UNIVERSITY OF ROME TOR VERGATA

DEPARTMENT OF ELECTRONICS ENGINEERING

Thesis

Features Analysis of Threats in Microprocessors: Attacks Detection & Mitigation Techniques

PhD Candidate:
Alessandro Palumbo
alessandro.palumbo@uniroma2.it

ADVISOR:
Prof. Giuseppe Bianchi
Prof. Marco Ottavi

CO-ADVISOR:
Prof. Luca Cassano

Academic Year: 2021/2022

Contents

Abstract

The continuous need to improve electronic device performance has led many companies to focus on designing hardware resources and moving industrial production of physical chips to other foundries. The *fab-less* trend of electronic companies, combined with the constantly increasing utilization of third party Intellectual Properties, has led to a new field of study that is called Hardware Security. The study of hardware security is relatively new, since hardware has traditionally been considered immune to attacks and is therefore used as the trust anchor or *root-of-trust* of a system. The main goal of my research was to combine the growing need for low-power devices justified by the IoT revolution with the security and trust issues related to microprocessor vulnerabilities. In detail, I focussed on methodologies and approaches for detecting (and possibly boycotting) attacks in microprocessor-based systems. In particular, I referred to Hardware Trojans Horses and malicious software, with particular emphasis on Microarchitectural Side Channel Attacks (e.g. Spectre and Meltdown attacks). In state-of-the-art, some techniques and methodologies at the circuit level are reported, ensuring microprocessor security and reliability. In my research project, I evaluated these kinds of algorithms implemented on dedicated hardware, operating in parallel to the microprocessor that has to be protected. Threats that may interfere with the circuit are divided into two typologies. Hardware Trojan Horses (which consist of malicious undesired modifications to the circuit designed) run on the hardware of malicious software programs. The effects and consequences of the attack depend on the scenario of usage of the attacked hardware circuit. Much detail about threats and possible attack consequences is reported in Chapter 1. In chapter 2 the security features and vulnerabilities of the RISC-V microprocessor architecture are explored.

The results of my research activity are reported in Chapters 3 and 4. These chapters describe security solutions for microprocessors, including those implemented by my research group and me. In particular, Chapter 3 reports methodologies and approaches implemented directly at the circuit level to build trusted hardware; in chapter 3 is illustrated a machine learning-based methodology for protecting the circuit under design from possible malicious tools is illustrated: the attacker is the tool, the victim is the circuit under development with the tool itself. Finally, the conclusion of my research activity, the open issues raised, and possible future work are discussed in 5

Chapter 1

Hardware Threats

In this chapter are presented which threats could boycott electronic device functionality.

1.1 Hardware Trojan Horses

Generally, considering malicious activities in microprocessors refers to issues involving software or information processed by the Operative System (OS). In this scenario, an attacker may be able to steal sensitive information by injecting malicious instructions. Software security focuses on malicious attacks on software, often exploiting different implementation bugs (such as inconsistent error handling and buffer overflows) and techniques to ensure reliable software operation in the presence of potential security risks [12]. Specifically, a Software Trojan Horse (STH) is a malware program with malicious code that let gains the attacker privileged access to the OS or interferes with the microprocessor activity [86]. On the other hand, an Hardware Trojan Horse (HTH) consists of an undesired and intentional circuit modification. Its presence is very hard to detect because the modification of the system is done to be silent most of the time and becomes active under specific rare conditions. It alters the nominal behavior of the system or it steals sensitive information processed by the

system. The basic structure of an HTH can include two main parts: trigger and payload [63]. The trigger monitors various signals and/or events in the circuit in order to evaluate the activation moment. Once the trigger detects an expected event or condition, the payload is activated to perform malicious behavior. Typically, the trigger is expected to be activated under sporadic conditions, so the payload often remains inactive. When the payload is not active, the system operates as a Trojan-free circuit, making it difficult to detect the circuit's undesired modification. According to the taxonomy presented in [87], HTHs may be classified based on their *triggering mechanism*, *payload*, and *insertion phase*.

An HTH may be triggered:

- *internally* by logical signals or sequences of logical signals, by physical quantities (e.g., internal temperature or voltage), by a hidden ad-hoc configured counter;
- *externally* by either received messages or commands or by physical interactions (e.g., the external temperature or voltage);

From the taxonomy point of view, there are also the *always-on* HTHs that become active as soon as the system is turned on. HTHs are classified also regarding the payload and their effect on the infested system:

- *Change functionality HTHs* alter the nominal functionality of the infected system (e.g., make the system execute a malicious code);
- *Information stealing HTHs* steal secret information from the system either through the available communication interfaces (e.g., by sending unauthorized messages to the attacker), or through covert side-channels (e.g., temperature or magnetic field)

- *Denial-of-service HTHs* stop system functioning (e.g., by introducing *NOP* instructions, or draining the system's batteries, or jamming the communication interfaces, or altering the timing behavior).

Figure 1.1 reports the Tehranipoor's HTHs taxonomy, based on activation ones and their payload. Much in detail *physical characteristics class* refers to HTHs manifestation. It could be divided, again, into functional and parametric classes: the first one describes hardware circuitry inclusion to implement malicious modules in the circuit; the second one concerns the modification of existing logic with malicious purposes. The *activation characteristics class* is referred to events that trigger the HTH and enable its function. The trigger signals may come from both the extern or the intern of the system (or the HTH is always on). In particular, the condition-based activation is usually associated with a rare environmental event, or a combination of logical internal signals, so that the chip easily overcome functional testing and rarely is detected by an *ATPG* (Automatic Test Pattern Generation) algorithm. The *action characteristic class* HTH goal is to modify the functionality or the specifications of the system attacked or steal information (also enabling and running malicious software). The *modify-function class* of HTHs refers to threats that change project functionalities using new hardware resources or modifying the existing ones. The HTHs of the *modify-specification class* consist to malicious modification of the circuit properties to downgrade the performance of the system o to perform a denial-of-service attack. It is worth mentioning that exist HTHs able to transmit information via Side Channel

Table 1.1: Software vs. Hardware Trojan Attacks [11]

	Software Trojan	**Hardware Trojan**
Activation	A type of malware that resides in a code and activates during its execution	Resides in Hardware (e.g., IC) and activates during its operation
Infection	Spreads through user interaction, e.g., downloading and running a file from the Internet	Inserted through untrusted entities in design or fabrication house
Remedy	Can be *removed in field* through S/W support	*Cannot be removed* once IC is fabricated

Attacks (much detail in the section 1.2.2).

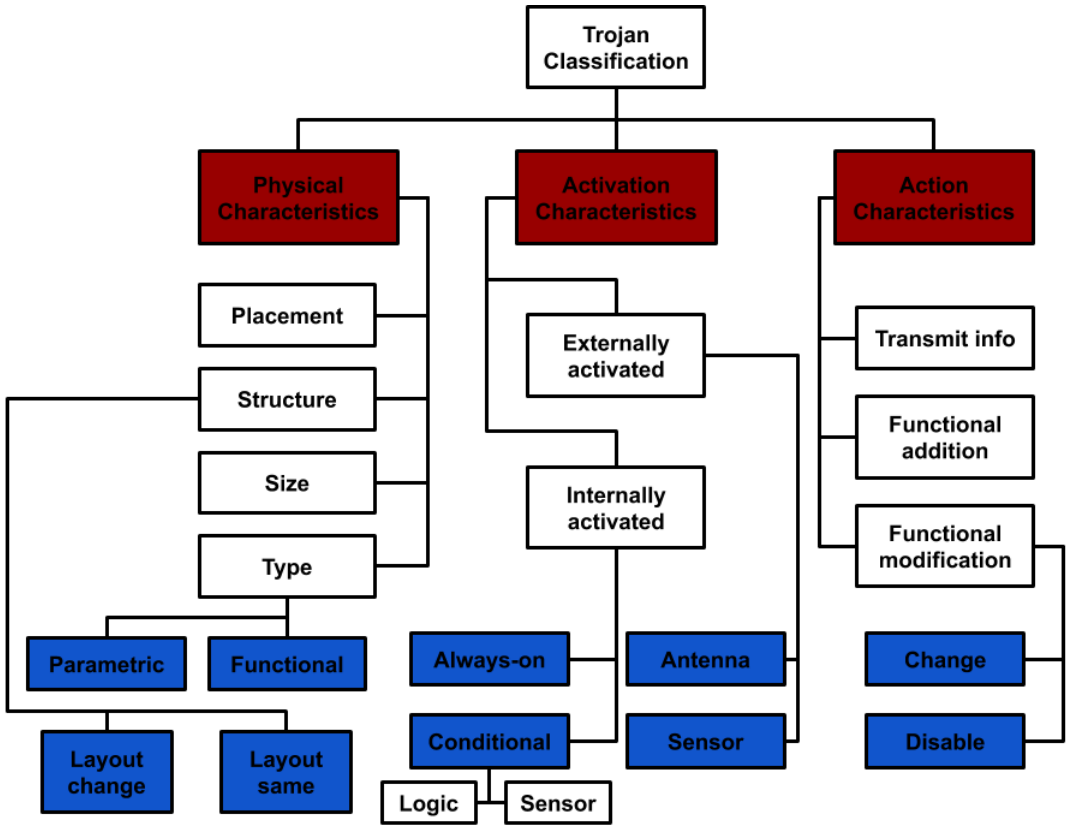


Figure 1.1: Detailed Trojan Taxonomy [86]

An high-level analysis of the differences between STHs and HTHs is reported in Table 1.1.

Looking at the insertion, HTHs may be inserted by providers in the purchased Third-Party Intellectual Property cores (3PIPs) via, for example, both rogue designers and the employed Computer Aided Design (CAD) tools; or by the foundry during chip fabrication.

In past years, HTHs were exclusively considered an academic issue because of the difficulty of insertion in real-world circuits and the limited advantages the attacker could count on. Nevertheless, it has recently been demonstrated that complex and dangerous *software-exploitable HTHs* can be inserted in real-world commercial microprocessors. This class of more powerful HTHs allows the attacker to execute his/her own malicious software, modify the running software, or acquire root privileges [45, 91, 93]. In 2018, an HTH, called the *Rosenbridge backdoor*, has been found in a commercial Via Technologies C3 processor [25, 2]. The Rosenbridge backdoor could be activated via software and allowed the attacker to enter supervisor mode. After the publication of [25] Via Technologies officially commented that this behavior was due to an undocumented feature meant for debug.

Given these premises, HTHs must nowadays be considered an issue by academy and industry. From a very high-level point of view, an HTH is a very hard-to-detect modification of a design that is meant to stay hidden most of the time, while in specific (usually rare) conditions, it alters the nominal behavior of the system, or it steals sensitive information. A produced system may be infected by HTHs belonging to 3PIPs providers [106], employees or malicious CAD tools [76] and mask providers and silicon foundries [9]. The real difficulty in detecting HTHs is related to their differences. In literature is known how physical inspection and reverse engineering

Table 1.2: Comparison Between Faults and Hardware Trojan Attacks [11]

	Fault	**Hardware Trojan**
Activation	Usually at known functional state	Arbitrary combination/sequence of internal circuit states (digital/analog)
Insertion Agent	Accidental (due to imperfection in manufacturing process)	Intentional (inserted by an adversary during IC design or fabrication)
Manifestation	Functional/parametric failure	Functional/parametric failure or information leakage

techniques are expansive and guarantee the reliability but not the trustworthiness chip population. Furthermore, Design For Testability (DFT) techniques generally do not trigger the HTH, which is usually associated with a very rare condition. This is due to the stealthy nature of HTHs and the vast spectrum of possible instances an adversary can exploit [11]. It is worth mentioning that a fault is not an HTH and vice versa. Regarding faults detection, much in detail refers to logical models (stuck-at faults, path delay faults, etc.) that describe physical defect due to manufacturing process imperfections. On the other hand, the HTHs, are intentionally inserted during the chip supply chain phases. In addition, the stealthy nature of HTHs is the principle difference concerning faults. In particular, if a fault is always activated at a known state, an HTH is built and customized for an activation that is difficult to be discovered. Fault detection techniques are implemented to generate test vectors that stimulate the most significant amount of logic possible. For checking a chip, it is necessary to test the circuit simulating all possible scenarios in which it will work. If the circuit has a fault or a production defect, it will be stimulated and an error occurs. The HTH may remains hidden even though the chip is produced with all possible input vectors [6]. In Table 1.2 are reported a summary of the difference between circuit faults and HTHs.

In [11] the authors present a methodology for HTH detection. Much in detail, they present an approach for raise low probability conditions at the internal circuit nodes in order to derive an optimal set of vectors to trigger each of the selected nodes individually. In this scenario, if occurs a different response of the circuit node respect to the desidered one, that "wrong" value will be characterized a single stuck-at fault.

1.1.1 Untrusted Foundries

The dramatic complexity of modern integrated circuits (ICs) and the continuous seek for low production cost and short time-to-market has led to a globalized design and fabrication process [24]. More and more often, the design of several hardware modules is outsourced 3PIPs are purchased, masks are also outsourced, and the final chip is fabricated by third-party foundries [75]. Nowdays the production circuits business trend is horizontal and not vertical: many facilities are involved in the chip supply chain (Figure 1.2). Such production model allows for a significant reduction of design cost and time at the expense of a substantial loss of trust in the delivered ICs [62]. It is impossible to ensure the trustworthiness of all the entities involved in such a globalized supply chain. Consequently, the produced system is exposed to several threats, among which are overproduction [40], counterfeiting [39] (Figure 1.3), license violation and abuse [26] HTHs insertion [87].

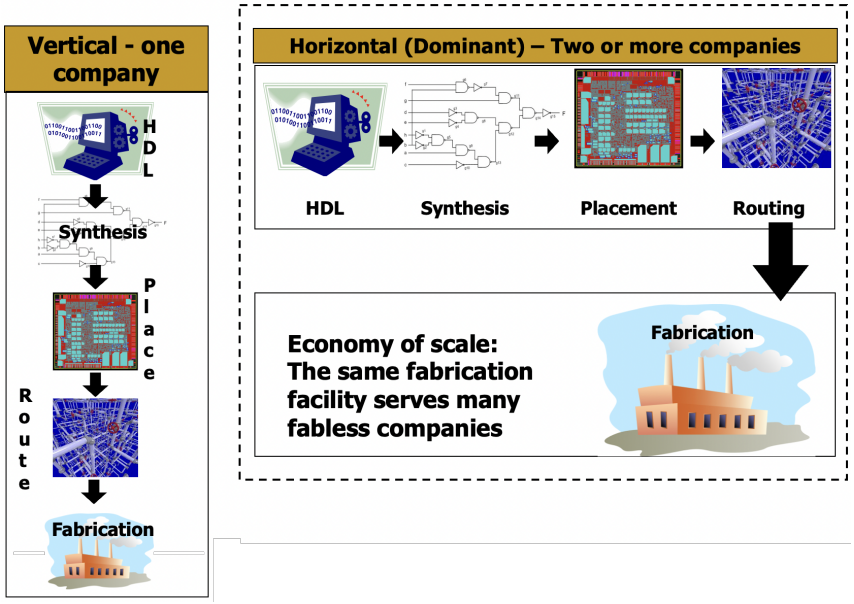


Figure 1.2: Industry business model [88]

In my PhD research activity I investigate both the HTHs insertion in circuits at Register Transfer Level (RTL) and Side Channel Attacks, which are physical attack to the circuit (much detail in section 1.2.

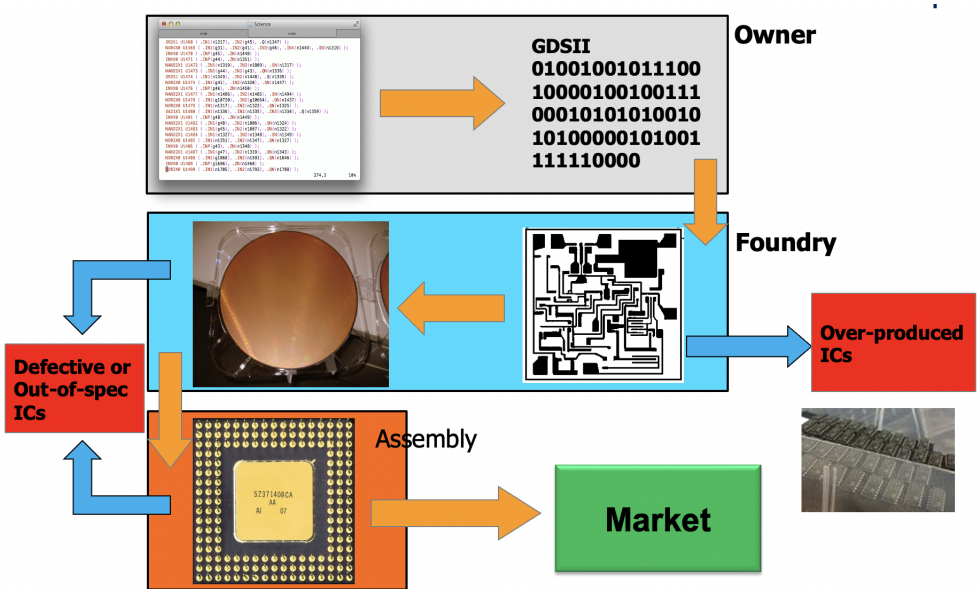


Figure 1.3: Counterfeiting possibility during chips production process [88]

1.1.2 Register Transfer Level Malicious Insertion

The decentralization of fabrication process, combined with the increase of 3PIPs and Electronic Design Automation and CAD tools, give rise to the possibility for an attacker to modify a project. In the circuit supply chain there are several stages whose final aim is to build a netlist of gates starting from hardware description language (HDL). In this phases a lot of malicious insertion could happen, in fact at the end of RTL design phase an attacker may put his/her malicious extra logic in the project modifing the circuit. HTHs at RTL level may be represented by two simple categories: (Figure 1.4)

- Combinational HTHs;

- Sequential HTHs.

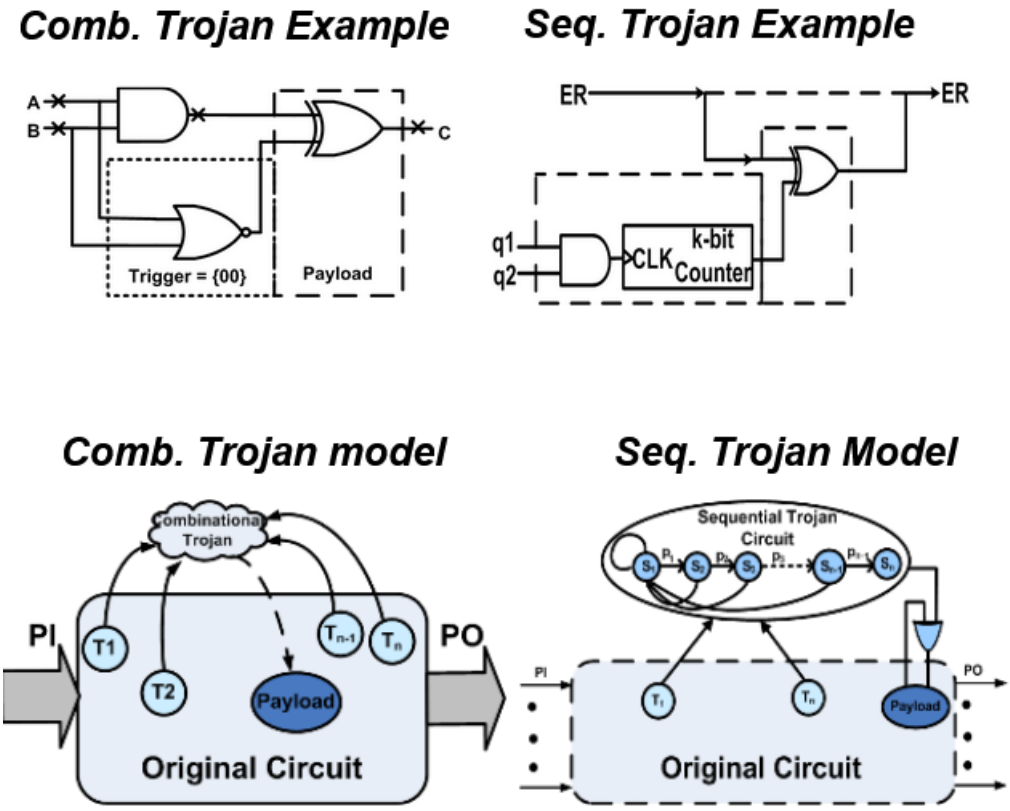


Figure 1.4: Examples of combinational and sequential HTH [88]

Furthermore, HTH rare activation conditions could challenge the detection through traditional logic testing techniques. As already said, other sources of untrusted hardware are 3PIPs and tools, which are known in literature for what concern cases of IP cloning, IP tampering and reverse engineering attacks. The vast majority of HTH are related to manipulations on RTL level, in which an attacker could inject malicious code. Thus an attacker may insert a malicious module in all of described steps

with the purpose of stealing secret keys, modifying functionality or also disabling the device (Figure 1.5).

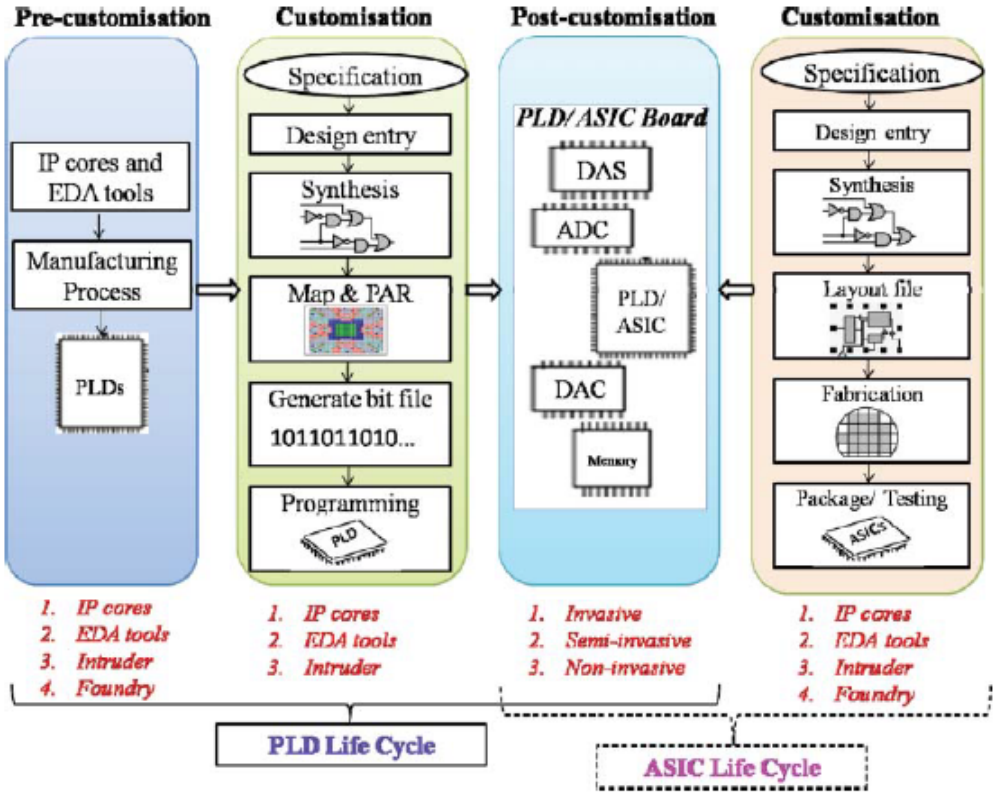


Figure 1.5: Feasible HT threats in PLD/ASIC life cycles [83]

1.2 Physical Attacks

Physical attacks are divided into three categories: noninvasive, semi-invasive, and invasive attacks. A noninvasive attack does not require any initial preparations of the device under test, and will not physically harm the device during the attack. The attacker can either tap the wires to the device, or plug it into a test circuit

for the analysis. Invasive attacks require direct access to the internal components of the device, which normally requires a well-equipped and knowledgeable attacker to succeed. Meanwhile, invasive attacks are becoming constantly more demanding and expensive, as feature sizes shrink, and device complexity increases. There is a large gap between noninvasive and invasive attacks. Many attacks fall into this gap, called semi-invasive attacks. They are not very expensive as classical penetrative invasive attacks, but are as easily repeatable as noninvasive attacks. Like invasive attacks, they require depackaging the chip in order to get access to its surface. However, the passivation layer of the chip remains intact, as semi-invasive methods do not require creating contacts to the internal wires. Reverse engineering, microprobing attack, and invasive fault injection attack are the most common physical attacks.

Side-channel attacks (SCAs) are a noninvasive kind attacks through which an attacker may takes advantages by analyzing features and information on channels apparently unrelated to the execution of the circuit function i.e., timing information, power consumption [56], thermal footprint [58] or electromagnetic emanation [53].

1.2.1 Rowhammer

For example, the Rowhammer attack 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

Listing 1.1: A code snippet representing the Rowhammer attack

```
mov (x1), %x0 ;read from address pointed by x1
mov (x2), %x3 ;read from address pointed by x2
cflush (x1) ;flushing x1
cflush (x2) ;flushing x2
```

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* [46]. 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 Listing 1.1 **??** represents the basic Rowhammer attack, assuming 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.

1.2.2 Microarchitectural Attacks

Microarchitectural SCAs (MSCA) consist in attacks to microprocessors system based. An attacker may finalize a MSCA without have a real physical access to the system under attack, but he/she can gains sensitive information regarding victim data looking at some microprocessor features during victim data computation. Indeed, such attacks only rely on the observation of the timing behavior of the system while running sensitive applications. The basic idea behind this family of 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: such correlation may represent an exploitable side channel in case the attacked and the attacker processes share the same cache space [33]. A well-known example of microarchitectural attack is *Meltdown* [52], where the attacker exploits out-of-order execution to break address space isolation without exploiting any software bug. Therefore, by exploiting Meltdown an attacker allows his/her own program to access the memory (and thus also secrets) of other programs and of the OS.

Multiple classifications have been proposed for MSCAs [82, 57]. Generally speaking they can be divided into two main families:

- **Time-driven**, where the attacker measures the execution time of the executed operations to extract sensitive information [5, 14, 97]. 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 are generally the data cache [41, 90], the instruction cache [3], and the branch prediction cache [4]. 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. A common access-driven attack consists is monitoring the data cache to learn which entries are accessed while performing, for example, AES.

Note that both time-driven and access-driven attacks rely on measuring time information. The key difference in timing-driven versus access-driven 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 difference gives access-driven attacks higher fidelity than time-driven attacks. Finally, one important requirement of all these attacks is that the attacker process needs to run in the same processor with the victim process, so that they can share the cache space. In my PhD research activity I focused on the following MSCAs.

1.2.2.1 Orchestration

Orchestration [30] consists in an attack that 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

Listing 1.2: A code snippet representing an orchestration attack

```
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
```

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 Listing 1.2 **??**. 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

Listing 1.3: A code snippet representing the Spectre attack

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

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.

1.2.2.2 Spectre

Spectre attack [48] has the same target of the Orchestration Attack, i.e. try to discover the secret data stored in a cache memory location. The difference is that in this attack the attacker takes advantage on 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 misspredict ion 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 Listing 1.3 **??**. 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`.

1.2.2.3 Flush+Reload

Flush+Reload 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 [79]. 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

Listing 1.4: A code snippet representing the Flush+Reload attack

```
mfence ;serializing instruction stream
lfence ;serializing instruction stream
rdtsc ;first time stamp generation
lfence ;serialising instruction stream
mov (x2), %x3 ;reading address previously written in x3
mov %1, (x2) ;storing the attacked address in x3
lfence ;serialising instruction stream
rdtsc ;second time stamp generation
sub1 (x3), (x2) ;computation taken time by victim
cflush 0(%1) ;flushing cache line
:"=2" (time) ;storing time variable in x2
:"3" (adrs) ;storing address to attack in x2
```

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 [102] or in the

snippet code reported in Listing 1.4**??**.

1.2.3 Military Point Of View

These series of issues and the impossibility to have a "*silver bullet*" solution for all the possible HTHs bring up national-security implications of staggering significance, making this field pervasive in military applications.A particular reference is "*the hunt for the kill switch*" that provides evidences on existence of a Commercial-Off-The-Shelf (COTS) microprocessors in the Syrian radar that might have been purposely fabricated with a hidden "backdoor" inside. By sending a pre-programmed code to those chips, an unknown antagonist had disrupted the chips function and temporarily blocked the radar [60]. This means that to overcome the possibility to deal with infected hardware, military rely only on trusted foundries, and while this approach may seem effective, it has its limitations because the majority of western foundries are woefully behind their foreign counterparts when it comes to the level of technology they can provide. This seriously limits access to more advanced chips which are required for modern avionics and weapons systems [74]. This scenario has brought the *Defense Advanced Research Projects Agency* (DARPA) in USA to release a program called *Integrity and Reliability of Integrated Circuits* (IRIS), and also other country created networks with expertise on this field and highly trained staff who works on it.

1.2.4 Space Point Of View

The idea of employing COTS components, 3PIPs cores and commercial CAD tools in space systems is becoming increasingly popular [29, 70]. Indeed, this design choice represents an interesting trade-off between performance and cost for application scenarios where the ICs production volume is extremely low, as the case of space applications.

The basic idea is to reuse legacy (i.e., not specifically designed for space) subsystems, components and 3PIPs in space systems to benefit from both reduced development cost and high performance [23].

When specifically looking at microprocessors, designers can either choose to define their own instruction set architecture (ISA) or use an already available ISA. While the employment of open-source software can be considered as a legacy procedure, hardware has not yet fully experienced the disruptive effects of openness. Nevertheless, over the last years the RISC-V architecture has risen in popularity, drawing the attention of several universities and companies previously focusing on other open and free ISAs, proprietary ISAs or even on ISAs designed in-house (with the big drawback of having to design and maintain a software ecosystem) [22].

While reducing costs, this design paradigm exposes the obtained system to a number of security threats both at design-time and at runtime. The production of commercial ICs is characterized by a globalized supply chain [24]. The benefit of such a globalized supply chain is a reduction of design cost and time that, on the other hand, comes at the cost of a loss of trust in the final ICs [62]. The consequence of is that it is very hard to ensure/assess the trustworthiness of all the parties involved in the supply chain. Therefore, the product is exposed to a number of threats: ICs may be overproduced by the foundry and sold in the black market [40]; defective or dismissed ICs may be delivered as good ones [39]; IP core licenses may be violated and IP cores may be overused [26, 15]; designs may be maliciously modified to insert stealthy unwanted functionalities in the final product, also known as Hardware Trojan Horses (HTHs) [87].

A novel menace that represents a serious security-issue for microprocessor-based systems and that raised in the last years are MSCAs [33] [81]. As discussed before

these kind of attacks do not require the attacker to have physical access to the system under attack.

In the past, hardware security was not considered an issue by the space industry because most attacks, e.g., fault attacks, required the attacker to have physical access to the attacked system, which indeed is not the case of space applications. Nevertheless, attacks based on MSCAs and HTHs may be successfully carried out by without any physical access (at the time of the attack) to the system. Indeed, once implanted, a HTH can be activated remotely through the execution of a trigger program as well as a MSCA can be carried out by forcing the execution of a piece of malicious software running in parallel with the attacked program. Therefore, hardware security issues should nowadays be tackled also when considering a digital design meant for a space application [16]. Microprocessors are present in On-Board Data Handling systems and On-Board Data Processing systems, both has hard cores in ASICs/SoCs and as soft cores in FPGAs. Microprocessors are key components to run the algorithms required for the Attitude Determination and Control System (ADCS) and execute ground-control or algorithm-based Command & Data Handling (C&DH) [31]. The orientation is critical for the satellite's mission objective (e.g. a wrong orientation would make communication with the ground station not possible). The execution of C&DH is critical too, given the master/agent relationship of ground stations to satellites [31].

1.2.4.1 Intentional Data Corruption

The data may be intentionally altered by an attacker at its source (e.g. sensor jamming). As a matter of fact, each satellite employs tens of sensors that are responsible for collecting data [31]. satellites typically have several generic sensors, for instance

an antenna, a Global Positioning System (GPS), a magnetometer, a sun sensor, a horizontal plane sensor, a star tracker, a angular rate sensor, and a temperature sensor [31]. There are sensors specific for certain applications, for example, a surveillance or remote sensing typically includes: digital cameras, light-detection and ranging (LiDAR) systems, synthetic aperture radar (SAR) systems, and multispectral and hyperspectral scanners [34].

Another possibility is that data is altered during its transmission from the instrument to the ground system (the processor and memories play a big role here). In the latter case, intentional data corruption may be due to the execution of malicious software. Along with corruption of information coming from the instruments, intentional data corruption could potentially lead to a catastrophic loss of the system, e.g. if, at the reception of a command, no action or a wrong action is taken by the spacecraft. For example, if a navigational maneuvering command is altered, the spacecraft may eventually enter an unusable orbit and miss the encounter with a target, or even be destroyed [17].

1.2.4.2 Software Threats

Users, system operators and programmers may make mistakes that can cause security problems [17].

Furthermore, the system operator may configure the system incorrectly, leading to security weakness. On the other hand, the programmer may introduce logic or implementation errors that may lead to system vulnerabilities [16].

Finally, external agents may try to use vulnerabilities to inject software or change configurations. The effect can be data loss, loss of spacecraft control, unauthorized spacecraft control, or mission failure. An example of mission failure is described in

[31]: in 2005, the Demonstration of Autonomous Rendezvous Technology (DART) spacecraft was launched into orbit with the goal of autonomously navigating around a satellite. Less than 11 hours into the mission, DART collided with the satellite. The collision was caused by a GPS sensor error that caused navigation calculations to be off by 0.6 m/s. This was likely caused by a human-created software error [31].

Software Threats (STs) may also come from the exploitation of hardware- and architecture-level vulnerabilities. For instance, high-performance processors may employ speculation and out-of-order execution. These features can be maliciously used to access protected processor memory locations, opening the way to so-called Microarchitectural Side-Channel Attacks (MSCAs) [33]. This type of SCAs is particularly critical (also for space applications) because it does not require physical access to the system under attack, relying on the monitoring of the features of interest for the attack itself, e.g., timing behavior, performance counters, of the attacked processor. Some examples of MSCA, are the *Meltdown* [52], *Spectre* [48] and *Foreshadow* [98], [92] where the attacker exploits out-of-order and speculative execution to break address space isolation without exploiting any software bug. Thus, these attacks allow the attacker to access unauthorized memory regions and steal information. As a result, a system built using trusted components may be successfully attacked.

1.2.4.3 Unauthorized Access

An access control strategy based on authentication allows operations only to authorized entities and block all of others. If there is no access control (or it is not strong), the result may be a malicious (unauthorized) access to memory locations of the system containing private information. The occurrence of these undesirable events can compromise the satellite mission, allowing third parties to take control of the system

or access confidential information [17].

A way to overcome authentication that could be exploited by an attacker is replay, i.e. the transmission from a different ground station to a spacecraft of recorded communications previously intercepted.

An unauthorized access could be also caused by an attack to the legitimate ground station, allowing the attacker to upload malicious commands to a spacecraft or obtain/corrupt confidential data. A similar attack, when commands are unknown or no previous communications were intercepted, may aim instead to shut down the ground station (or network) [17, 16].

1.2.4.4 Malicious hardware

Issues related to hardware trust arise from involvement of untrusted entities in the life cycle of the designed system, including untrusted IP providers, design team components, CAD tools and fabrication, test, and distribution facilities [12]. It is possible that malicious HW finds its way in the system when COTS components are employed. These components usually do not have a trusted supply chain and are deployed to mass markets. As a matter of fact, the supply chain for COTS components consist of many steps, from the design to fabrication services [12]. Potentially, in any of these phases, an operator may modify the circuit in a malicious way.

An undesired addition or modification to existing circuit elements, done in order to apply malicious activity is called Hardware Trojan (HT). HTs can change the functionality of a circuit, reduce its reliability or leak valuable information [12]. It has been known that *software-exploitable HTs* may be integrated in Microprocessor. Thanks to an HT, attackers may be able to execute their own malicious software, to modify the running software or to acquire root privileges [45, 91, 93]; or they

may be able to enter in supervisor mode thanks to a backdoor activated via software [2]. In this way, attackers may be able to access "forbidden" memory regions and obtain confidential information. As a result, a circuit with trustable components could be attacked; an attack-proof circuit does not necessarily have components that have not been tampered with. On the other hand, rad-hard components are usually more trustable, as they are developed under the supervision of governmental agencies. Similar situation for FPGA designs.

More recently, a new security-related menace raised. Indeed, it has been demonstrated that HTHs may be introduced in the designed circuit by the employed CAD tool [76, 72]. In [32, 44] the don't care conditions of the design are exploited to insert HTHs both in the RTL code or gate-level netlist. In [71] a high-level synthesis tool able to maliciously modify the system under design has been presented. Indeed, starting from a high-level specification, i.e., a C/C++/SystemC, of the desired functionality the malicious CAD tool produces an HTH-infested implementation of the corresponding IP core. The same authors also demonstrated that several types of HTHs could be inserted in the produced IP core. The presented CAD tool was indeed able to insert HTHs downgrading performance, changing the implemented functionality and draining the battery of the system. Finally, in [8] it has been demonstrated that all electronic CAD tools, i.e., high-level synthesis, logic synthesis, physical design, verification, test, and post-silicon validation, may represent threat vectors. Very similar considerations can be made when looking at the FPGA scenario instead of the ASIC one. It has been demonstrated that CAD tools represent a threat for the security and trust of FPGA-based systems [85, 104, 27]. Indeed, malicious CAD tools may tamper the produced bitstream before FPGA configuration to introduce HTHs in the system [28, 84].

1.2.5 Impact of threats on different space missions

The impact of threats depend first of all on the mission orbit [16].

1.2.5.1 Orbits

On the one hand, Geostationary Orbit (GEO) satellites are more secure than Low-Earth Orbit (LEO) missions because they require ground stations with higher transmission power and larger antennas, limiting the amount of attackers capable of communicating with them. On the other hand, they are more vulnerable than LEO satellites because they are continuously visible in a certain geographical area [17], while LEO satellites are visible only for certain time intervals from a certain ground station. However, constellations increase the vulnerability of the space system, because in this case there are several satellites in orbit, increasing the visibility window from a specific point on Earth [17]. Finally, deep-space/interplanetary missions require even larger antennas and higher power compared to a GEO satellite to attack [17].

1.2.5.2 Type of missions

The described threats have different impact and likelihood depending on the type of mission [16].

1.2.5.2.1 Earth Observation Satellites Earth Observation Satellites can be either scientific or a critical asset of governments (e.g. meteorological). In the latter case the most probable threat is unauthorized access, which is typically countered with encryption of commands and mission data and authenticated commands. Also malicious hardware is a possible threat that can be mitigated with analysis of hardware functionality, or careful selection of a trusted supplier. Data corruption may also be

an issue, providing wrong information to the final user.

1.2.5.2.2 Communication Satellites These satellites are meant to provide a service with a usually very high availability [16]. For example, in the case of a GEO telecommunication satellite the whole space system is expected to provide a certain service 99.9% of the time [49]. Therefore, these type of satellites are particularly vulnerable, as even attacks that only aim to slightly reduce their availability can have a huge impact on the service. An example of the possible impact of an attack on a communication satellite is reported in [31]: in 2003, the Telestar 12 satellite was intentionally attacked using an uplink station that sent contradictory frequency to the satellite, which overrode the signal, effectively blocking television programming.

1.2.5.2.3 Navigation Satellites Navigation satellites are often dual-use satellites, therefore belonging to critical government infrastructures. An example is the GPS, employed by individuals, companies and the military. Like communications satellites, the loss of navigation satellite systems would result in potential loss of life, safety of individuals, and critical infrastructure [16].

1.2.5.2.4 Science missions They are usually not a part of a critical infrastructure. As a result, even if they are exposed to the same threats as other missions, the impact of these threat is much less critical [17]. For instance, in the case of an attack causing the loss of a scientific mission, the damage is limited to the financial investment in the mission [17]. However, these investment might be very high. For instance, the cost of the recently launched James Webb Space Telescope reached 10 billion USD [69]. The major risk in this case is that the loss of a similar investment may damage the reputation of an entity and even lead to budget cuts for future missions from

founders.

Chapter 2

On the Security of RISC-V Processors

When specifically looking at microprocessors, designers can either choose to define their own instruction set architecture (ISA) or use an already available ISA. While the employment of open-source software can be considered as a legacy procedure, hardware has not yet fully experienced the disruptive effects of openness. Nevertheless, over the last years the RISC-V architecture has risen in popularity, drawing the attention of several universities and companies previously focusing on other open and free ISAs, proprietary ISAs or even on ISAs designed in-house (with the big drawback of having to design and maintain a software ecosystem) [22].

2.1 The RISC-V Instruction Set Architecture

RISC-V was originally developed by UC Berkeley to support computer architecture research and education oriented at hardware implementations because a flexible, open and free ISA fit for such purpose was not available [96]. The spread of an open and free ISA like RISC-V already enabled a vast field of research activities for terrestrial application (e.g. security, AI, etc.), as accessing proprietary architectures is costly and limits what can be done with a certain product or within a certain research activity. The availability of a software ecosystem supported by a large open community ignited

an unprecedented amount of developments, with several announcements and/or releases of open-source implementations. The adoption of a popular free and open ISA can thus lead to shorter time to market and lower costs thanks to reuse.

DARPA funded RISC-V in its very beginning and is continuing to fund other activities related to the spin-in of open-source IPs in trustable electronic systems [77]. The reason behind this is that open-source IPs and open ISAs can reduce the costs, time, and complexity required for secure and trusted custom SoC design, as detailed information available with open-source IP can be found by inspection. Ad-hoc improvements or modifications for secure and trusted application are much easier, avoiding the need to design everything from scratch and focusing only on the critical part and thus ultimately increasing reuse [77].

The work in [20] is an example of how the openness of RISC-V enables academia to work on systematic approaches to eliminating attack surfaces instead of fixing specific security holes and provide strong security guarantees against cache timing and memory access pattern attacks. Another example is [59], which proposes an extension of RISC-V against temporal and spatial memory attacks.

The main feature of the RISC-V ISA specifications is modularity, which allows to cover a large spectrum of applications and to increase software reuse (adding new instructions as optional extensions instead of releasing new versions of the whole ISA). The RISC-V manual is structured in two volumes, one for the user-level ISA and the other describing the privileged architecture.

RISC-V allows both standard and non-standard extensions (defined outside the specifications). The user-level RISC-V ISA is defined as a base integer (I) ISA, which must be present in any implementation, plus optional extensions to the base ISA. A subset of the integer base (E) can optionally be implemented when an implementation

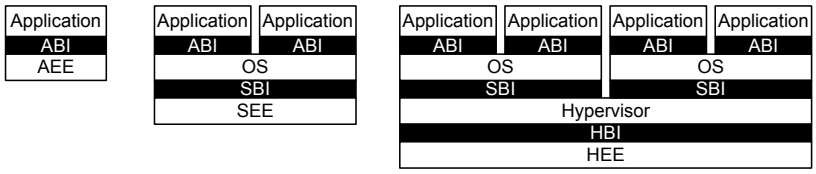


Figure 2.1: Left: Single application; Center: Multiprogrammed execution of multiple applications; Right: Multiprogrammed OSs supported by a single hypervisor [95]

targets small 32-bit microcontrollers, with 16 general purpose registers instead of 32. The standard defines a "general" subset (G) (comprising the IMAFD subsets) as the set of extension required for general purpose computing systems.

sectionRole of the Instruction Set Architecture

The main tool available at software level to preserve the security of a system is memory isolation (i.e. each process has its own address space, which isolates memory between programs) [80]. Isolation implies that even if the security of a partition is compromised, the attacker cannot breach the security of other partitions. Memory isolation provides also fault isolation (a fault in one partition does not affect others, e.g. an error in an application will not crash the OS) [80] and is a very sought-after capability in modern dependable embedded systems.

The Linux kernel provides separation between kernel services and user-space applications. However, in this way the kernel is a single-point of failure in terms of security, especially keeping into account that the kernel is 18 million lines of code. Therefore, it may be responsible of a high number of vulnerabilities, given its huge attack surface [35].

Isolation can be improved employing hypervisors to provide isolation between different instances of OSs running on multicore processors. Isolation is currently being investigated in space applications to improve safety behavior, i.e. to avoid that the failure of an application running on an OS may impact the execution of another

application running in a different OS on the same processor [37].

Given the increasing complexity of software systems over time, designers increasingly rely on 3rd-party software components, typically provided as software libraries. When the code of these libraries is open source, an analysis and security validation is possible. Proprietary software instead may come as linkable object modules. Therefore, it is often required to enforce the separation of the various software components within the system [35].

The use of general-purpose processors in space systems is no the rise, as they allow the use of use largely supported Linux-like OSs for general-purpose computing, greatly increasing software modules reuse and enabling complex applications (running third-party code in protected mode being one of the most requested features, lately).

2.1.1 RISC-V software stacks and privilege levels

The main tool to address isolation in an embedded system based on RISC-V processors is the use of privilege levels. Furthermore, the modular nature of RISC-V allows different software stack implementations with different levels of security [23]. For instance, Figure 2.1 shows the stacks described in the RISC-V Privileged Specification [95]. On the left, a simple stack implementation is shown. The application interacts via an Application Binary Interface (ABI) with the Application Execution Environment (AEE). In this simple case the ABI is composed of the user-level ISA subsets implemented. At the center, multiple applications run and communicate via the ABI with the OS (the latter in this case providing the AEE). In turn, the OS interface with a supervisor execution environment (SEE) via a Supervisor Binary Interface (SBI). An SBI comprises the user-level and supervisor-level ISA together with a set of SBI function calls. The SEE can be a simple bootloader and BIOS-style IO

system on a low-end hardware platform, or a virtual machine in a high-end server, or a thin translation layer over a host operating system in an architecture simulation environment. On the right, RISC-V runs a virtual machine monitor configuration where multiple OSs run on top of a hypervisor. Each OS communicates via an SBI with the hypervisor, which provides the SEE. The hypervisor communicates with the hypervisor execution environment (HEE) using a hypervisor binary interface (HBI) to isolate the hypervisor from the hardware platform.

Privilege levels are used to provide protection between different components of the described software stacks. At any time, a RISC-V thread is running at some privilege level. The possible levels are:

- The machine level (M-mode) has the highest privileges and is the only mandatory privilege level for a RISC-V hardware platform. Code run in machine-mode is to be trusted, as it has access to the machine implementation.

- Supervisor mode: added to provide isolation between a supervisor-level Operating System (OS) and the SEE.

- User-mode for application code. Many RISC-V implementations will also support at least user mode (U-mode) to protect the rest of the system from application.

An attempt to perform operations not permitted by the current privilege mode will cause an exception to be raised. These exceptions will normally cause traps into an underlying execution environment [95].

The Hypervisor (H) mode (designed to support Type-1 hypervisors) has been removed and the encoding space reserved, as the RISC-V community is focusing on

hypervisor support via an extended S mode suitable for both Type-1 and Type-2 hypervisors[1].

Implementations can choose which modes to implement (except for the mandatory M-mode), trading off reduced isolation for lower implementation cost [95]. Typical solutions are:

- M: simple embedded systems where security is not a concern. This solution will provide no protection against incorrect or malicious application code.
- M, U: secure embedded systems
- M, S, U: for systems running Unix-like operating systems. Supervisor mode (S-mode) can be added to provide isolation between a supervisor-level operating system and the SEE.

A thread normally runs application code in U-mode until some trap (e.g., a supervisor call or a timer interrupt) forces a switch to a trap handler, which usually runs in a more privileged mode. The hart will then execute the trap handler, which will eventually resume execution at or after the original trapped instruction in U-mode. Application code is usually run in U-mode until some trap such as a supervisor call or a timer interrupt forces a switch to a trap handler, which usually runs in a more privileged mode. The hart will then execute the trap handler, which will eventually resume execution at or after the original trapped instruction in U-mode. Traps that increase privilege level are termed vertical traps, while traps that remain at the same privilege level are termed horizontal traps. The RISC-V privileged architecture pro-

[1]A type-1 hypervisor (bare-metal) runs directly on the host's hardware. A type-2 hypervisor (hosted) resides above a conventional host operating system and provides a full set of virtual hardware resources to the above guest OS [80]

vides flexible routing of traps to different privilege levels. Each privilege level has a core set of privileged ISA extensions with optional extensions and variants.

2.1.2 RISC-V extensions for security

Beside different privilege modes, the RISC-V ISA provides several extensions to increase security. The reminder of this section describes them.

2.1.2.1 Physical Memory Protection

The M-mode supports an optional standard extension for Physical Memory Protection (PMP). The PMP extension defines control registers to specify access privileges (read, write, execute) for each physical memory region. Attempting to fetch an instruction from a PMP region that does not have execute permissions or to load from a physical memory location within a PMP region without read permissions raises an exception.

The use of the PMP provides many security features. For example, threads should be prevented from modifying or even reading the data of shared libraries. Therefore, the memory region in which the shared library data reside can be set as execute only using PMP.

Even if the PMP is mainly used to prevent threads running in lower privilege levels (e.g. U and S modes) from accessing privileged memory contents, the "lock" feature provides protection even with only the M-mode implemented. As a matter of fact, if a PMP entry is locked, writes to the configuration register and associated address registers of the entry are ignored and PMP restrictions are valid also for the M-mode. Locked PMP entries may only be unlocked with a system reset.

2.1.2.2 Cryptographic extension

The basic requirement for security in spacecrafts is the use of encrypted TC/TM signals with AES (which is emerging as a de-facto standard also in space) [54].

The implementation of a cryptographic extension increases performance and hides implementation details from software, reducing the attack surface [55]. However it is often preferable to implement encryption and decryption of telemetry and commands in a separated hardware component or board, as it is required that the keys are not accessible from software.

2.1.2.3 User-level interrupt extension (N)

This extension allows isolation of interrupts, as the interrupt handlers can be executed at user-level, thus being unable to compromise the isolation model [35].

2.1.3 Limits of isolation and need for hardware solutions

Most attacks to isolation exploit physical access to the processor to perform Fault Injection (FI) [64]. In space this may be more complex than in applications like mobile and Internet of Things (IoT), although in principle still possible. For instance, in [64] it is shown how knowledge of power consumption for each instruction can be employed to skip an instruction during context switch, allowing access to PMP protected portions of memory. On the other hand, the previously presented attacks, i.e., MSCAs and HTHs, can be successfully carried out by the attacker without any physical access (at the time of the attack) to the system under attack. Indeed, HTHs can be implanted at design time by one of the many untrusted parties involved in the design. Once implanted, a HTH can "easily" be activated through the execution of a trigger program. Similarly, a MSCA is a piece of malicious software running in parallel

with the attacked software. Therefore, MSCAs and HTHs have to be considered serious threats also in the space scenario, where it is extremely difficult for the attacker to have physical access to the system. While the security features natively integrated in the RISC-V architecture do not protect the system from MSCAs and HTHs, we argue that innovative security solutions (coming from research) should be integrated in the RISC-V architecture to make space missions adopting this microprocessor secure and trustable.

Chapter 3

Malicious Activities Detection in Microprocessors: Methodologies against Untrustable Users

There are various detection methods in the state of art to detect malicious activities in microprocessors. This chapter describes some system-level methods to protect microprocessor-based systems from MSCA and HTH. In particular, the proposed architectures aim to provide secure and trusted program execution through systems built with untrusted components. The main idea behind most of these methods is to introduce an Hardware Security Checker (HSC) capable of monitoring the fetching activity and processed data as of the microprocessor to both detect potentially suspicious activity such as e.g., HTHs and/or STHs and errors at the same time. Generally speaking, such HSCs work in two phases: in a first *training* step they learn which instructions/sequences of instructions are legitimate, while, in subsequent *monitoring* step, they check whether the fetching activity of the microprocessor corresponds to previously learned legitimate profiles or not.

In [107] is proposed an HSC, depicted Figure 3.1), which consists of three hardware monitors: an *IO tracker (ITR)*, a *memory tracker (MTR)* and a *dynamic reconfigurable processing unit (RCP)*. The RCP collects logs produced by the MTR and by

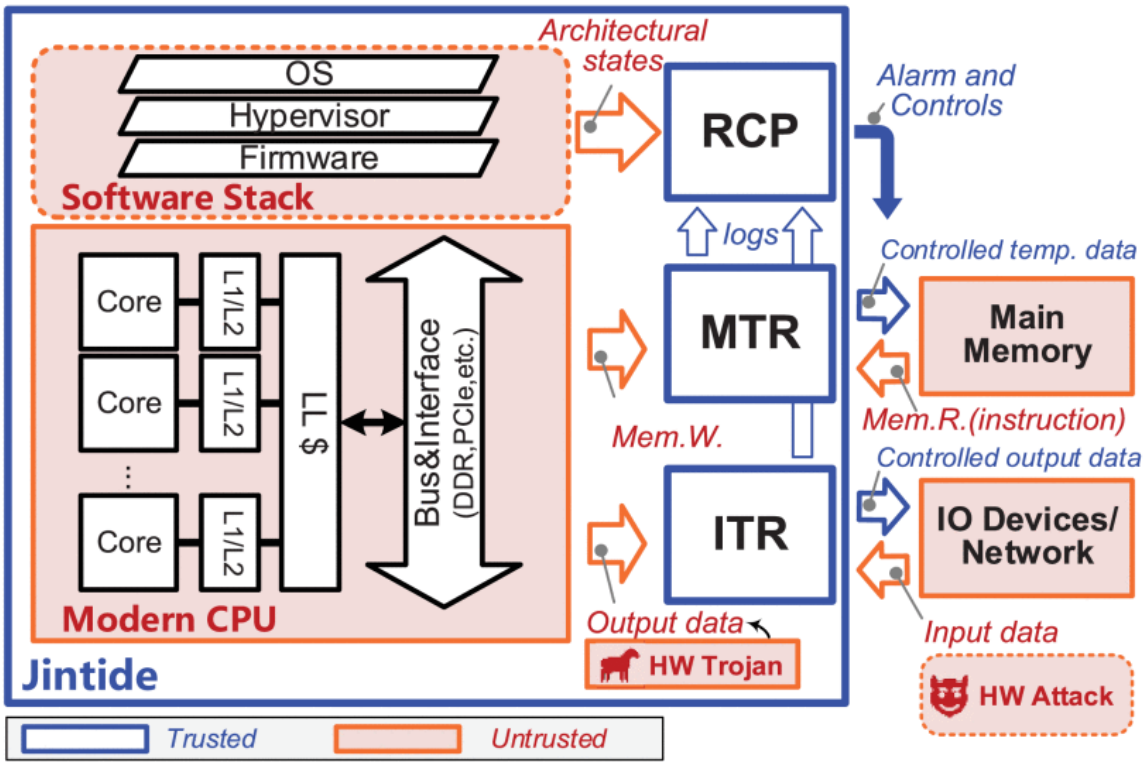


Figure 3.1: The Jintide architecture proposed in [107]

the ITR. In these logs there are information about both instruction memory read and write operations (including data input and data output). Moreover, the RCP collects information about the architecture status coming from the system hypervisor. This information is used by the RCP to *rerun* the instructions just executed by the microprocessor and to check whether the produced output and state after this check execution are the same as the ones after the nominal execution. Moreover, the RCP after collects this information and the one coming from the system hypervisor system. If the produced output and the microprocessor state are the same as the ones after the nominal execution, the [107] permits the operation; otherwise, it raises an alarm. In addition, the RCP collected information with previously defined patterns that are used to characterize specific malicious activities. By checking the executed instructions and produced results and whether the state of the processor is consistent with

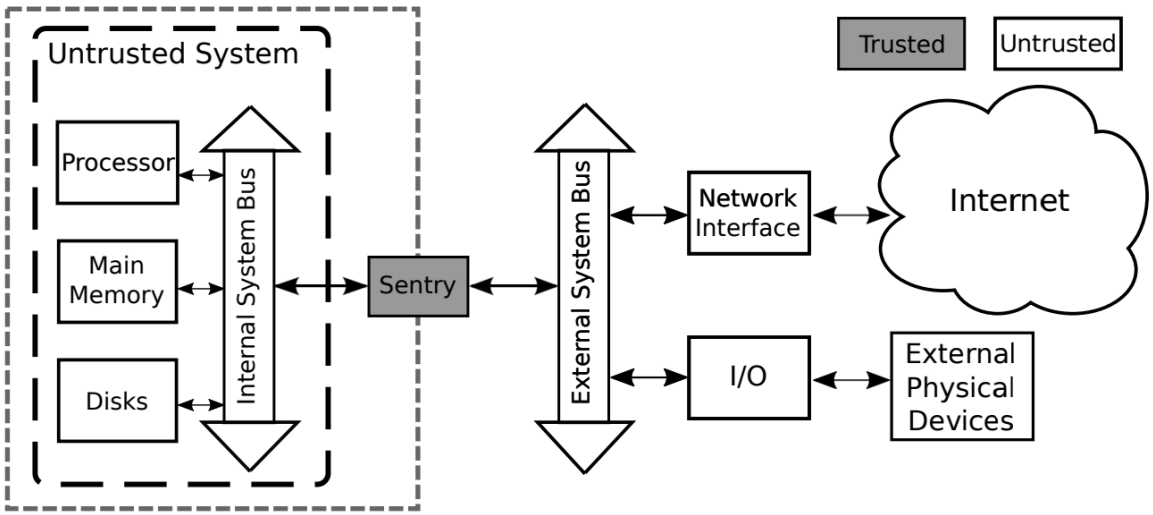


Figure 3.2: The TrustGuard approach proposed in [103]

the executed instructions themselves, Jintide is able not only to detect the activation of attacks (either Hardware Trojans or Microarchitectural side-channel) but also the occurrence of faults.

In [103] the authors present the *Sentry* security co-processor and its working method, dubbed *TrustGuard*. The idea behind TrustGuard solution is to isolate the microprocessor (together with the memory and storage units) and the I/O interfaces. Such isolation is implemented by the Sentry co-processor that acts as a filter between the microprocessor and the interfaces, as shown in Figure 3.2. The Sentry co-processor is designed jointly with the main processor, and it includes a combination of re-execution functional units and encrypted memory integrity schemes. In this scenario is needed continuous communication between the untrusted/unprotected processor and memory and Sentry. Indeed, Sentry is in charge of monitoring the state and the activity of the processor and checking whether the previously defined security rules are met or not. In there is also a shadow register file, containing all architectural register state produced by the verified instruction sequence, in order to ensure that

the processor does not misreport execution and state information, As for Jintide, also TrustGuard is able to detect both the activation of attacks (either Hardware Trojans or Microarchitectural side-channel) and the occurrence of faults.

In [105] the authors proposed an HSC dubbed *SCIFinder*. It results useful for the identification of security-critical properties to be used in the dynamic verification of a processor for the detection of HTHs. The workflow of SCIFinder has four phases:

1. Observe the state of the microprocessor while executing a number of programs. In this activity, are collected a set of likely processor invariants defined over software-visible processor state.

2. Given a list of known design flaws, use human expertise and judgment to classify each flaw as either a functional bug or a potential security vulnerability.

3. Among the previously identified invariants, define the Security-Critical Invariants (SCI) as those violated by the security vulnerabilities.

4. Via machine learning techniques is found additional SCI in the set of processor invariants.

It is worth mentioning that all these HSCs are able to detect both HTHs and Software threats. Much in detail *Jintide*, is also able to detect Spectre [48] and Meltdown [52] attacks.

In the state of art there are many techniques for boycotting and detecting HTHs and STHs. A specific one may be cache coloring [68]: cache is divided into regions and data are mapped into another on, depending on the data application. More in general, cache partitioning techniques result useful for boycott cache-based attacks,

as reported in [65]. Another solution against these threats is dynamically locking cache lines, as reported in [94]. Finally, another strategy to defend against hardware speculation attacks in microprocessors is making speculation invisible in the data cache hierarchy [100]. At this propose in [38] the authors present a speculative buffer, called SpecBuf. It is implemented as a part of Miss Status Holding Registers in BOOM's microprocessor [107] L1 data cache. SpecBuf holds speculative data in the until commit where the data will be transferred to the L1 cache. Typically, L2 data cache, refills would write the refill data into the tag and data arrays before waking up the corresponding MSHRs to return the load data to the core. Implementing SpecBuf, is modified cache refills to instead write the refill data into per-MSHR cache line buffers.

3.1 Probabilistic Data Structures

In the following paragraphs are reported the highlights of structures based on probabilistic computations, evaluated in microprocessor malicious activities detection scenario. The probabilistic data structures are highly adopted in big data analysis and are characterized by fast processing and efficient handling of the space. Much in detail, they result useful for evaluating membership check, counting event occurs, data frequency estimations or data similarity estimations.

3.1.1 Bloom Filters

Bloom filters (BFs) consist in probabilistic data structures that store membership information of a set of elements. A BF consist in a bit array where each dataset element is mapped array locations. The address of each location is depending on the output of an hash function calculated on the element itself. Two operations can be

performed on BFs: load and query. Loading a BF means storing in it all the elements of the set of interest. To do so, for each element x_i in the set and for each hash function $hash_j$ the bit array location associated with address $hash_j(x_i)$ is set to 1 (initially all bit arrays are 0), as reported in Figure 3.3. Querying a BF means checking whether an element is in the set of interest, and thus that the associated location(s) of the bit array is (are) set to 1. For a given element x_i that has to be checked and for each hash function $hash_j$ the BF raises an alarm if at least one of the bit array locations associated with address $hash_j(x_i)$ is found to be 0, as shown in Figure 3.4. A key feature of BFs is that, although querying may result in false positive in element presence, it is always true that there could not be false alarms. In other words, if a query of an element returns a positive there is a chance for it not to be in the set (undetected alarm in proposals [13, 67]); however if a negative is returned, it is not possible that the element is in the set (false alarm in proposals [13, 67]). Nevertheless, the percentage of such false element presences can be calculated as a functions of m, the size of the bit array (expressed in bits), of n, the number of elements that have been inserted in the BF and of k, the number of employed hash functions. The formula to calculate the undetected alarms rate (UAR) reported in equation 3.1.1. Therefore, as m increases, UAR decreases. The value of k that minimizes the UAR is given by equation 3.1.2.

$$UAR \approx (1 - e^{-\frac{n \cdot k}{m}})^k \tag{3.1.1}$$

$$k_{opt} = \frac{m}{n} \cdot ln2 \tag{3.1.2}$$

Reliability and security are crucial features in microprocessors. Via BFs is possible to ensure that errors do not affect the system behavior, thus the presence of this

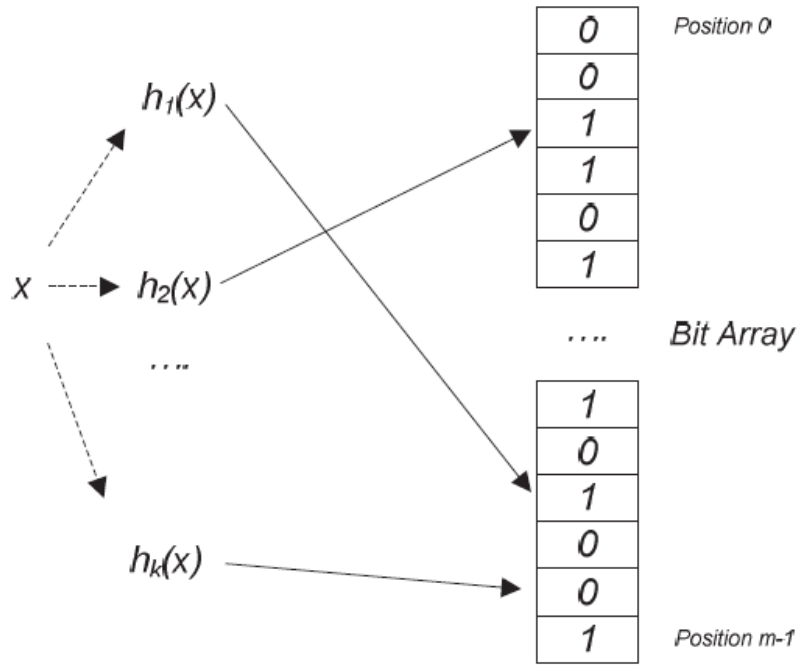


Figure 3.3: Diagram of a Bloom Filter [73]

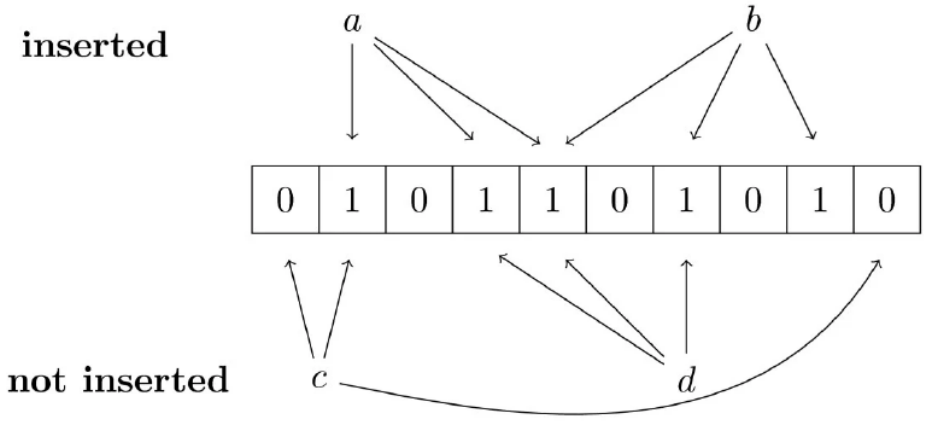


Figure 3.4: Example in which a and b are present in the Bloom Filter initial dataset, c is not, d is a false presence

architecture in global design should not to be considered as a further vulnerability [73]. The power of this choice is that these architectures are completely transparent w.r.t. the normal functioning of the system.

3.1.1.1 Bloom Filter-based Security Checking Module to Protect Microprocessor

In [13] the authors proposed a HSC based on Bloom filters to protect the system against HTHs infecting the memories (the architecture approach is shown in Figure 3.5). The HSC is programmed while the software is installed in the instruction memory. During this phase, the HSC stores information about legal instructions and corresponding memory addresses are concatenated and given as input to a number of hash functions composing the Bloom filter. The output of such has functions is then used to point to a number of array location where a logical 1 is stored to keep track of the legitimacy of the current instruction and memory address couple. At runtime, the HSC monitors memory location accessed by the microprocessor and the corresponding fetched instructions and based on this information it queries the Bloom filter. In particular, if the accessed memory address and the corresponding fetched instruction are legitimate, the response of the Bloom filter will be 1, otherwise it will be 0. In this way, the HSC is able to signal at runtime any illegitimate memory access and instruction execution due to both the activation of a HTH and the occurrence of a fault. Some [13] authors and I, proposed in [67] a lighter version of the HSC, with the very same capabilities, but implemented employing a significantly smaller amount of resources and power consumption. Both HSCs are implemented in a microprocessor based system, between core and instruction memory. Much in detail, in [13] the authors presented an hardware implementation of a HSC considered as unstrustable component the memory. On the other hand in [67] we have considered as untrusted

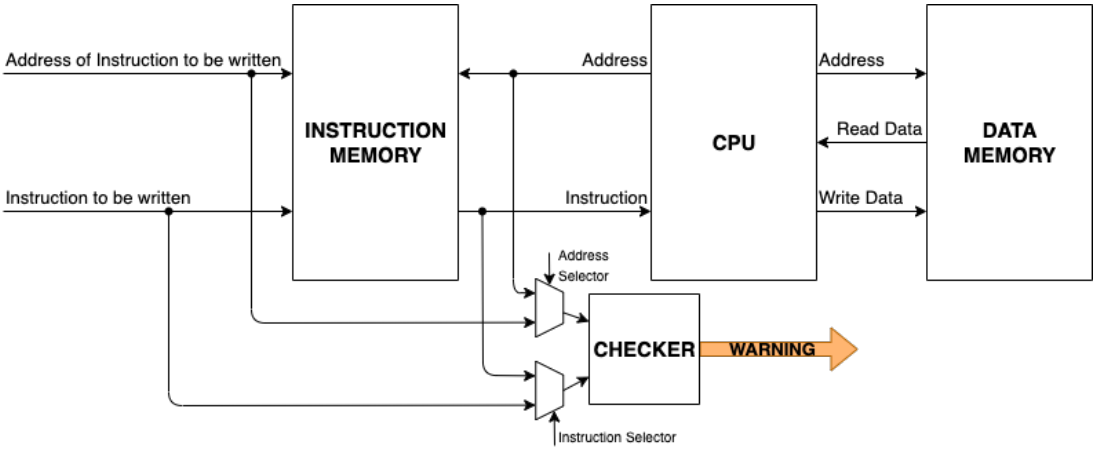


Figure 3.5: The security approach architecture proposed in [13] and [67]

component the core. Both papers evaluated the impact of HTHs that aim at changing the functionality of the system by forcing the CPU to execute an unwanted program. In particular, their considered HTH models are the following (and shown in Figure 3.7):

1. An HTH that modifies the memory address required for instruction and injects a not legit instruction;

2. An HTH that forces the memory to access an incorrect location injecting an illegal instruction;

3. An HTH that alters the content of the program counter and injects a legit instruction in a wrong moment of program execution.

3.1.1.1.1 The Security Checker architecture The architecture of [67] HSCs is depicted in Figure 3.6. The HSC takes in input memory addresses, instructions and a signal that specifies whether the HSC is working in configure or in query mode. As output, the HSC produces a warning. When working in configure mode, both the

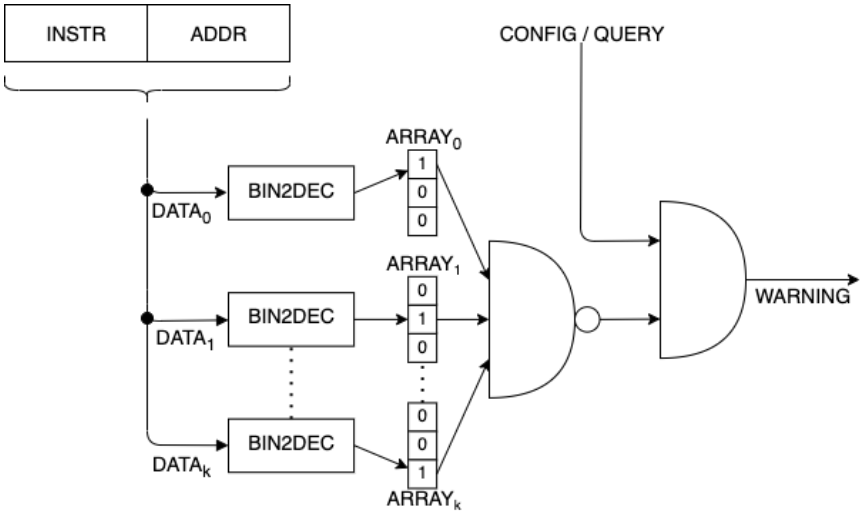
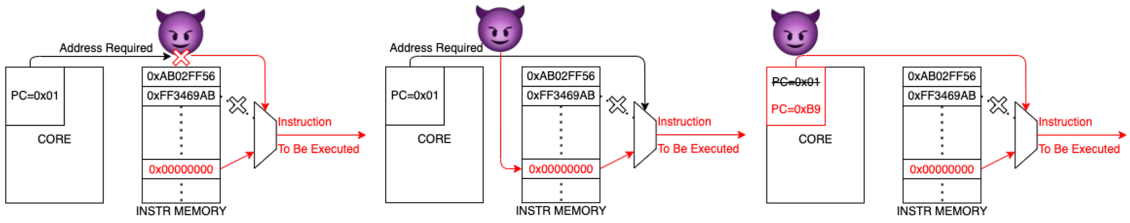


Figure 3.6: Hardware Security Checker Architecture proposed in [67]

addresses and instructions come from the user space that is installing a program in the instruction memory of the system; on the other hand, when working in query mode, addresses come from the core while the instructions come from the instruction memory. A combination of address and instruction is then used, in both configuring and querying phases of the HSC, to address a number of bit arrays within the HSC. Referring to k as the number of bit arrays in the HSC and we call it the *fragmentation factor*. The content of these k bit arrays is set at configuration time to keep track of all the address-instruction pairs legal for the program that is going to be installed. At query time the content of these bit arrays is read to check whether the current address-instruction pair is legal or not. As it can be noticed from Figure 3.6 a warning is raised if at least one of the accessed bit array locations is set to 0 and if the HSC is working in query mode.

3.1.1.1.2 The Security Checker configuration and usage As previously said, the HSC takes in input an address and an instruction. Such two input data are combined within the HSC and then fragmented into a number of data chunks ($\texttt{DATA}_0$



(a) HTH Model 1 (b) HTH Model 2 (c) HTH Model 3

Figure 3.7: Hardware Trojan Horses models for evaluation of [67, 13] Security Checkers

up to $\texttt{DATA}_k$ in Figure 3.6). In particular, being n the size in bit of addresses and instructions in the considered architecture, $\texttt{DATA}_0$ is composed of the first n/k bits of the address paired with the first n/k bits of the instruction, $\texttt{DATA}_1$ is composed of the second n/k bits of the address paired with the second n/k bits of the instruction and so on. The produced bit groups are then decoded and used to access specific locations of a number of bit arrays.

When working in configure mode, both the addresses and instructions come from the user space that is installing the program in the instruction memory of the system. After pairing the address and the instruction and producing the k data chunks such chunks are used as memory addresses to access the corresponding bit arrays. In particular, a 1 is written in each bit array location addressed by the corresponding chunk to *teach* to the HSC that the specific address-instruction pair is legal for the program.

Figures 3.8, 3.9 depict an example configuration procedure for two consecutive instructions in a system having 32 bit long addresses and instructions and where the HSC has a fragmentation factor of 4. It is worth observing that for different address-instruction pairs one or more data chunks may point to the same location of the corresponding bit array. This is the case of the first data chunk in the example that points to the middle bit in both Figures 3.8 and 3.9. This does not represent a

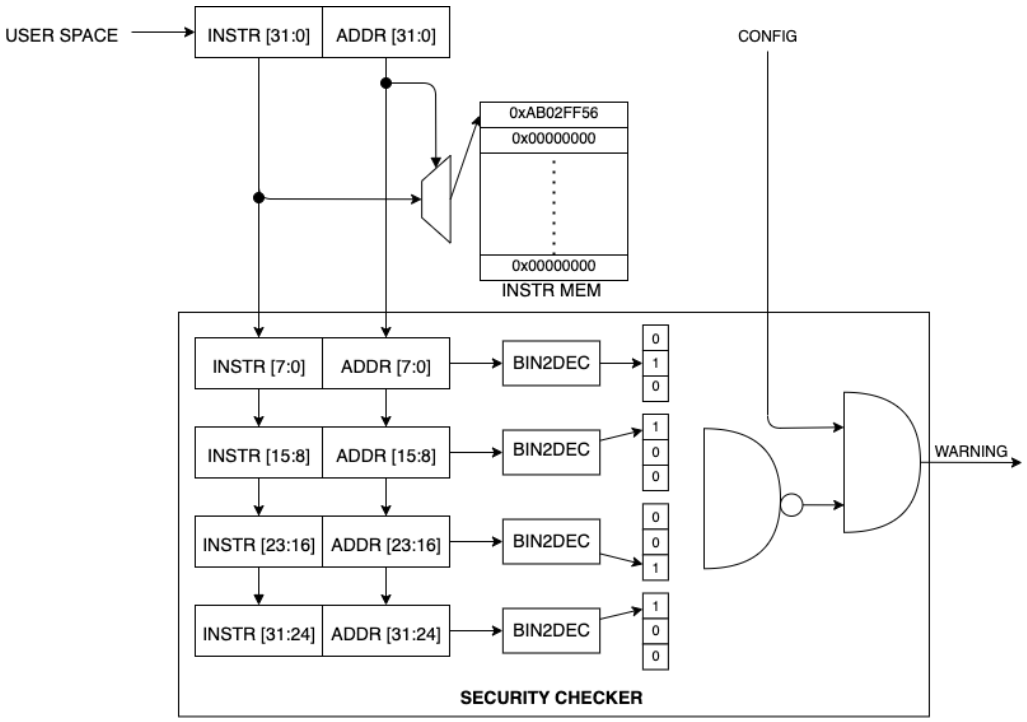


Figure 3.8: [67] Configuration phase: writing the first program instruction

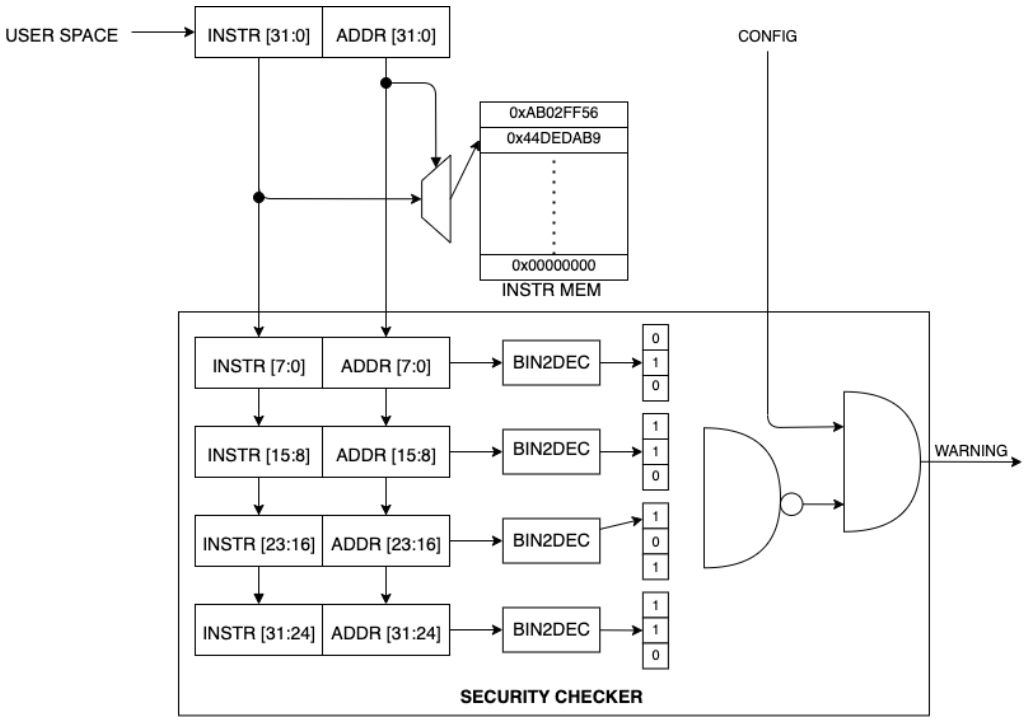


Figure 3.9: [67] Configuration phase: Writing the second program instruction

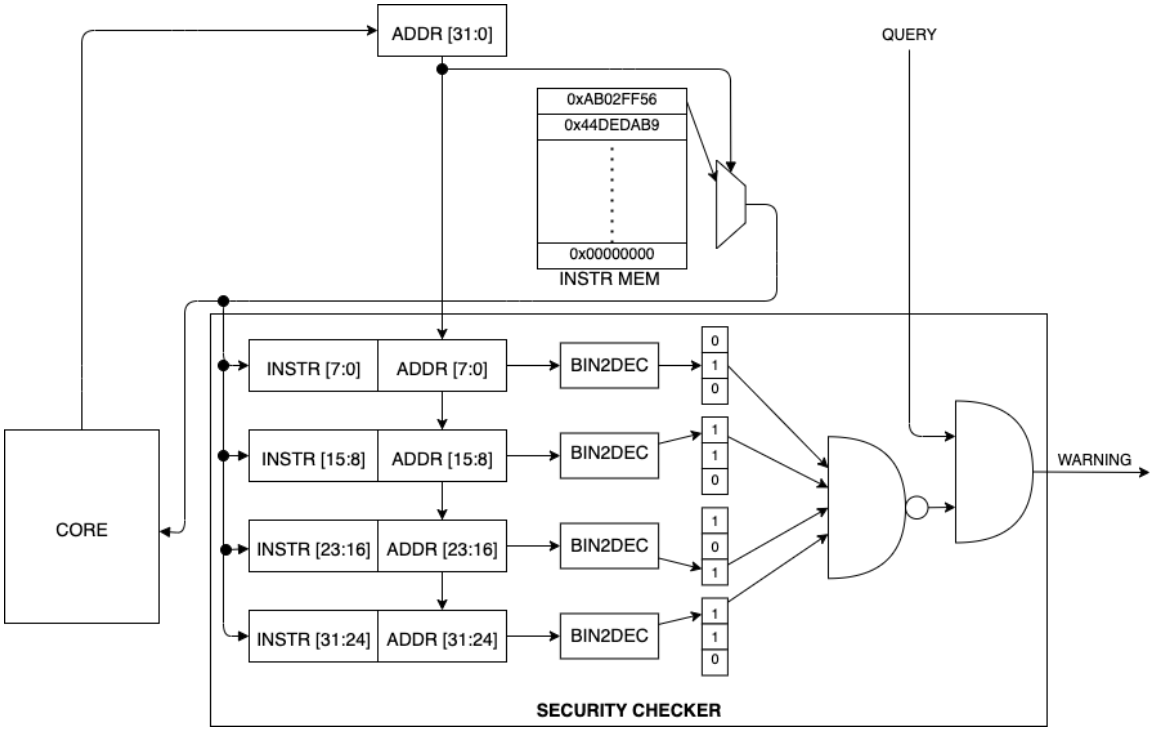


Figure 3.10: [67] Query phase: legit instruction read

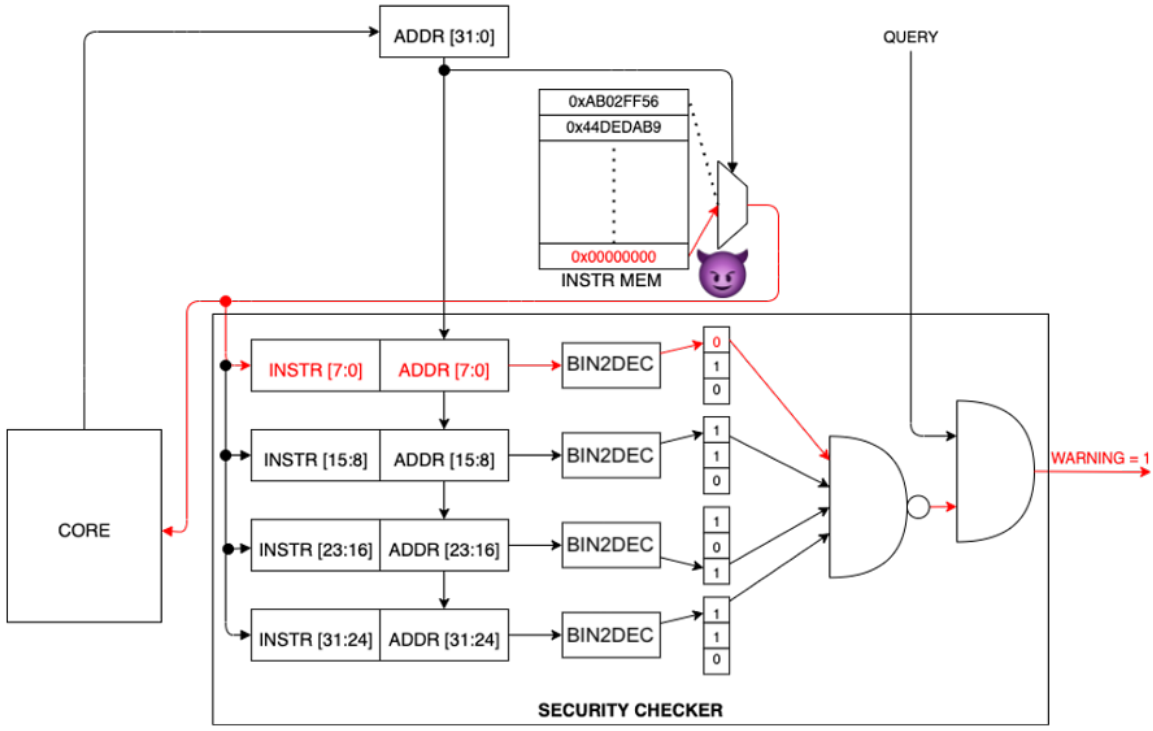


Figure 3.11: [67] Query phase: illegal instruction read

Table 3.1: The considered benchmark programs in [67, 13]

Benchmark	**#Instructions**
`Binary Search (BinS)`	215
`Matrix Multiplication (MM)`	216
`Bubble Sort (BubS)`	268
`Quick Sort (QS)`	1023
`Sudoku Solver (SS)`	475
`Motion Detection (MD)`	934

Table 3.2: Resource occupation and working frequency comparison between [13, 67] proposals

Bench.	**Proposal in [67]**				**Proposal in [13]**			
	#LUTs	#FFs	BRAM size	Freq. (MHz)	#LUTs	#FFs	BRAM size	Freq. (MHz)
`BinS`	75 (0.49%)	31 (0.31%)	208 Kbit	275 MHz	880 (5.83%)	84 (0.85%)	32 KBit	112 MHz
`MM`	75 (0.49%)	31 (0.31%)	208 Kbit	275 MHz	880 (5.83%)	84 (0.85%)	32 KBit	112 MHz
`BubS`	75 (0.49%)	31 (0.31%)	208 Kbit	275 MHz	880 (5.83%)	84 (0.85%)	32 KBit	112 MHz
`QS`	75 (0.49%)	31 (0.31%)	208 Kbit	275 MHz	880 (5.83%)	84 (0.85%)	32 KBit	112 MHz
`SS`	75 (0.49%)	31 (0.31%)	208 Kbit	275 MHz	1539 (10.19%)	89 (0.90%)	64 KBit	106 MHz
`MD`	75 (0.49%)	31 (0.31%)	208 Kbit	275 MHz	1539 (10.19%)	89 (0.90%)	64 KBit	106 MHz

problem, i.e., does not lead to false alarms, as those are triggered when no address-instruction pair in the program maps to the positions selected [67].

When working in query mode, addresses come from the core and, after reading the instruction memory, the instructions come from the instruction memory itself. The k data chunks are generated exactly as previously described but in this phase the content of the bit arrays is read. As soon as at least one of the read values is 0, the HSC raises an alarm. Figure 3.10 depicts an example query procedure for an example address in a system having 32 bit long addresses and instructions and where the HSC has a fragmentation factor of 4. Therefore, the main target of those HTHs is infesting the fetching unit of the core.

3.1.1.1.3 Experimental results The experiments have been conducted in the evaluation of both HSCs ([13] and [67]) on the following benchmarks: Binary Search

(`BinS`), Matrix Multiplication (`MM`), Bubble Sort (`BubS`), Quick Sort (`QS`), Sudoku Solver (`SS`) and Motion Detection (`MD`) (Table 3.1). For both evaluation campaigns have been considered the RI5CY [36] version of PULPINO [89], which is a small 4-stage RISC-V core. When synthesized on a Xilinx Artix XC7A35T, RI5CY requires 15097 LUTs and 9881 FFs and it works at about 50MHz. Table 3.2 reports the resource occupation comparison between the two HSCs and their relative overhead with respect to the PULPINO core ([67] HSC overhead is benchmark independent), while in Table 3.3 are reported their rate detection comparison (in [13] there is not an evaluation for HTH model 3). *False Negatives (FNs)* rates refer to those cases where the HTH is activated but the checker did not detect it; *False Positive (FPs)* rates indicate the cases where no HTHs are activated but the checker raised a false alarm. The activation of HTHs belonging to the previous models by modifying at a random time the instruction memory address from which the microprocessor fetches an instruction. In particular, in order to emulate an HTH that makes the microprocessor run a malicious program, have been simulated 10,000 randomly generated HTH activation cases (for each HTH model). Similarly, have been run 10,000 times benchmarks in which no HTH was activated for the evaluation of program execution in which the checker raised an alarm over the total number of runs. FN rates and [67] HSC area occupation depend on fragmentation factor. With $k = 1$ there will not occur any false negatives and false positives because, but the checker will require a 2^{64} bits memory, which is a too big number and the relative implementation is totally unfeasible. For the same reason, also a checker with a $k = 2$ (that would require two 2^{32} bits memories and have FN rates lower with respect to the HSC with $k = 4$) can be considered unfeasible for an embedded system. Solutions having $k = 8$ or greater (eight 2^8, sixteen 2^4 bits memories and so on) achieved extremely poor accuracy. Therefore, the

Table 3.3: FP and FN rates when the HTH modifies the accessed instruction memory location (HTH model 1 and 2)

Bench.	**Accuracy in [67] HTH Model 1 & 2**		**Accuracy in [13] HTH Model 1 & 2**		**Accuracy in [67] HTH Model 3**	
	FP	FN	FP	FN	FP	FN
`BinS`	0%	0%	0%	0.523%	0%	2.25%
`MM`	0%	0%	0%	0.520%	0%	0.40%
`BubS`	0%	0%	0%	0.572%	0%	3.01%
`QS`	0%	0%	0%	0.607%	0%	3.91%
`SD`	0%	0%	0%	0.249%	0%	2.83%
`MD`	0%	0%	0%	0.912%	0%	2.83%
AVG	0%	0%	0%	0.663%	0%	2.18%

only feasible checker configuration that provides acceptable accuracy (whose results are presented in tables 3.2 and 3.3 is the one having $k = 4$, thus requiring four 2^{16} bits memories.

3.1.2 Count-Min Sketch

A *Count–Min Sketch (CMS)* is a probabilistic data structure useful to estimate the occurrence frequencies of a stream of events belonging to different types [19]. 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

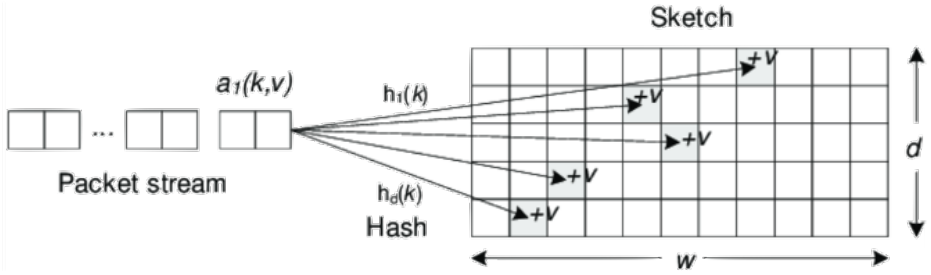


Figure 3.12: Count-Min Sketch Updating phase

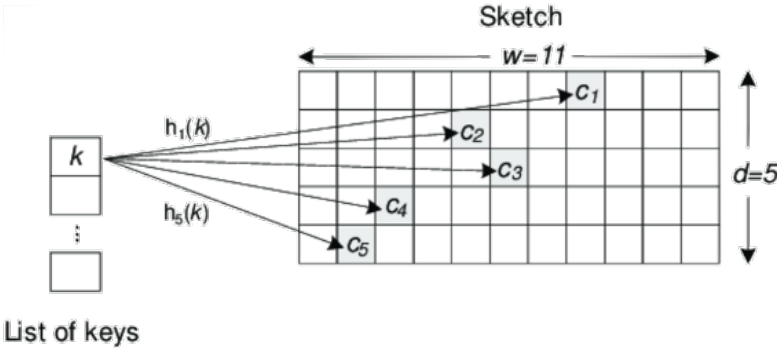


Figure 3.13: Count-Min Sketch Estimating phase

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 (Figure 3.12); the `Estimate(x)` operation provides the CMS estimation of the frequency of x (Figure 3.13). 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: $FP_p = e^{-k}$). 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. The probability that the difference between a value estimated by a CMS and the corresponding expected value is larger than $(e \cdot I)/m$ (where I corresponds to the possible different data input) is always smaller than $FP_p = e^{-k}$ [19], therefore, defining t:

$$\frac{e \cdot I}{m} = t \tag{3.1.3}$$

and, by inverting the formula is:

$$m = \frac{e \cdot I}{t} \tag{3.1.4}$$

A correlation between m, I and t is reported in Figure 3.14. At the end of the day, having I number of input and k number of hash functions, defining a certain acceptable false positive rate, it is possible to design how many counters (m) are needed.

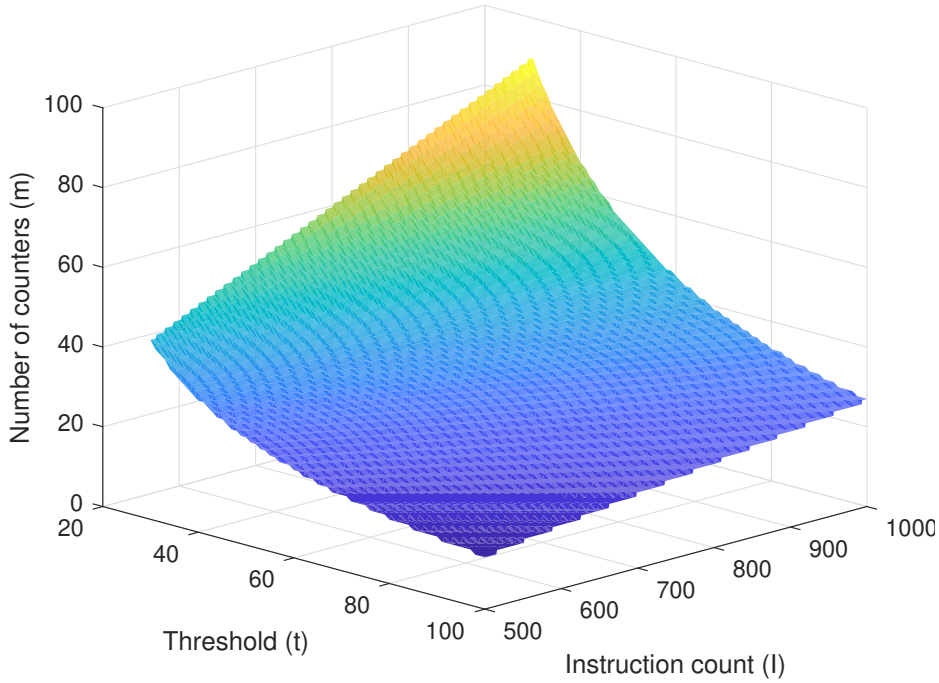


Figure 3.14: Correlation among m, I and t [19]

3.1.2.1 Count-Min Sketch-based Security Checking Module to Protect Microprocessor

In [7] I and some colleagues of mine, proposed a novel approach to detect and signal the occurrence of MSCAs into microprocessor-based systems. The architecture of this solution (depicted in Figure 3.16) relies on the insertion of an HSC on the instruction bus, between the instruction memory and the microprocessor. The HSC, by observing all the fetching activity performed by the microprocessor, is able to detect whether an attack is going on or not. In other words, it raises an alarm as soon as it recognizes the critical instructions composing the signature of an attack among the fetched instructions. More in detail, the HSC module works on a time-window base: within a time window (whose duration can be programmed by the user) the HSC observes and keeps track of all the instructions fetched by the microprocessor. Moreover, thanks to a Count-Min Sketch, the HSC is able to estimate the occurrence frequency of a

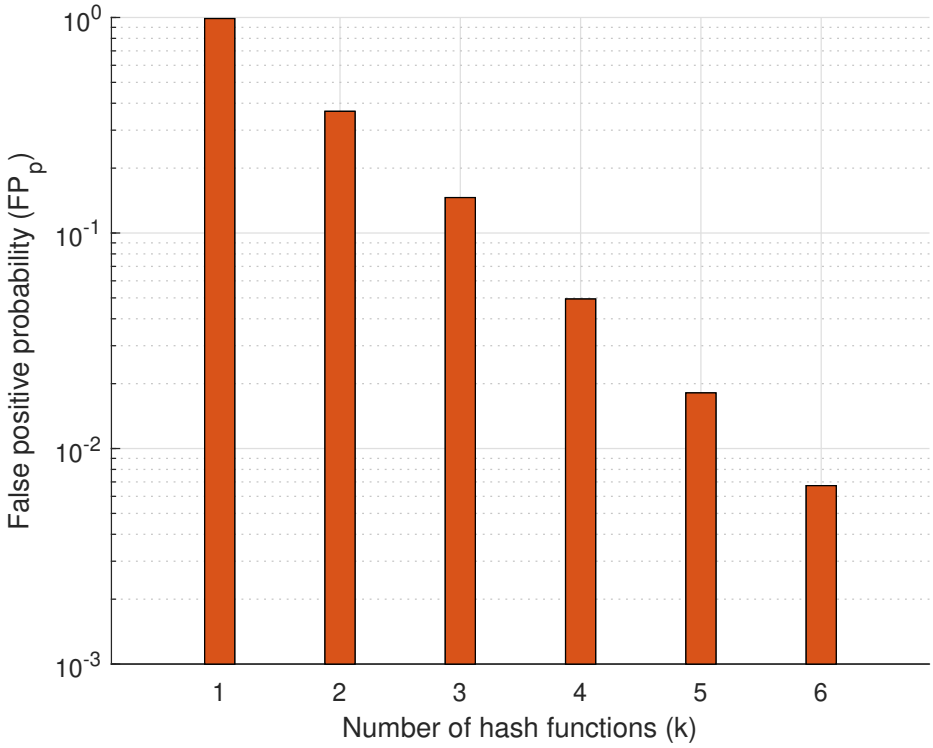


Figure 3.15: Correlation between k and theoretical worst case FP_p [19]

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. The attack models are described within the [7] 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.

If there is a match between at least one of the MSCA attack models programmed and the microprocessor 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 signal is raised. The duration of the time window and the attack models to check are programmable data, via an AXI bus, as represented by the red arrow in Figure 3.16. Thanks to its programmability, it is possible to always keep updated

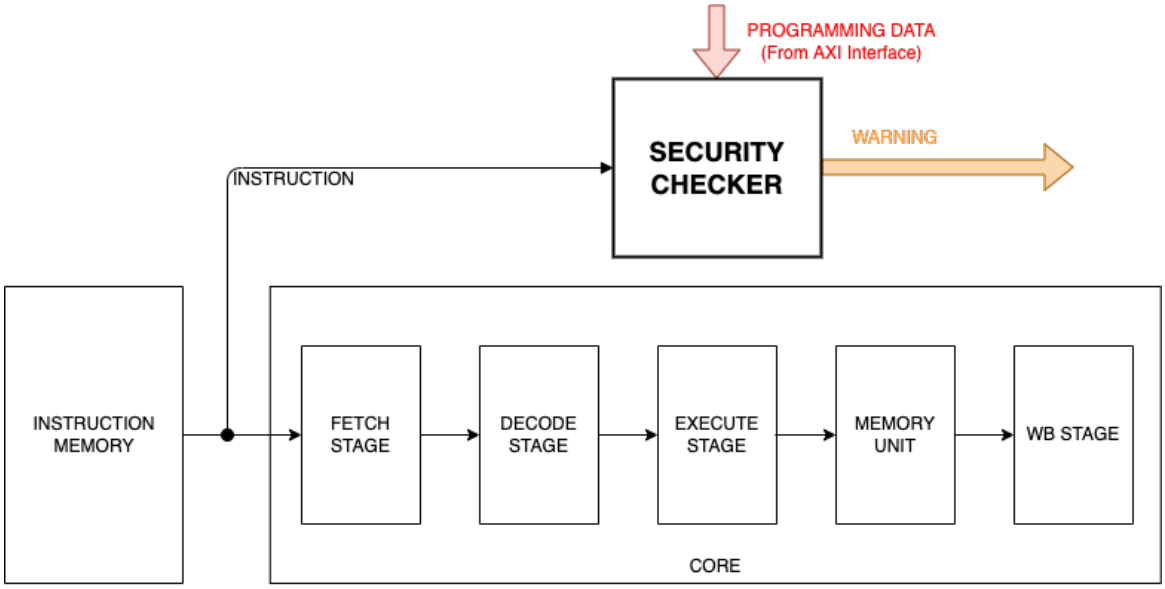


Figure 3.16: The architecture of the secured system including the checker proposed in [7]

the list of MSCAs the solution proposed in [7] is in charge to detect. Note that the [7] HSC is platform-independent and can be employed between instruction memory and core (to monitor its fetching activity), irrespective of the specific microprocessor features (e.g., out-of-order, in-order, or speculative execution).

3.1.2.1.1 The Hardware Security Checker architecture An high-level representation of [7] HSC is depicted in Figure 3.17. As previously mentioned it works at run-time and, without stopping the program execution core. Before running the system (or during its configuration) the HSC 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 HSC reads every fetched instruction: this information is analyzed 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 verifies if there are matches of

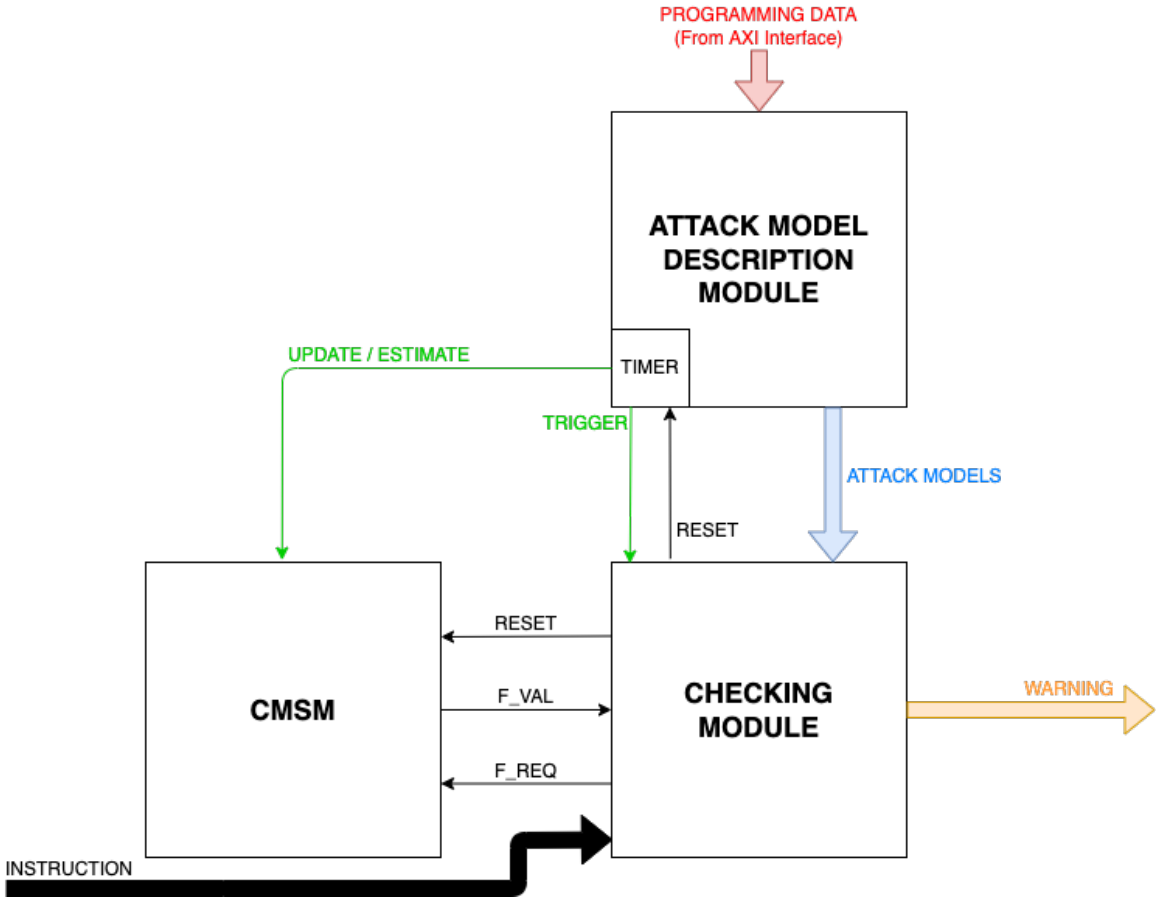


Figure 3.17: The internal structure of the [7] security checker

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 exceeding the programmed threshold, a warning is raised. At the end of these operations, irrespective of attack detection, the CMSM is reset and the next time-window starts. The workflow of [7] HSC is reported in Figure 3.18.

By summarizing, the possible conditions of the microprocessor's fetching activity that may be verified by [7] checker at the end of a time window are:

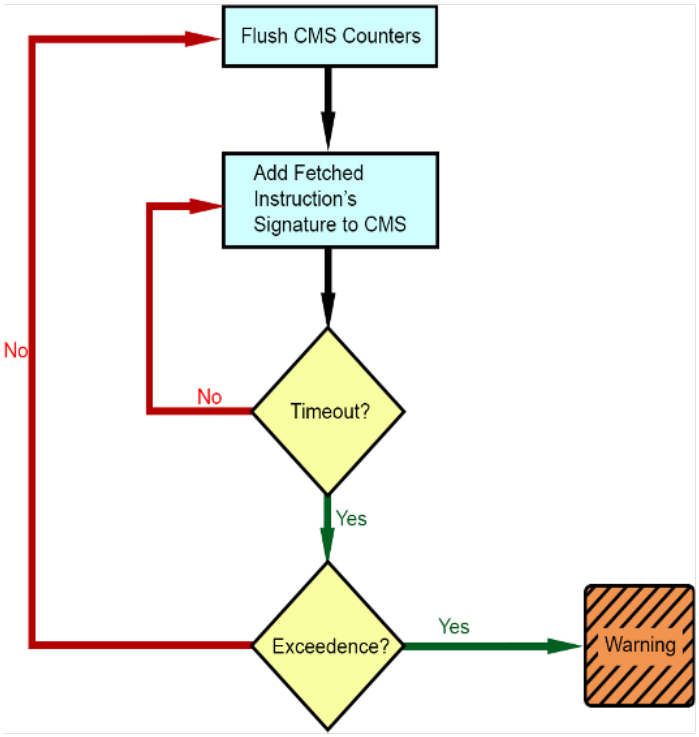


Figure 3.18: The Workflow of the Hardware Security Checker CMS-based [7]

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

3.1.2.1.2 The Attack Model Description Module architecture The *Attack Model Description Module (AMDM)* is in charge of storing the attack description models specified by the user. The AMDM is depicted in Figure 3.19: it is a table containing the description of the attack models the user wants to take into account plus a programmable timer. The content of the attack description table (which is actually a memory) and the duration of the timer's countdown are the components

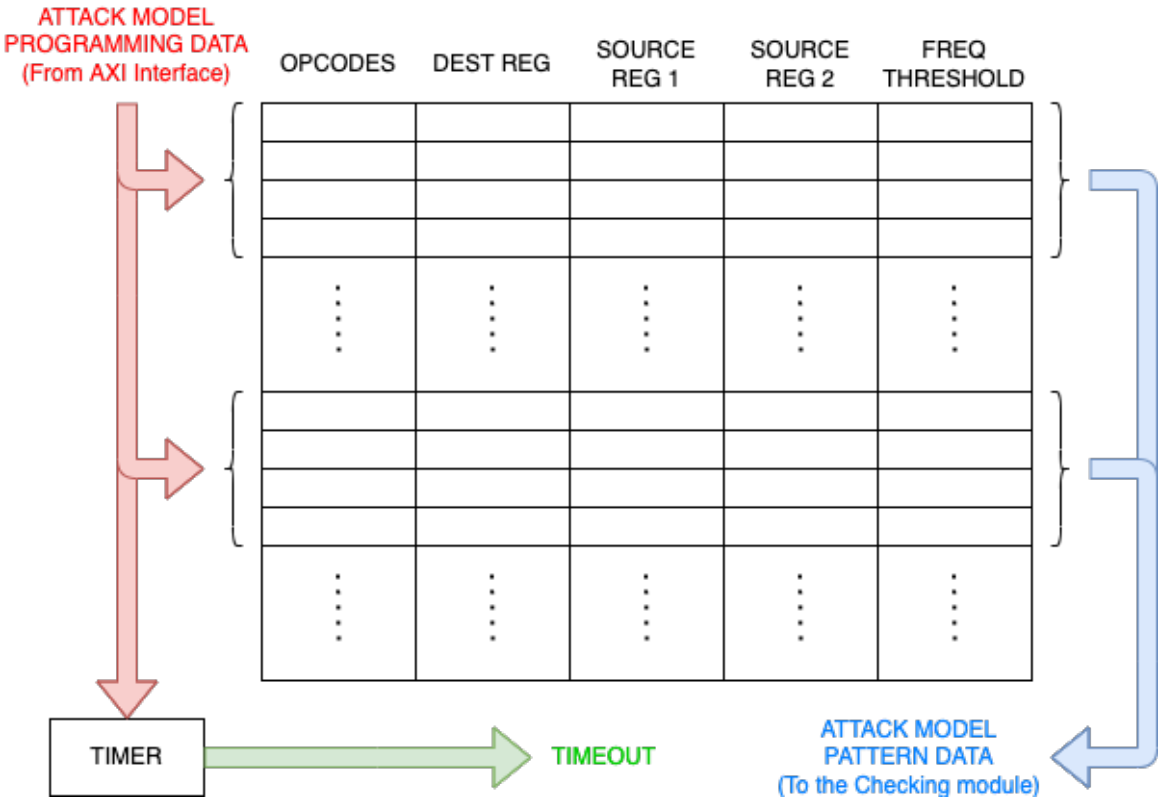


Figure 3.19: The architecture of the [7] Attack Model Description Module

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. The attack models evaluated in [7] are composed of a pattern `ID` and a set of *prototype instructions* characteristic of the attack itself. An instruction prototype is described by an opcode, a set of labels that represent the destination and source registers and a frequency threshold. In Table 3.4 are reported the complete [7] attack models description (the - means that no second source register is expected), where both the previously discussed Orchestration attack (first block of rows) and Spectre attack (second block of rows) are modeled. 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.

3.1.2.1.3 The Count Min Sketch Module architecture The *Count-Min Sketch Module (CMSM)* is in charge of monitoring the activity of the microprocessor. The architecture of the CMSM is depicted in Figure 3.20. 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.

Table 3.4: Attack models descriptions for the Orchestration, Spectre, Flush+Reload, Rowhammer Attacks in [7]

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	-	
3	ld	A	B	-	40
	mov	A	B		
	mov	B	A		
	sub	B	A	-	
	cflush	B			
4	ld	A	B	-	40
	mov	A	B		
	mov	C	D		
	cflush	A			
	cflush	A			

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

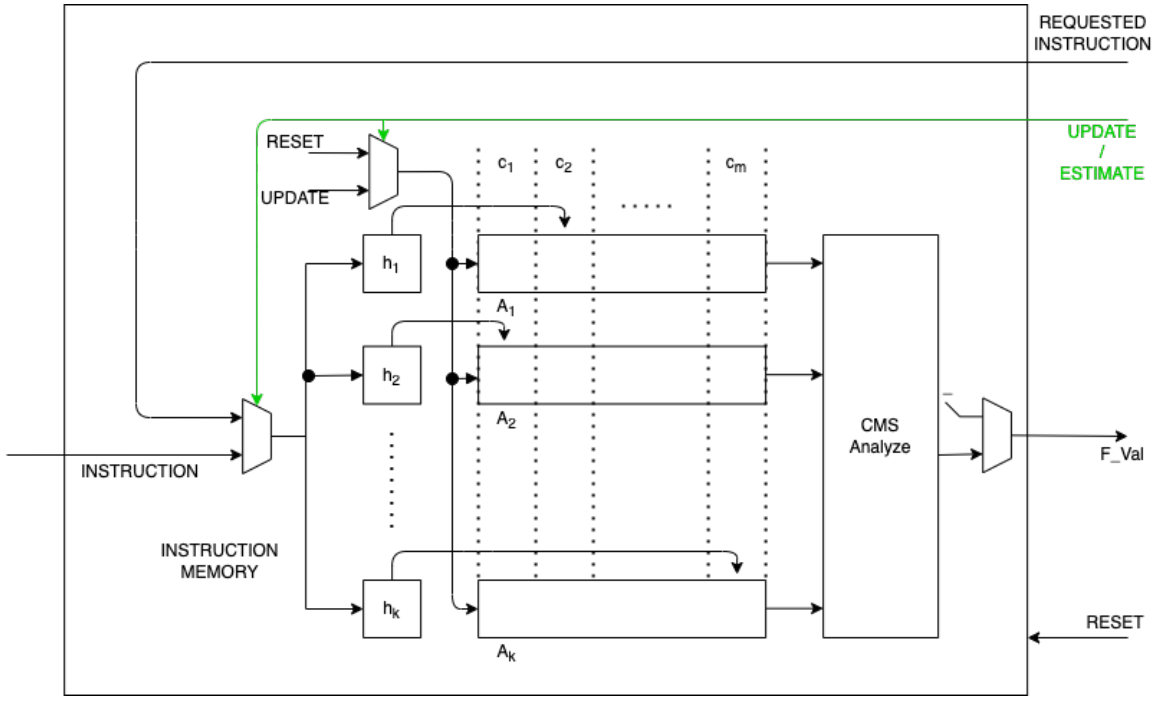


Figure 3.20: The architecture of [7] Count-Min Sketch module.

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.

3.1.2.1.4 The Checking Module architecture The *Checking Module (CM)* is in charge of detecting suspicious activity of the microprocessor and, if necessary,

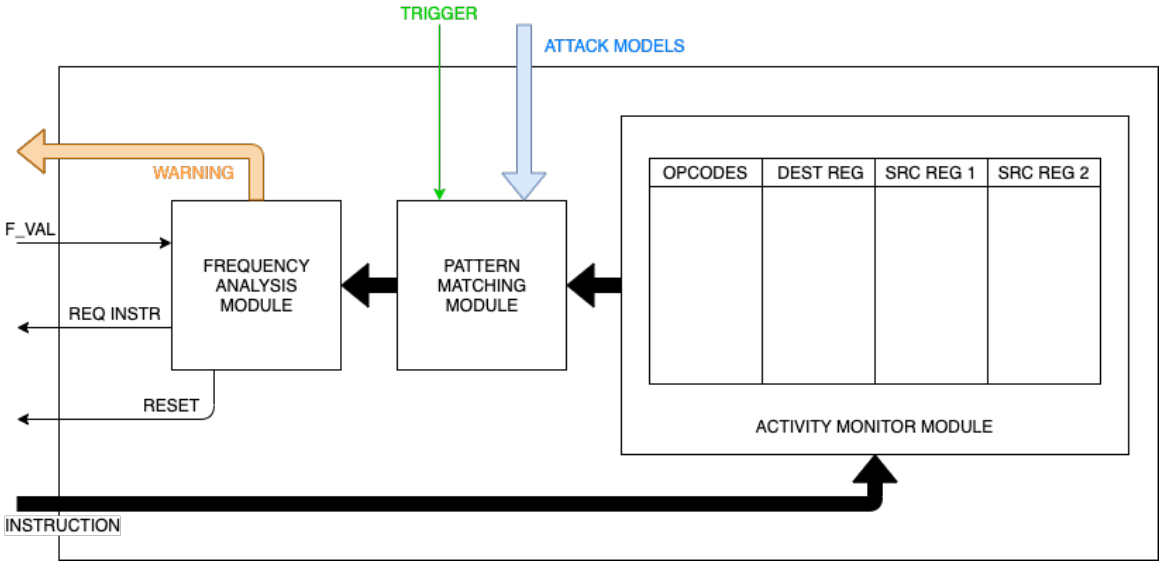


Figure 3.21: The architecture of [7] Checking module

raising warnings. The architecture of the CM is depicted in Figure 3.21. 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.

3.1.2.1.5 The security checker design flow The HSC proposed in [7] ensures a 100% detection probability of the programmed attacks (no false negatives). On the other hand, a theoretically possible vulnerability of their 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. Given a specific set of the checker in terms of number of hash functions number k and counters per hash function m, the worst case false positive rate can be calculated at design time. More in detail, referring to equation 3.1.3 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 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 (calculated as reported in 3.1.3) should be identified as the minimum number of times the attack code should be executed in order for the attack to be successful. Figure 3.14 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 $FP_p = e^{-k}$ in CMSs estimates [19]. It is possible to invert this formula and calculate the number of hash functions:

$$k = \lceil \ln \frac{1}{FP_p} \rceil \tag{3.1.5}$$

Therefore, it is possible to calculate the value of k as a function of the desired worst case false positive probability. Figure 3.15 correlates the values of k and those of 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.

3.1.2.1.6 Experimental Results For the experimental campaign the [7] HSC is implemented in the RSD core [61] which is a 32-bit, speculative out-of-order, superscalar, two-fetch front-end and five-issue back-end pipelines RISC-V core. The synthesis and the implementation of the considered microprocessor have been performed on Vivado targeting a Virtex7 `xc7z020clg484-1` FPGA. The obtained core counted

Table 3.6: The considered benchmarks programs in [7]

Benchmark	**#Instructions**	**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

18334 LUTs, 10885 FFs, 4512 LUTRAM cells and 17 BRAM cells and worked at 57MHz with an estimated power consumption of about 0.926W. 3.7 reports the results of these experiments. In order to assess both the effectiveness and the efficiency of the solution in [7] have been considered several implementations of the checker. In particular, they tried the number of hash functions k ranging from 1 up to 6 and for every value of k they 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. The overhead of each different implementation of the considered $<k, m>$ HSC's configuration tuples implementations of the HSC is reported in Table 3.7 (the first row indicates the core unprotected). Regarding attack detections, in [7] have been considered three versions of three attacks(Spectre, Orchestration, Rowhammer and Flush+Reload attacks) and four not malicious programs. Table **??** reports the detail about the instructions of the attacks.

For each of the HSC's configurations and for each of the attacks have been ran 100 random simulations of each of the four benchmark programs where an attack was activated. The not malicious benchmarks considered (and where are injected the attacks) are reported in Table 3.6. The [7] HSC detect always the attacks but "sometimes" it raises a false alarm (a false positive event). Much in detail the false positive rates for the considered checker's configurations when considering the Orchestration, Spectre, Rowhammer and Flush+Reload attacks are shown in Figures 3.22, 3.23, 3.24

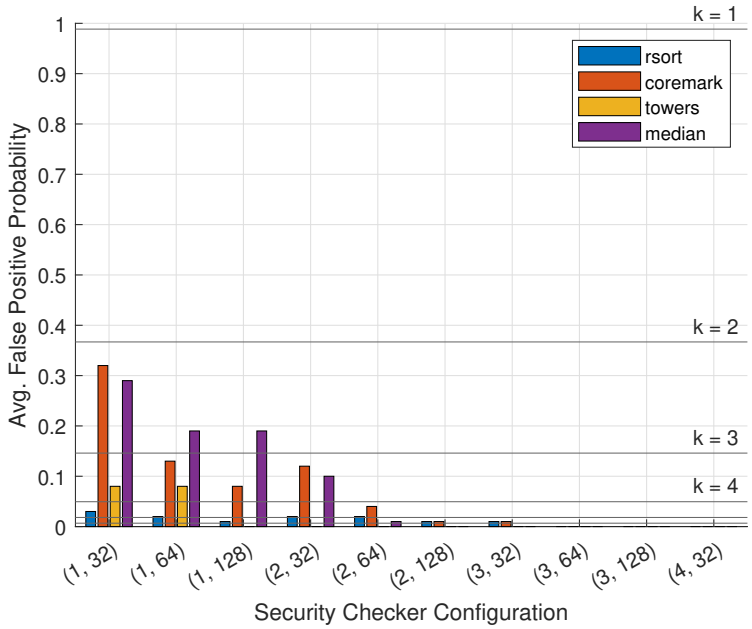


Figure 3.22: Average False Positive Probability (FP_p) when attacking several configurations of the SC with the Orchestration Attack in [7]

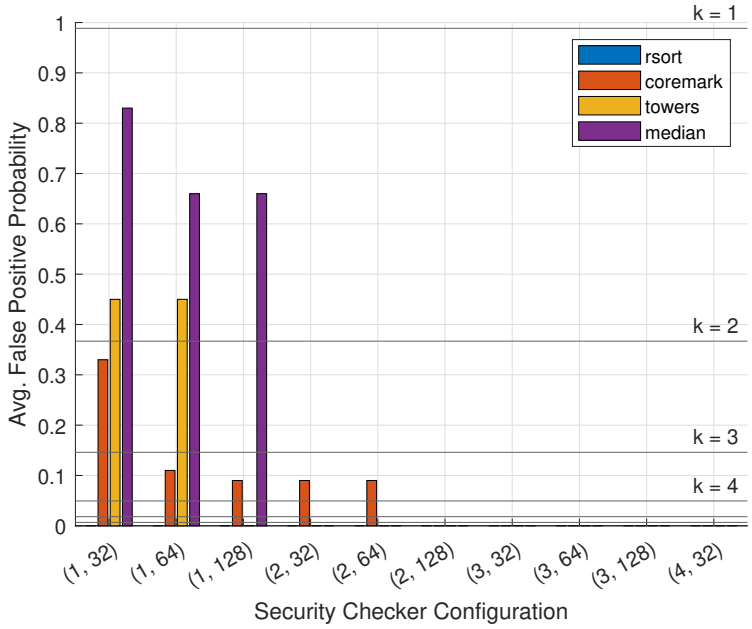


Figure 3.23: Average False Positive Probability (FP_p) when attacking several configurations of the SC with the Spectre Attack in [7]

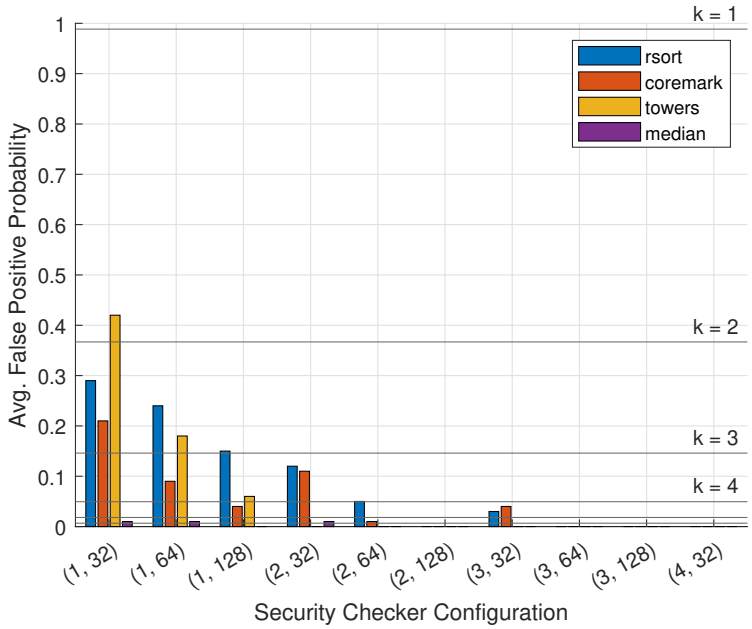


Figure 3.24: Average False Positive Probability (FP_p) when attacking several configurations of the SC with the Rowhammer Attack in [7]

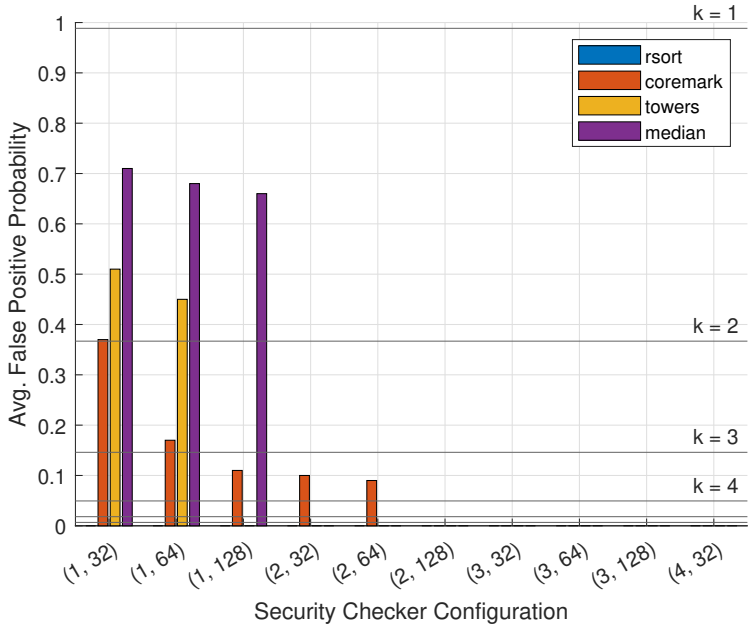


Figure 3.25: Average False Positive Probability (FP_p) when attacking several configurations of the SC with the Flush+Reload Attack in [7]

Table 3.7: Synthesis results: resource occupation, power consumption and working frequency

#Config.	#LUTs	#LUTRAMs	#FFs	#BRAMs	Power Consumption	Freq.
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

Table 3.8: Rates of detection and overhead comparison between HSCs for SCAs detection

	[7]	**[101]**	**[108]**	**[107]**	**[42]**
Detection	100%	99%	100%	100%	100%
Area	10%	NA	NA	25%	8%
Power	4%	NA	28%	41%	NA
Slowndown	0%	21%	1%	15%	4%

and 3.25, respectively <k, m> HSC's configuration tuples implementations (configuration from <4, 64> up to <6, 128> are not reported in the Figures but their false positive rates are 0%).

In Table 3.8 there is a comparison between HSCs presented in this chapter, able to detect SCAs. All of them are able to detect both HTHs and Software threats. Much in detail, Jintide is also able to detect Spectre [47] and Meltdown [52] attacks

Chapter 4

Methodologies against untrusted CAD Tools: Can Machine Learning Provide an Answer?

Software exploitable HTHs inserted into commercial CPUs may allow an attacker to run malicious software or to gain unauthorized privileges. Recently a novel menace was raised: HTHs inserted by CAD tools [76, 72]. In [32, 44] the don't cares of the design are exploited to insert HTHs both in the RTL code or gate-level netlist. In [71] a black-hat high-level synthesis tool has been presented. Its workflow starts from high-level specifications of the desidered function to be implemented, i.e., a C/C++/SystemC, and the tool produces an HTH-infested hardware implementation of the corresponding IP core. In the state of art there are many reports demonstrating the HTHs injection in the produced IP core. Such HTHs may have different malicious goals: downgrading system performance, changing the implemented functionality and draining the battery of the system. In addition, in [8] the authors demonstrate that all electronic CAD tools, i.e., high-level synthesis, logic synthesis, physical design, verification, test, and post-silicon validation, are potential threat vectors to different degrees. Similar considerations can also be made when looking at the FPGA scenario instead of the ASIC one. It has indeed been demonstrated that CAD tools may

represent a serious threat to the security and trust of FPGA-based systems [85, 104, 27]. In particular, in [28, 84], it has been demonstrated that malicious CAD tools may tamper the produced bitstream before FPGA configuration, introducing HTHs in the design. These reasons confirm, again, that HTHs have to be considered a serious threat and not only an academy case of study. Given this discussion, it is crucial to provide designers with effective tools to detect malicious modifications introduced in the system by the employed CAD tool before sending the design to the foundry (in the case of an ASIC design) or before integrating it into the final system (in the case of an FPGA-based design).

The new paradigm, dubbed *Security Rule Checking*, has been proposed to drive the implementation and the adoption of CAD tools in order to support the verification of the trustworthiness and security of systems throughout the whole design process [99]. In [21] the authors present a methodology where reverse engineering of the generated configuration bitstream is exploited to extract circuit designed layout implemented onto the FPGA device. Given this (possibly infected) layout, it is then compared with the "ideal" layout coming from the trusted design phases to find differences, and thus, to signal the presence of a malicious insertion. As in [21] the authors say, receiving the circuit layout via reverse engineering techniques is far from trivial. Much in detail, in FPGA scenarios, the same verification activities are much harder when dealing with the final configuration bitstream, since they would require strong reverse engineering capabilities, which are not always available in a design team. Indeed, as in [10] the author reported, via reverse engineering computation the structure of the logic netlist, i.e., retrieving the number of LUTs and FFs employed in the design, is "quite simple", but knowing the content of such LUTs and the structure of the routing, i.e., the actual functionality implemented in the device, is a much harder

task. In [66], the authors (including me) presented a machine learning methodology to identify these differences between the "golden" circuit reference and the possible infected one without any reverse engineering computation.

4.1 The Considered Threat Model

In the comparative analysis proposed in [66] we considered HTHs injected by the employed CAD tool when translating the final design files of microprocessor softcores meant to be implemented onto SRAM-based FPGA devices into the configuration bitstream. Therefore, the considered attacker is the CAD tool generating the bitstream, while all other tools are considered to be trusted or verifiable. This is a reasonable assumption because the generated bitstream is far from trivial to verify the final output among all the intermediate phases producing outputs of the FPGA-based design flow and thus the most suitable for the insertion of a HTH. Indeed, all the intermediate outputs of the FPGA-based design flow can be verified via formal equivalence checking and formal property verification, logic testing, design-rule checking and simulation.

Regarding the specific HTHs, in [66] have been considered four models inspired to the originally presented in the Trust-Hub repository [78]:

- `Clk_Mod`, that is an HTH able to slow down the microprocessor's clock thus impacting on programs' throughput. This consists of a denial-of-service HTH and it may be either triggered or always-on, depending on the configuration.

- `Critical`, that is an HTH that inserts additional combinational logic in a path of the circuit. If inserted on the critical path of the circuit, this HTH has an impact on its timing behavior, and thus, possibly, on performance. This HTH is denial-of-service and always-on.

- `MitM`, that is an HTH that emulates a malicious man in the middle between the microprocessor and the instruction memory. Indeed, this HTH interferes with the fetching activity of the microprocessor by modifying the fetched instructions, thus forcing the execution of a malicious program. This HTH bay be triggered for changes system functionality.

- `Fetch`, that is an HTH that interferes with the fetching activity of the microprocessor by altering the instruction memory address from which instructions are loaded, thus again, forcing the execution of a malicious program. Also this HTH is may be triggered and changes functionality.

In the final dataset, also the estimation of resources of different Vivado implementation strategies has been evaluated. In practice, four HTHs are evaluated and one golden circuit. Each of those have been implemented with four different Vivado strategies reported in Table 4.1. At the end of the day, the trojaned microprocessors were 16 in total, while the golden ones were 4. The software benchmarks considered was 5 and each run of those pruduced a dataset sample. The trojaned samples were 80 ($4_{TrojanedMicro} \times 5_{Benchs} \times 4_{Implementations} = 80$), while the golden ones were 20 ($4_{Implementations} \times 5_{Benchs}$). In total the whole dataset contained 100 samples. Finally a classification has been performed exploiting a machine learning algorithm, with the purpose of creating an overall evaluation methodology to test the trustworthiness of an SoC. This algorithm takes as input some golden micro-controller version features values and the values of the same features from different Trojaned versions.

Table 4.1: Vivado strategies and implementation settings

N° Strategy	**Synthesis**	**Implementation**	**Opt. Design**
1	Vivado Default	Vivado Default	Yes
2	Vivado Default	Vivado Default	No
3	Flow Area Opt. medium	Ultra Low Fast Design Method	Yes
4	Flow Area Alternate Routability	Ultra Low Fast Design Method	Yes

Table 4.2: The considered benchmarks

Benchmark	**Description**
`Coremark`	CPU performance benchmark
`Median`	Median image filter
`Multiply`	Numbers multiplication
`Rsort`	Sorting algorithm
`Towers`	Solver for the tower puzzle

4.2 The Considered Machine Learning models

In literature are reported many ML classification algorithms: supervised, unsupervised, both linear and non-linear. Each of those may be directly applied to the HTH detection problem. Following the no-free lunch (NFL) theorem, there is no single algorithm suitable for all problems, so a wide number of them must be tested and adjusted using cross-validation techniques with grid-search for fine hyper-parameter tuning, as explained in [51, 43]. For [66] experimental comparison have been considered the following machine learning models. Much detail about they can be found in [50]:

- **Logistic Regression**, which is a fast, simple, linear regression model. Although these advantages, logistic regression often shows underfitting since it cannot capture non-linear data patterns. Still, logistic regression provides a performance baseline for comparison and a hint about which features in the considered dataset are most relevant in the classification task.

- **Decision Tree**, that classifies instances of a problem by sorting them based on the values of a set of features. Nodes in a DT represent features, while branches represent values that the associated features may assume. Leaves represent the result of the classification. Instances of the problem are classified starting from the root and traversing it based on the actual values of the features. Examples of tree algorithms include ID3, C4.5 or C5.0 algorithms.

- **Random Forest**, which combines a number of decision trees to build an ensemble classifier that outperforms individual decision trees by always selecting the class identified by the majority of the trees in the considered forest.

- **Support Vector Machines**, which represent the instances of a dataset as points of an n-dimensional space. An SVM is an ML technique based on the identification of one or more hyperplanes (based on the number of classes of the problem) used to isolate the classes the elements of the available training set belongs to. In particular, an SVM identifies the hyperplanes having the largest margin, i.e., the distance between the hyperplane and the closest instance of every class, to minimize the classification error.

- **Gradient Boosting Machine** exploits an ensemble of prediction models, typically a decision tree, producing a majority-based classification. GBM is especially suitable for classification problems suffering class imbalance, as it is the case of the HTH dataset of this work.

4.3 Hardware Trojan Horse Detection Flow

The verification flow is meant to be placed at the microprocessor design-time flow. Much in detail, it needs to be executed before deploying the microprocessor softcore

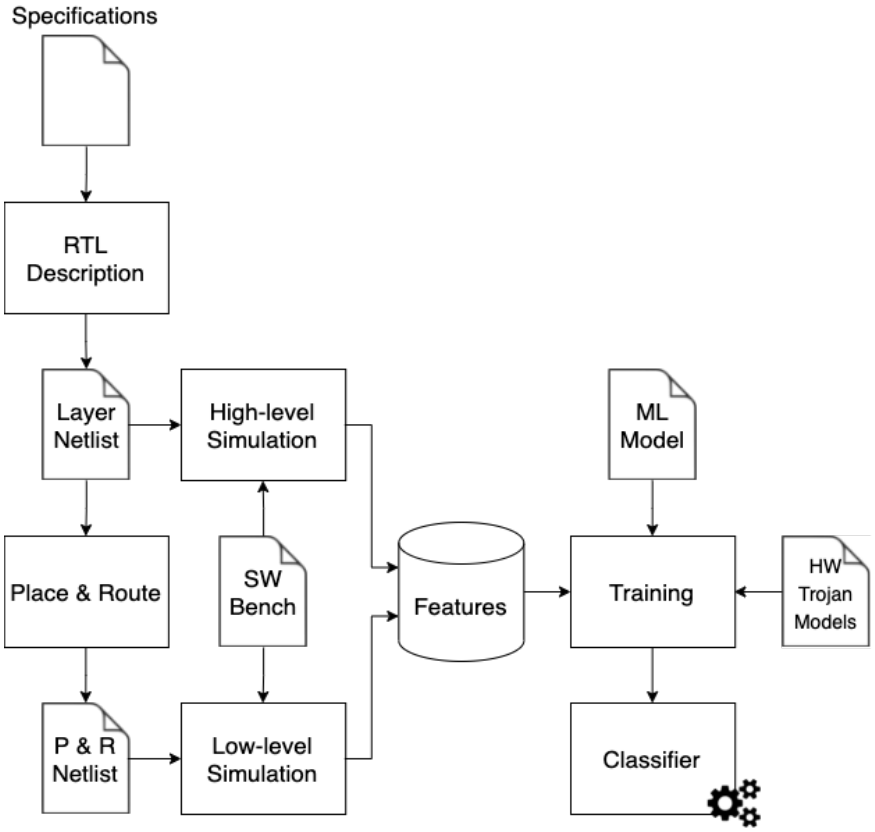


Figure 4.1: The flow for the definition and training of the classifier in [66]

under analysis in the final system. The idea is that, since, as discussed before, the bitstream generation tool is considered untrsuted, the generated bitstream needs to be verified against the output of the previous steps of the design flow (which are considered either trusted or verifiable). To accomplish that, need to be exploited trusted simulations at the logic netlist-level and at the post-place&route-level in order to extract a number of features, e.g., the number of microprocessors executed Load instructions or its average power consumption. These features are in turn used to train a ML-based classifier. It finally verified the behavior of the real system when the (possibly infected) bitstream downloaded onto the considered FPGA device. Indeed, in this final verification phase, the ML-based classifier is fed with features coming from the execution of the real system configured onto the target FPGA. The flow would

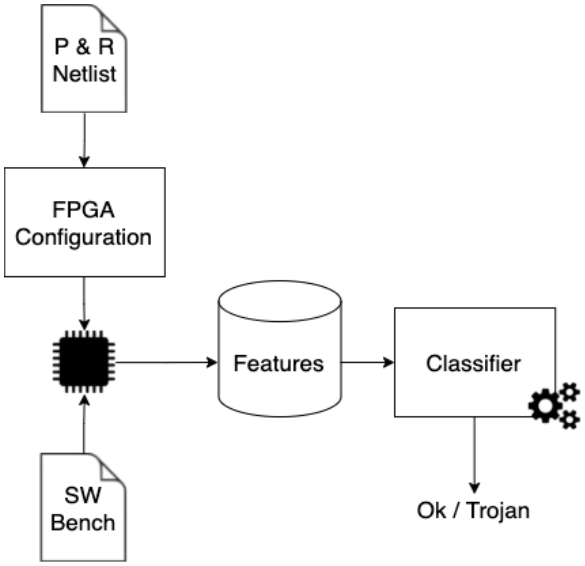


Figure 4.2: The flow for the bitstream verification in [66]

be divided into two parts to be executed one after the other: the *classifier definition flow* and the *bitstream verification flow*. In the first one the designer extracts circuit-level and runtime features for training the ML-based classifier. In the second part of the flow take place the verification of whether the produced bitstream (and thus the obtained microprocessor) is HTH-free (and it can be deployed in the final system), or not.

4.3.1 The classifier definition flow

Figure 4.1 reported the steps required to define and train the ML-based classifier. On the left-side of the flow, there is the standard FPGA-based design flow. The two intermediate artifacts of the logic synthesis and place&route phases, namely the logic netlist and the post place&route netlist, would be exploited in two simulation processes: a *high-level simulation* and a *low-level simulation* where the microprocessor logic and post place&route netlists are together simulated with the considered software benchmark(s). The high-level simulation allows to extract runtime features

like statistics on the executed instructions and the number of clock cycles required to complete a given instruction as well as details related to the logic netlist, such as the number of LUTs and FFs. On the other hand, the low-level simulation allows to extract fine-grained circuit-level features like details about the power consumption and the timing of the circuit. All the features considered in the proposed comparison are the ones reported in Table 4.3. On the top part of the table are reported the features related to the high-level simulation; the ones reported in the bottom of that table correspond to low-level estimations. The samples of all of those have been evaluated by running five different programs (much detail in Table 4.2) in each microprocessor circuit considered. Such benchmarks have been considered also as a feature. At the end of the day, the user knows which program its microprocessor is running if its circuit is not under attack. In the following paragraphs are reported some detail regarding features estimations.

4.3.1.1 Feature Power Consumption

Vivado IDE provides a report power analysis with high level of confidence that is based on post-implementation simulations. In particular the power consumption evaluation of the circuit designed relies on the fidelity of system behavior with respect to post-implementation test-bench, that is considered to be the most reliable since it focuses only on real signals that are synthesize and implemented by the process. Thus using Vivado settings is possible to generate SAIF (*Switching Activity Interchange Format*) log files that collect, through test-bench simulations, wave dumps (Figure 4.3). In addition, is possible to specify which signals have to be included in those log files, how long time have to be run the simulation (and the data estimation collection) and other detail shown in Figure 4.4. Power consumption traces are suitable for HTHs

Table 4.3: The set of considered features

Feature	**Description**
`Benchmark`	Program under execution
`Cycles`	# clock cycles to execute the program
`InstrRet`	# instructions retired in the program
`LSUs`	# waiting cycles to access data memory
`FetchWait`	# waiting cycles before instruction fetch
`Load`	# load instructions
`Store`	# store instructions
`Jump`	# jump instructions
`CondBr`	# conditional branches
`TakCBran`	# taken conditional branches
`CompIns`	# compressed
`MulWait`	# cycles for mul. completion
`DivdWait`	# cycles for div. completion
`LUTs`	# Look Up Tables
`FFs`	# Flip Flops
`AvgDynPow`	Avg. dyn. power consumption
`AvgPower`	Avg. total power consumption
`Timing`	Worst negative slack
`Temperature`	Temperature trend

detection. Indeed, in Figure 4.5 there is a comparison of power estimation between the golden circuit and the Clk_Mod trojaned one: their values are quite different. In this specific case of an HTH able to modify the circuit clock frequency, the impact in the power consumption is viewable also according to the following formula:

$$P_d = \alpha f_{clk} C V_{dd}^2 \tag{4.3.1}$$

where:

- α = switching activity.
- f_{clk} = clock frequency.
- **C** = transistors capacity.
- V_{dd} = power supply.

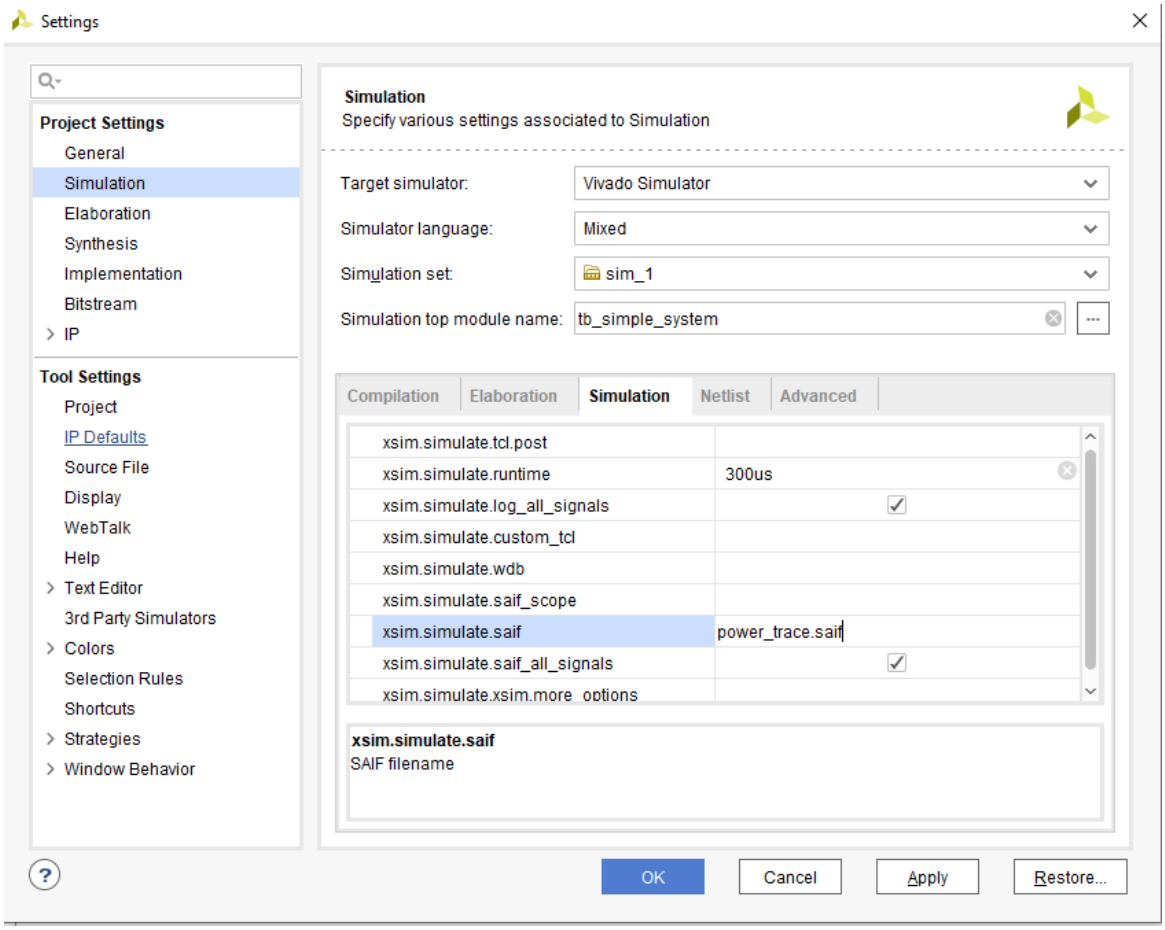


Figure 4.3: Settings example for SAIF file generation

4.3.1.2 LUT, FF and Timing

The number of LUTs and FFs are given by Vivado after system synthesis. Timing specification is built through a clock constraint that specifies system operational clock frequency. Vivado returns information about timing margin calculated with respect to this constraint.

4.3.1.3 Temperature

This feature has been obtained via the **Vivado XADC monitor**. Zynq-7000 device, where al the estimations have been conducted, provides a temperature sensor directly

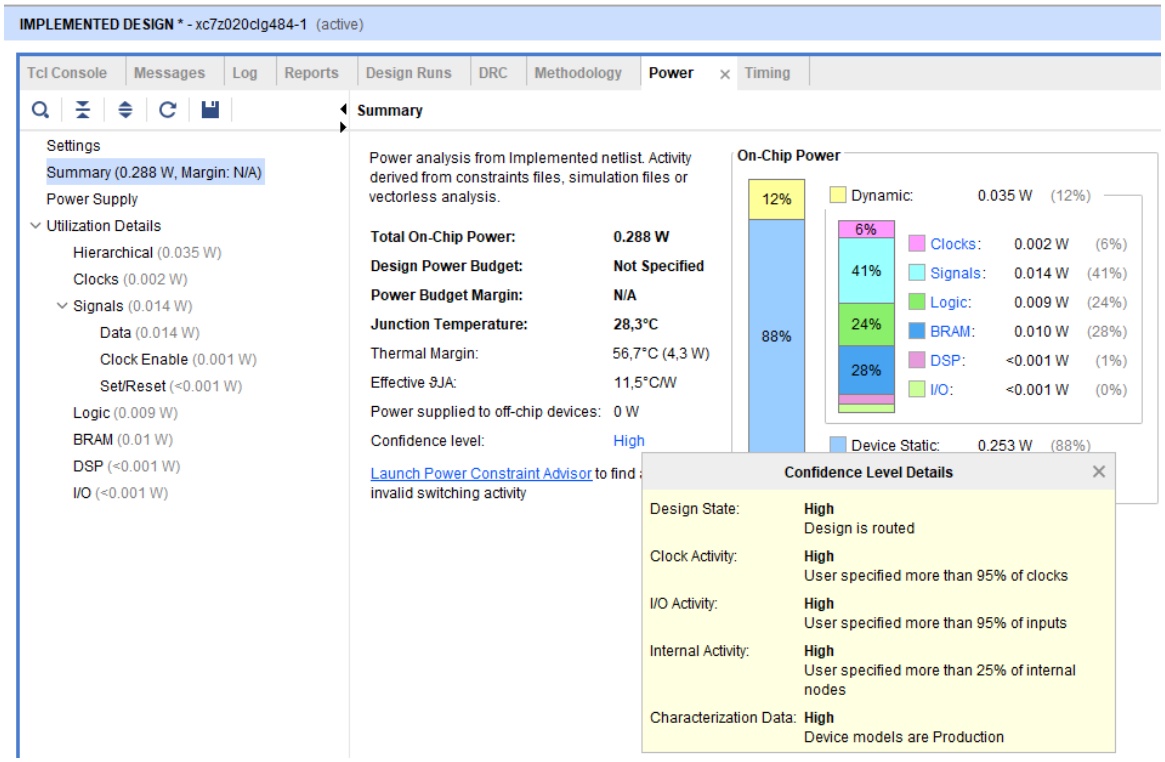


Figure 4.4: Example of power estimation with SAIF file

linked to XADC signals that monitors FPGA runtime variations. The on-chip temperature sensor has a maximum measurement error of ±4°C over a range of –40°C to +125°C. This sensor is located in the center of the die to provide a good level of confidence regardless Vivado implementation positional choices. Essentially these estimations are collected following this procedure:

1. Turn-on the board.
2. After 20 minutes load the bit-stream of desired implementation and collect temperature data samples.
3. Leave reset low (reset active low) for 1 minutes then deactivate it and start program execution for other 4 minutes.
4. Turn-off the board for 30 minutes before restart the process.

It has been noted that after about 20 minutes the temperature increase associated to power supply becomes negligible and measurements result reliable. The visual representation of runtime measurement is shown in Figure 4.6.

4.3.1.4 Performance Counters and high-level features

As previously described, high-level features (reported in high part of Table 4.2) are used to obtain a fingerprint of microprocessor execution runs. In each program run, the core, stores information about the operations performed in its *CSR* (*Control Status Register*).

4.4 Experimental Setup & Evaluation

The processor selected for the experimental campaign is the IBEX core [1] which is a 32-bit, low-power, in-order, two stage pipelined RISC-V core often employed in IoT systems. The synthesis and the implementation of the considered microprocessor have been performed on a Xilinx XC7Z020 1CLG484C Zynq-7000 FPGA. Accuracy, Sensitivity (also called Recall or True Positive Rate), are calculated according the following equations:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{4.4.1}$$

$$Sensitivity = \frac{TP}{TP + FN} \tag{4.4.2}$$

$$Specificity = \frac{TN}{FP + TN} \tag{4.4.3}$$

$$Precision = \frac{TP}{TP + FP} \tag{4.4.4}$$

where TP, TN, FP, FN represent the True Positives, True Negatives, False Positives and False Negatives, respectively, being HTH-infested circuits the positive class. Furthermore, we considered the Cohen's Kappa coefficient (κ), that can be calculated as

follows:

$$\kappa = \frac{O - E}{1 - E} \tag{4.4.5}$$

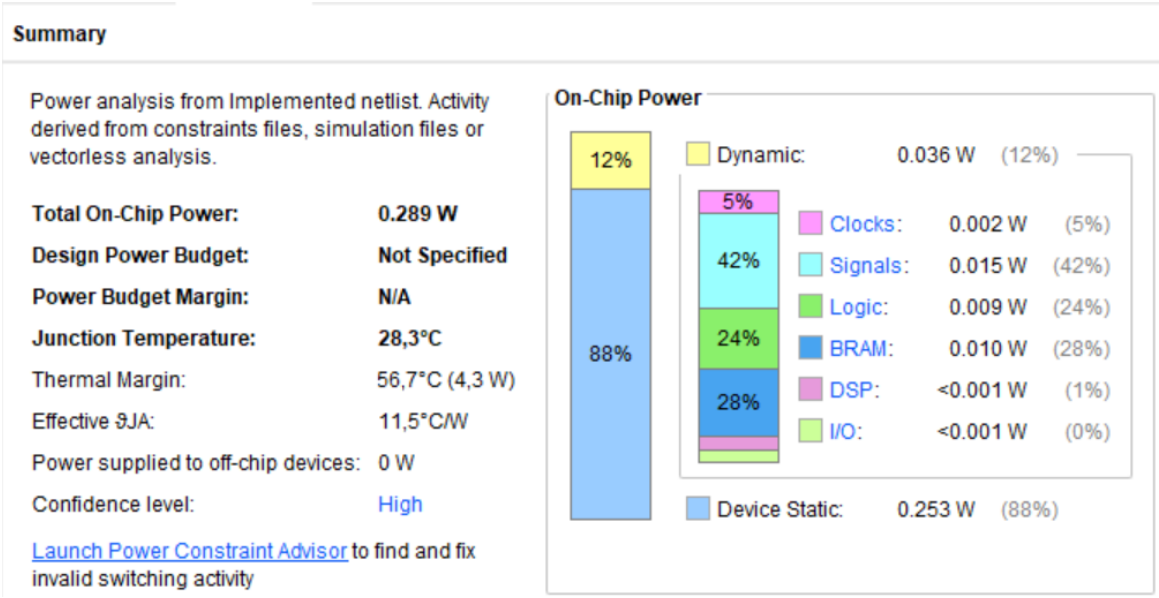
where O represents the Observed accuracy and E the Expected accuracy, i.e. the accuracy provided by a dummy classifier that always selects the majority class. The difference between the Observed and Expected Accuracy is reflected by the Kappa coefficient showing the benefits of the ML model with respect to a random classifier. Typically, κ values above 0.6 reflect substantial agreement between observed and expected accuracy, while values above 0.8 represent nearly perfect agreement [18]. The whole dataset, contein 100 samples (80 trojaned and 20 golden). The first experiment campaign has been conducted maintaining the HTH not triggered (the ones that are not always on) in the circuit; the second one has been conducted in the scenario where the HTHs are triggered and infesting the microprocessor. For both campaigns have been done estimation splitting the training and test sets in different ways (80%-20% and 65%-35%). All the detail about Accuracy, Sensitivity and κ for the different ML algorithms considered in both the scenarios with HTHs triggered and untriggered are reported in Tables 4.4, 4.5, 4.6, 4.7. The "ranking" of the most important features to consider for maximize the accuracy in GBM and C5.0 ML algorithms is in Figure 4.7. The correlation between the features considered is in Figure 4.8.

Table 4.4: ML performance results for splits: 65/35% (top) and 80/20% (bottom) when HTHs are not triggered

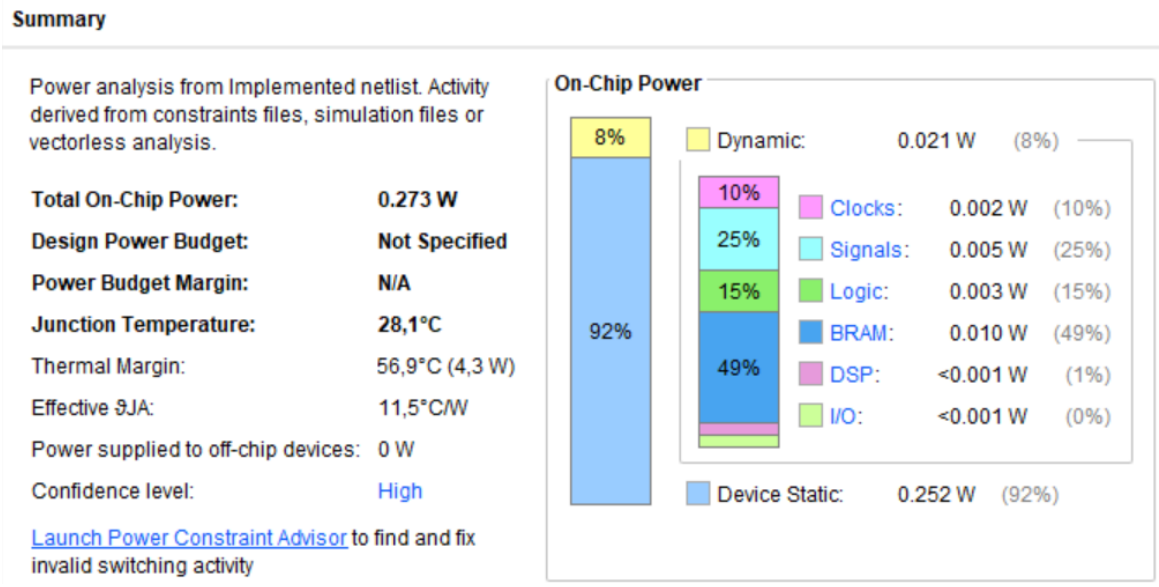
ML Model (65/35%)	**Accuracy**	κ
Dummy Classifier	0.5700	0
LR with Regularization	0.9368	0.8671
SVM (Radial Basis Function)	0.9451	0.8850
DT (algorithm C5.0)	0.9864	0.9718
Random Forest	0.9872	0.9729
Gradient Boosting Machine	0.9887	0.9763
ML Model (80/20%)	**Accuracy**	κ
Dummy Classifier	0.5700	0
LR with Regularization	0.9578	0.9119
SVM (Radial Basis Function)	0.9578	0.9122
DT (algorithm C5.0)	0.9894	0.9777
Random Forest	0.9921	0.9834
Gradient Boosting Machine	0.9960	0.9917

Table 4.5: ML performance results for splits: 65/35% (top) and 80/20% (bottom) when HTHs are triggered

ML Model (65/35%)	**Accuracy**	κ
Dummy Classifier	0.5700	0
LR with Regularization	0.9503	0.8971
SVM (Radial Basis Function)	0.9473	0.8898
DT (algorithm C5.0)	0.9834	0.9661
Random Forest	0.9894	0.9779
Gradient Boosting Machine	0.9894	0.9784
ML Model (80/20%)	**Accuracy**	κ
Dummy Classifier	0.5700	0
LR with Regularization	0.9578	0.9124
SVM (Radial Basis Function)	0.9526	0.9016
DT (algorithm C5.0)	0.9907	0.9810
Random Forest	0.9947	0.9890
Gradient Boosting Machine	0.9934	0.9864



(a) Power estimation of Golden reference in [66]



(b) Power estimation of Clk_Mod trojaned reference in [66]

Figure 4.5: Power estimation comparison between of two circuit reference in [66]

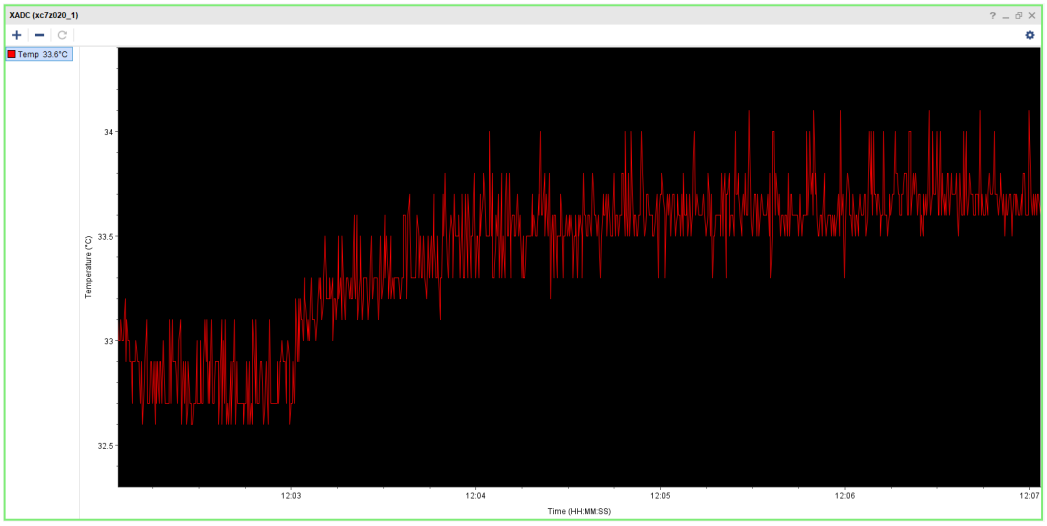


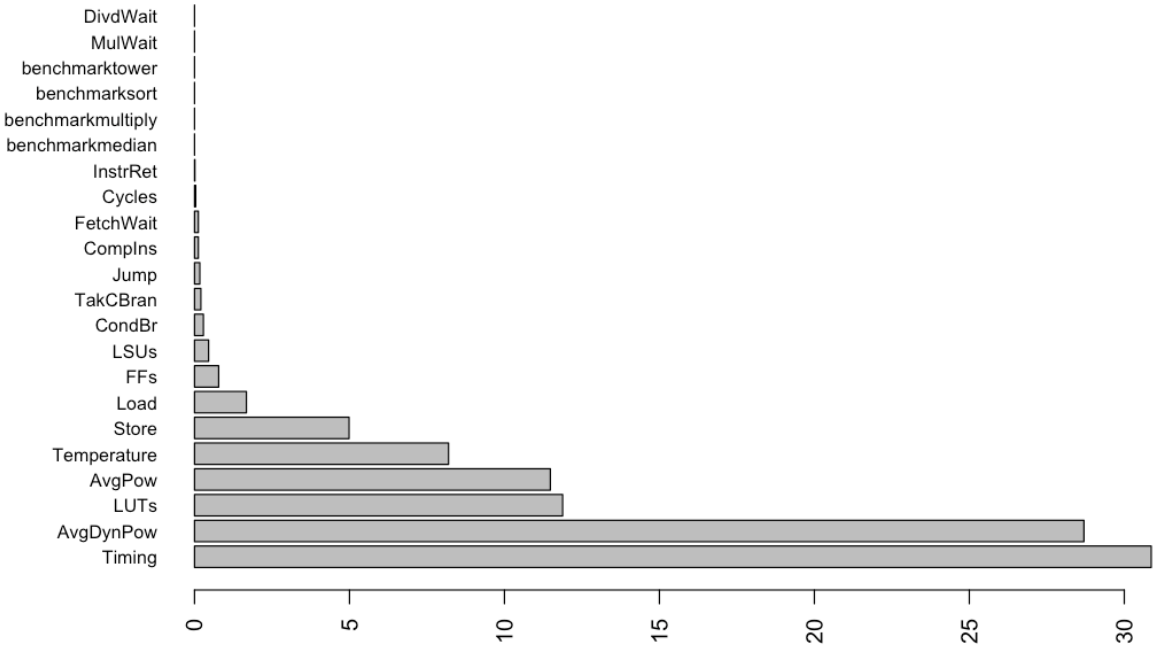
Figure 4.6: Esample of XADC monitor visualization for temperature estimations in [66]

Table 4.6: Performance metrics of the classifiers on the 80/20 train/test split when the HTHs are not triggered.

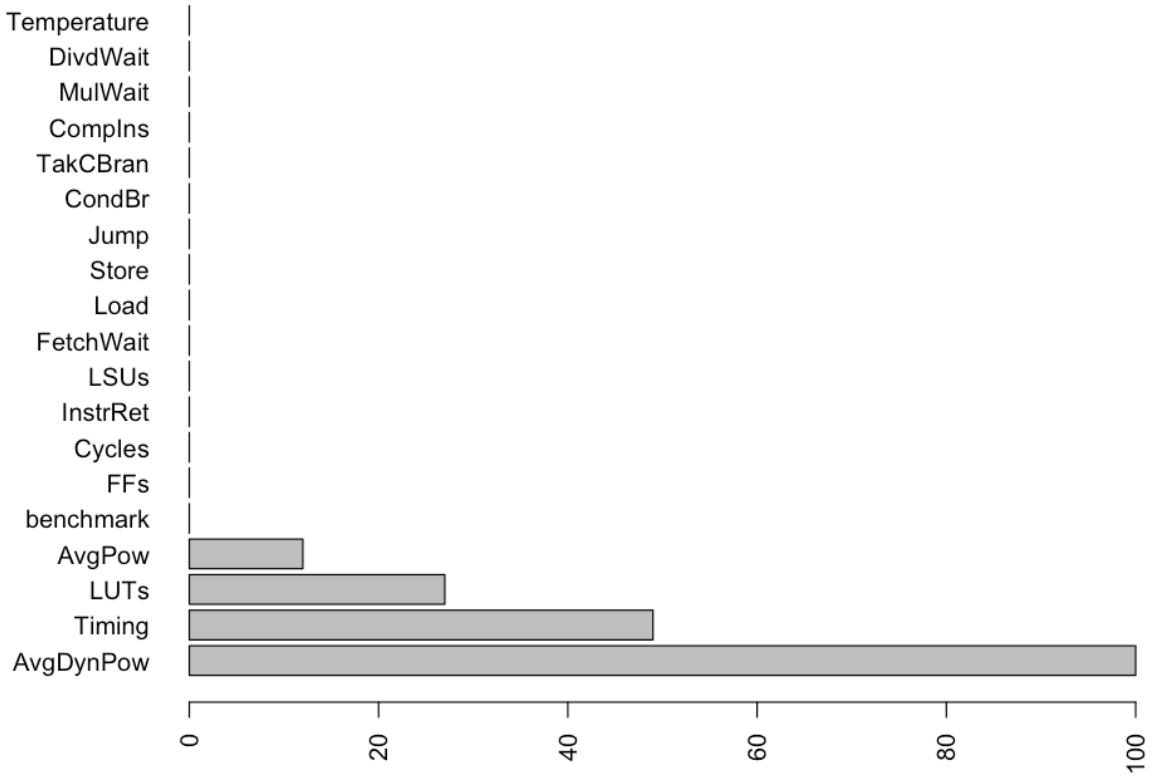
Metric	**LR**	**SVM**	**C5.0**	**RF**	**GBM**
κ	0.9119	0.9122	0.9777	0.9834	0.9917
Sensitivity	0.9320	0.9406	0.9906	0.9898	0.9929
Specificity	0.9723	0.9651	0.9947	0.9963	0.9984
AUC-ROC	0.9904	0.9936	0.9975	0.9999	1.0000

Table 4.7: Performance metrics of the classifiers on the 80/20 train/test split when the HTHs are triggered

Metric	**LR**	**SVM**	**C5.0**	**RF**	**GBM**
κ	0.9124	0.9016	0.9810	0.9890	0.9864
Sensitivity	0.9671	0.9523	0.9921	0.9945	0.9984
Specificity	0.9760	0.9687	0.9807	0.9932	0.9869
AUC-ROC	0.9919	0.9921	0.9956	0.9998	0.9986



(a) GBM



(b) C5.0

Figure 4.7: Variable importance for HTHs detection in [66]

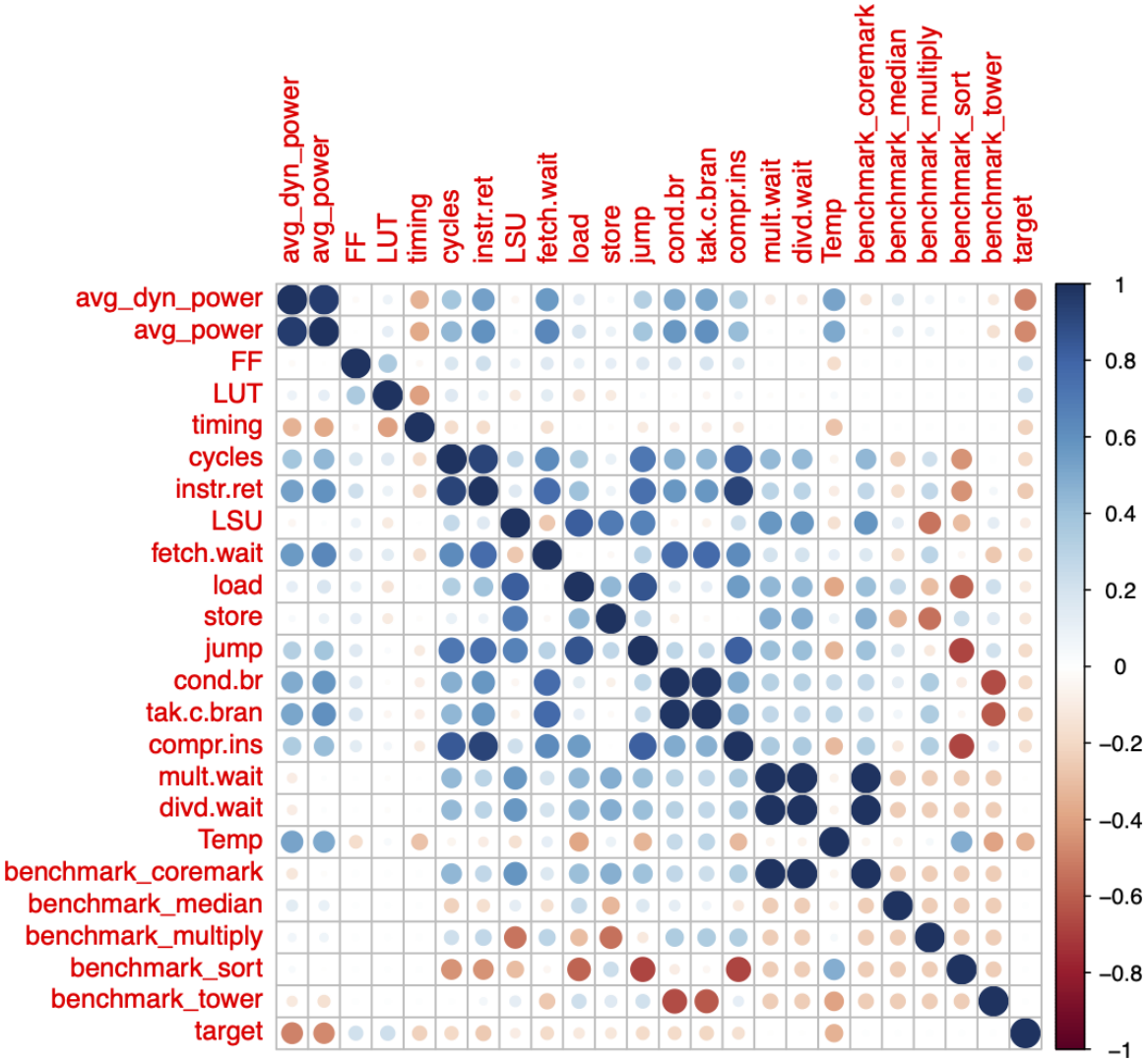


Figure 4.8: Correlation between features

Chapter 5

Conclusion, Open Issues & Future Work

The growing interest in deploying microprocessors as well as the rise of new hardware-oriented menaces, e.g., MSCAs and HTHs, make security (and hardware security, in particular) an open and severe issue. As have discussed, the RISC-V microprocessor, is receiving a growing interest also in the space application field, already implements several security features. Indeed, Physical Memory Protection (PMP), Cryptographic extension and User-level interrupt extension already protect a RISC-V based system against several attacks, e.g., data corruption and unauthorized access. On the other hand, the protection of RISC-V processors and other ISA architecture against MSCAs and HTHs is still an issue. For this reason, a number of hardware-based security checkers suitable for microprocessors have been explored.

Much in detail, in this thesis, have been presented security checking modules to protect microprocessor-based systems against those hardware Trojan horses that try to force the system to run a malicious program. In particular, such checkers, at configuration time store information about legit data while at runtime they check if there are malicious activities. HSC based on Probabilistic Data Structures and Machine Learning have been exhaustively discussed. In addition, it has been presented

a comparative analysis of the effectiveness of several ML models in detecting the presence of HTHs eventually injected by CAD tools in the bitstreams and a detailed feature importance ranking has been discussed. The considered multi-parametric approach led to promising results and confirms literature trend to prefer utilization of several observable features. The exploitation of ML models confirms the generalization potential of the proposed approach.

Apart from the protection against MSCAs and HTHs, it results still open two main open issues related to the trusted adoption of microprocessors. The first issue is the lack of a trusted and robust tool chain: indeed it has been demonstrated that malicious modifications can be introduced in the designed system by the employed CAD tools. Therefore, the availability of trusted tools and of methodology to ensure the trustworthiness of the employed tools it is vital. The second main issue is related to the integration of third party intellectual property cores (3PIPs) within a microprocessor based system. A trusted interaction between dedicated hardware accelerators and the main processor is fundamental to ensuring the security of the entire system. Therefore, methodologies to verify trusted behaviors and hardware-level mechanisms, like hardware-based firewalls, to isolate untrsuted components and limit their dangerousness should be investigated.

As future work it will be interesting to evaluate circuitry integrated in microprocessors for boycotting data errors and attacks. It may be feasible to adopt Error Correcting Code algorithms both for reliability and security scopes. Another paradigma to investigate is obfuscation: it may result interesting evaluate hardware solutions able to produce the same output of the microprocessor without protection, with the same input, but computing different operations in order to boycott a possible attacker looking for Side Channel features to gain information.

List of Figures

Bibliography

[1] *Ibex RISC-V Core*.

[2] project:rosenbridge, last access Feb. 2022.

[3] Onur Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18, 2007.

[4] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 312–320, 2007.

[5] Onur Acıiçmez, Werner Schindler, and Çetin K. Koç. Cache based remote timing attack on the aes. In Masayuki Abe, editor, *Topics in Cryptology – CT-RSA 2007*, pages 271–286, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[6] Sally Adee. The hunt for the kill switch. *iEEE SpEctrum*, 45(5):34–39, 2008.

[7] Kerem Arıkan, Alessandro Palumbo, Luca Cassano, Pedro Reviriego, Salvatore Pontarelli, Giuseppe Bianchi, Oğuz Ergin, and Marco Ottavi. Processor security: Detecting microarchitectural attacks via count-min sketches. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(7):938–951, 2022.

[8] Kanad Basu, Samah Mohamed Saeed, Christian Pilato, Mohammed Ashraf, Mohammed Thari Nabeel, Krishnendu Chakrabarty, and Ramesh Karri. Cadbase: An attack vector into the electronics supply chain. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24(4):1–30, 2019.

[9] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. Stealthy dopant-level hardware trojans. In *Cryptographic Hardware and Embedded Systems*, 2013.

[10] Florian Benz, André Seffrin, and Sorin A. Huss. Bil: A tool-chain for bitstream reverse-engineering. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 735–738, 2012.

[11] Swarup Bhunia, Michael S Hsiao, Mainak Banga, and Seetharam Narasimhan. Hardware trojan attacks: Threat analysis and countermeasures. *Proceedings of the IEEE*, 102(8):1229–1247, 2014.

[12] Swarup Bhunia and Mark Tehranipoor. *Hardware security: a hands-on learning approach*. Morgan Kaufmann, 2018.

[13] Alperen Bolat, Luca Cassano, Pedro Reviriego, Oguz Ergin, and Marco Ottavi. A microprocessor protection architecture against hardware trojans in memories. In *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. IEEE, 2020.

[14] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[15] A. Carelli, C. A. Cristofanini, A. Vallero, C. Basile, P. Prinetto, and S. Di Carlo. Securing bitstream integrity, confidentiality and authenticity in reconfig-

urable mobile heterogeneous systems. In *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pages 1–6, 2018.

[16] Luca Cassano, Stefano Di Mascio, Palumbo Alessandro, Menicucci Alessandra, Furano Gianluca, Bianchi Giuseppe, and Ottavi Marco. A lightweight security checking module to protect microprocessors against hardware trojan horses. In *2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2022.

[17] CCSDS. Security threats against space missions. *Informational Report (CCSDS 350.1-G-2)*, 2015.

[18] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[19] Graham Cormode and Senthilmurugan Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55:58–75, 04 2005.

[20] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, 2016.

[21] Wafi Danesh, Joshua Banago, and Mostafizur Rahman. Turning the table: Using bitstream reverse engineering to detect fpga trojans. *Journal of Hardware and Systems Security*, 5(3):237–246, 2021.

[22] Stefano Di Mascio, Alessandra Menicucci, Gianluca Furano, Claudio Monteleone, and Marco Ottavi. The case for risc-v in space. In *International Con-*

ference on Applications in Electronics Pervading Industry, Environment and Society, pages 319–325. Springer, 2018.

[23] Stefano Di Mascio, Alessandra Menicucci, Eberhard Gill, Gianluca Furano, and Claudio Monteleone. Leveraging the openness and modularity of risc-v in space. *Journal of Aerospace Information Systems*, 16(11):454–472, 2019.

[24] DIGITIMES. Trends in the global IC design service market. http://www.digitimes.com/news/a20120313RS400.html?chid=2.

[25] Christopher Domas. Hardware backdoors in x86 cpus. *Black Hat*, pages 1–14, 2018.

[26] Adam P. Donlin, Prasanna Sundararajan, and Bernard J New. Method and system for secure exchange of ip cores, Aug. 2010. US Patent 7,788,502.

[27] Adam Duncan, Fahim Rahman, Andrew Lukefahr, Farimah Farahmandi, and Mark Tehranipoor. Fpga bitstream security: A day in the life. In *2019 IEEE International Test Conference (ITC)*, pages 1–10, 2019.

[28] Maik Ender, Pawel Swierczynski, Sebastian Wallat, Matthias Wilhelm, Paul Martin Knopp, and Christof Paar. Insights into the mind of a trojan designer: the challenge to integrate a trojan into the bitstream. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pages 112–119, 2019.

[29] Stefano Esposito, Cristian Albanese, Monica Alderighi, Fabio Casini, Luca Giganti, Maria Livia Esposti, Claudio Monteleone, and Massimo Violante. Cots-based high-performance computing for space applications. *IEEE Transactions on Nuclear Science*, 62(6):2687–2694, 2015.

[30] Mohammad Rahmani Fadiheh, Dominik Stoffel, Clark Barrett, Subasish Mitra, and Wolfgang Kunz. Processor hardware security vulnerabilities and their detection by unique program execution checking. 1812.04975.pdf, December 2018.

[31] Gregory Falco. *When Satellites Attack: Satellite-to-Satellite Cyber Attack, Defense and Resilience.*

[32] Nicole Fern, Shrikant Kulkarni, and Kwang-Ting Tim Cheng. Hardware trojans hidden in rtl don't cares — automated insertion and prevention methodologies. In *2015 IEEE International Test Conference (ITC)*, 2015.

[33] Apostolos P Fournaris, Lidia Pocero Fraile, and Odysseas Koufopavlou. Exploiting hardware vulnerabilities to attack embedded system devices: a survey of potent microarchitectural attacks. *Electronics*, 6(3):52, 2017.

[34] Wenxue Fu, Jianwen Ma, Pei Chen, and Fang Chen. *Remote Sensing Satellites for Digital Earth*, pages 55–123. Springer Singapore, Singapore, 2020.

[35] Cesare Garlati and Sandro Pinto. A clean slate approach to linux security risc-v enclaves. In *Proceedings of the Embedded World Conference, Nuremberg, Germany*, 2020.

[36] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K Gürkaynak, and Luca Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[37] Francisco Gómez, Miguel Masmano, Vicente Nicolau, Jan Andersson, Jimmy Le Rhun, David Trilla, Felipe Gallego, Guillem Cabo, and Jaume Abella Ferrer. De-risc–dependable real-time infrastructure for safety-critical computer systems. *Ada User Journal*, 41(2):107–112, 2020.

[38] Abraham Gonzalez, Ben Korpan, Ed Younis, and Jerry Zhao. Spectrum: Classifying, replicating and mitigating spectre attacks on a speculating risc-v microarchitecture.

[39] U. Guin, K. Huang, D. DiMase, J. M. Carulli, M. Tehranipoor, and Y. Makris. Counterfeit integrated circuits: A rising threat in the global semiconductor supply chain. *Proc. IEEE*, 102(8):1207–1228, 2014.

[40] U. Guin, Ziqi Zhou, and A. Singh. A novel design-for-security (dfs) architecture to prevent unauthorized ic overproduction. In *2017 IEEE 35th VLSI Test Symposium (VTS)*, pages 1–6, 2017.

[41] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[42] Austin Harris, Shijia Wei, Prateek Sahu, Pranav Kumar, Todd Austin, and Mohit Tiwari. Cyclone: Detecting contention-based cache information leaks through cyclic interference. 2019.

[43] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[44] Wei Hu, Lu Zhang, Armaiti Ardeshiricham, Jeremy Blackstone, Bochuan Hou, Yu Tai, and Ryan Kastner. Why you should care about don't cares: Exploiting internal don't care conditions for hardware trojans. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017.

[45] Y. Jin, M. Maniatakos, and Y. Makris. Exposing vulnerabilities of untrusted computing platforms. In *Proc. Int. Conf. Computer Design*, pages 131–134, 2012.

[46] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur 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*, page 361–372, 2014.

[47] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[48] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution, December 2018.

[49] B. Kosinski and K. Dodson. Key attributes to achieving >99.99 satellite availability. In *2018 IEEE International Reliability Physics Symposium (IRPS)*, pages 6A.3–1–6A.3–10, March 2018.

[50] Sotiris B Kotsiantis, I Zaharakis, P Pintelas, et al. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160(1):3–24, 2007.

[51] Max Kuhn and Kjell Johnson. *Applied predictive modeling*. Springer, 2013.

[52] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[53] Jake Longo, Elke De Mulder, Dan Page, and Michael Tunstall. Soc it to em: electromagnetic side-channel attacks on a complex system-on-chip. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 620–640. Springer, 2015.

[54] David Lopez and Enrique Fraga. Tm/tc encryption system. In *14th International Conference on Space Operations*, page 2330, 2016.

[55] Tao Lu. A survey on risc-v security: Hardware and architecture. *ArXiv*, abs/2107.04175, 2021.

[56] Chao Luo, Yunsi Fei, Pei Luo, Saoni Mukherjee, and David Kaeli. Side-channel power analysis of a gpu aes implementation. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 281–288. IEEE, 2015.

[57] Yangdi Lyu and Prabhat Mishra. A survey of side-channel attacks on caches and countermeasures. *Journal of Hardware and Systems Security*, 2(1):33–50, 2018.

[58] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal covert channels on multi-core platforms. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 865–880, 2015.

[59] Arjun Menon, Subadra Murugan, Chester Rebeiro, Neel Gala, and Kamakoti Veezhinathan. Shakti-t: A risc-v processor with light weight security extensions. In *Proceedings of the Hardware and Architectural Support for Security and Privacy*, pages 1–8. 2017.

[60] Subhasish Mitra, H-S Philip Wong, and Simon Wong. The trojan-proof chip. *IEEE Spectrum*, 52(2):46–51, 2015.

[61] Satoshi Mitsuno, Junichiro Kadomoto, Toru Koizumi, Ryota Shioya, Hidetsugu Irie, and Shuichi Sakai. A high-performance out-of-order soft processor without register renaming. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 73–78, 2020.

[62] Mohammad Tehranipoor and Cliff Wang. *Introduction to Hardware Security and Trust*. Springer-Verlag New York, 2012.

[63] Adib Nahiyan and Mark Tehranipoor. Code coverage analysis for ip trust verification. In *Hardware IP security and trust*, pages 53–72. Springer, 2017.

[64] Shoei Nashimoto, Daisuke Suzuki, Rei Ueno, and Naofumi Homma. Bypassing isolated execution on risc-v with fault injection. Cryptology ePrint Archive, Report 2020/1193, 2020. `https://ia.cr/2020/1193`.

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

[66] Alessandro Palumbo, Luca Cassano, Bruno Luzzi, José Alberto Hernández, Pedro Reviriego, Giuseppe Bianchi, and Marco Ottavi. Is your fpga bitstream hardware trojan-free? machine learning can provide an answer. *Journal of Systems Architecture*, 128:102543, 2022.

[67] Alessandro Palumbo, Luca Cassano, Pedro Reviriego, Giuseppe Bianchi, and Marco 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)*, pages 1–6, 2021.

[68] Colin Percival. Cache missing for fun and profit, 2005.

[69] José Bezerra Pessoa. Space age: Past, present and possible futures. *Journal of Aerospace Technology and Management*, 13, 2021.

[70] Michel Pignol. Cots-based applications in space avionics. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 1213–1219. IEEE, 2010.

[71] Christian Pilato, Kanad Basu, Francesco Regazzoni, and Ramesh Karri. Black-hat high-level synthesis: Myth or reality? *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(4):913–926, 2018.

[72] Miodrag Potkonjak. Synthesis of trustable ics using untrusted cad tools. In *Proceedings of the 47th Design Automation Conference*, pages 633–634, 2010.

[73] Pedro Reviriego, Salvatore Pontarelli, Juan Antonio Maestro, and Marco Ottavi. A method to protect bloom filters from soft errors. In *2015 IEEE International*

Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS), pages 80–84. IEEE, 2015.

[74] Catherine Rooney, Amar Seeam, and Xavier Bellekens. Creation and detection of hardware trojans using non-invasive off-the-shelf technologies. *Electronics*, 7(7):124, 2018.

[75] M. Rostami, F. Koushanfar, J. Rajendran, and R. Karri. Hardware security: Threat models and metrics. In *Proc. Int. Conf. Computer-Aided Design*, pages 819–823, 2013.

[76] J. A. Roy, F. Koushanfar, and I. L. Markov. Extended abstract: Circuit cad tools as a security threat. In *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, 2008.

[77] Linton G Salmon. A perspective on the role of open-source ip in government electronic systems. In *7th RISC-V Workshop Proceedings*, 2017.

[78] Bicky Shakya, Tony He, Hassan Salmani, Domenic Forte, Swarup Bhunia, and Mark Tehranipoor. Benchmarking of hardware trojans and maliciously affected circuits. *Journal of Hardware and Systems Security*, 1(1):85–102, 2017.

[79] Chaoqun Shen, Congcong Chen, and Jiliang Zhang. Micro-architectural cache side-channel attacks and countermeasures. 12 2020.

[80] Rui Shu, Peipei Wang, Sigmund A Gorski III, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. A study of security isolation techniques. *ACM Comput. Surv.*, 49(3), October 2016.

[81] 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*, 20(1):465–488, 2018.

[82] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. Systematic classification of side-channel attacks: A case study for mobile devices. *IEEE Communications Surveys & Tutorials*, 20(1):465–488, 2017.

[83] G Sumathi, L Srivani, D Thirugnana Murthy, K Madhusoodanan, and SAV Satya Murty. A review on ht attacks in pld and asic designs with potential defence solutions. *IETE Technical Review*, 35(1):64–77, 2018.

[84] Sandeep Sunkavilli, Zhiming Zhang, and Qiaoyan Yu. Analysis of attack surfaces and practical attack examples in open source fpga cad tools. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, pages 504–509, 2021.

[85] Sandeep Sunkavilli, Zhiming Zhang, and Qiaoyan Yu. New security threats on fpgas: From fpga design tools perspective. In *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 278–283, 2021.

[86] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE design & test of computers*, 27(1):10–25, 2010.

[87] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Design & Test of Computers*, 27(1), 2010.

[88] Mohammad Tehranipoor and Cliff Wang. *Introduction to hardware security and trust*. Springer Science & Business Media, 2011.

[89] Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini, Germain Haugou, Eric Flamand, Frank K Gurkaynak, and Luca Benini. Pulpino: A small single-core risc-v soc. In *3rd RISCV Workshop*, 2016.

[90] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[91] N. G. Tsoutsos and M. Maniatakos. Fabrication attacks: Zero-overhead malicious modifications enabling modern microprocessor privilege escalation. *IEEE Trans. Emerging Topics in Computing*, 2(1):81–93, 2014.

[92] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018. See also technical report Foreshadow-NG [98].

[93] Xinmu Wang, Tatini Mal-Sarkar, Aswin Krishna, Seetharam Narasimhan, and Swarup Bhunia. Software exploitable hardware trojans in embedded processor. In *2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 55–58. IEEE, 2012.

[94] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505, 2007.

[95] Andrew Waterman and Krste Asanovic. The risc-v instruction set manual volume ii: Privileged architecture document version 20190608-priv-msu-ratified. Technical report, RISC-V Foundation, 2019.

[96] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, and Volume I User level Isa. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2, 2014.

[97] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. A cache timing attack on aes in virtualization environments. In *International Conference on Financial Cryptography and Data Security*, pages 314–328. Springer, 2012.

[98] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018. See also USENIX Security paper Foreshadow [92].

[99] Kan Xiao, Adib Nahiyan, and Mark Tehranipoor. Security rule checking in ic design. *Computer*, 49(8):54–61, 2016.

[100] 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)*, pages 428–441, 2018.

[101] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution in-

visible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441. IEEE, 2018.

[102] Yuval Yarom and Katrina 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`.

[103] Hansen Zhang, Soumyadeep Ghosh, Jordan Fix, Sotiris Apostolakis, Stephen R. Beard, Nayana P. Nagendra, Taewook Oh, and David 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*, ASPLOS '19, page 361–377, New York, NY, USA, 2019. Association for Computing Machinery.

[104] Jiliang Zhang and Gang Qu. Recent attacks and defenses on fpga-based systems. 12(3), 2019.

[105] Rui Zhang, Natalie Stanley, Christopher Griggs, Andrew Chi, and Cynthia Sturton. Identifying security critical properties for the dynamic verification of a processor. *ACM SIGARCH Computer Architecture News*, 45(1):541–554, 2017.

[106] Xuehui Zhang and Mohammad Tehranipoor. Case study: Detecting hardware Trojans in third-party digital IP cores. In *Proc. Hardware-Oriented Security and Trust*, pages 67–70, 2011.

[107] Jianfeng Zhu, Ao Luo, Guanhua Li, Bowei Zhang, Yong Wang, Gang Shan, Yi Li, Jianfeng Pan, Chenchen Deng, Shouyi Yin, Shaojun Wei, and Leibo Liu. Jintide: Utilizing low-cost reconfigurable external monitors to substantially

enhance hardware security of large-scale cpu clusters. *IEEE Journal of Solid-State Circuits*, 56(8):2585–2601, 2021.

[108] Jianfeng Zhu, Ao Luo, Guanhua Li, Bowei Zhang, Yong Wang, Gang Shan, Yi Li, Jianfeng Pan, Chenchen Deng, Shouyi Yin, Shaojun Wei, and Leibo Liu. Jintide: Utilizing low-cost reconfigurable external monitors to substantially enhance hardware security of large-scale cpu clusters. *IEEE Journal of Solid-State Circuits*, 56(8):2585–2601, 2021.