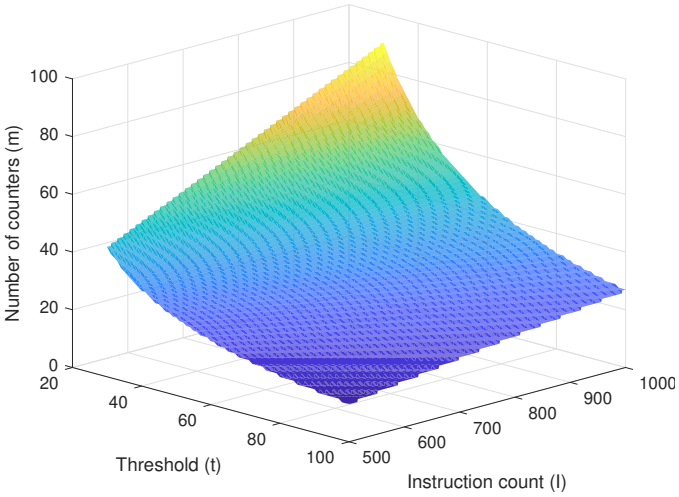

Fig. 9. Correlation among m, I and t.

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:

$$\frac{e \cdot I}{m} = t \tag{1}$$

and, by inverting the formula:

$$m = \frac{e \cdot I}{t} \tag{2}$$

Therefore it is possible to calculate the value of m as a function of I and t. Figure 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 $FP_p = e^{-k}$. We can invert this formula and calculate:

$$k = \lceil \ln \frac{1}{FP_p} \rceil \tag{3}$$

Therefore, it is possible to calculate the value of k as a function of the desired worst case false positive probability. Figure 10 correlates the values of k and those of FP_p. It is possible to

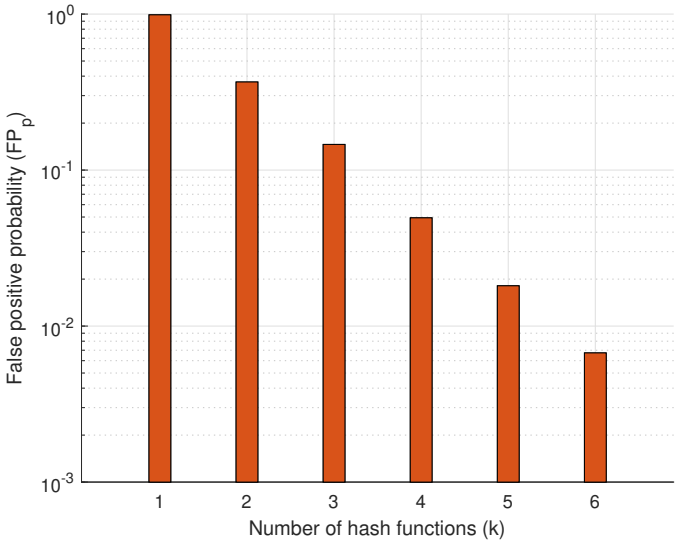

Fig. 10. Correlation between k and FP_p

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* FP_p values will be discussed in the next section.

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, super-scalar, two-fetch front-end and five-issue back-end pipelines RISC-V core with 16KByte 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 18334 LUTs, 10885 FFs, 4512 LUTRAM cells and 17 BRAM cells and worked at 57MHz with an estimated power consumption of about 0.926W.

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, t to 100 and the size of the counters has been fixed to 8 bits.

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 load, store, branches and Finally, the same information is reported in Table III for the four software benchmark that we considered as case the study programs being executed under attack.

[3]It is worth mentioning that in this paper 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.

TABLE II
The considered attacks

Attack	Instr.	Loads	Stores	Branches	Jumps
OrcV1	143363	32345	32004	18495	3552
OrcV2	141705	33057	36000	17010	3272
OrcV3	141537	33905	39723	15894	3052
SpectreV1	139454	72	46213	46195	98
SpectreV2	139452	72	46286	46196	90
SpectreV3	139195	80	46127	46075	100
RowHammerV1	126933	42962	42962	21481	3
RowHammerV2	128565	42838	42838	21419	3
RowHammerV3	128193	42714	42714	21357	3
Flush+ReloadV1	283673	39941	58991	98870	6711
Flush+ReloadV2	283732	39943	58993	98896	6711
Flush+ReloadV3	285365	39944	58994	98875	6711

TABLE III
The considered benchmarks programs

Benchmark	Instr.	Loads	Stores	Branches	Jumps
Coremark	412286	75002	25518	81799	32209
Rsort	297714	37900	28682	10787	4617
Towers	9638	2449	2412	303	912
Median	7388	1995	402	2075	665

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 Attack Model Description Module 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.

A second set of experiments aimed at measuring the amount 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 thresh-

TABLE IV
SYNTHESIS RESULTS: RESOURCE OCCUPATION, POWER CONSUMPTION AND WORKING FREQUENCY

#Checker configuration	#LUTs	#LUTRAMs	#FFs	#BRAMs	Power Consumption	Working Frequency
0	18334	4512	10885	17	0.926 W	57 MHz
1-32	18980 (+3.52%)	4520 (+0.18%)	11518 (+5.82%)	17	0.960 W (+3.67%)	57 MHz
1-64	18981 (+3.53%)	4520 (+0.18%)	11518 (+5.82%)	17	0.960 W (+3.67%)	57 MHz
1-128	18975 (+3.50%)	4512	11510 (+5.74%)	17.5 (+2.94%)	0.960 W (+3.67%)	57 MHz
2-32	19024 (+3.76%)	4528 (+0.35%)	11535 (+5.97%)	17	0.961 W (+3.78%)	57 MHz
2-64	19034 (+3.82%)	4528 (+0.35%)	11535 (+5.97%)	17	0.961 W (+3.78%)	57 MHz
2-128	19024 (+3.76%)	4512	11519 (+5.82%)	18 (+5.88%)	0.964 W (+4.10%)	57 MHz
3-32	19058 (+3.95%)	4536 (+0.53%)	11552 (+6.13%)	17	0.962 W (+3.89%)	57 MHz
3-64	19063 (+3.98%)	4536 (+0.53%)	11552 (+6.13%)	17	0.962 W (+3.89%)	57 MHz
3-128	19049 (+3.90%)	4512	11528 (+5.91%)	18.5 (+8.82%)	0.965 W (+4.21%)	57 MHz
4-32	19082 (+4.08%)	4544 (+0.71%)	11569 (+6.28%)	17	0.962 W (+3.89%)	57 MHz
4-64	19092 (+4.13%)	4544 (+0.71%)	11569 (+6.28%)	17	0.962 W (+3.89%)	57 MHz
4-128	19066 (+3.99%)	4512	11537 (+5.99%)	19 (+11.76%)	0.967 W (+4.43%)	57 MHz
5-32	19114 (+4.25%)	4552 (+0.89%)	11586 (+6.44%)	17	0.963 W (+4.00%)	57 MHz
5-64	19124 (+4.31%)	4552 (+0.89%)	11586 (+6.44%)	17	0.963 W (+4.00%)	57 MHz
5-128	19090 (+4.12%)	4512	11546 (+6.07%)	19.5 (+14.71%)	0.969 W (+4.64%)	57 MHz
6-32	19198 (+4.71%)	4566 (+1.20%)	11591 (+6.49%)	17	0.965 W (+4.21%)	57 MHz
6-64	19208 (+4.77%)	4566 (+1.20%)	11591 (+6.49%)	17	0.965 W (+4.21%)	57 MHz
6-128	19116 (+4.27%)	4512	11555 (+6.16%)	20 (+17.65%)	0.971 W (+4.86%)	57 MHz

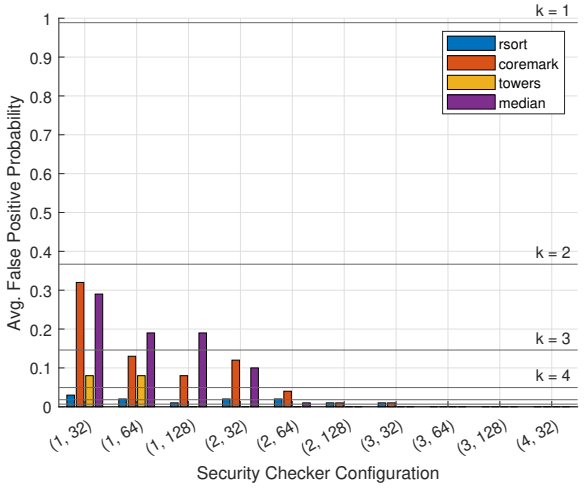

Fig. 11. Average False Positive Probability (FP_p) when attacking several configurations of the SC with the Orchestration Attack

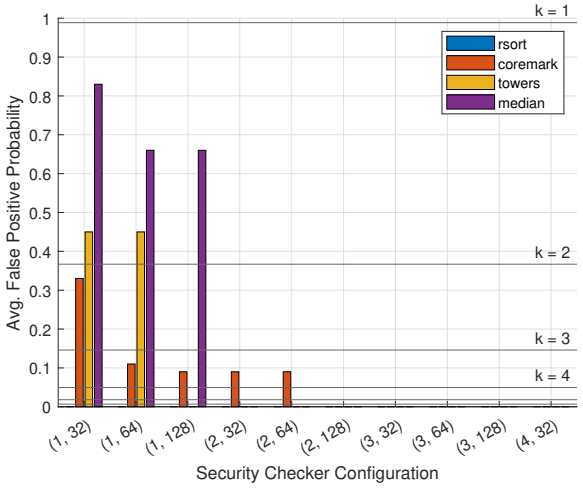

Fig. 12. Average False Positive Probability (FP_p) when attacking several configurations of the SC with the Spectre Attack

old). In this way we emulated an attacker that *wakes up* the Checking Module (by executing at least once the attack code) and that then tries to cheat on the Count-Min Sketch Module 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 Figures 11, 12, 13 and 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[4]. More important, such real false positive rates are always below the worst case values (that are represented in the figures by the black horizontal lines[5]) 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.

[4]Note that SC configuration from <4, 64> up to <6, 128>are 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%.

[5]Note 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.

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 the used LUTs, LUTRAMs, FFs and BRAMs, the total power consumption and the working frequency. The first row of the table reports these values for the unprotected core and then the remaining rows report the absolute values and the increase w.r.t. the unprotected core for each of the considered $<k, m>$ checker's configuration tuples. As it can be observed from the table, the LUTs overhead ranges from about 3.50% and 4.71% and a slightly larger overhead can be observed in terms of FFs.

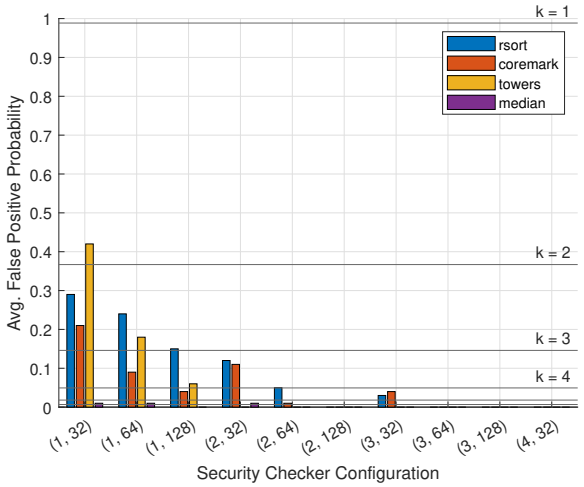

Fig. 13. Average False Positive Probability (FP_p) when attacking several configurations of the SC with the Rowhammer Attack

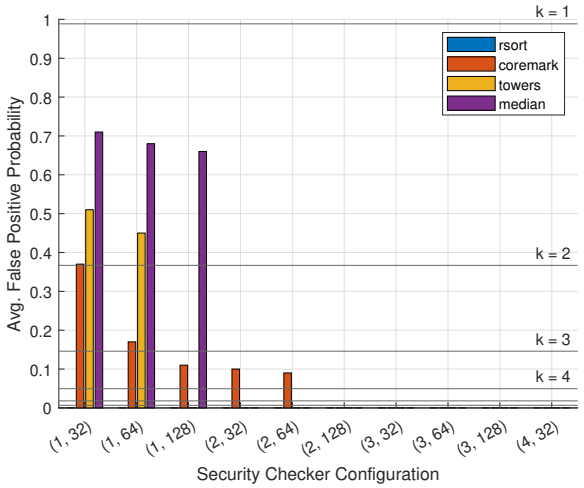

Fig. 14. Average False Positive Probability (FP_p) when attacking several configurations of the SC with the Flush+Reload Attack

It is to be noted that the reported overhead values do not take into account instruction and data memories. Indeed, in the scenario where the protected core is integrated into a system-on-chip with memory cores, the introduced overhead would of course be much smaller. The power consumption increase (based on Vivado estimations) ranges from 3.67% up to 4.86%, which is totally acceptable if we consider that the checker is able to protect the microprocessor from microarchitectural and denial-of-service attacks exposing a configurable small amount of false positives. Finally, no working frequency reduction is caused by the proposed checker.

The $<3,\ 64>$ checker configuration could be considered as a reference for the target microprocessor, attacks and benchmark programs, since it exposes approximately 0% false positive rate for both orchestration and spectre attacks. This configuration introduces about 10% area overhead (LUTs plus FFs plus LUTRAMs), employs one additional BRAM w.r.t. the unprotected core and adds about 4% power consumption with no performance overhead. We also analysed the overhead of the proposed security solution in a 7nm library ASIC implementation working at 200 MHz: in this case the area overhead on the synthesized circuit has been about 18% with about 23% power consumption overhead and, again, no working frequency reduction. Of course these overhead values may be further reduced by finely optimizing the ASIC implementation (out of the scope of this paper).

TABLE V
COMPARISON BETWEEN OUR SOLUTION AND RECENT WORKS

	Our	[32]	[33]	[34]	[35]
Detection	100%	99%	100%	100%	100%
Area	10%	NA	NA	25%	8%
Power	4%	NA	28%	41%	NA
Slowndown	0%	21%	1%	15%	4%

D. Comparison

Finally we compared the the best configuration for the proposed solution, i.e., the one having a $<3,\ 64>$ checker configuration, with four recent related works, namely InvisiSpec [32], Jintide [33], Trust Guard [34] and Cyclone [35]. 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 that the ones introduced by the methodology proposed in this paper 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 checking module presented in this paper 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, 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 a 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 behaviour, 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 paper.

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 non existing attack occurrences by exploiting the not null false positive rate of the adopted Count-Min Sketch architecture. However, as it has been discussed in the paper, the adopted configuration of the security checker 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 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 is 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 the paper.

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], [36] or vector permutation [9]. Although being effective these methods suffer from being extremely hard-to-implement and platform-dependent. Moreover, they significantly slowdown 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. in [37] 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], [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 details, cache is divided into regions and data are mapped into a cache region or another depending on the specific application. Percival [38] suggested to avoid cache sharing or selectively evicting cache based on thread. Page in [39] proposed cache partitioning to block cache-based side- channel attacks while Wang and Lee in [40] proposed the Partition Locked cache (PLcache) to dynamically lock cache lines. Finally, Yan ed al. proposed a strategy to defend against hardware speculation attacks in multiprocessors by making speculation invisible in the data cache hierarchy [41]. All these techniques either limited 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 [42] and decision trees [43], [44], and the observation of hardware performance counters to detect microarchitectural attacks has been proposed in the very last years [45]. 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, to 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. Conclusions

We have presented a methodology based on Count-Min Sketches for protecting microprocessor-based embedded systems against microarchitectural attacks. We add a security checking module 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 a 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

[1] L. Wang and S. Köse, "When hardware security moves to the edge and fog," in *2018 IEEE 23rd International Conference on Digital Signal Processing (DSP)*, 2018, pp. 1–5.

[2] S. R. Chhetri, S. Faezi, N. Rashid, and M. A. Al Faruque, "Manufacturing supply chain and product lifecycle security in the era of industry 4.0," *Journal of Hardware and Systems Security*, vol. 2, no. 1, pp. 51–68, 2018.

[3] D. K. Oka, "Securing the automotive critical infrastructure," in *Cyber-Physical Security*. Springer, 2017, pp. 267–281.

[4] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, "Systematic classification of side-channel attacks: A case study for mobile devices," *IEEE Communications Surveys Tutorials*, vol. 20, no. 1, pp. 465–488, 2018.

[5] A. P. Fournaris, L. Pocero Fraile, and O. Koufopavlou, "Exploiting hardware vulnerabilities to attack embedded system devices: a survey of potent microarchitectural attacks," *Electronics*, vol. 6, no. 3, p. 52, 2017.

[6] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. (2018, Dec.) Spectre attacks: Exploiting speculative execution. [Online]. Available: https://spectreattack.com/spectre.pdf

[7] Y. Lyu and P. Mishra, "A survey of side-channel attacks on caches and countermeasures," *Journal of Hardware and Systems Security*, vol. 2, no. 1, pp. 33–50, 2018.

[8] E. Käsper and P. Schwabe, "Faster and timing-attack resistant aes-gcm," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2009, pp. 1–17.

[9] M. Hamburg, "Accelerating aes with vector permute instructions," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2009, pp. 18–32.

[10] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 45–60.

[11] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity." in *NDSS*, 2015, pp. 8–11.

[12] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 871–882.

[13] M. Godfrey and M. Zulkernine, "Preventing cache-based side-channel attacks in a cloud environment," *IEEE transactions on cloud computing*, vol. 2, no. 4, pp. 395–408, 2014.

[14] Y. Zhang and M. K. Reiter, "Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 827–838.

[15] T. Kim, M. Peinado, and G. Mainar-Ruiz, "{STEALTHMEM}: System-level protection against cache-based side channel attacks in the cloud," in *21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 189–204.

[16] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *Proceedings of the 2009 ACM workshop on Cloud computing security*, 2009, pp. 77–84.

[17] J. Kong, O. Aciiçmez, J.-P. Seifert, and H. Zhou, "Hardware-software integrated approaches to defend against software cache-based side channel attacks," in *2009 IEEE 15th international symposium on high performance computer architecture*. IEEE, 2009, pp. 393–404.

[18] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *Journal of Algorithms*, vol. 55, pp. 58–75, 04 2005.

[19] S. Mitsuno, J. Kadomoto, T. Koizumi, R. Shioya, H. Irie, and S. Sakai, "A high-performance out-of-order soft processor without register renaming," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, 2020, pp. 73–78.

[20] A. Bolat, L. Cassano, P. Reviriego, O. Ergin, and M. Ottavi, "A microprocessor protection architecture against hardware trojans in memories," in *2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2020, pp. 1–6.

[21] A. Palumbo, L. Cassano, P. Reviriego, G. Bianchi, and M. Ottavi, "A lightweight security checking module to protect microprocessors against hardware trojan horses," in *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2021.

[22] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, "Systematic classification of side-channel attacks: A case study for mobile devices," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 465–488, 2017.

[23] O. Acıiçmez, W. Schindler, and Ç. K. Koç, "Cache based remote timing attack on the aes," in *Topics in Cryptology – CT-RSA 2007*, M. Abe, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 271–286.

[24] M. Weiß, B. Heinz, and F. Stumpf, "A cache timing attack on aes in virtualization environments," in *International Conference on Financial Cryptography and Data Security*. Springer, 2012, pp. 314–328.

[25] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on aes to practice," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 490–505.

[26] O. Aciiçmez, "Yet another microarchitectural attack: exploiting i-cache," in *Proceedings of the 2007 ACM workshop on Computer security architecture*, 2007, pp. 11–18.

[27] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, 2007, pp. 312–320.

[28] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz. (2018, Dec.) Processor hardware security vulnerabilities and their detection by unique program execution checking. 1812.04975.pdf. [Online]. Available: https://arxiv.org/pdf/1812.04975.pdf

[29] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014, p. 361–372.

[30] C. Shen, C. Chen, and J. Zhang, "Micro-architectural cache side-channel attacks and countermeasures," 12 2020.

[31] Y. Yarom and K. Falkner, "Flush+reload: a high resolution, low noise, l3 cache side-channel attack," Cryptology ePrint Archive, Report 2013/448, 2013, https://ia.cr/2013/448.

[32] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 428–441.

[33] J. Zhu, A. Luo, G. Li, B. Zhang, Y. Wang, G. Shan, Y. Li, J. Pan, C. Deng, S. Yin, S. Wei, and L. Liu, "Jintide: Utilizing low-cost reconfigurable external monitors to substantially enhance hardware security of large-scale cpu clusters," *IEEE Journal of Solid-State Circuits*, vol. 56, no. 8, pp. 2585–2601, 2021.

[34] H. Zhang, S. Ghosh, J. Fix, S. Apostolakis, S. R. Beard, N. P. Nagendra, T. Oh, and D. I. August, "Architectural support for containment-based security," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 361–377.

[35] A. Harris, S. Wei, P. Sahu, P. Kumar, T. Austin, and M. Tiwari, "Cyclone: Detecting contention-based cache information leaks through cyclic interference," 2019.

[36] M. Matsui and J. Nakajima, "On the power of bitslice implementation on intel core2 processor," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2007, pp. 121–134.

[37] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*. Springer, 2006, pp. 1–20.

[38] C. Percival, "Cache missing for fun and profit," 2005.

[39] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," 2005.

[40] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th annual international symposium on Computer architecture*, 2007, pp. 494–505.

[41] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 428–441.

[42] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.
[43] H. Wang, H. Sayadi, S. Rafatirad, A. Sasan, and H. Homayoun, "Scarf: Detecting side-channel attacks at real-time using low-level hardware features," in *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2020, pp. 1–6.
[44] H. Wang, H. Sayadi, G. Kolhe, A. Sasan, S. Rafatirad, and H. Homayoun, "Phased-guard: Multi-phase machine learning framework for detection and identification of zero-day microarchitectural side-channel attacks," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, 2020, pp. 648–655.
[45] H. Wang, H. Sayadi, A. Sasan, S. Rafatirad, T. Mohsenin, and H. Homayoun, "Comprehensive evaluation of machine learning countermeasures for detecting microarchitectural side-channel attacks," in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, 2020.

Kerem Arıkan is an undergraduate Electrical and Electronic Engineering student at TOBB University of Economics and Technology, Ankara, Turkey, who is also minoring in Computer Engineering. His current research is focused on Side Channel Attack detection and mitigation techniques. He did an internship at Kasırga Bilişim Elektronik Ltd. Şti. located in TOBB ETÜ Research Center and an internship at University of Rome Tor Vergata Italy as a research assistant where he worked on fault tolerance and probabilistic data structures.

Alessandro Palumbo is a PHD student of the University of Rome Tor Vergata in Electronics Engineering. He was a researcher assistant for the CNIT, since April 2018 to October 2019. He partecipated to two EU projects: SESAMO and 5G-PICTURE. He took master's degree in Electronics Engineering for Telecommunications and Multimedia at the University of Tor Vergata. His research interests include CPU microarchitectures, In particular: Machine Learning techniques and Probabilistic data structures for attacks detection.

Luca Cassano is an Assistant Professor at Politecnico di Milano, Italy. He received the B.S., M.S. and Ph.D. degrees in Computer Engineering from the University of Pisa, 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. With his Ph.D. thesis, titled "Analysis and Test of the Effects of Single Event Upsets Affecting the Configuration Memory of SRAM-based FPGAs", he won the European semifinals of the 2014 TTTC's E. J. McCluskey Doctoral Thesis Award.

Pedro Reviriego received the M.Sc. and Ph.D. degrees in telecommunications engineering from the Technical University of Madrid, Madrid, Spain. From 1997 to 2000, he was an Engineer with Teldat, Madrid, working on router implementation. In 2000, he joined Massana to work on the development of 1000BASE-T transceivers. From 2004 to 2007, he was a Distinguished Member of Technical Staff with the LSI Corporation, working on the development of Ethernet transceivers. From 2007 to 2018 he was with Nebrija University. He is currently with Universidad Carlos III de Madrid working on approximate data structures, dependable and secure systems, and high speed packet processing.

Salvatore Pontarelli is Assistant Professor at Department of Computer Science, Sapienza University of Rome. He received a master degree in electronic engineering at University of Bologna and the PhD degree in Microelectronics and Telecommunications from the University of Rome Tor Vergata. His main research interests are the design of high-speed hardware architectures for programmable network devices, the implementation of hash based data structures (Bloom filters, cuckoo Tables, etc.), and network dataplane programmability. He participated in research projects funded by public bodies (FP7, H2020) and private companies, and he collaborated with various manifacturers of internet network devices (Cisco and Mellanox Technologies).

Giuseppe Bianchi is Full Professor of Networking at the University of Roma Tor Vergata. His research activity includes wireless networks (an area where he has carried out pioneering research work on WLAN modelling and assessment), programmable network systems, security monitoring and vulnerability assessment, traffic modelling and control, and is documented in about 280 peer-reviewed international journal and conference papers, accounting for more than 20.000 citations. He has coordinated six large scale EU projects, and has been (or still is) editor for several journals in his field, including IEEE/ACM Trans. on Networking, IEEE Trans. on Wireless Communications, IEEE Trans. on Network and Service Management, and Elsevier Computer Communications.

Oğuz Ergin is a professor in the department of computer engineering in TOBB University of Economics and Technology, Ankara, Turkey. He received his BS in electrical and electronics engineering from Middle East Technical University, MS and PhD in computer science from State University of New York at Binghamton. He was a senior research scientiest in Intel Barcelona Research Center prior to joining TOBB ETÜ. He is currently leading a research group in TOBB ETÜ working on energy-efficient, reliable and high performance computer architectures.

Marco Ottavi (M'03–SM'10) is currently is an Associate Professor at the University of Twente, in the Netherlands and an Associate Professor at the University of Rome Tor Vergata, in Italy. In 2009 he received a prestigious "rientro dei cervelli" Fellowship awarded by the Italian Ministry of University and Research. Previously he was with AMD, Sandia National Labs and with the ECE Department, Northeastern University, Boston, MA, USA. His research interests include design issues in nanotechnology for emerging computing paradigms, computer architecture for dependable systems, reliability modeling and fault-tolerant design techniques. From 2011 to 2014, he was the Chair of COST Action IC1103 "Manufacturable and Dependable Multicore Architectures at Nanoscale."