A Comprehensive Survey of Attacks without Physical Access Targeting Hardware Vulnerabilities in IoT/IIoT Devices, and their Detection Mechanisms

NIKOLAOS-FOIVOS POLYCHRONOU, Univ. Grenoble Alpes, CEA, LETI, DSYS, FRANCE
PIERRE-HENRI THEVENON, Univ. Grenoble Alpes, CEA, LETI, DSYS, FRANCE
MAXIME PUYS, Univ. Grenoble Alpes, CEA, LETI, DSYS, FRANCE
VINCENT BEROULLE, Univ. Grenoble Alpes, Grenoble INP, LCIS, FRANCE

With the advances in the field of the Internet of Things (IoT) and Industrial IoT (IIoT), these devices are increasingly used in daily life or industry. To reduce costs related to the time required to develop these devices, security features are usually not considered. This situation creates a major security concern. Many solutions have been proposed to protect IoT/IIoT against various attacks most of which are based on attacks involving physical access. However, a new class of attacks has emerged targeting hardware vulnerabilities in the micro-architecture that do not require physical access. We present attacks based on micro-architectural hardware vulnerabilities and the side effects they produce in the system. In addition, we present security mechanisms that can be implemented to address some of these attacks. Most of the security mechanisms target a small set of attack vectors, or a single specific attack vector. As many attack vectors exist, solutions must be found to protect against a wide variety of threats. This survey aims to inform designers about the side effects related to attacks and detection mechanisms that have been described in the literature. For this purpose, we present two tables listing and classifying the side effects and detection mechanisms based on the given criteria.

CCS Concepts: • **Security and privacy** → *Intrusion detection systems*; **Embedded systems security**; **Hardware security implementation**; **Side-channel analysis and countermeasures**; *Intrusion detection systems*; **Embedded systems security**; **Hardware security implementation**.

Additional Key Words and Phrases: IoT, IIoT, security, attacks, hardware vulnerabilities, side effects, detection, detection mechanisms

Authors' addresses: Nikolaos-Foivos POLYCHRONOU, nikolaos.polychronou@cea.fr, Univ. Grenoble Alpes, CEA, LETI, DSYS, 17 Avenue des Martyrs, F-38000, Grenoble, Isere, FRANCE; Pierre-Henri THEVENON, Pierre-henri.THEVENON@cea.fr, Univ. Grenoble Alpes, CEA, LETI, DSYS, 17 Avenue des Martyrs, F-38000, Grenoble, Isere, FRANCE; Maxime PUYS, Maxime.PUYS@cea.fr, Univ. Grenoble Alpes, CEA, LETI, DSYS, 17 Avenue des Martyrs, F-38000, Grenoble, Isere, FRANCE; Vincent BEROULLE, vincent.beroulle@lcis.grenoble-inp.fr, Univ. Grenoble Alpes, Grenoble INP, LCIS, 50 Rue Barthélémy de Laffemas, Valence, Drome, FRANCE, 26000.

1084-4309/2021/1-ART1 $15.00
https://doi.org/10.1145/3471936

ACM Reference Format:
Nikolaos-Foivos POLYCHRONOU, Pierre-Henri THEVENON, Maxime PUYS, and Vincent BEROULLE. 2021. A Comprehensive Survey of Attacks without Physical Access Targeting Hardware Vulnerabilities in IoT/IIoT Devices, and their Detection Mechanisms. *ACM Trans. Des. Autom. Electron. Syst.* 1, 1, Article 1 (January 2021), 39 pages. https://doi.org/10.1145/3471936

1 INTRODUCTION

Internet of Things (IoT) and Industrial IoT (IIoT) devices are becoming increasingly popular and now represent a significant part of the computational devices market. IoT and IIoT range from simple to more complex devices. Their primary principle is the interaction with the user or environment and the exchange of data mainly through a communication interface. IoT and IIoT can be invasive in individual lives or the industrial environment. They also handle a large amount of information and data that they either process, store, or transmit through a network. A high percentage of this information is sensitive to the user or the industry. For example, a voice controller handles user requests and consequently processes patterns corresponding to their habits or daily life. A smart lock may store the entry passwords to an individual's house. An actuator or automated machine programmed to produce/design a product of some security concern. Due to their popularity and the amount of sensitive information they can potentially handle, IoT and IIoT are the object of increasing attention from attackers. Attackers take advantage of the lack of security features in IoT/IIoT to penetrate the system and undertake their malicious activity. Attack campaigns such as Stuxnet [20] and Mirai [47] demonstrated the high risk of cyber-attacks. These campaigns led designers to place more emphasis on security for IoT/IIoT.

In today's systems two main classes of vulnerabilities are found: software and hardware vulnerabilities. Software vulnerabilities are flaws or defects in the software code that allow attackers to exploit the Operating System (OS), or applications in the system, to gain some privileges. In contrast, hardware vulnerabilities are flaws present in the hardware system. For example, the Rowhammer [91] attack exploits faults generated in the Dynamic Random Access Memory (DRAM) due to repeated access to the same memory locations over a short period of time. Hardware vulnerabilities allow attackers to directly exploit interactions with the system's electronic components, without the need for a software vulnerability, and regardless of the OS.

Traditionally, hardware attacks are used to extract information through leakages such as computing time, power consumption, electromagnetic radiation, or injection of faults into the hardware. Thus, if attackers have physical access to the device, they can employ methods such as laser injection fault attacks [18, 89], the Joint Test Action Group (JTAG) interface [59, 80] or voltage/clock glitch attacks [11, 48] to attack the system.

Due to the complex architectures of modern systems, the attack surface has increased. A new class of Software Attacks Targeting Hardware Vulnerabilities (SATHV) in the various units of the system has emerged. The units targeted include the memory (e.g., cache, DRAM), debugging interfaces, power, and frequency management modules or solutions used to optimize the computation time such as the out-of-order and speculative execution. In contrast with the usual hardware attacks mentioned earlier, which require physical access, SATHV attacks can be performed remotely. If attackers do not have physical access to the targeted device, they must access it using communication interfaces such as WiFi or Bluetooth. They can then perform a remote access attack such as clock/voltage fault attacks [78, 90] or a JavaScript Cache side-channel attack (Cache SCA) [34, 70]. Cache SCA [57], Rowhammer [91], Spectre [46] and Meltdown [56] correspond to some of the most

serious non-invasive attacks to have occurred that rely on hardware vulnerabilities. This new class of attack poses a severe threat to the security of modern devices, and specific countermeasures must consequently be deployed.

It is common to develop patches to resolve problems related to software vulnerabilities following their detection. In contrast, hardware vulnerabilities tend to be less extensively researched than software vulnerabilities. This distinction could be justified by the fact that security researchers have more limited access to deployed system architectures. Typically, software tools (e.g., anti-virus, firewalls, anti-malware) can ensure the system is safe from attacks involving injection of malicious code taking advantage of software vulnerabilities. Anti-virus software compares information obtained from the system with known malware information stored in databases. Firewalls filter information coming from the internet. Anti-malware systems work like anti-viruses, but using different methods like signature check, heuristics, sandboxing. However, although these security features may be able to detect the software part of an attack, they cannot always detect hardware-related vulnerabilities [52, 68]. Thus, if the attacker induces a fault in the hardware, these software tools often have no access to this information [52].

To make the system safe with respect to hardware attacks, several physical protections have been developed. Such as active shields protecting against invasive probing attacks [24], fault-tolerant redundant hardware systems [10] to prevent fault attacks, dual coil [40] or light sensors [12] to protect against electromagnetic attacks. Because of the extensive research into physical protections, we consider that they are already implemented in systems which are physically accessible to prevent physical attacks. Based on this assumption, this survey does not include attacks like electromagnetic emission analysis, power consumption analysis, leakage power analysis, fault attacks like electromagnetic, clock, voltage, temperature glitches, laser attacks, cold boot, bus probing, etc.

In contrast to physical attacks, it is more challenging to secure the system against SATHV. Both hardware and software detection mechanisms have been proposed in the literature to address SATHV. As previously mentioned, a software implementation of a detection mechanism frequently has no access to hardware information and it raises the load of the processor. As a result, hardware-implemented detection mechanisms are favored. These mechanisms have the advantage of not degrading the system performance as much as a software implementation, and can directly access hardware information that could allow attacks to be detected.

The information that both software and hardware detection mechanisms rely on to detect the attacks are the side effects of SATHV. The term side effects, which will be used throughout the survey, includes all the signals, configurations, and modifications of the system that change their nominal value or area of values due to the execution of SATHV alongside legitimate software. The side effect selection is critical to determine the detection capability of a detection mechanism.

Threat model. In our threat model, we make the following assumptions. **Assumption 1** is that the attackers have no physical access to the targeted system. In general, the industrial environment provides limited access to systems, and consumer IoT are stored within the user's home. Moreover, when IoT devices can become a target of their own users, we consider physical protections to exist. Examples of such protections include an integrated secure element or detection of attempts to dismantle the device. Above, we mentioned some of the physical protections that can be used to protect the system against physical attacks. **Assumption 2** is that the attackers in our threat model had access to the system. During

this access, they introduced or implemented a piece of malicious code on the system. This could be achieved for example by loading a corrupted application, or introducing a bug into the application code, or the OS. Furthermore, network interfaces e.g., WiFi, Bluetooth, Ethernet, can be used to gain access. **Assumption 3** is that devices equipped with an OS are targeted. Some examples of OS in our threat model are Linux, embedded Linux, and Android. We will not consider devices like micro-controllers, as we are only interested in devices with more complex microprocessor architectures such as smartphones. The attackers' aim is to gain additional levels of privilege or to extract sensitive information from the system. **Assumption 4** is that the attacker may already have privileges. Privilege escalation can be achieved due to an OS bug. OS bugs are frequently discovered and allow attackers to gain access to the - normally protected - system information. The attacks studied could be considered malware and are based on software code targeting a hardware vulnerability in the system microarchitecture.

Problematic. As more SATHV appear each year, this type of attack needs to be counteracted. One way to do this is by mitigating the attacks, and another way is by monitoring the system to detect them. Because mitigation does not resolve the problem, monitoring, detection, and reaction represent a more definitive solution. To help designers take SATHV into account, they must be aware of most of the existing SATHV and the hardware vulnerabilities they target. In addition, detection mechanisms addressing SATHV described in the literature should be presented.

Because SATHV can induce a fault in the hardware, they can bypass all the security checks present in the OS. In addition, software-implemented mechanisms only have access to a limited amount of information on hardware side effects which could reveal the attacks. Furthermore, the hardware side effects access to software are architecture-independent. As mentioned above, software solutions introduce large performance overheads, as both detection mechanisms and running applications requiring access to the Central Processing Unit (CPU). The attack may detect the detection mechanism, and in response hide its malicious behavior [65]. This is possible as the attack and the detection mechanism share system resources. Finally, depending on the application, some software detection mechanisms might be unacceptable. For all these reasons, software solutions to SATHV are limited. However, hardware implementations also have limitations. For example, the cost of the added design time, extra features in the system design, and the difficulty updating hardware after its production counteracts the benefits of this type of approach. In addition, most of the detection mechanisms implemented in the literature are focused on the detection of specific attacks. For example [21] detects only three variants of Cache CSA, [3] detects only Meltdown, and [55] detects only Rowhammer and Spectre. The aforementioned solutions have no general fair cost strategy to detect a large set of attacks, and the mechanisms require update depending on the application and the platform. This raises severe problems as new attacks appear every year and a large database of attack vectors exists. Detection mechanisms limited to specific attacks which cannot be upgraded in the future to include new attacks will not be adopted by vendors.

During an attack of the system, numerous side effects appear. Detection mechanisms rely on these side effects to detect the attack. It is thus important to create a collection of attack side effects, but also to list the side effects used by the detection mechanisms described in the literature. This database could serve as a tool for security designers, who want to find out what is used in current detection mechanisms that might be useful for modification or inclusion in their own mechanisms.

Related work. Many surveys of security and privacy in embedded systems, and more specifically in IoT and IIoT have been published. Here, we will initially present and categorize existing surveys. We will predominantly focus on surveys dealing with attacks on IoT/IIoT systems and security mechanisms. Most of the surveys conducted previously present mitigation techniques. However, as indicated above, mitigation techniques target specific attacks and can only limit the impact or make the attack more difficult to implement without resolving the issue.

In 2019, Ehret et al. [27] presented a survey of hardware security techniques targeting low-power System on Chip (SoC) designs. The techniques presented mostly mitigate attacks without detecting them, and require the attacker to try harder. The authors classified the techniques based on which attacks each technique could mitigate. In another study published in 2019, Lokhande et al. [58] presented attacks targeting hardware architecture to bypass OS security. The authors discussed the operations performed at the hardware level and the different exploits that rely on faults in these operations. The surveys focused on a limited set of attacks and ultimately the authors questioned the impact on system security. Also in 2019, Szefer [88] presented the key features of the processor micro-architectural functional units, which make covert and side channel attacks possible. After presenting several covert and side channel attacks, Szefer et al. present some software and hardware defenses. The defenses presented focused only on mitigating the attacks, not on their detection. In 2020, Sengupta et al. [83] presented attacks to the different system layers, e.g., physical, network, software, data. These authors also presented countermeasures for each layer, thus producing a taxonomy of the security research in IoT and IIoT. In a similar survey also published in 2020, Kwong et al. [50] attempted to present a comprehensive review of all major security attacks faced by the IoT industry and the existing mitigation techniques. The authors discuss the potential benefits and shortcomings of the different techniques by analyzing their pros and cons. However, the survey only dealt with software attacks and remained relatively general, not going deep into the source of the vulnerabilities. Finally, another comprehensive survey from 2020 was published by Akram et al. [2]. This survey only related to cache side-channel attacks (cache SCA) and how they could be detected. The survey covered a broad range of detection mechanisms for cache SCA, and presented a taxonomy of these mechanisms.

Contributions. The key contributions of the survey presented in this article, compared to the previously published surveys in the area of IoT/IIoT, are as follows: it is the first survey to list most of the SATHV, alongside their targeted hardware vulnerabilities, and their side effects. The side effects are also classified based on a set of proposed criteria. Detection mechanisms implemented in modern devices, considering the previously mentioned threat model are listed, and the mechanisms studied are finally classified according to a specified set of criteria.

This survey thus provides an insight into SATHV and the hardware vulnerabilities targeted. We further discuss the side effects of each attack, which correspond to the footprints of each attack. These attack footprints can be used to implement detection mechanisms to monitor for the side effects and thus detect attacks. As previously mentioned, the goal is to address multiple threats simultaneously. We present a taxonomy of the side effects, from which the reader can determine whether attacks cause similar side effects, have some correlation, or are completely different.

Finally, hardware mechanisms based on detection of side effects described in the literature are presented. These mechanisms attempt to address the security issues targeted by the

attacks included in our threat model. These mechanisms are classified according to the criteria we define and describe. We have focused our survey on detection mechanisms, as we want to react instantaneously to find and delete the malware. In most cases, mitigation techniques (countermeasures or mechanisms to slow the attack) are specific to one attack and cannot completely protect against all attacks. In contrast, detection mechanisms could be used to deal with multiple attacks and eliminate threats. We believe design detection mechanisms addressing all attacks are needed because systems have become too complex to address the increasing demands, for example, on memory and speed. The increase in system complexity in turn augments the number of possible attacks. Attackers can use more than one attack to intrude the system, potentially bypassing detection mechanisms implemented for a specific attack or attack variant. A detection mechanism addressing a variety of attacks is therefore desirable. Designers of detection mechanisms can refer to our table and list of detection mechanisms to extract information that will help with their goal.

To the best of our knowledge, this is the first survey describing software attacks targeting hardware vulnerabilities in the context of micro-architecture and presenting relevant detection mechanisms. The goal of this article is to help security designers to obtain information on the detection mechanisms that have been implemented, and the side effects through which to detect attacks. Because detection mechanisms implemented in the literature focus on specific attacks rather than a set of multiple attacks, this survey should help designers make their system more secure following their vulnerability analysis. With this knowledge, and the annotated lists of side effects and detection mechanisms presented in this survey, security designers could benefit by designing their detection mechanisms depending on the limitations of their application and their system.

In summary, our key contributions are:

- Presentation of software attacks targeting hardware vulnerabilities in the system micro-architecture and listing their side effects.
- Presentation of registers and signals to help extract information to detect SATHV attacks.
- Introduction of a set of criteria by which to classify the side effects of the attacks.
- Discussion of detection mechanisms addressing the attacks presented. Detection mechanism designers could benefit by acquiring information on the limitations and advantages of the current detection mechanisms. Subsequently, they could use this knowledge to improve these implementations through modifications, or use the implementation idea to create a more complex and capable mechanism to detect a broader range of attacks.
- Classification of the detection mechanisms according to a set of criteria. The criteria will help detection mechanism designers to choose the most appropriate and cost-effective mechanisms for their platform and applications, and based on their vulnerability analysis.

Outline. The remainder of the paper is structured as follows. In section 2, we present SATHV targeting IoT and IIoT systems. The attacks presented target hardware vulnerabilities in the system architecture, and the methods used to exploit these vulnerabilities are implemented in software. We also list the side effects of these attacks. In section 3, we go into more technical detail for each of the side effects listed for each attack. The side effects of the attacks are summarized in a table. In section 4 we provide a set of classification criteria for the side effects presented in section 3. In section 5, we discuss detection mechanisms described in the literature, which address the security issues associated with the attacks presented. We define some classification criteria and propose a taxonomy table for the detection

mechanisms. Finally, in section 6, we discuss some choices, limitations, and improvements in relation to the survey presented, before concluding our article. The conclusion analyzes choices and limitations with regard to the data used for the classifications and analysis of the vulnerabilities, and proposes some improvements to be implemented in our future work.

2 IOT/IIOT ATTACKS, SYSTEM HARDWARE VULNERABILITIES, AND SIDE EFFECTS

In this section, we focus on software attacks targeting the system micro-architecture and involving hardware vulnerabilities. Hardware vulnerabilities may be the result of a design flaw, an implementation flaw, or an incorrect configuration. Attacks that target a hardware vulnerability but require physical access to the system are not included as they are beyond the scope of our threat model. In addition, as it is not possible to list all of the attacks presented in the literature, we present attack vectors that target different system units. If an attack vector has several variants, we present the variant that best describes the vulnerability. These choices were made as we wished to focus on the side effects of each attack vector. Furthermore, variants of an attack frequently present similar side effects. Where this is not the case, we have clearly stated the variants of the attack presenting distinct side effects.

To begin with, we will give some background information on the terms used in this section. This will help the reader to follow our analysis. Next, we divide the attacks according to the unit they target, e.g., the memory, the out-of-order and speculation units, the debug interface, the power, frequency, and thermal units. Subsequently, we provide a general insight into each attack in three parts. We start with a quick description of the vulnerability followed by some ways of exploiting it that have been mentioned in the literature, without going into detail on how they work and how to perform them – such detail can be found in the bibliography. Finally, we list the side effects produced by each of the attack vectors described. Later in the paper, we will present further technical details of the side effects and how they affect the system, and can be detected.

It should be noted that we list side effects related to both ARM and Intel architectures, as well as running Linux OS. ARM is a leading architecture in IoT/IIoT devices, whereas Intel focuses more on high-performance processors. Despite this, Intel has also increased its attention on the IoT market, for example through the design of the 11th Gen Core. The side effects reported in the referenced literature and presented in this survey relate to these architectures and OS. Nonetheless, this choice does not limit the range of our threat model, which covers multiple architectures and rich OSs. The architecture will be mentioned before the presentation of the side effects, to notify readers and avoid confusion. If the side effects are observable only in a specific architecture, this information is mentioned.

2.1 Background

In the following subsections, we will use some terms during the SATHV analysis. This subsection is included to help the reader better understand the concept of the attacks described.

TrustZone is a security feature introduced by ARM. It is an extension implemented in the hardware that aims to provide a secure execution environment. The environment is made secure by splitting the resources of the microprocessor into two execution worlds, normal and secure [64]. The hardware restricts the access of normal applications to the resources labeled as secure.

Differential Fault Analysis (DFA) is a technique that forces a cryptographic implementation to compute incorrect results and attempts to take advantage of them. Using DFA it is

possible to retrieve cryptographic keys by analyzing and comparing the correct and faulty output pairs. DFA is used to inject faults with clock glitches, voltage glitches, lasers, etc. A DFA in the AES implementation was demonstrated in [1].

2.2 Memory Attacks

We will start by describing attacks dealing with vulnerabilities related to memory. These attacks target information stored in the memory, by corrupting it, or using it as a side-channel. We describe attacks related to cache and DRAM memory. The attacks presented are cache SCA, Rowhammer, DRAMA and DMA attacks.

Cache SCA. Cache SCA are among the most basic side channels exploited. The fundamental hardware vulnerability exploited by a cache covert channel is that it is faster to access directly from the cache than the main memory. In addition to the time consideration, Cache SCA channels take advantage of other hardware vulnerabilities. Thus, the Flush+Flush [37] attack exploits the hardware vulnerability of the timing difference when serving the *"flush"* instruction. When the targeted address to be *flushed* is available in the cache memory, the flush instruction deals with the request more slowly. Attackers can gain knowledge on the execution path by measuring the aforementioned timing-based side effects.

Various techniques have been used to perform a cache SCA. Some of the exploits are: Evict+time [72], Prime+Probe [57], Flush+Flush, Flush+Reload [93], Evict+Reload [39], Prime+Count [22]. To present an example of a technique, during Flush+Reload the attacker flushes a targeted address. If the victim uses this address, the system will reload it as it was just flushed by the attacker leading to a cache miss. After the victim executes, the attacker reloads the address. If the victim reloaded the address, a hit occurs e.g., the time needed to access the address is shorter. If the victim did not use the address, a miss occurs e.g., a longer access time is observed as the data must be retrieved from the main memory.

Some of the exploits relying on cache SCA are presented here. [72] attack AES OpenSSL on x86, whereas [74] and [13] attack the RSA OpenSSL and RSA SGX SDK version. In [39] attackers observed the cache to extract the users' keystrokes, and in [71], they attacked the javascript sandbox. The cache side-channel is also the covert channel used for other vulnerabilities such as Meltdown [56] and Spectre [45]. [94] attacks AES in TrustZone and [22] uses the side channel to transfer images from the secure world to the normal world in TrustZone. Both these approaches attack the AES by extracting the key based on the access pattern of the AES T-table. Furthermore, attacks on the *Translation Lookaside Buffer* (TLB) caches have been reported, such as attacks on the implementation of cryptographic algorithms [33], or which bypass the kernel *Address Space Layout Randomization* (ASLR) protection [36].

Most cache SCA rely on timing differences either when loading values from the cache/main memory or serving a *flush request*. To obtain these timing differences, attackers monitor the *Performance Monitoring Units* (PMUs) for ARM [8] or Performance Counter Monitor (PCM) for Intel [92]. For a cache SCA to work, the attacker must know how the physical addresses in the main memory are mapped to the virtual addresses used by the process. In most cache SCA techniques, this is necessary because a process using only virtual addresses must know where in the cache the targeted memory contents are mapped as most cache architectures are physically addressed. The translation information can be found in the page tables stored by the OS. In Linux, the attacker can use an OS file (*proc/PID/pagemap*) to obtain this information and find the translations. If this option is unavailable, the attacker

could try to reverse-engineer the mapping through more complex techniques [75] such as huge pages.

Having seen the resources used by attackers to mount a successful attack, we observed that access to the PMUs or *proc/PID/pagemap* is not necessary. The main side effects of cache SCA are related to the use of caches and small loops. These effects increase several stats such as cache misses [21, 23], cache hits [21], cache accesses [21], number of executed instructions [21], number of CPU clock cycles [21], speculative branches taken [23], retired branches [23]. The side effects observed values will be different for each architecture. Because of the distinct implementations, Intel is typically more vulnerable to cache SCA than ARM. Some of the reasons are the existence of the *clflush* instruction and *DC CVAC, DC CIVAC, DC ZVA* in Intel and ARMv8, which are not present in ARMv7. In addition, the random replacement policy in ARM caches makes eviction techniques more complex. Nevertheless, the pattern of side effects remains the same for both architectures – resulting in increased or decreased values. This comment applies to all attack vectors which use or bypass the cache, as we will explain later.

Finally, because the attacks are implemented in a loop, attackers must prevent the interrupt scheduler from inducing noise by interrupting the attack process. Following the same logic, attackers have to prevent the CPU and the compiler from reordering the instructions in their attack loop.

Rowhammer. Rowhammer is a hardware vulnerability in the Dynamic Random Access (DRAM) memory, that was first presented in [43]. Modern DRAMs have an increased cell density, which increases the electrical interaction between cells. The vulnerability exploited by Rowhammer is due to charge disturbances in DRAM capacitors, caused by the repeated access to neighboring rows. Attackers can modify the stored context of a targeted address, by repeatedly accessing the two neighboring rows that *sandwich* it. To do this, attackers use the pseudocode listed in Algorithm 1. The Rowhammer vulnerability depends on several parameters such as the frequency of the row activation interval, the refresh interval, and the data patterns.

The Rowhammer vulnerability has been exploited in a variety of scenarios. In [77] and [91] attackers induced faults in an OS page table and effectively gained some privileges. In [51] and [19] attackers performed a DFA on the RSA key. These examples showcase the potentially severe consequences of the Rowhammer vulnerability.

Algorithm 1 Rowhammer pseudocode

```
1: while time <DRAM refresh interval do
2:     LD/ST address 1
3:     LD/ST address 2
4:     flush address 1
5:     flush address 2
6:     memory barrier
7: end while
```

The Rowhammer attack involves a considerable number of steps, each of which produces side effects. The primary side effect of the attack is the increased number of accesses to the same memory locations, which the attacker uses to induce faults in the neighboring rows. The attack shares many similarities with cache SCA, as the attacker must bypass the cache levels to directly address the DRAM. This explains why the attack induces an increased

number of cache misses [54], hits [35] and accesses [54]. Interestingly, interaction with the CPU is unnecessary. [29] presents the GLitch attack, which performed the Rowhammer from the GPU, thus bypassing the CPU. Through control of a malicious website, attackers can get remote code to execute on a smartphone without relying on any software bug, as demonstrated by the researchers.

To use this hardware vulnerability, attackers must also know the virtual to physical mapping of addresses. However, sometimes this is not enough as the memory controller maps the physical addresses differently to the DRAM addresses. As the attacker must know the exact location of the addresses neighboring the target in the DRAM physical address space, finding these neighbors in the physical address space is not sufficient for a successful attack. These DRAM mappings are not always available from vendors, and thus the mapping functions must be reverse engineered to perform the attack. As the attack might require reverse engineering, timing differences of accesses within the DRAM must be calculated using the PMUs.

Furthermore, attackers also need to prevent the interrupt scheduler from inducing noise by interrupting the attack process. Similarly, the CPU, the compiler, and the memory controller must be prevented from reordering instructions and memory accesses. The overall number of page faults (known as the page fault ratio) is low [73]. This side effect is visible because the attack repeatedly accesses the same DRAM cells, though the attack process has limited interaction with the OS page fault handler. Repeatedly accessing the same addresses does not force the OS page fault handler to bring a missing page in the page table, thus producing the low page fault ratio. In addition, the number of branches taken is increased and the branch mispredictions are reduced [55].

DRAMA. This attack was presented in [75]. The hardware vulnerability exploited is the timing difference between accessing data from the *DRAM row buffer* or the DRAM. The row buffer acts like a cache for a DRAM row. A request served from an active row is faster than a request that requires a new row to be accessed. [75] presents two attack scenarios using the DRAM as a covert channel. In one of the covert channels, sender and receiver occupy different rows in the same DRAM bank. The sender continuously accesses a row in the DRAM. The receiver also continuously accesses a target address in the same DRAM and measures the average access time. If the addresses map to different rows, a longer access time is observed due to the row conflict. In contrast, if the addresses map to the same row, a decreased access time is observed. Sender and receiver can map the access time difference to '0' and '1' to send a message.

The DRAMA attack shares some side effects with the Rowhammer attack. Firstly, an increased number of accesses to the same memory row is observed. The attack also induces an increased number of cache misses, hits and accesses because the attackers must address the DRAM directly, by bypassing the caches. Reordering of instructions from the CPU and compiler, and reordering of memory accesses through the memory controller must be prevented. Interrupts are also prevented in the DRAMA attack.

Furthermore, the attackers must synchronize the covert channel. To do so, the attacks make use of *Time Stamp Counters* (TSC). TSC is a 64-bit register in x86 processors that counts the number of cycles since reset. The counter is used only to synchronize the two channels, so any precise counter could be used as an alternative, such as the PMU cycle counter. The side effects of reverse engineering are also visible here. Finally, PMUs are used to measure the timing difference of accessing an active or inactive row buffer.

Direct Memory Access (DMA) attacks. The DMA feature was introduced to allow peripherals rapid access to the memory without the involvement of the host CPU. The peripherals gain access to the whole host memory, completely bypassing the CPU. In this scenario, the CPU can perform other tasks while DMA transfers occur. This option gives freedom to a peripheral to read/write to all the DMA-accessible memory ranges. The hardware vulnerability exploited in this attack is the potential direct access to the memory bypassing the CPU, OS, and hardware access verifications.

Some attacks implemented that exploited the DMA vulnerability are DAGGER [87], ThunderClap [60], [63], and pcileech [30]. DMA attacks using the GPU's ability to access the DMA feature have been presented [95]. The attack needs a malicious driver and access to the kernel. As the authors propose, this access can be obtained using kernel exploits, by bypassing capability checks, or by exploiting kernel module loader weaknesses.

During a DMA attack, side effects are primarily observed on the system bus. A higher bus activity can be observed, as the attacker tries to read and write to memory locations using the DMA functionality, without interference from the CPU. Several events related to bus statistics can be observed, for example, memory bus transactions including burst transactions, invalidate transactions, partial reads, and writes.

2.3 Transient Execution Unit Attacks

With the aim of increasing CPU speed, vendors sometimes allow hardware to perform operations for subsequent instructions ahead of time or even out-of-order. To achieve this, the CPU must be able to predict the control flow, data dependencies, or even data. If the CPU prediction is correct, the operation continues. If the prediction was incorrect, it flushes the pipeline, restores the context before the prediction, and continues with the correct data. This offers the benefit that the time spent performing instructions ahead of time, is almost the same as stalling the CPU to wait for the calculated data to arrive. This ability gave rise to a new class of attacks called Transient Execution Attacks. Although the architectural effects and results of transient instructions are discarded, micro-architectural side effects remain after the transient execution. An example of such residual effects is the cache state that can subsequently be exploited by the attacker [16].

Meltdown. Meltdown is one of the most recent severe attacks exploited [56] corresponding to a hardware vulnerability related to the out-of-order execution. The hardware vulnerability exploited is access to restricted memory locations bypassing security checks due to the out-of-order execution. A vulnerable CPU allows an unprivileged process to load data from a privileged (kernel or physical) address into a temporary CPU register [56]. The attacker can then extract the information using a cache side-channel. Meltdown mostly affects Intel-based processors, but IoT devices based on ARM processors have also been reported to be vulnerable. Some vulnerable ARM-based processors including the ARM-Cortex A72 used in the Raspberry Pi 4 and Exynos 9820 used in Samsung Galaxy S10. A research from Graz university [38] also indicates that the Meltdown vulnerability could be exploited in the Samsung Galaxy S7 due to its use of an Intel chip. Since the original paper was published, several variants of the attack have been reported [14], [15], [26], [85], [44].

The primary side effect of Meltdown is related to the access to restricted memory. When a malicious process attempts to access a memory location that is not part of the process, the request will eventually fail, creating a segmentation fault. The access violation is caught after the content of the requested memory location is transferred to the cache. When the violation is determined, the requesting process is notified by the OS, e.g., by raising a *SIGSEGV* event

in Linux [3]. Segmentation faults that occur in memory locations close to each other can be marked as suspicious side effects, and can be extracted by kernel tracing. As Meltdown uses the cache as a side-channel to extract the information after inducing the fault, all the cache side effects listed above will also be visible here. The number of cache loads is a new interesting feature measured in the literature [28].

Spectre. Spectre [45] is another novel attack that was presented alongside the Meltdown vulnerability. Spectre is a hardware vulnerability related to speculative execution. The vulnerability relies on dedicated control or data flow prediction mechanisms, thus it transiently bypasses software-defined security policies (e.g., bound checking, function call/return abstractions, memory stores) to leak secrets out of the program's intended code/data paths [16]. If the processor does not know the future instruction stream for a program, it will speculate on the branch path. To succeed, the CPU saves the context of the current state and speculatively executes instructions. By speculative instruction execution, attackers can access context for which they should not have permission, and extract it through a cache side-channel. Following its publication, like Meltdown, many variants of Spectre emerged [44], [46], [49], [41].

For Spectre, the side effects observed are an increased number of branches and branch mispredictions during attacks. In contrast, the branch miss rate is decreased [55].

$$branch\_miss\_rate = branch\_mis - predictions/branches$$

This is because the attacker tries to train the branch predictor by calling the conditional branch many times with different input values that make the condition true [55]. As explained for Meltdown above, Spectre also uses the cache side-channel, and will thus produce similar side effects. The number of cache loads and cache references [55] also increases. Although Spectre has many variants, the side effects remain the same for most of them. According to [17], which examined the different Spectre variants and conducted a classification of them, three out of four variants use the same training and extraction methods, with the only difference being the mechanism exploited. As the aforementioned side effects rely on training of the predictor and the cache SCA to extract the information, three out of four variants present similar side effects. Only Spectre variant 4, which relies on the load/store dependencies, uses an alternative training method and might induce different side effects. For the other variants, the side effect values might differ, but the patterns should remain the same.

2.4 Debug Unit Attacks

The processor's debug unit is a hardware module designed to help developers debug hardware and software running on the processor. In combination with a software debugger, it allows developers to debug application software, operating systems, or processor-based hardware systems. Unrestricted access to this information leaves the system vulnerable to attackers. For our purposes, we refer to the debug component inside the system, and not the debug interface which requires physical access.

Nailgun. The Nailgun attack was presented in [67]; it targets ARM cores. This attack was made possible as the hardware does not check access privileges when in the debug state. Using this hardware vulnerability consequently, a component with low privilege can access resources of a higher privilege mode. Nailgun exploits this hardware vulnerability to attack the system, specifically by misusing the ARM debug components *Embedded Trace Macrocell* (ETM), *Program Trace Macrocell* (PTM) [5], [4] to extract sensitive information.

The attack relies on authentication signals [6] (signals that control whether non-invasive or invasive debugging is allowed in the non-secure or secure state) and whether they are enabled in the targeted system (vendors suggest that these signals be disabled after the development phase, but this is not the case for all manufacturers). This attack requires no physical access, unlike JTAG/debug interface attacks. The Nailgun attack has been possible since ARMv7. In this model, an on-chip processor can debug another processor (the debug target) on the same chip. Two attack scenarios are possible, depending on the authentication signals and whether the CPU inserts the debug state during the attack.

In the non-invasive scenario, Nailgun can infer the AES encryption key, based on the AES table-lookup implementation in the secure world. The ETM debug component can reconstruct the instruction accessed and data addresses involved in the encryption algorithm execution. The secret key can be inferred by reverse engineering the page table entries accessed.

In the invasive scenario, Nailgun demonstrates how an application running in non-secure EL1 (privilege level for the rich-OS) can arbitrarily execute payloads in EL3 (privilege level for the firmware). The attackers used a payload to extract fingerprint data from the secure world as proof of concept. The main steps in this type of attack are as follows: initially, the attackers must induce the debug state. Next, they must change the register value containing the first instruction after exiting the debug state. An *Secure Monitor Call (SMC)* instruction is inserted to take the processor to EL3. Then, the register value must be modified, holding the instruction indicated by the exception vector in EL3. The modified instruction will point to an arbitrary payload introduced by the attacker. The processor will execute the payload, and finally, the attacker must restore the modified register values to their initial values.

The ETM is used by the attacker to monitor the addresses accessed by a secure-world application. In the invasive attack scenario, the attacker uses a memory-mapped interface (*Debug Communications Channel*) to access system registers allowing communication between the attacker-controlled processor and the victim core. In addition, the attacker accesses system registers to insert and execute instructions when the system is in the debug state. The attacker modifies the register, which holds the first instruction to be executed after leaving the debug state. Moreover, the attacker modifies the instruction to which the exception vectors point [67]. When a secure application is running, whether these registers and features are used during the debug state can be checked. In ARM architecture, access to the system registers of ETM and PMU can raise an interrupt, thus preventing access to the EL3 secure OS [66].

eXecute-Only Memory (XOM). XOM memory was introduced to protect one vendor's Intellectual Property (IP) from other developers, even though they retain access to the binary code through the debug interface. Thus, code can be executed but read or write access to it is prevented [7]. Despite this protection, attackers can halt a processor when it is executing sensitive information. If the execution stops, the sensitive information is retained in the CPU resisters or the Static Random Access Memory (SRAM) contents. The attacker may then be capable of partially recovering information by looking at the remnant information [7]. Interrupts or debugging features, such as halting the processor, can be used to stall the execution and extract information. Consequently, the hardware vulnerability of restricted access data being available in system registers after an execution halting, allows sensitive information to be extracted, when no permission has been granted.

In [82], the attackers used the single-step feature of the debug interface or an interrupt-driven approach, to execute elements in the XOM and reverse engineer the targeted instructions. The interrupt-driven approach does not require any debugging capabilities but only privileged code execution on the target device.

Some vendors do not allow the processor to be stalled when executing code contained in the XOM. However, halting is permitted outside this region when the processor executes a function or an interrupt handler. For example, STM32Lx and STM32F4 allow the halt, but not STM32F7 and STM32H7.

The XOM attack presented above causes the following side effects. To start with, the attacker must use the single-step debug feature during execution of the protected code. If this option is unavailable, interrupts may be raised after the protected code is executed. The two above side effects are visible multiple times in the same memory context. The attacker executes a loop to obtain output information multiple times by processing the targeted memory context for different inputs. For the method with interrupts, the *System Tick* (SYSTick) timer is used to synchronize the interrupts [82].

2.5 Power and Clock Management Unit Attacks

The power and clock management hardware units responsible for providing the necessary voltage levels and clocks throughout the system. The need for power and energy efficiency has led to hardware-software management mechanisms. Dynamic Voltage and Frequency Scaling (DVFS) is a technique designed to save energy based on energy demands during runtime. When applying this technique, hardware regulators collaborate with the OS, which keeps track of the runtime demands. Flip-flops in synchronous circuits must follow specific constraints [81], so the system operates correctly. Both frequency and power have an impact on these constraints, which explains why the operating points of the device must be calculated in accordance with frequency and voltage levels. A deviation in the (voltage, frequency) operating points from safe levels can lead the hardware to produce faulty results. The hardware is vulnerable to these deviations, and attackers can take advantage of these vulnerabilities to attack the system.

Clkscrew [90] is an attack performed by software with low privileges which gains access to the power management hardware. The attack is implemented by a CPU with at least two cores. Because cores are powered at the same voltage, the attacker modifies the frequency of the victim core to induce a timing fault. This type of timing fault is a hardware vulnerability that is mostly exploited using hardware attacks, such as clock glitches [48]. In this case, it is exploited using software code that pushes the frequency operation point in the power management hardware to its limits.

As proof of concept, the authors implemented two attacks. In the first attack, they induced a fault in an AES round and performed a differential fault analysis. In the second, they induced a fault in the RSA key during TrustZone execution. A crafted signature signed by the fault key was loaded to allow a malicious app to be introduced into the secure world of TrustZone.

Voltjockey is another attack based on the power unit [78]. The attack is very similar to Clkscrew, but the error is due to an increase in propagation delay, as the result of a modified voltage. Like Clkscrew, the Voltjockey timing fault is a hardware vulnerability mostly exploited using hardware attacks such as voltage glitches [11]. The attacker core runs at low frequency, whereas the victim core runs in high frequency. By inducing a sudden drop in the voltage, the timing constraint is not respected. The low-frequency core is unaffected,

but the high-frequency victim core is severely affected. The proof of concepts is similar to the Clkscrew attack. Both attacks were demonstrated in commodity ARM devices.

The side effects mentioned below relate to Linux-ARM devices. However, similar effects could be observed with other platforms and OS, as DVFS is a common feature in most modern devices.

The Clkscrew and Voltjockey attacks need access to a malicious driver, to allow them to change the configurations of the frequency and voltage. To modify the frequency, the Phase-Locked Loop (PLL) unit register configurations are modified, and frequency configurations outside the operation range are used. To modify the voltage, the *Power Management Integrated Circuit* (PMIC) drivers, and *Subsystem Power Manager* (SPM) registers are altered to change the supplied power configurations. Operating Performance Points (OPPs) other than those specified by the vendors are used. Interrupts of the victim cores are disabled during the attack period, to reduce the noise of the targeted time of the induced fault. Finally, both attacks use cache side channels to profile the executed code, consequently triggering the same side effects as cache SCA.

2.6 Thermal Monitor Unit Attacks

Temperature Side Channel. The temperature is another interesting feature of the system, that is not often exploited. Temperature plays a vital role in the correct operation of a device, and temperature leakage measurements could be used as a side-channel. Modern CPUs use temperature monitors to prevent the system from overheating and manage power consumption. This information can be retrieved from special CPU registers. The thermal channels are interesting, as a defender could reset resources (e.g., cache, registers) to eliminate the side channel. However, the heat produced during the computation can be observed even after the end of the process, making it difficult to eliminate this covert channel. This hardware vulnerability due to the thermal footprint of system operation can be exploited to extract sensitive information.

A thermal side channel was presented in [61]. The authors of that study observed that the higher the operating frequency, the higher the core temperature. In the first attack scenario, the residual heat in one core is monitored. In the second scenario, the temperature of a core is monitored from a second core, thus establishing a covert channel between the two cores. The attacker performs a computationally intensive process (the DVFS will decide to increase the frequency) to send a *'1'* and remains idle to send a *'0'*. This attack was demonstrated on an Intel platform.

When the attackers use the available thermal monitor to establish a covert channel, they have to access the *Digital Thermal Sensor* (DTS) data through the coretemp kernel module. Coretemp is a Linux-specific module that allows access to Intel DTS. A thermal fingerprint that differs from the nominal fingerprint may be observed because the attacker changes the CPU's operating frequency from high to idle. By doing so, the temperature of the CPU changes accordingly. As explained before, when the CPU operates at a high frequency, the temperature is higher than when it operates at idle state frequency.

2.7 Summary

The attacks mentioned here cover a large surface in IoT/IIoT systems that are equipped with an OS and can perform complex functionalities. Based on the information presented, the reader should have a good understanding of the existing class of software attacks targeting hardware vulnerabilities. The vulnerabilities were presented briefly by referring to the exploits targeting the vulnerabilities. The side effects, which we list in this section,

will be more fully explained in section 3. As several attacks manipulate the memory, they share a large number of side effects. Table 1 presents attack vectors manipulating the cache or the main memory. We chose not to list attack variants, as the vulnerability is mostly the same. Because they are omitted from our threat model, we did not list invasive attacks on the system, such as laser fault attacks, clock, and voltage glitch attacks, cold boot, and bus probing. In addition, electromagnetic side-channel analysis attacks, power consumption analysis, and leakage power analysis were not mentioned, because we focus on attacks that can be mostly mounted from inside the system.

Table 1. Attack vectors related to memory manipulation

	Attacks							
	Cache CSA	Rowhammer	DRAMA	DRAM covert channel	DMA	Meltdown	Spectre	Clkscrew/ Voltjockey
Cache access/ manipulation	X	X	X	X		X	X	X
DRAM manipulation		X	X	X	X			
Cache as the covert channel	X					X	X	X
DRAM as a covert channel			X	X				

3 ATTACK SIDE EFFECT ANALYSIS

In this section, we list and explain the side effects presented briefly in section 2. In the previous section, we focused on the attack procedures, targeted vulnerabilities, and goals. Here, we list and describe files, hardware events and registers, which can be used to obtain information to detect the attacks. Furthermore, we provide an insight into why the attacks produce these side effects. This information is important as the detection mechanism designers may have to deal with unexpected results. Examples of such unexpected incidences are evasive SATHV, which try to hide their malicious activities by inserting normal operations during code execution. The inserted normal operations will shift the behavior of the system, but result in side effects with normal values. Knowing why an attack produces particular side effects should help the designer take into consideration possible ways an attacker might try to modify the side effects being monitored.

If our threat model is limited to a specific attack vector, monitoring multiple side effects of the attack could increase the probability of its detection. In contrast, if our threat model includes all possible attack vectors in the device to be protected, we must carefully select the side effect set to be monitored. To our benefit, several attack vectors induce the same side effects. In both cases, detection mechanism designers can use our list of side effects to construct a first monitoring set in their threat model. Finally, we present a table listing all the side effects for attack vectors which present some similarities in their side effects set. The table does not include side effects which are only visible for a single attack vector as we considered it obvious that attacks with unique side effects will need a dedicated detection mechanism. By presenting only attack vectors with some similarity in their side effects, the designer can potentially find an optimum monitoring set to detect them all. Hereafter, the terms indicated in parentheses correspond to the short terms entered in the tables.

3.1 Side effects obtained through PMU registers

The PMUs are special-purpose registers which allow us to gather hardware-specific events related to the CPU units. These units support CPU execution profiling. Detailed information on the events gathered for ARM and Intel architectures can be found in [9] and [25].

Attackers use these units as support during attacks. These features cannot simply be disabled as legitimate processes also use the PMU, for example to perform scheduling and timing of interruptions. Furthermore, designers use the PMUs for debugging, tuning, and to compare their applications. Multiple attack vectors, presented in the previous section, manipulate the PMUs to exploit the targeted vulnerabilities. For example, cache attacks [37, 57, 72] rely on timing differences to distinguish a cache miss from a cache hit. Attackers access the PMU cycle counters to extract timing information. The TSC Intel counter, or the *Cycle Counter Register* (CCNT) in ARM, measure the CPU cycles providing the timing information. The ARM PMU cache refill event[1] is used by the Prime+Count [22] to perform its attack.

For the benefit of a detection mechanism, these special-purpose registers can also be accessed to extract information relating to the PMU states. Initially, the detection mechanism can monitor access to the PMU and debug units. In ARM, the system can be configured to raise an interrupt whenever there is a read in the PMUs. In addition, a detection mechanism can monitor various PMU hardware events, such as events related to the cache, branch prediction, bus access, etc.

As some attacks manipulate the cache state, it may be relevant to monitor cache-relevant hardware events (e.g., cache misses, hits, accesses, etc.) to observe differences from the nominal behavior. A cache miss describes the situation when we search for, but do not find, the desired content in the cache layers. The cache content is written to the cache following transfer from the main memory. A cache hit describes the situation when the desired content is directly served from a cache layer. Cache access (or cache references) count any load or store operation or page table walk access, which performs a look-up in the cache. A cache load counts the number of loads to the cache, and is used more for the last-level cache. Cache miss events are a good indicator of cache manipulation. When an attacker evicts the targeted cache address, an increase in cache misses is observed. When the target application accesses these targeted lines, a cache miss will occur, and the requested information must be obtained from the main memory. Rowhammer and DRAMA require the cache to be bypassed, as they require direct access to the DRAM. To bypass the caches, Rowhammer flushes/evicts the cache address corresponding to the targeted physical address. The goal is to directly obtain the information from the DRAM and not from the copies stored in the caches. When Rowhammer accesses the evicted address once again, a cache miss occurs at all levels in the cache. Furthermore, in cache SCA the eviction will cause the targeted application to return misses from these locations, and the requested information must be obtained from the main memory. Attackers can thus time their accesses to observe which locations are accessed by the application targeted. As previously mentioned, during Flush+Reload, a miss will occur when the target application does not access this location. In contrast, a hit will indicate access by the victim. As a result, cache access events are also increased due to the intensive manipulation of the cache state. When an attacker uses eviction techniques instead of *flushing*, multiple loads must be performed to evict the targeted address. Table 1 provides a summary of the attacks that manipulate the state of the cache.

[1]The cache refill event counts how many cache lines have been updated.

To obtain valuable information, attacks must execute their malicious code multiple times, in a loop. An attacker must execute the same piece of code multiple times to reduce noise or calculate every possibility, such as all possible key values during AES encryption. Examples are loops which manipulate the cache state to bring it into a known state, or the loop continuously accessing the same DRAM location to induce a Rowhammer fault. These loops introduce CPU overheads, and an attack detection mechanism can therefore monitor PMU events to detect deviations from the nominal state. Two events are relevant to the observed overhead: the number of instructions executed, and the number of clock cycles. The number of instructions executed (Num_exec_instr)[2] counts the instructions executed inside the specified timing window from the counter activation. The number of clock cycles (Num_clk_cycles)[3] counts the number of CPU clock cycles completed during the same time window. If the attacker executes the malicious code in a loop, the number of instructions executed and clock cycles completed will increase.

In addition, side effects of execution of continuous loops are visible in the branch prediction unit, e.g., due to an increased number of branches taken, and a decreased number of branch mispredictions. Branch mispredictions decrease as the loop takes the same branch multiple times, although they may also increase as in the case of mistraining during execution of Spectre. The effects of using continuous loops can be observed through PMU events related to branch control. The retired branches taken (Ret_Br_taken) event counts when the last *micro-operations* (uOp) of a branch instruction retire[4]. Another event to monitor is the number of speculative retired branches taken (Spec_Ret_Br_Taken), which counts the number of speculative and retired macro conditional branches taken[5]. In addition, the event mispredicted branches (Br_Mis_Prdct) counts when the last uOp of a branch instruction retires, when the misprediction of the branch prediction hardware is corrected at execution time[6].

The bus is another part of the system that can be monitored to identify side effects. DMA attacks make heavy use of the bus. Hardware events that can be monitored through the CPU are available. The events (BUS_STATS)[7] count memory bus activity. Because Rowhammer and DRAMA both make extensive use of the bus during attack loops, we therefore believe similar side effects should be observed for these attacks.

For all the events noted above, a detector can be deployed in both Intel and ARM architectures. Events and registers may have different names, and consequently different platforms may also be present in the same architecture. For Intel, in the table, we use the event names from Intel Ivy Bridge, and for ARM events, we use names from ARM Cortex-A57.

3.2 Other side effects

Apart from the PMUs that provide us with direct access to hardware-related side effects, we can check other units, registers, or files for any modifications. Examples are OS files, the interrupt controller, and memory-mapped registers.

To perform an attack, the attacker must have knowledge of the system and they must bring it to a known state. Working with this information, they will try to eliminate noise and

[2]Event name *INST_RETIRED* for Intel and ARM.
[3]Event name *CPU_CLK_UNHALTED* for Intel and *CPU_CYCLES* for ARM.
[4]Event name *BR_INST_RETIRED* in Intel and *BR_RETIRED* in ARM.
[5]Event name *BR_INST_EXEC.TAKEN_CONDITIONAL* in Intel and *BR_PRED* in ARM.
[6]Event name *BR_INST_RETIRED.MISPRED* in Intel and as *BR_MIS_PRED* in ARM.
[7]Event name *BUS_TRANS_* in Intel and *MEM_ACCESS_* in ARM.

thus take better measurements. Attacks that manipulate the cache state or the main memory need information on the virtual to physical mapping. This information is necessary as the last level cache (shared memory in most systems) is usually physically indexed, but the processes use virtual addresses. Each process has its own virtual memory mapping. Attackers will therefore need to translate their virtual address to the physical address of the last-level cache. Using this information, they can evict the targeted physical addresses, which are the same between processes. For example, during a Rowhammer attack, the attackers must know which virtual address maps to the targeted DRAM's physical address. Repeatedly accessing the memory address can cause a fault to appear in the neighboring row. The side effect of using the Linux virtual file *proc/PID/pagemap*, which lets a user-space process determine which physical frame each virtual page maps to, can be observed. However, in most recent kernel versions, the pagemap is only accessible with root privileges. The side effects of using huge pages can also be observed, these are used as an alternative to reverse engineering the mapping. As mentioned above, when a huge page is used, e.g., a 2-MB page, the last 21 bits of the address are used as an offset. These 21 bits are the same for the virtual and physical addresses, and consequently only a small number of bits must be reversed-engineered. This method requires more effort from the attacker point of view [62]. To refer to these two methods providing the mappings, we use the event name (Access *proc/PID/pagemap*, use of transparent huge pages).

Many exploits target attack-specific events during execution of the victim code, e.g., to induce a fault in the last round of AES, or try to perform their exploit within a specific time window. For example, in Rowhammer to induce a fault, the same location must be accessed a certain number of times. The CPU optimizing the execution of processes may interfere with execution of the malicious code, resulting in noisy measurements for the attacker. An example of an attempt to avoid this consequence is the side effect of disabling the interrupt scheduler. The attacker disables the interrupt scheduler in the victim process as otherwise the interrupt scheduler will introduce noise into the measurements. This noise is introduced due to the uncertainty of the time required to execute the victim process when an interrupt is raised during the process. The time to serve the interrupt is unknown. For the example of AES, an interrupt is raised and the time required to serve it increases the difficulty of inducing the fault at the opportune moment.

Similar noise is induced due to the side effects of re-ordering instructions or memory accesses. The CPU uses instruction re-ordering to speed up execution of a process by allowing instructions to be executed out of the original program order by running ahead of sequential instruction code and exploiting existing instruction-level parallelism [69]. The CPU accomplishes instruction re-ordering using *re-order buffers* (ROBs). Re-ordering introduces noise because the attacker does not know the exact order in which the instructions execute or their timing. We refer to this event as (CPU instr re-order dsbl). In addition, the compiler re-orders instructions to speed up code execution. This introduces noise into the data measured by the malicious loop because the instruction stream might not be that expected. This side effect, which we refer to as (Compiler instr re-order dsbl) is not listed in our table because no detection mechanism can monitor it, and it is therefore of no use to the designer. However, we believe it is important that readers be aware of this side effect.

Another unit that induces noise by re-ordering is the memory controller. This component re-orders memory accesses to speed up access to the main memory and optimize memory system performance. If, for example, the memory controller re-orders accesses during a DRAMA attack, a faulty covert channel might be created. We refer to this event as (Mem cntrl access re-order). The memory controller can be monitored to extract side effects regarding

Rowhammer. To induce a fault in the DRAM using the Rowhammer attack, the attacker must access the same memory location multiple times. The event (Num_Mem_ADDR_)[8] counts the number of total memory accesses to the same cells in a time window chosen by the designer as a threshold for attack detection. This event assumes knowledge of the physical memory addresses accessed by the process.

Furthermore, attackers who wish to establish a covert channel must synchronize transmission and reception. Synchronization reduces the uncertainty of sampling outside the appropriate window. The TSC, as previously mentioned, corresponds to a register counting the number of CPU cycles since the last reset. This register provides us with the highest possible precision, it can be accessed using the *RDTSC* instruction in Intel. The TSC register is Intel-specific. Any other precise counter could be used as an alternative. For example, *SYSTick* counts down to zero and generates an interrupt with the aim of providing a fixed time interval between interrupts.

Side effects are also visible through OS events. During Meltdown, segmentation faults (SIGSEGV) can be monitored. A SIGSEGV occurs when a reference to a variable falls outside the segment where that variable resides or writes to a location in a read-only segment. However, the defender must take care as the *Transactional Synchronization Extensions* (TSX) extension and *Restricted Transactional Memory* (RTM) interface can be used to bypass segmentation faults. In addition, during the Rowhammer attack, the page fault miss rate is reduced. A page fault occurs when a running process accesses a memory page that is not currently mapped in the Memory Management Unit (MMU). A low page fault miss rate can be produced when a process runs in a loop repeatedly accessing the same addresses, as in the case of Rowhammer. In this case, the page resides in the page table, and any reference to it will constitute a hit.

To detect thermal covert channels and power-frequency fault attacks the nominal configuration of the *On-Chip Thermal Sensors* and Operating Performance Points *OPP* can be monitored. A detection mechanism can monitor access to on-chip thermal sensors. The attacker accesses the sensors to measure the temperature and establish the temperature covert channel, for example using *Intel Digital Thermal Sensors* and the *Coretemp* file in Linux. Consequently, monitoring the system temperature to detect abnormal temperature fingerprints may also help detect the attack. Moreover, attacks targeting the Power and frequency units such as [90] and [78] modify OS files and system registers dedicated to clock management or operating voltage. A detection mechanism may not accept configurations that differ from the ones in the *OPP* Linux file as pairs not specified in this file may not be nominal. As mentioned above, the *OPP* file stores frequency and voltage pairs supported by the device. Consequently, monitoring configurations of the memory-mapped registers of the *Power Management Unit* and *PLL* may help detect malicious changes.

Attacks exploiting the debug interface capabilities also leave traces. When a malicious code accesses the ETM to extract some information, a mechanism can be used to detect this access. As mentioned for the PMU access, the system can be configured to raise an interrupt and trap it to ensure EL3 is maintained (security level of the firmware/secure monitor [76]) during access to the ETM registers. We can also check accesses to other system registers if certain instructions[9] are used [66]. Attackers can use the *SMC* instruction to change from the non-secure to the secure world during debugging. Furthermore, the use of *SMC* exceptions can be monitored. The effects introduced are all extracted from ARM architecture.

[8] *Num_Mem_ADDR_1 >x OR (Num_Mem_ADDR_1 >x & Num_Mem_ADDR_2 >y)*

[9] Instructions (MRC, STC),(MRS) and (MCR, LCD),(MSR) to move and read register values in ARM

Finally, attackers who attempt to extract information from a protected memory (XOM) will need to execute the same context multiple times and raise multiple interrupts inside the XOM memory. As explained in subsection 2.4, the targeted instruction must be executed for different inputs and the different outputs extracted from the system registers. From the input-output combination, attackers can reverse-engineer the context of the location targeted. However, because attackers must halt the execution after each targeted instruction, they must raise an interrupt. Multiple execution and interruption patterns inside the protected memory may be used to raise an alarm.

3.3 Summary of attack vector side effects

From this presentation of all the side effects reported in the literature for the main attack vectors, we have produced a summary Table 2. In this table, we summarize attack vectors producing side effects with at least some overlap. As mentioned before, an attack vector producing unique side effects requires a specific detection mechanism monitoring a set of these effects if the attack is to be detected. In contrast, attack vectors which share some side effects allow designers to identify an optimum set of side effects to be monitored. This avoids the need to monitor all side effects and results in cheaper implementation. For this reason, we do not present the debug unit and thermal unit attack vectors, as their side effects are unique compared to the others. Our intention is to aid the reader to better compare the side effects of different types of attack. For the two attack vectors not listed in the table, the reader should refer to subsection 2.4 and subsection 2.6.

From Table 2, the reader has access to a list of the side effects in the third column. For each side effect, we have indicated which attack vectors produce it. Readers could use this information to choose side effects, depending on the types of attack they want to detect. Furthermore, readers can choose attacks to detect and determine the most appropriate side effects to monitor. For example, PMUs only allow a specific number of events to be monitored. The designers should choose among the side effects those that correlate the best with the chosen attacks, as some cover a broad range of attacks.

4 CLASSIFICATION OF SIDE EFFECTS

4.1 Side effect criteria

The goal of this section is to define some criteria by which to classify the side effects. The selection of side effects to be used in the implementation of a detection mechanism will depend on these criteria and the application monitored. For example, some side effects are deeply embedded in the system micro-architecture, and observation of these depends on the knowledge and privileges available to the designer. Designers are ultimately responsible for determining the parameters that best serve their design and the system's limitations. It is worth mentioning that these criteria are defined by us, but other criteria could be equally valid to classify side effects. The criteria we chose are as follows: *Hardware or Software side effect*, *Register or Interface based*, and *Usage in an existing detection mechanism*. We will start by describing each criterion and we subsequently present the taxonomy, Table 3 provides a summary of the classification.

Hardware or Software side effects. The nature of the side effect is determinant in the implementation of a detection mechanism. A software side effect may be more easily accessed from a software-implemented detection mechanism than through a hardware mechanism. This could be due to the difficulty in translating the software side effect in a form that a hardware mechanism could process to detect an attack. For example, segfaults are available

Table 2. List of side effects induced by different attacks

			Attacks						
			Memory				Trans exec		Power, freq, Thermal Unit
			Row-hammer	CSA	DRAMA	DMA	Meltdown	Spectre	Clkscrew Volt-jockey
		Side effects							
PMUs	Cache	Cache miss	X	X	X		X	X	X
		Cache hit	X	X	X				X
		Cache access	X	X	X			X	X
		LLC Load					X	X	
	CPU stats	Num_exec instr		X					X
		Num_clk cycles		X					X
	Branch cntrl	Ret_Br taken	X	X				X	
		Spec_Ret Br_Taken		X				X	
		Br_Mis Prdct	X					X	
	Bus stats	BUS_STATS	X		X	X			
		Use of PMUS	X	X	X		X	X	X
	OS	SIGSEGV					X		
		Low page fault miss rate	X						
	Virtual to physical mapping translation or reverse engineering	Access proc/PID/pagemap, use of transparent huge pages	X	X	X				X
	Interrupts	Interrupt disable	X	X	X				X
re-order		CPU instr re-order dsbl	X	X	X				X
	Mem_cntrl	Mem cntrl access re order	X		X				
		Num_Mem_ADDR_	X		X				
	Synchronization	Time Stamp Counter (TSC) for synchronization			X				

only as software events. On the other hand, a hardware side effect might be inaccessible through a software mechanism due to a lack of privileges or accessibility of software to the hardware itself.

Source. A criterion corresponding to the source of the side effect can be defined. In our case, the source may be a register or interface. A register is a special memory in the system, used to store information locally in the system. An interface is a layer used by several system modules to communicate with each other. A side effect could be a value stored in a register that needs to be checked to determine whether it retains the desired nominal value, or an interface event e.g., an interrupt raised that effects a change in the system. The observability of the side effect in the register or the interface is worth discussing. Some registers or interfaces are easily observed with a debugger, whereas others are hidden for non-authorized users.

Used in an existing detection mechanism. This criterion could benefit designers if the side effect is already used by some existing detection mechanism. For example, the side effect could be used in several detection mechanisms because it is simple to acquire and correlates strongly with an attack. On the other hand, a side effect may not have been used, because it is difficult to obtain, e.g., a software implementation may not have the required privileges to extract information from the hardware. Alternatively, the side effect may only correlate weakly with the attack, thus making other side effects more suitable. It should be noted that PMUs can only be used to count a certain number of events, even if the options of events to count are numerous. In addition, if a designer uses Neural Networks (NNs), the complexity of the system increases with the number of inputs used. These are some reasons why certain side effects may not have been used by designers.

Other. Other criteria that can be used to classify side effects could be the availability of the side effect depending on the platform. For example, as indicated, cache SCA detection mechanisms use performance counters. Cache load events are only available on some Intel architectures, and Intel SGX security implementation forbids access to these performance counters during enclave execution. These constraints have led security designers to investigate new ways of detecting attacks. In addition, some events provided by the ARM debug infrastructure may not be available on Intel platforms. The debug unit side effects used to date are all ARM-specific events. Intel provides other debugging events that were not included in this study. Another interesting fact is that some attacks are more efficient on certain platforms or architectures, e.g., cache SCA on Intel platforms due to the existence of the clflush instruction.

Moreover, the availability of the side effects could refer to the moment at which the attack side effect becomes available. The attack side effect could be obtained during the attack or once the attack has been completed. The availability of side effects during an attack determines whether we can prevent the attack in the first place, or whether we can simply prevent further exploitation by the attacker. This comes with the cost that the attackers were able to manipulate the system at least once before any intervention can be made to stop them. For example, segmentation faults are only available after the OS recognizes the violation. Furthermore, segmentation faults may be produced by benign applications. Because of this potential confusion, we may not be able to detect the attack immediately. The attacker may thus manipulate the system several times before a possible attack is identified. For side effects obtained from PMUs, the effects are available only if the specific counter is active. In addition, the availability of events depends on the frequency at which the counted values are checked. For example, if a software mechanism is implemented that regularly checks the PMU, it might be possible to detect the attack, but only at the cost of a high performance overhead. In contrast, if the PMU is checked less often, the attack might be missed. In this scenario, the designer must determine the best trade-off. Similar logic applies when register values must be checked, to detect possible use of non-nominal values.

Another interesting criterion is the efficiency of attack detection based on the information provided by side effects. Side effect efficiency in this context is not the same as detection accuracy. Rather, the attack could leave behind only some traces, which could be useful for evaluation. In addition, these traces might not correlate to a high degree with the attack. This situation will limit our capacity to detect the attack using current mechanisms. Furthermore, multiple side effects might be used to create a side effects pattern. Single value side effects typically require simpler implementations, setting thresholds for values between a set range. On the other hand, to monitor one or more side effect patterns requires

more complex implementations. The detection method used by the mechanisms might not rely on a single value, but rather on side effect waveforms throughout the attack. Complex techniques involving neural networks are now being used to detect side effects, such as in [21], [23], and [79]. For example, cache hits or cache accesses do not provide enough information to identify a malicious process, and instead, the designer must use a combination of cache side effects to come to a decision. Alternatively, if the designer knows the physical addresses accessed, the frequency of access to specific locations provides enough information to detect the Rowhammer or DRAMA attacks based on only one side effect. For the last two criteria, not enough non-trivial information could be extracted to consider including them in our classification table.

4.2 Taxonomy of side effects

Table 3 presents a summary of the side effects mentioned in this paper. The taxonomy table classifies these side effects based on the defined criteria. Readers can thus compare the side effects, find where they are employed, which attacks produce them, and finally choose those that best fit their mechanisms. Side effects are listed in the second column and the defined criteria in the second row. From this presentation, we can make some interesting observations. The side effects could be SW or HW. Depending on the nature of the side effect, as previously mentioned, a detection mechanism may not have access to it. For example, SIGSEGV is difficult to obtain from a HW implementation. In addition, as *SIGSEGV* is a side effect visible only after the attack, it is not an ideal candidate for monitoring. In contrast, register-accessible side effects might be obtained more quickly directly through the HW than through the SW.

In the table, we indicate some detection mechanisms relying on the side effects. These detection mechanisms will be presented in more detail in section 5.

5 ATTACK DETECTION MECHANISMS

A detection mechanism is an added layer of system protection. Its function is to detect malicious behavior in the system and notify the system so that corresponding actions are initiated. Detection relies on the observation of side effects left behind by an attack. As IoT and IIoT are becoming more frequent targets for attackers, today detection mechanisms can be considered a necessity. Some detection mechanisms are already integrated into these systems. In this section, we present detection mechanisms based on detection of events that are already implemented for the attack vectors listed in this survey. We also define some classification criteria, which we use to present the taxonomy presented in Table 4. We believe these criteria will help designers to find a model that fits their conceptual plan or to compare their implementation with solutions that have already been implemented. For example, the accuracy criterion can help designers to compare how accurately their model detects the attack or to choose the most accurate implemented model, and integrate it into their model.

5.1 Memory attacks

5.1.1 Rowhammer detection mechanisms.

ARMOR. ARMOR was presented in [31], it is a hardware detection mechanism designed as a run-time memory hot-row detector, that monitors the activation stream at the memory interface level. ARMOR can detect which rows are in danger of being hammered by using counters to count the number of activations per row. It does not require knowledge of the logical-to-physical mapping of the memory. The storage overhead depends on the size of the

Table 3. Classification of side effects

		Criteria				
	Side effects	HW/SW	Source	Detection Mechanism	Other	Attacks
Cache	Cache miss	HW	Reg	[21] (L3 misses), [23] (L3-L2-L1 misses), [55] (LLC miss)	Availability[1]	Rowhammer, CSA, Meltdown, Spectre, DRAMA
	Cache_hit	HW	Reg	[21] (L2 hits)	Availability[1]	Rowhammer, CSA, DRAMA
	Cache access	HW	Reg	[21] (L3 accesses), [55] (LLC references)	Availability[1]	Rowhammer, CSA, Spectre, DRAMA
	Cache Ld	HW	Reg	No	Available only in some Intel platforms	Spectre, Meltdown
Instr	Num_Exec Instr	HW	Reg	[21]	Availability[1]	CSA
	IPC	HW	Reg	[23]	Availability[1]	CSA
Branches	Spec_Ret Br_Taken	HW	Reg	[23]	Availability[1]	CSA, Spectre
	Ret_Br Taken	HW	Reg	[55]	Availability[1]	Spectre, Rowhammer
	Br_Mis Prdct	HW	Reg	[55]	Availability[1]	Spectre, Rowhammer
	SIGSEGV	SW	Kernel interface	[3]	Available after the fault	Meltdown
Bus	Num_Mem ADDR	HW	Interface	[31], [53], [84], [43]	Efficient to detect DRAM attacks	Rowhammer, DMA, DRAMA
	BUS_TRANS	HW	Reg	[86]	Efficient to detect DRAM attacks	DMA, SCA, Rowhammer, DRAMA
Units	SPM_PMIC	HW/SW	Reg	No	Availability[2]	Clkscrew, Voltjockey
	PLL	HW	Reg	No	Availability[2]	Clkscrew, Voltjockey
	OPP	SW	DVFS OS -file	No	Available immediately OS file	Clkscrew, Voltjockey
	DTS	HW/SW	Reg Interface	No	Availability[2]	Thermal monitor SCA
	Intrp dsbl	HW	Reg	No	Availability[2]	Rowhammer, CSA, DRAMA, Spectre, Meltdown, Clkscrew, Voltjockey

[1] Available to export only if selected.
[2] Depends on frequency with which the nominal register value is checked.

memory monitored and the nature of the attacks: one-sided Rowhammer or double-sided Rowhammer; the double-sided Rowhammer doubles the storage overhead. ARMOR provides 99.99% accuracy on the targeted addresses and the number of activations. It mitigates attacks by informing the memory controller about the targeted row, which triggers the Targeted Row Refresh (TRR) command to refresh the victim rows. In addition, it uses a buffer as a cache for the hammered rows to send them out of the memory. The performance overhead of ARMOR only depends on the number of hammered rows and is only caused by the need to refresh the two rows adjacent to the victim row.

Counter-Based Tree Structure. This solution, presented by [84], represents another approach using counters. Instead of using counters for each row, the proposed solution tries to minimize the number of counters used by assigning one counter to a group of rows in each bank. If the bank has N rows and M counters are used, when the counter reaches a threshold, the DRAM module performs $\frac{N}{M} + 2$ refreshes on the $\frac{N}{M}$ of the group plus the two adjacent rows. Because a significant number of rows must be refreshed when the threshold is reached, the performance overhead increases dramatically. To address this increase, the mechanism uses a counter-based tree, which classes the rows in groups of variable size depending on their access frequency. Thus, hot rows are mapped to smaller groups, and consequently victim rows can be more precisely identified.

Time Window Counter (TWiCe). TWiCe is a detection mechanism proposed by [53]. TWiCe is also a counter-based detection solution. Like Counter-Based Tree structures, TWiCe tries to minimize the number of counters used. It assigns a counter to a row only if the row is activated, and periodically invalidates the counters associated with rows that are not frequently activated. The number of counters used depends on the characteristics of the DRAM module; each bank has a counter table. TWiCe, rather than being implemented as a memory controller or DRAM extension, relies on Register Clock Drivers (RCD). The solution adds a new DRAM command ARR (Adjacent Row Refresh), which notifies the DRAM of the need to refresh the rows adjacent to the aggressor. The DRAM is involved because neither the RCDs nor the memory controller knows the logical-to-physical mapping.

5.1.2 Cache Side Channel.

Hardware Performance Counters. Hardware performance counters are the most common tool used to detect cache SCA. In [21], they were used to implement a detection mechanism that could be run in user space (HPC-Chiappetta). The perf-stat command in Linux runs and gathers performance counter statistics from the PMU. Due to the limited resolution of the perf-stat (the time interval between two consecutive samples is 100ms, which is much longer than the time necessary to complete some attacks), HPC-Chiappetta was developed as a new process with a higher resolution. This new process can run from the user space, but requires the same privileges as the process monitored. Three detection mechanisms were developed: The first is based on the correlation between the victim and the spy process, for which the Last Level Cache (LLC) accesses were shown to be a good correlation indicator. The second approach is based on anomaly detection, whereby malicious processes were considered normal and all others as anomalies. The third solution is based on Neural Networks (NNs), using the LLC accesses and misses, L2 cache hits, total execution cycles, and executed instructions to train the system. The proposed solution demonstrated high accuracy and fast detection time. Another machine learning approach presented in [23] uses the Instruction Per Cycle (IPC), L3 cache miss, L2 cache miss, L1 cache miss, and the speculative and retired branch counters to train the system. In this solution (HPC-Cho), the Intel PCM from the user-space was used to access the counters. The authors also modified the tool to enhance the resolution. The detection range for attacks depends on the counters selected.

MASCAT. All the above examples are run-time detection mechanisms. MASCAT [42] is an offline detection solution, which statically analyses binary elf files. In these files, MASCAT searches for characteristics exhibited by micro-architectural attacks. MASCAT can detect cache and DRAM attacks. For cache SCA, MASCAT looks for the use of high-resolution timers, memory barriers that serialize reads, and cache evictions. For DRAM SCA, it looks

for cache evictions, the use of fine-grained timers, and memory barriers. The offline tool uses these patterns to detect malicious binary code before it is loaded onto the device.

5.2 JTAG monitor

The debug interface is a critical part of system development. It gives developers a lot of freedom, allowing them to develop a fault-free system. However, attackers can also take advantage of this feature to explore the state of the system. In [79], the authors proposed a two-layer learning-based protection scheme to enhance the existing security of JTAG. In their implementation, layer 1 performs a basic check that verifies whether basic rules for JTAG operations are violated. A basic check should detect attacks that violate the basic rules of JTAG operation, such as illegal operation code (opcode). Layer 2 classifies the sequence of opcodes as normal or attacking, using a support vector machine (SVM) classifier. The authors use a hardware implementation because a JTAG attack should be detected in real-time, when a JTAG instruction is loaded.

5.3 DMA attacks

The authors of [86] implemented the Bus Agent Runtime Monitor (BARM), a DMA attack detection mechanism, as a Linux kernel module. For their implementation, they use an Intel architecture. The method is based on modeling the expected memory bus activity and comparing it to the actual activity. Any additional bus activity when accessing the platform's main memory is measured. This additional bus activity is the Achilles heel of DMA-based attacks, which BARM uses to reveal and stop the attack. The mechanism counts events called BUS_TRANS_MEM, which summarize all bursts (full cache line), partial reads/writes (non-burst), and invalid memory transactions. The other counters used by BARM are general-purpose counters that count certain BUS_TRANS_MEM events. The general approach is to count bus events caused by user space and kernel space processes with a single counter. All other processor bus transactions, from other bus system masters, can be distinguished by the extensions .THIS_AGENT and .ALL_AGENTS.

5.4 Thermal monitor

In [32] on-chip monitors are proposed as a means to identify attacks on systems. The authors use a dedicated hardware system to monitor for and prevent attacks. In their implementation, they use a processing monitor, which verifies the run time behavior and compares it to the expected behaviour determined by static analysis. In addition, they implement a thermal monitor, which monitors the temperature at various points in the physical core to detect abnormal patterns. Abnormal temperature patterns may be used in an attack to slow or halt the processor. In the implementation described, ring oscillator-based sensors are used, which provide a digital output and can be included in sensors for microprocessors. The ring oscillator is by its nature unstable and oscillates at a frequency determined by the delay across each inverter. This delay is also temperature-dependent. The resolution of the thermal monitor is two degrees per degree Celsius, with very good temperature linearity in the range close to operating temperature limits. The monitor presented uses a threshold to define abnormal temperatures. The threshold can be adapted to the running process to allow for processes that use more power and consequently dissipate more thermal energy. This adaptability eliminates the problem of designing a monitor that is too conservative or too optimistic, which would leave some paths open to attackers that would not be detected. Major vendors also use thermal monitors to monitor the system temperature as part of power and temperature management.

5.5 Transient execution attacks

Transient execution attacks, like Meltdown and Spectre, rely - as explained above - on flaws in CPU design. Patches to resolve the problem have been implemented, but due to the number of different variants cannot address all of these types of attack. To fully address the problem, new designs must be implemented. Detection mechanisms proposed for these attacks exploit information from side effects of the flaws and the side-channel analysis used to extract the information.

5.5.1 Meltdown and Spectre. During the Meltdown attack, the attacker process requests access to a memory location not belonging to it. Eventually, the request fails, causing the OS to raise a segmentation fault (*SIGSEGV*). The authors of [3] proposed MeltdownDetector (MeltDetector), implemented in software as a modified Linux or a dynamic instrumentation script using SystemTap. The monitor filters out benign $segfaults$. If $segfaults$, occurring at closely spaced memory addresses increase above a threshold, an alarm is raised and the IDs of the processes inducing the warnings are reported. The mechanism induces a flush of the caches if a $segfault$ is detected, even if the monitor raises an alarm. The monitor does this to prevent leakage of even a single byte of information. This mechanism introduces a performance overhead, because of the time required to flush the cache, and because benign $segfaults$ may occur. For example, some processes use $segfaults$ to speed up some types of operation. The designers argue that these are occasional or rare. Another limitation of the design is that if the attack produces a certain type of $segfault$, it might be able to bypass the monitor. The accuracy of the monitor, measured by computing the proportion of bytes that are successfully read/leaked, is less than 3%. Thus, using nothing but $segfaults$ to detect the attack is not efficient. For example, attackers could abuse the *Restricted Transactional Memory* (RTM) interface in Intel, if this is available, to suppress an exception event [28]. In addition, Spectre attacks do not raise a page fault, as this scenario is handled inside the CPU. To detect these attacks, the authors proposed to monitor LLC loads and misses or LLC references and misses. The corresponding monitor must be implemented in the OS to access these counters. The sampling frequency of the counters plays a vital role in the occurrence of false positives, and attackers could time their attack to bypass detection, by reading a small number of bytes and then sleeping for a period of time. In [54], a monitor for Spectre attacks is presented. This monitor uses the performance counters to gather the necessary information to train the machine learning classifier. The events monitored are retired branch instructions, branch mispredictions, LLC misses, and LLC references. It also calculates the branch miss rate and the LLC miss rate:

$$branch\_miss\_rate = branch\_mispredictions/branches$$

$$LLC\_miss\_rate = LLC\_misses/LLC\_references$$

The monitor (LI-Spectre) successfully detects Spectre attacks with high accuracy. In a more recent version [55], the authors also detected Rowhammer with the same implementation (LI-Rowhammer). We believe that this detection mechanism can detect three out of four Spectre variants because the side effects monitored concern mistraining of the branch predictor and the cache as a side-channel to extract the information. The only variant that might remain undetectable is Spectre variant 4 as the branch predictor plays no role in this attack. However, the cache is still used to extract the information.

5.6 Detection mechanism classification criteria

We have identified several criteria that can be used to compare and characterize detection mechanisms for the different attack vectors. It should be noted that this list of criteria is not exhaustive. Nevertheless, it includes a set of criteria that we consider to be the most important based on our study of the attack vectors and their detection mechanisms. The criteria presented are the following:

CPU or Hardware modification. This criterion applies primarily to detection mechanisms implemented in hardware. A detection mechanism must inevitably modify the system when it is added. The question here is whether the detection mechanism needs to modify the CPU or *Instruction Set Architecture* (ISA), or a hardware module of the system to obtain information or to be added as an extra feature. For example, a detection mechanism could be added as an independent module in the system, connected to the main bus, or it could be part of the CPU or perhaps another controller. Some mechanisms also require the OS to be modified.

Privilege level. To implement detection mechanisms, the privilege level (user, OS, or system) accorded to the defender is important. A user-level mechanism may not have access to higher-level privilege mechanisms, for example, PMUs are only accessible from the OS level, or it must rely on more noisy results, provided by the operating system to the user for safety reasons. A system-level mechanism has access to the hardware itself, increasing the number of resources and the speed with which the necessary information can be obtained.

Detection Accuracy. Detection accuracy should be considered as a primary criterion for comparing detection mechanisms or identifying a mechanism as useful or not. Detection accuracy can be divided into True Positives (T_P) and True Negatives (T_N). True positives are observations where the actual and predicted attacks were true. True negatives are observations where the actual and predicted normal applications were true. Inaccuracy is also divided into False Positives (F_P) and False Negatives (F_N); in this case, False positives or False alarms are observations where normal applications are reported as attacks. False negatives are observations where actual attacks were present but were reported as normal applications. A common metric used to assess detection accuracy is sensitivity. Sensitivity is defined as:

$$Sensitivity = \frac{T_P}{T_P + F_N}$$

Another metric is the F-score, which measures the global trade off between precision and sensitivity, and is defined as:

$$F_{score} = \frac{2 * (precision * sensitivity)}{precision + sensitivity}$$

where precision (the proportion of positive observations that are truly positive) is the number of True positives divided by the number of True positives and False positives:

$$precision = \frac{T_P}{T_P + F_P}$$

This metric is used as some mechanisms are very good at detecting an attack, but are overly sensitive, leading to a high number of False alarms.

Detection Overhead. Detection overhead is another metric we considered. The mechanisms used for detection necessarily incur some overhead in the system. We primarily consider the performance overhead, and more specifically the runtime detection overhead. The runtime detection overhead could be due to slowing down of the process that we want to secure due to the mechanism running in order or in parallel. This overhead will depend on how much the detection mechanism interferes with the running process, to make a decision based on the information obtained.

Storage - Area Overhead. No implementation of a detection mechanism comes for free for the system. Thus, any detection mechanism must be implemented as a dedicated hardware module. Consequently, there will be memory requirements and space required to build it, both of which add to the cost of the system. In addition, software implementation requires extra memory to be made available. In limited memory environments, like IoT and IIoT applications, limiting this overhead is crucial.

Detection Speed. The speed at which the detection mechanism detects an attack is also another essential criterion for evaluating any detection mechanism. A mechanism's detection speed depends on the attack vector. Some attacks need to perform a small number of steps before the attack succeeds, while others must perform complex executions before even starting the actual attack. For example, the Rowhammer attack requires multiple preparatory steps before the attacker can perform an attack with a useful impact. In contrast, a fault attack delivers a more direct and rapid impact on the system. A detection mechanism could detect the attack after the leakage or fault, or before the attacker can retrieve any information or induce a negative effect on system results. It is, naturally, preferable that detection mechanisms detect attack vectors before they produce useful results for the attackers.

Other. Another criterion that could be used to compare the detection mechanism is online or offline detection. A detection mechanism could be implemented offline to detect malicious code before it is loaded by the system. This could be a first defence against attacks. Offline detection relies on known attack patterns, thus it cannot detect new or modified attack vectors. In contrast, online detection uses real-time and offline information to detect an attack. Based on the online information obtained, an online detection mechanism can detect modified attack vectors. MASCAT is the only offline detection mechanism presented in this survey, which explains why this criterion is not included in our summary table.

5.6.1 Table presentation. We present all the information from this section in Table 4, which classifies the detection mechanisms according to the criteria defined above. The following remarks are necessary before presenting the classification table. N\A ("Not Applicable") was used when we were unable to find the value for a criterion in the literature, or when the criterion does not apply to the particular detection mechanism. In the accuracy criterion, the sensitivity metric was used for most of the detection mechanisms, except that presented in [21], for which the F_{score} was used. Readers could use the information presented in this table to choose from among the detection mechanisms presented those that best fit their design criteria. In addition, designers could use it to compare the detection mechanisms implemented in their systems to the mechanisms presented, in accordance with our defined criteria.

6 CONCLUSION

In this paper, we performed a survey on a specific class of malicious attack vectors targeting IoT and IIoT devices. These attacks target both the computer micro-architecture hardware

Table 4. Taxonomy table of the detection mechanisms

Mechanism	Criteria					
	Modification	Privilege	Accuracy	Detection Overhead	Speed	Storage Area
[31] ARMOR	HW-Memory controller	System	99.99%	less than 0.09%	Before faults	800Bytes-1.6KB for 4GB of RAM
[84] Counter-Based Tree Structure	HW-DRAM	System	N/A	N/A	Before faults	Number of counters
[53] TWiCe	HW-DRAM	System	N/A	less than 0.3% [2]	Before faults	2.71KB per 1GB DRAM bank
[21] HPC-Chiappetta	No	OS/User	F-score [1]	less than 2.3%	less than 2.6ms [3]	N/A
[23] HPC-Cho	No	OS/User	more than 91.9%	less than 1.1%	less than 2.4s [4]	N/A
[42] MASCAT	SW	User	100%, less than 4% F_P	No overhead	N/A	No overhead
[79] JTAG monitor	HW	System	more than 87.8% for SVM	No overhead	520 clk cycles [5]	1.79% of chip area
[3] MeltdownDetector	OS	OS	more than 99%	OS less than 0.97%	N/A	N/A
	SystemTap - Linux			SystemTap less than 0.38%		
[54] LI-Spectre	SW	OS/User	100%, 0.77% F_P	N/A	N/A	N/A
[55] LI-Rowhammer	SW	OS/User	100%, 0% F_P	N/A	N/A	N/A
[86] BARM	SW	OS	N/A	3.50%	N/A	N/A

[1] F-score=0.509 for AES (Anomaly detection), F-score=0.932 for Neural network (AES), F-score=1 for Anomaly detection and Neural network (ECDSA).
[2] The overhead is due to additional row activations.
[3] Time before completion of the attack in a same-OS scenario.
[4] Time needed to detect an attack, when the detection limit is set to 5s for safe operation.
[5] CPU clock cycles needed to make a per-instruction prediction.

vulnerabilities and side-channel leakages. The attacks presented showcased the use of software code to exploit hardware vulnerabilities in the target systems and allow the attacker to extract sensitive information, implant malicious code, or gain access to privileged code. We referred to these attacks as Software Attacks Targeting Hardware Vulnerabilities (SATHV). Even the debug unit attack presented in this survey can be performed remotely. As previously said, this is possible due to the new debug capabilities introduced in recent ARM architectures, which no more require physical access to the device. We did not consider attacks requiring physical access and assumed the existence of physical protections or barriers making physical access to the devices difficult, due to environmental limitations. We focused on recently

exploited attacks and presented the side effects left behind during the attack. In addition to these attacks, we presented an overview of existing detection mechanisms. In section 3 and section 4 we discussed the side effects, and later in section 5 we listed and explained several mechanisms that could be used to detect this type of attack.

Furthermore, we proposed a number of classification criteria based on side effects which should allow designers to select the most appropriate observations for their application. For each attack, we presented only the side effects reported in the literature or exploited by detection mechanisms, other side effects may also exist. In the related works, detection mechanism designers used the side effects presented to detect potential attacks due to the high correlation of these side effects with the attack vector to be protected against. However, this side-effect selection correlates best in their model of implementation. This does not necessarily mean that another set of side effects will not be suitable or more appropriate in different implementations of the same mechanism. Another reason governing selection could be the ease of access to the side effects, allowing a less complicated implementation of detection mechanisms.

We believe that IoT and IIoT Systems require more robust security solutions. The criteria defined here should help designers to compare a wide variety of relevant aspects when designing a new detection mechanism. Most of the solutions presented were developed to deal with a specific attack vector. The detection mechanisms presented only consider a limited subset of attacks. No overall solution is available which would simultaneously consider multiple attack vectors and vulnerabilities. However, designers do not know in advance which attack vectors will be used, and must therefore implement and concurrently optimize multiple detection mechanisms. The information presented here can support designers in their selection of a first set of side effects to be monitored and provide a first insight into previously implemented detection mechanisms that should help them achieve the best attack detection possible.

Finally, we present some perspectives for future work. In this survey, we used data extracted from published scientific papers to generate a collection of side effects due to SATHV execution. The platforms and architectures were often different between studies. However, we know that the platform used to evaluate a system plays a major role as the side effects will change depending on the system configuration, for example, cache size, cache replacement policy, type of branch predictor, etc. A better approach would be to implement all the attacks and detection mechanisms on the same platform. This would allow us to measure all the side effects induced and make a more precise classification. This experimentation will create a database of platform-specific side effects. A more general experimental platform is necessary to create a database of SATHV side effects. The same considerations apply to the detection mechanisms. We are also interested in studying whether the implementation of detection mechanisms in the platform affects system behavior with respect to side effects. Additionally, we believe, that implementing different mechanisms in a single platform is not efficient. Attacks are constantly increasing, and as previously indicated, a global and re-configurable/upgradable solution is required. In addition, after analyzing the efficiency of an overall supervision solution monitoring the software-related side effects exploiting hardware vulnerabilities, we will add some physical attacks to our threat model.

ACKNOWLEDGMENTS

This work benefitted from funding through the French government's IRT Nanoelec program, reference ANR-10-AIRT-05.

REFERENCES

[1] Eltayeb Salih Abuelyaman and Balasubramanian Devadoss. 2005. Differential Fault Analysis. In *Proceedings of The 2005 International Conference on Internet Computing, ICOMP 2005, Las Vegas, Nevada, USA, June 27-30, 2005*, Hamid R. Arabnia and Rose Joshua (Eds.). CSREA Press, 535.

[2] Ayaz Akram, Maria Mushtaq, Muhammad Khurram Bhatti, Vianney Lapotre, and Guy Gogniat. 2020. Meet the Sherlock Holmes' of Side Channel Leakage: A Survey of Cache SCA Detection Techniques. *IEEE Access* 8 (2020), 70836–70860. https://doi.org/10.1109/ACCESS.2020.2980522

[3] Taha Atahan Akyildiz, Can Berk Guzgeren, Cemal Yilmaz, and Erkay Savas. 2019. MeltdownDetector: A Runtime Approach for Detecting Meltdown Attacks. *IACR Cryptol. ePrint Arch.* 2019 (2019), 613. https://eprint.iacr.org/2019/613

[4] ARM. 2009. ARM1176JZF-S technical reference manual (Revision H). https://developer.arm.com/documentation/ddi0301/h. Accessed on 26.6.2020.

[5] ARM. 2013. CoreSight Technical Introduction (Version 1.0). https://developer.arm.com/documentation/epm039795/latest. Accessed on 26.6.2020.

[6] ARM. 2016. ARMv8-M Processor Debug (Version 1.0). https://developer.arm.com/documentation/100734/0100/. Accessed on 26.6.2020.

[7] ARM. 2017. Arm Compiler User Guide (Version 6.9). https://developer.arm.com/documentation/100748/0609. Accessed on 26.6.2020.

[8] ARM. 2017. ARM Cortex-R52 Processor Technical Reference Manual (Version 1.0). https://developer.arm.com/documentation/100026/0100. Accessed on 26.6.2020.

[9] ARM. 2018. Arm Cortex-A76AE Core Technical Reference Manual (Version 0.1). https://developer.arm.com/documentation/101392/0000/. Accessed on 26.6.2020.

[10] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. 2006. The Sorcerer's Apprentice Guide to Fault Attacks. *Proc. IEEE* 94, 2 (2006), 370–382. https://doi.org/10.1109/JPROC.2005.862424

[11] Noemie Beringuier-Boher, Kamil Gomina, David Hely, Jean-Baptiste Rigaud, Vincent Beroulle, Assia Tria, Joel Damiens, Philippe Gendrier, and Philippe Candelier. 2014. Voltage glitch attacks on mixed-signal systems. In *2014 17th Euromicro Conference on Digital System Design*. IEEE, 379–386.

[12] Vincent Beroulle, Philippe Candelier, Stephan De Castro, Giorgio Di Natale, Jean-Max Dutertre, Marie-Lise Flottes, David Hély, Guillaume Hubert, Régis Leveugle, Feng Lu, Paolo Maistri, Athanasios Papadimitriou, Bruno Rouzeyre, Clément Tavernier, and Pierre Vanhauwaert. 2014. Laser-Induced Fault Effects in Security-Dedicated Circuits. In *VLSI-SoC: Internet of Things Foundations - 22nd IFIP WG 10.5/IEEE International Conference on Very Large Scale Integration, VLSI-SoC 2014, Playa del Carmen, Mexico, October 6-8, 2014, Revised and Extended Selected Papers (IFIP Advances in Information and Communication Technology, Vol. 464)*, Luc Claesen, María Teresa Sanz-Pascual, Ricardo Reis, and Arturo Sarmiento-Reyes (Eds.). Springer, 220–240. https://doi.org/10.1007/978-3-319-25279-7_12

[13] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*, William Enck and Collin Mulliner (Eds.). USENIX Association. https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser

[14] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[15] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2019. Breaking Virtual Memory Protection and the SGX Ecosystem with Foreshadow. *IEEE Micro* 39, 3 (2019), 66–74. https://doi.org/10.1109/MM.2019.2910104

[16] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[17] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[18] Gaetan Canivet, Paolo Maistri, Régis Leveugle, Jessy Clédière, Florent Valette, and Marc Renaudin. 2011. Glitch and Laser Fault Attacks onto a Secure AES Implementation on a SRAM-Based FPGA. *J. Cryptol.* 24, 2 (2011), 247–268. https://doi.org/10.1007/s00145-010-9083-9

[19] Pierre Carru. 2017. Attack trustzone with rowhammer. https://grehack.fr/2017/program.

[20] Thomas M. Chen and Saeed Abu-Nimeh. 2011. Lessons from Stuxnet. *Computer* 44, 4 (2011), 91–93. https://doi.org/10.1109/MC.2011.115

[21] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. 2016. Real time detection of cache-based side-channel attacks using hardware performance counters. *Appl. Soft Comput.* 49 (2016), 1162–1174. https://doi.org/10.1016/j.asoc.2016.09.014

[22] Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. 2018. Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 441–452. https://doi.org/10.1145/3274694.3274704

[23] JongHyeon Cho, TaeHyun Kim, TaeHun Kim, and Youngjoo Shin. 2019. Real-Time Detection on Cache Side Channel Attacks using Performance Counter Monitor. In *2019 International Conference on Information and Communication Technology Convergence, ICTC 2019, Jeju Island, Korea (South), October 16-18, 2019*. IEEE, 175–177. https://doi.org/10.1109/ICTC46691.2019.8939797

[24] Jean-Michel Cioranesco, Jean-Luc Danger, Tarik Graba, Sylvain Guilley, Yves Mathieu, David Naccache, and Xuan Thuy Ngo. 2014. Cryptographically secure shields. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2014, Arlington, VA, USA, May 6-7, 2014*. IEEE Computer Society, 25–31. https://doi.org/10.1109/HST.2014.6855563

[25] Intel Corporation. 2017. *Intel® 64 and IA32 Architectures Performance Monitoring Events*. Intel Corporation.

[26] Intel Corporation. 2020. Q2 2018 Speculative Execution Side Channel Update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html. (2020).

[27] Alan Ehret, Karen Gettings, Bruce R. Jordan, and Michel A. Kinsy. 2019. A Survey on Hardware Security Techniques Targeting Low-Power SoC Designs. In *2019 IEEE High Performance Extreme Computing Conference, HPEC 2019, Waltham, MA, USA, September 24-26, 2019*. IEEE, 1–8. https://doi.org/10.1109/HPEC.2019.8916486

[28] David Fiser and William Gamazo Sanchez. 2018. Detecting attacks that exploit meltdown and spectre with performance counters. TrendMicro - https://www.trendmicro.com/en_us/research/18/c/detecting-attacks-that-exploit-meltdown-and-spectre-with-performance-counters.html.

[29] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 195–210. https://doi.org/10.1109/SP.2018.00022

[30] Ulf Frisk. 2016. pcileech. https://github.com/ufrisk/pcileech accessed 26.6.2020.

[31] Mohsen Ghasempour, Mikel Lujan, and Jim Garside. 2015. ARMOR: A Run-time Memory Hot-Row Detector.(2015).

[32] Guy Gogniat, Tilman Wolf, and Wayne P. Burleson. 2006. Reconfigurable Security Support for Embedded Systems. In *39th Hawaii International International Conference on Systems Science (HICSS-39 2006), CD-ROM / Abstracts Proceedings, 4-7 January 2006, Kauai, HI, USA*. IEEE Computer Society. https://doi.org/10.1109/HICSS.2006.409

[33] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 955–972. https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[34] Daniel Gruss, David Bidner, and Stefan Mangard. 2015. Practical Memory Deduplication Attacks in Sandboxed Javascript. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9326)*, Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl (Eds.).

Springer, 108–122. https://doi.org/10.1007/978-3-319-24174-6_6

[35] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2018. Another Flip in the Wall of Rowhammer Defenses. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 245–261. https://doi.org/10.1109/SP.2018.00031

[36] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 368–379. https://doi.org/10.1145/2976749.2978356

[37] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9721)*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.). Springer, 279–299. https://doi.org/10.1007/978-3-319-40667-1_14

[38] Daniel Gruss, Michael Schwarz, and Moritz Tipp. 2020. https://www.youtube.com/watch?v=UTSJf05pw-0

[39] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 897–912. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss

[40] Naofumi Homma, Yu-ichi Hayashi, Noriyuki Miura, Daisuke Fujimoto, Daichi Tanaka, Makoto Nagata, and Takafumi Aoki. 2014. EM Attack Is Non-invasive? - Design Methodology and Validity Verification of EM Attack Sensor. In *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8731)*, Lejla Batina and Matthew Robshaw (Eds.). Springer, 1–16. https://doi.org/10.1007/978-3-662-44709-3_1

[41] Jann Horn. 2018. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[42] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2018. MASCAT: Preventing Microarchitectural Attacks Before Distribution. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY 2018, Tempe, AZ, USA, March 19-21, 2018*, Ziming Zhao, Gail-Joon Ahn, Ram Krishnan, and Gabriel Ghinita (Eds.). ACM, 377–388. https://doi.org/10.1145/3176258.3176316

[43] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 361–372. https://doi.org/10.1109/ISCA.2014.6853210

[44] Vladimir Kiriansky and Carl A. Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *CoRR* abs/1807.03757 (2018). arXiv:1807.03757 http://arxiv.org/abs/1807.03757

[45] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *CoRR* abs/1801.01203 (2018). arXiv:1801.01203 http://arxiv.org/abs/1801.01203

[46] 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. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1–19. https://doi.org/10.1109/SP.2019.00002

[47] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey M. Voas. 2017. DDoS in the IoT: Mirai and Other Botnets. *Computer* 50, 7 (2017), 80–84. https://doi.org/10.1109/MC.2017.201

[48] Thomas Korak, Michael Hutter, Baris Ege, and Lejla Batina. 2014. Clock Glitch Attacks in the Presence of Heating. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014, Busan, South Korea, September 23, 2014*, Assia Tria and Dooho Choi (Eds.). IEEE Computer Society, 104–114. https://doi.org/10.1109/FDTC.2014.20

[49] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*, Christian Rossow and

Yves Younan (Eds.). USENIX Association. https://www.usenix.org/conference/woot18/presentation/koruyeh

[50] Deepa Krishnan and Adesh Mallya. 2020. A Survey on Security Attacks in Internet of Things and Challenges in Existing Countermeasures. In *Proceedings of International Conference on Wireless Communication*. Springer, 463–469.

[51] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. 2020. Rambleed: Reading bits in memory without accessing them. In *41st IEEE Symposium on Security and Privacy (S&P)*.

[52] Nica Latto. 2020. What Are Meltdown and Spectre? Avast Academy - Security - Other threats. https://www.avast.com/c-meltdown-spectre Acessed 7 December 2020.

[53] Eojin Lee, Ingab Kang, Sukhan Lee, G. Edward Suh, and Jung Ho Ahn. 2019. TWiCe: preventing row-hammering by exploiting time window counters. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman (Eds.). ACM, 385–396. https://doi.org/10.1145/3307650.3322232

[54] Congmiao Li and Jean-Luc Gaudiot. 2018. Online Detection of Spectre Attacks Using Microarchitectural Traces from Performance Counters. In *30th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2018, Lyon, France, September 24-27, 2018*. IEEE, 25–28. https://doi.org/10.1109/CAHPC.2018.8645918

[55] Congmiao Li and Jean-Luc Gaudiot. 2019. Detecting Malicious Attacks Exploiting Hardware Vulnerabilities Using Performance Counters. In *43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, Milwaukee, WI, USA, July 15-19, 2019, Volume 1*, Vladimir Getov, Jean-Luc Gaudiot, Nariyoshi Yamai, Stelvio Cimato, J. Morris Chang, Yuuichi Teranishi, Ji-Jiang Yang, Hong Va Leong, Hossain Shahriar, Michiharu Takemoto, Dave Towey, Hiroki Takakura, Atilla Elçi, Susumu Takeuchi, and Satish Puri (Eds.). IEEE, 588–597. https://doi.org/10.1109/COMPSAC.2019.00090

[56] 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. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[57] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 605–622. https://doi.org/10.1109/SP.2015.43

[58] Vaibhav G. Lokhande and Deepti Vidyarthi. 2019. A study of hardware architecture based attacks to bypass operating system security. *Secur. Priv.* 2, 4 (2019). https://doi.org/10.1002/spy2.81

[59] Fabien Majéric, Benoit Gonzalvo, and Lilian Bossuet. 2018. JTAG Fault Injection Attack. *IEEE Embed. Syst. Lett.* 10, 3 (2018), 65–68. https://doi.org/10.1109/LES.2017.2771206

[60] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. 2019. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/thunderclap-exploring-vulnerabilities-in-operating-system-iommu-protection-via-dma-from-untrustworthy-peripherals/

[61] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. 2015. Thermal Covert Channels on Multi-core Platforms. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 865–880. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/masti

[62] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9404)*, Herbert Bos, Fabian Monrose, and Gregory Blanc (Eds.). Springer, 48–65. https://doi.org/10.1007/978-3-319-26362-5_3

[63] Benoît Morgan, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. 2016. Bypassing IOMMU Protection against I/O Attacks. In *2016 Seventh Latin-American Symposium on Dependable Computing, LADC 2016, Cali, Colombia, October 19-21, 2016*. IEEE Computer Society, 145–150. https://doi.org/10.1109/LADC.2016.31

[64] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. 2016. TrustZone Explained: Architectural Features and Use Cases. In *2nd IEEE International Conference on Collaboration and Internet Computing, CIC 2016, Pittsburgh, PA, USA, November 1-3, 2016*. IEEE Computer Society, 445–451. https://doi.org/10.1109/CIC.2016.065

[65] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards Transparent Tracing and Debugging on ARM. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 33–49. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ning

[66] Zhenyu Ning and Fengwei Zhang. 2019. Hardware-Assisted Transparent Tracing and Debugging on ARM. *IEEE Trans. Inf. Forensics Secur.* 14, 6 (2019), 1595–1609. https://doi.org/10.1109/TIFS.2018.2883027

[67] Zhenyu Ning and Fengwei Zhang. 2019. Understanding the Security of ARM Debugging Features. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 602–619. https://doi.org/10.1109/SP.2019.00061

[68] Graz University of Technology. 2018. Meltdown and Spectre Vulnerabilities in modern computers leak passwords and sensitive data. website page. https://meltdownattack.com/ Found in Questions and Answers - Can my antivirus detect or block this attack.

[69] José R. García Ordaz, Marco Antonio Ramírez Salinas, Luis A. Villa Vargas, Herón Molina Lozano, and Cuauhtémoc Peredo Macías. 2012. A Reorder Buffer Design for High Performance Processors. *Computación y Sistemas* 16, 1 (2012). http://cys.cic.ipn.mx/ojs/index.php/CyS/article/view/1369

[70] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 1406–1418. https://doi.org/10.1145/2810103.2813708

[71] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 1406–1418. https://doi.org/10.1145/2810103.2813708

[72] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3860)*, David Pointcheval (Ed.). Springer, 1–20. https://doi.org/10.1007/11605805_1

[73] Mathias Payer. 2016. HexPADS: A Platform to Detect "Stealth" Attacks. In *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9639)*, Juan Caballero, Eric Bodden, and Elias Athanasopoulos (Eds.). Springer, 138–154. https://doi.org/10.1007/978-3-319-30806-7_9

[74] Colin Percival. 2005. Cache missing for fun and profit. BSDCan.

[75] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 565–581. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl

[76] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.* 51, 6 (2019), 130:1–130:36. https://doi.org/10.1145/3291047

[77] Rui Qiao and Mark Seaborn. 2016. A new approach for rowhammer attacks. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2016, McLean, VA, USA, May 3-5, 2016*, William H. Robinson, Swarup Bhunia, and Ryan Kastner (Eds.). IEEE Computer Society, 161–166. https://doi.org/10.1109/HST.2016.7495576

[78] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. 2019. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 195–209. https://doi.org/10.1145/3319535.3354201

[79] Xuanle Ren, Ronald D. Blanton, and Vítor Grade Tavares. 2016. A Learning-Based Approach to Secure JTAG Against Unseen Scan-Based Attacks. In *IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2016, Pittsburgh, PA, USA, July 11-13, 2016*. IEEE Computer Society, 541–546.

https://doi.org/10.1109/ISVLSI.2016.107

[80] Kurt Rosenfeld and Ramesh Karri. 2010. Attacks and Defenses for JTAG. *IEEE Des. Test Comput.* 27, 1 (2010), 36–47. https://doi.org/10.1109/MDT.2010.9

[81] Nahi Jnanena Sadrusham. 2015. Timing Constraints. http://asic-soc.blogspot.com/2015/02/timing-constraints.html. Accessed on 26.6.2020.

[82] Marc Schink and Johannes Obermaier. 2019. Taking a Look into Execute-Only Memory. In *13th USENIX Workshop on Offensive Technologies, WOOT 2019, Santa Clara, CA, USA, August 12-13, 2019*, Alex Gantman and Clémentine Maurice (Eds.). USENIX Association. https://www.usenix.org/conference/woot19/presentation/schink

[83] Jayasree Sengupta, Sushmita Ruj, and Sipra Das Bit. 2020. A Comprehensive Survey on Attacks, Security Issues and Blockchain Solutions for IoT and IIoT. *J. Netw. Comput. Appl.* 149 (2020). https://doi.org/10.1016/j.jnca.2019.102481

[84] Seyed Mohammad Seyedzadeh, Alex K. Jones, and Rami G. Melhem. 2017. Counter-Based Tree Structure for Row Hammering Mitigation in DRAM. *IEEE Comput. Archit. Lett.* 16, 1 (2017), 18–21. https://doi.org/10.1109/LCA.2016.2614497

[85] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *CoRR* abs/1806.07480 (2018). arXiv:1806.07480 http://arxiv.org/abs/1806.07480

[86] Patrick Stewin. 2013. A Primitive for Revealing Stealthy Peripheral-Based Attacks on the Computing Platform's Main Memory. In *Research in Attacks, Intrusions, and Defenses - 16th International Symposium, RAID 2013, Rodney Bay, St. Lucia, October 23-25, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8145)*, Salvatore J. Stolfo, Angelos Stavrou, and Charles V. Wright (Eds.). Springer, 1–20. https://doi.org/10.1007/978-3-642-41284-4_1

[87] Patrick Stewin and Iurii Bystrov. 2012. Understanding DMA Malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece, July 26-27, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7591)*, Ulrich Flegel, Evangelos P. Markatos, and William K. Robertson (Eds.). Springer, 21–41. https://doi.org/10.1007/978-3-642-37300-8_2

[88] Jakub Szefer. 2019. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *J. Hardware and Systems Security* 3, 3 (2019), 219–234. https://doi.org/10.1007/s41635-018-0046-1

[89] Shahin Tajik, Heiko Lohrke, Fatemeh Ganji, Jean-Pierre Seifert, and Christian Boit. 2015. Laser Fault Attack on Physically Unclonable Functions. In *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2015, Saint Malo, France, September 13, 2015*, Naofumi Homma and Victor Lomné (Eds.). IEEE Computer Society, 85–96. https://doi.org/10.1109/FDTC.2015.19

[90] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 1057–1074. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang

[91] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1675–1689. https://doi.org/10.1145/2976749.2978406

[92] Thomas Willhalm, Roman Dementiev, and Patrick Fay. 2012. Intel performance counter monitor-a better way to measure cpu utilization. *Dosegljivo: https://software. intel. com/en-us/articles/intelperformance-countermonitor-a-better-way-to-measure-cpu-utilization.[Dostopano: September 2014]* (2012).

[93] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[94] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. 2016. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. *IACR Cryptol. ePrint Arch.* 2016 (2016), 980. http://eprint.iacr.org/2016/980

[95] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. 2017. Understanding The Security of Discrete GPUs. In *Proceedings of the General Purpose GPUs, GPGPU@PPoPP, Austin, TX, USA, February 4-8, 2017*. ACM, 1–11. https://doi.org/10.1145/3038228.

3038233