Delft University of Technology

Survey on Architectural Attacks
A Unified Classification and Attack Model

Ghasempouri, Tara; Raik, Jaan; Reinbrecht, Cezar; Hamdioui, Said; Taouil, Mottaqiallah

DOI
10.1145/3604803

Publication date
2023

Published in
ACM Computing Surveys

Citation (APA)
Ghasempouri, T., Raik, J., Reinbrecht, C., Hamdioui, S., & Taouil, M. (2023). Survey on Architectural Attacks: A Unified Classification and Attack Model. *ACM Computing Surveys*, *56*(2), Article 42. https://doi.org/10.1145/3604803

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Green Open Access added to TU Delft Institutional Repository

'You share, we take care!' - Taverne project

https://www.openaccess.nl/en/you-share-we-take-care

Otherwise as indicated in the copyright section: the publisher is the copyright holder of this work and the author uses the Dutch legislation to make this work public.

Survey on Architectural Attacks: A Unified Classification and Attack Model

TARA GHASEMPOURI and JAAN RAIK, Tallinn University of Technology, Estonia
CEZAR REINBRECHT, SAID HAMDIOUI, and MOTTAQIALLAH TAOUIL, Delft University of Technology, The Netherlands

According to the World Economic Forum, cyberattacks are considered as one of the most important sources of risk to companies and institutions worldwide. Attacks can target the network, software, and/or hardware. Over the years, much knowledge has been developed to understand and mitigate cyberattacks. However, new threats have appeared in recent years regarding software attacks that exploit hardware vulnerabilities. This article defines these attacks as architectural attacks. Today, both industry and academia have only limited comprehension of architectural attacks, which represents a critical issue for the design of future systems. To this end, this work proposes a new taxonomy, a new attack model, and a complete survey of existing architectural attacks. As a result, it provides the tools to understand architectural attacks in more depth and to start building improved designs and protection mechanisms.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Security and privacy** → **Security in hardware**; • **Security in hardware** → **Hardware attacks;**

Additional Key Words and Phrases: Architectural attacks, IP attacks, functionality attacks, data attacks, attack model

ACM Reference format:
Tara Ghasempouri, Jaan Raik, Cezar Reinbrecht, Said Hamdioui, and Mottaqiallah Taouil. 2023. Survey on Architectural Attacks: A Unified Classification and Attack Model. *ACM Comput. Surv.* 56, 2, Article 42 (September 2023), 32 pages.
https://doi.org/10.1145/3604803

1 INTRODUCTION

The importance of cybersecurity grows every year. Future projections foresee a total market growth from 155.83 billion US dollars in 2022 to 376.32 billion US dollars by 2029 [1]. The main reason is the deployment of new technologies like the **Internet of Things** (**IoT**) and 5G communications. These technologies significantly increase the number of devices and their connectivity [2], creating new opportunities for cyberattacks.

There are many types of cyberattacks [3]. They can be organized based on the target system (network, software, or hardware) [4]. Attacks can also be classified based on the attack vector

Authors' addresses: T. Ghasempouri and J. Raik, Tallinn University of Technology, Computer Systems Department, Akadeemia tee, Tallinn, Estonia; emails: tara.ghasempouri@taltech.ee, jaan.raik@taltech.ee; C. Reinbrecht, S. Hamdioui, and M. Taouil, Delft University of Technology, Quantum and Computer Engineering Department, Mekelweg, Delft, The Netherlands; emails: cezar.rwr@gmail.com, m.taouil@tudelft.nl.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0360-0300/2023/09-ART42 $15.00
https://doi.org/10.1145/3604803

(i.e., the actor responsible for performing the attack), which may be software (logical) or hardware (physical) [5]. Combining both target and vector reveals the main types of threats present in the cyberattack field. For example, a virus is a software attack that targets another piece of software. A **distributed denial of service** (**DDoS**) [6] is a software attack that targets the network. Furthermore, a fault injection attack [7] is a hardware attack that targets the hardware. Most of the presented examples have been known in academia and industry for many years, where mature protection mechanisms already exist. However, in recent years, new attacks where software attacks target hardware have rapidly emerged. This new attack segment is defined as ***architectural attacks*** (**ArchA**). These attacks can compromise the entire system by exploiting a hardware vulnerability while being operated by software. As software, they can be executed remotely. In addition, the continuous increase in architectural complexity and connectivity in current and future chips shows a clear trend that architectural attacks will increase. Therefore, there is an urgent need to understand how they behave, what they have in common, and how they work, to build intelligent and efficient countermeasures.

There are two main steps toward the comprehension of architectural attacks (ArchA). The first step is to define an appropriate taxonomy. A taxonomy such as this can reveal the key characteristics of each attack type. Currently, there are proposed classifications for only a subset of ArchA. Cache-based attacks are the most studied group of attacks, with many proposed classifications [8–10]. More recently, transient execution attacks have gained popularity, and some classifications have also been proposed [11]. Other schemes have also been proposed to classify software attacks that exploit hardware based on their level of sharing in the system [12, 13]. However, no scheme has yet successfully assembled all ArchA into a single taxonomy.

Furthermore, the attacks are categorized based on three metrics, i.e., what, where, and how. The nature of an attack is generally determined by the target components. Hence, it should be clear what the aim of the attack is. Is the aim to steal data, to modify the functionality of the system, or for another purpose? This is the reason for identifying of metric "what." On the other hand, if the target components are hardware, the description should clearly categorize which elements of the hardware are targeted by the attack. This is why we introduced the metric "where." An understanding of this is essential for analyzing an attack. Last but not least, for better identification of the attack, how it is executed should be determined. Is the attack performed by manipulating the victim, injecting false data, or by observing the behavior of the victim? Hence, we determined the metric "How." In this taxonomy, all the above questions are answered, and a new and unified classification is proposed.

The second step required to understand ArchA relies on having a representative attack model. An attack model is a formal description that organizes any attack as a sequence of generic actions. Consequently, such models define the patterns inside each attack, revealing its *backbone*. In a way similar to the taxonomy problem, most attack models describe only cache attacks [14, 15]. Other works have tried to describe how attacks behave, but they lack the required formalism to be considered as an attack model [16–18]. Therefore, it is clear that current taxonomies and attack models cannot organize and describe architectural attacks because they can only refer to a subset of attacks. This article proposes a new taxonomy and attack model to organize and describe ArchA. As a result, a survey of existing attacks can be presented, showing how each one fits into our classifications and modeling. Hence, the main contributions of this research are the following:

- Proposal of a new taxonomy for architectural attacks using three metrics: what, where, and how.
- Proposal of a new attack model for architectural attacks that contains five steps: Setup, Trigger, Operate, Retrieve, and Evaluate.

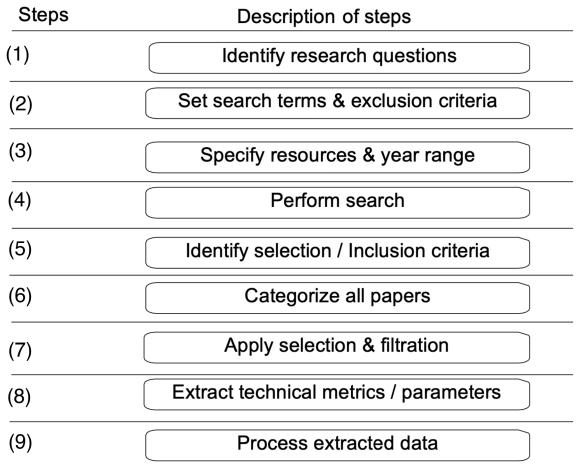

Fig. 1. Steps of Kitchenham and Charter's methodology.

— A survey of existing architectural attacks, framing them into the proposed classification and model.

The remainder of the article is organized as follows. Section 2 systematically provides a literature review. Section 3 summarizes related works that address proposed taxonomies and attack models. Section 4 describes the proposed attack model. Section 5 introduces the proposed taxonomy. Then, a survey of existing attacks is presented in three sections: Section 6 regarding IP attacks, Section 7 regarding Functionality attacks, and Section 8 regarding Data attacks. Section 9 provides a discussion of prospective developments, and Section 10 concludes the article.

2 RESEARCH METHODOLOGY

The literature covers a wide range of attacks that can compromise an entire system by exploiting a hardware component while being operated by software; i.e., architectural attacks (ArchA. The methodology used is based on Kitchenham and Charter's methodology [19], shown in Figure 1. The actions in each step of Kitchenham and Charter's methodology are described below. *Step 1: Identify research questions*: The objective of this survey is set to answer research questions concerning ArchA, as follows:

— RQ1: What is a general taxonomy for ArchA that can represent the main features of each type of attack? This question is essential because no scheme has successfully put all ArchA together in the same taxonomy.
— RQ2: What is a formal description to model ArchA? This question is important because describing attack models in a formal sequence of actions can clearly represent the patterns inside the attacks.

Step 2: Set Search Terms and Exclusion Criteria: The second stage of conducting this review is to identify the search terms such as keywords used in searching and collecting relevant papers to answer the research questions. The keywords below with the following strategy were used. The search began with generic keywords; i.e., "attacks," "hardware," and "software,", because the focus of this survey was on attacks that target hardware by manipulating software. Afterwards, detailed keywords such as "processors," "cache memor," and communication channels were added. Exclusion criteria were as follows:

Table 1. Search Result Count of Respective Keyword Combinations

Search keywords	Number of articles
"hardware," "software," "attack"	2,065
"hardware," "attack," "processor"	521
"hardware," "attack," "cache memory"	105
"hardware," "attack," "communication"	1,605

— Non-peer-reviewed sources such as online articles.
— Attack mitigation papers that do not incorporate the attack mechanism itself.
— Articles that discuss attacks only at software level.

Step 3: Specify Resources and Year Range: The following digital libraries were used to obtain a wide range of related articles: IEEE Explorer, Google Scholar, ACM Digital Library, Elsevier, and Springer. All articles between 2011 and 2021 were considered. *Step 4: Perform Search*: Initially, a total of 2,065 papers were found, based on the initial search terms mentioned earlier. The papers were then filtered to more focused categories by a procedure to be described next. *Step 5: Identify selection/inclusion criteria*: Analyzing the articles revealed that these attacks are mainly performed on three domains: "Processor," "Cache memory," and "Communication channel." Hence, the search space was narrowed by the corresponding keywords. *Step 6: Categorize all papers*: Table 1 represents the number of articles for each search, based on the keywords defined in previous steps. *Step 7: Apply selection and filtration*: The selection and filtration process are explained below.

— Remove duplicate or multiple versions of the same articles.
— Apply inclusion and exclusion criteria to avoid any irrelevant papers.
— Remove review papers from the collected papers.
— Apply quality assessment rules to include qualified papers (quality was determined based on the parameters mentioned in the next step) that best address the research questions.
— Look for additional related papers using the reference lists of the collected papers and repeat the same process again.

Step 8: Extract technical metrics/parameters: The list of papers to be considered in this review was identified by applying parameters such as **quality assessment rules** (**QARs**). The QARs are important to ensure proper evaluation of research paper quality. Therefore, 10 QARs were specified, each worth 1 mark out of 10. The score on each QAR could be 1 or 0, where 1 indicates "answered," whereas 0 indicates "not answered." The selected article should obtain a score of 10 out of 10 to be selected; otherwise, it is excluded. The QARs are listed below:

— QAR1: Are the research objectives identified clearly?
— QAR2: Are the techniques of the implementation well defined and deliberated?
— QAR3: Is the platform (in this case, processor, cache memory, and communication protocol) that the ArchA will target clearly defined?
— QAR4: Does the paper include practical experiments regarding the proposed technique?
— QAR5: Are the experiments well designed and justified?
— QAR6: Is the proposed algorithm applied on a hardware component?
— QAR7: Is the method effectiveness reported?
— QAR8: Is the proposed ArchA model design compared to others?
— QAR9: Are the methods used to analyze the results appropriate?
— QAR10: Does the overall study contribute significantly to the area of research?

Overall, after Steps 7 and 8, 98 useful papers were identified. *Step 9: Process extracted data:* In this step, the objective is to analyze the final list of papers to extract the needed information;

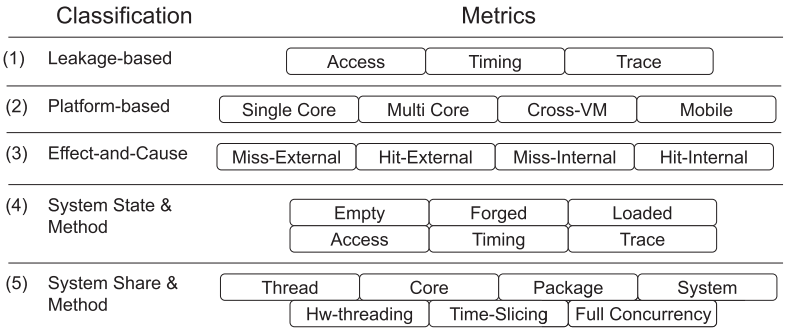

Fig. 2. Classifications of architectural attacks with metrics.

i.e., provide answers to the given research questions. First, general details are extracted from each paper, such as the paper number, paper title, publication year, and publication type. Then, more specific information is sought, such as the algorithm model, the category of hardware component, and whether the paper addresses cache memory, processor, communication channel, or a combination. Finally, any details directly related to the research questions are pursued. Table 1 shows the number of collected papers after processing all steps of Kitchenham and Charter's methodology in the column "final number of collected papers," which is equal to 95.

3 RELATED WORK

This section presents related work on classifying and modeling of architectural attacks (ArchA). To the best of the author's knowledge, only five classifications exist for architectural attacks, all of which are studied and analyzed in this section. It is noteworthy that each classification focuses on a different characteristic to organize the attacks. The existing classifications are presented below, followed by the existing attack models.

3.1 Classifications of Architectural Attacks

Five classifications have been proposed for architectural attacks. Each proposal focuses on a different characteristic to organize the attack. Figure 2 shows each classification and its metrics, followed by a description and an analysis of its benefits and limitations.

3.1.1 Leakage-based Classification. Based on the leakage source, the authors in [8] classified architectural attacks on caches into three types: (i) timing, (ii) access, and (iii) trace. In timing-driven attacks, the attacker performs an attack based on different temporal responses of components or operations. In access-driven attacks, the attacker performs an attack based on user behavior of components or functions. Finally, in trace-driven attacks, the attacker performs an attack based on high-level monitoring information from internal system monitors or some specialized external instruments (e.g., probing the power consumption).

However, novel attack techniques have emerged that do not exploit side-channel attacks. Hence, the above classification is not complete because it considers only information leakage. For example, Rowhammer attacks only inject faults in main memory to gain privilege or alter system functionality. Consequently, this case has no leakage behavior.

3.1.2 Platform-based Classification. In this classification scheme, attacks are classified based on the hardware platform [13]. According to the authors, there are four possible platforms: (i) single-core, (ii) multi-core, (iii) cross-VM, and (iv) embedded/mobile. The single-core platform represents

the most popular platform used in desktop/laptop computers. Attacks in this category require malicious processes to run in parallel with victim processes because there is only a single core in the system. Such a malicious process is also defined as a spy process. The multi-core class includes desktop/laptop computers and more complex devices like servers. In this category, it is possible to attack using a spy process that runs on a different core from the victim or by forcing the attacker's application to run in a separate core. The third type, i.e., cross-VM, refers to virtualized systems. In a virtualized environment, the attacker and victim processes run in different **virtual machines** (**VMs**). Hence, the attack must interact with the other VM indirectly, which defines the term cross-VM. Even if the actual hardware is not disclosed in these systems, it is still possible to attack them by exploiting the vulnerabilities of VM implementations. The last category refers to embedded or mobile computer systems. These systems typically have limited performance because they are constrained devices. In this scenario, the operating system can be lightweight, or even absent. At the same time, the software typically has fewer restrictions on accessing the hardware configuration and low-level information.

Although the platform-based classification is interesting, it clarifies that several existing attacks might fit into two or more categories when the attacker can apply a specific attack. For example, the pure timing attack of Bernstein can be mounted on single-core, multi-core, and embedded/mobile platforms. Therefore, the classification is not helpful in understanding the basic principles of these attacks.

3.1.3 Cause-and-Effect Classification. The Cause-and-Effect classification proposed by [10] is dedicated to cache attacks. The effect refers to cache accesses, which can be cache misses or cache hits. The cause specifies which process made the effect happen; this is external when performed by the attacker process and internal when performed by the victim process. Consequently, this classification has four categories: (i) Miss-External, (ii) Hit-External, (iii) Miss-Internal, and (iv) Hit-Internal. In the Miss-External class, attacks exploit cache misses caused by the victim process that are indirectly induced by the attacker. For example, in the Prime+Probe attack, the attacker accesses the cache before and after the victim's computation. Hence, the cache misses reveal the locations used by the victim. Note that the cache misses only occurred because the attacker had already accessed these locations. Conversely, Hit-External exploits cache hits when the attacker accesses the cache. The same idea is applied for Miss-Internal and Hit-Internal, with the difference that the victim triggers the miss or hit behavior. Pure timing attacks typically fall into the internal classes. The attacker only collects the final time of the operations, i.e., the variation in timing is a direct consequence of misses or hits during the victim's access.

This classification suits cache attacks, but cannot be directly applied to general architectural attacks such as attacks related to processing or communication. Note that this classification can be updated by encompassing different architectural attacks by changing the options related to the effect.

3.1.4 System State and Method Classification. The System State and Method classification is proposed in [9] and uses two metrics to classify attacks. The first one is the system state before the victim's operation. The system state refers to the cache's content, which can be empty, forged, or loaded. Empty state means that the cache has not been initialized; the initial state does not matter for the attack. The forged state refers to a cache initialized by the attacker, where the attacker writes specific content to manipulate cache behavior. The loaded state refers to an initialized cache with some (or all) contents from the victim already loaded. As a result, the attacker can quickly identify changes in the cache. The second metric specifies the method used to collect the cache leakage. In this case, the same metrics as the leakage-based classification are used, namely, timing, access, and trace.

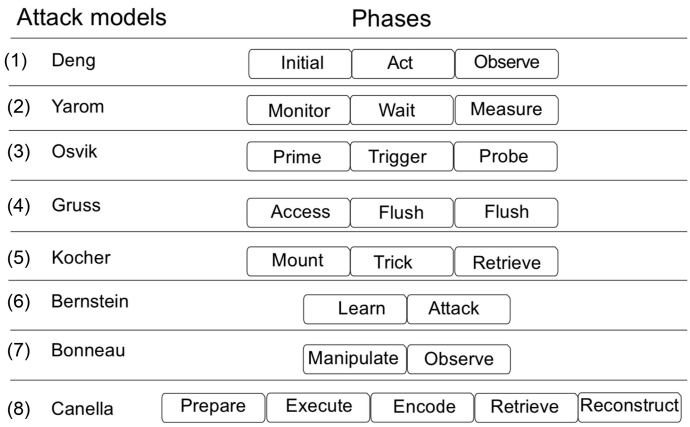

Fig. 3. State-of-the-art attack models and their phases.

This classification also specifically addresses cache attacks. As discussed in the Introduction, ArchA can target any system component, including processing elements and communication structures. Hence, this classification is also limited for the present purposes.

3.1.5 Process Sharing and Method Classification. The survey in [12] proposed a classification based on resource sharing and degree of concurrency. Sharing of resources can occur on different levels, such as thread, core, package, or system level. Consequently, the classes related to sharing are thread-shared, core-shared, package-shared, **non-uniform memory access** (**NUMA**)-shared, and system-shared. With respect to the degree of concurrency, three categories are applicable: full concurrency (multi-core), time-sliced execution on a single core, and hardware threading (SMT). Note that this classification focuses on the relation between different processes running on a complex system and how vulnerabilities arise from such concurrent and sharing behavior. In [12], only cache attacks have been classified under this scheme. However, this classification could embed many other logical side-channel attacks. Many attacks use a spy process (a malicious process sharing the same core) or spy node (a malicious core sharing the system).

Although this classification can organize most logical SCAs, it does not help understand the main aspects of the attacks. It only defines the system configuration where attacks take place.

All in all, as discussed above, there are existing classifications only for subsets of ArchA. For instance, cache-based attacks are the most studied group of attacks and have many proposed classifications. Transient execution attacks have also been studied and classified by many authors. Other schemes have also been proposed to classify software attacks that exploit hardware based on their level of sharing in the system. However, no scheme has yet successfully put together all ArchA into the same taxonomy. Therefore, the authors believe that a unified taxonomy covering all attacks instead of just a subset is essential.

4 ATTACK MODELS

This section explains state-of-the-art ArchA models. They all divide an attack into phases; i.e., a series of steps that describe an attack. Understanding and comparing the models is essential because different authors have different views of the same attack; this results in models with different numbers of phases. Even when authors agree on the number of phases, the phase definitions can be different. Figure 3 shows the attack models. It is apparent that the models use different numbers and different definitions of phases. The following subsection describes the eight attack models.

4.1 Deng

The approach in [14] is related to side-channel attacks on caches. In this work, several well-known attacks such as Flush+Reload, Evict+Time, Prime+Probe, Flush+Flush, Evict+Reload, Bernstein, and Cache Collisions are analyzed. The approach divides these attacks into three steps: (i) during the first phase, a memory access sets a single cache block to some known initial state; (ii) during the second phase, some action like fetch can be done by the victim or attacker; (iii) during the third phase, a final action is taken to derive information by observing timing. This takes a short time if there is a hit and a longer time if there is a miss.

4.2 Yarom

The work in [15] analyzed the Flush+Reload attack on caches, which was found to be composed of three phases: (i) during the first phase, the monitored memory line is flushed from the cache hierarchy; (ii) the attacker then waits to allow the victim time to access the memory line before the third phase; (iii) in the third phase, the attacker reloads the memory line, measuring the time to load it. If, during the wait phase, the victim accesses the memory line, the line will be available in the cache, and the reload operation will take a short time. If, on the other hand, the victim has not accessed the memory line, the line will need to be brought in from memory, and the reload will take significantly longer.

4.3 Osvik

The approach in [18] divides Prime+Probe attacks into three steps: (i) in Prime, the attacker fills (parts of) the cache with data; (ii) the attacker triggers a sensitive operation (e.g., encryption); and (iii) in Probe, the attacker reads the data written in the Prime step and evaluates which addresses were used by the victim based on observation of cache misses.

4.4 Gruss

[20] analyzed the Flush+Flush attack in caches with hierarchy. This approach decomposed the attack into three phases: (i) in the first step, the attacker accesses the memory location that is cached; (ii) in the second phase, the victim only flushes the shared line. Because the line is present in the last-level cache by inclusiveness, it is flushed from this level; (iii) a bit also indicates that the line is present in the L1 cache, and therefore must also be flushed from this level. To transmit a 0, the attacker stays idle. The victim flushes the line (step 1). Because the line is not present in the last-level cache, it is also not present in the lower levels, which results in faster execution of the clflush instruction. Hence, only the attacker process performs memory accesses, whereas the receiver only flushes cache lines. To send acknowledgment bytes, the victim performs memory accesses, and the attacker runs a Flush+Flush attack.

4.5 Kocher

Spectre, an attack that can be performed on the CPU as well as on caches, is divided into three steps in [17] as follows: (i) mounting, where the attacker introduces a sequence of instructions into the process address space; (ii) trick, where the attacker induces the CPU to perform a transient execution; and (iii) retrieve, where the attacker gathers information through the covert channel.

4.6 Bernstein

Although the Bernstein attack is divided into three phases by the approach in [14], it is composed of two parts in [21]: (i) a learning phase, where statistical reference models of cache behavior are developed; and (ii) an attack phase, where multiple encryptions are collected and correlated with the reference models.

4.7 Bonneau

In [22], the Spectre attack in AES is divided into two phases: (i) manipulation, in which the attacker manipulates the input message of the encryption algorithm to force collisions of cache addresses (cache hits) when the key hypothesis is correct; and (ii) observation, in which the attacker observes whether there is a reduction in the required number of traces to retrieve the key.

4.8 Canella

[16], on the other hand, divides Spectre and Meltdown into five phases: (i) preparation of a micro architecture; (ii) execution for a trigger instruction; (iii) encoding unauthorized data by transient instructions through a micro-architectural covert channel; (iv) the CPU retires the trigger instruction and flushes the transient instructions; and (v) reconstructing the secret from the micro-architectural state. As discussed, there is no general formal structure for modeling attack phases. In some articles, authors split particular attacks into two phases, whereas in others, they split them into three phases. A unified attack model that covers all ArchA is currently missing. This implies that the underlying patterns that reveal the attack mechanisms are not well-defined and understood.

To move beyond the state of the art, this article first proposes a new taxonomy for ArchA using three metrics: what, where, and how. These metrics can cover all architectural attacks. Second, this work propounds an innovative attack model that contains five steps: Setup, Trigger, Operate, Retrieve, and Evaluate. In the following sections, these aspects are further presented in more detail.

5 PROPOSED TAXONOMY

Architectural attacks are software attacks that exploit any hardware vulnerability. Modern integrated circuits like systems-on-chip or multi-processor systems-on-chip are complex components, from hardware accelerators to memories and interfaces. Hence, architectural attacks encompass a wide range of possibilities because an adversary might theoretically exploit any hardware component. The literature contains several descriptions of software attacks targeting hardware components. However, due to the widely varying nature of hardware components like memories, processors, and accelerators, the attacks were organized into separate classifications. It was observed in the previous section that memory attacks are classified in the cache attack category and processor attacks in the transient execution category, whereas others might fall under more generic concepts like side-channel attacks. To bring all architectural attacks together in the same scheme, this paper proposes a novel *taxonomy*. As a result, the metrics and classification proposed here provide a systematic way to analyze and evaluate such attacks. Our proposed taxonomy uses three metrics as criteria:

- Target: **What** the attacker is looking for.
- Location: **Where** is the victim's vulnerability.
- Method: **How** the attacker exploits the vulnerability.

Figure 4 presents the taxonomy as a hierarchical arrangement of the proposed metric criteria. In addition, the figure shows where all attacks evaluated in this survey are located in the taxonomy. In the following subsections, each criterion is described in more detail.

5.1 Target (*What*)

The attack goal is the main objective behind the attack. Three options cover all possibilities:

- IP: The attack looks for intellectual property information, such as implementation details of a design or some specific software/application. Engineering information is valuable in the

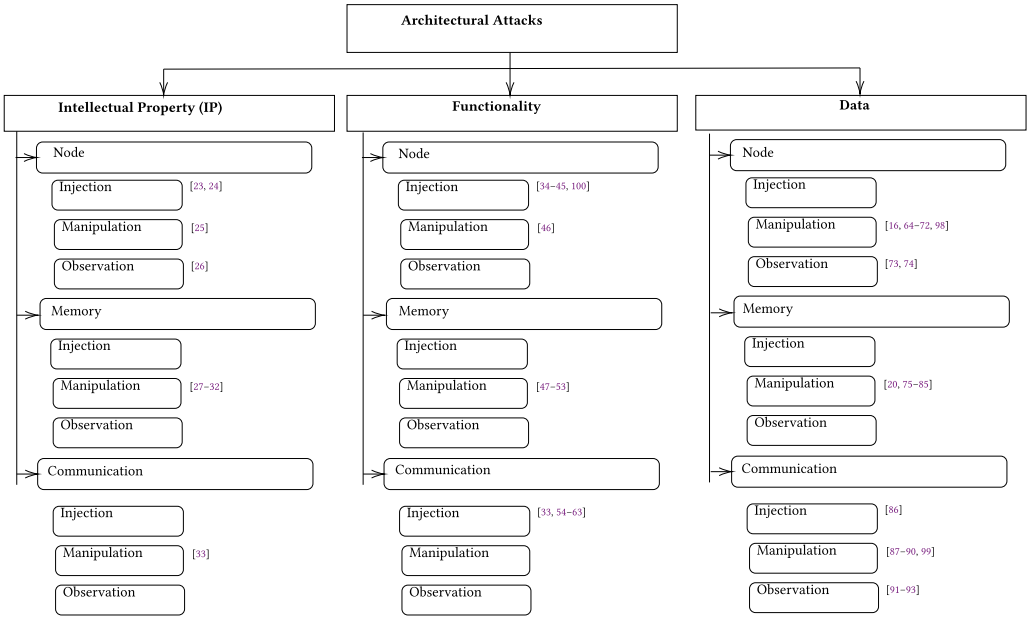

Fig. 4. Proposed architectural attack taxonomy.

global market because it makes it possible to avoid months of research and development and reduces risks when building something new.

- Functionality: In this case, an attacker aims to modify temporarily or permanently the functionality of a system. Some examples of functionality exploitation are bypassing a security check, providing privilege escalation, and decreasing overall system reliability.
- Data: When the goal is data, the attacker aims to steal or corrupt data inside a system. In most cases, important information is encrypted. Hence, the first target of most data attacks is to retrieve the secret key of the system.

5.2 Location (*Where*)

The location refers to where the vulnerability is. Because architectural attacks are software exploiting hardware, location criteria relate only to hardware components. Because different types of components exist in hardware, these criteria are classified into three parts:

- Processing: Processing elements are processors, hardware accelerators, and co-processors. Any element responsible for an operation or task in the system can be classified as Processing.
- Memory: Memory relies on storage components. Some examples are cache memories, flash memories, **read-only memories** (**ROMs**), and **static and dynamic RAMs** (**SRAMs** and **DRAMs**).
- Communication: Communication components are responsible for interfacing with other components. Internally, communication is provided by a bus system (e.g., AMBA AHB or AMBA AXI) or even a **network-on-chip** (**NoC**). These components interface with internal elements—Processing and Memory—and among each other. Moreover, there are communication components dedicated to external interfaces. Examples are Ethernet, UART serial, and Bluetooth.

5.3 Method (*How*)

These criteria focus on how the attacker triggers a specific vulnerability present in the hardware. Three options are given:

- Injection: In this method, an attacker can force a trigger to exploit a certain vulnerability. Fault attacks like Buffer Overflow use the injection method, where the adversary overwrites part of memory to change the victim's behavior.
- Manipulation: In this method, an attacker manipulates a component from the system to trigger a vulnerability. This method is used when the attacker cannot create or force a trigger to exploit a certain vulnerability. There are several reasons for this, but the main reason relies on permission accesses.
- Observation: The observation method occurs when the vulnerability does not need a trigger. In this case, the vulnerability is always present; i.e., the victim is always leaking information. In this method, the attacker only needs to find a means to monitor the victim's behavior and process the leakage into meaningful data.

6 IP ATTACKS

IP attacks target extracting sensitive information related to an engineering process. Such attacks aim to gain a technical advantage, whether to replicate, fake, or reuse a design. Examples of sensitive technical information are the algorithm used, the division between software and hardware for a certain task, or even non-disclosed functions inside the system. Therefore, this section focuses on software attacks that can extract sensitive technical data, also known as intellectual property. Figure 4 presents references for existing IP ArchA under the "**Intellectual Property (IP)** attack" branch.

6.1 Node

This subsection presents attacks that aim to reveal design secrets from nodes like processors and **graphical processing units (GPUs)**. According to the taxonomy being discussed here, node attacks are categorized into three main groups: injection (suffix -INJ), manipulation (suffix -MAN), and observation (suffix -OBS). Next, each group is described in more detail.

Group IP-NODE-INJ: This group contains attacks that generally inject random inputs to force the processor into abnormal conditions, revealing unexpected features or instructions. Later on, the attacker monitors processor behavior to discover anomalies.

Attack Formula: The steps of the proposed attack formula can be described as follows:

- S: In the setup phase, the attacker defines the main parameters to be explored in a node architecture. Examples are instruction opcodes, microcode values, and special hardware parameters.
- T: The attacker crafts new data based on the defined parameter to the attack and applies them to the node architecture. This crafted information is from the specification and configures an injection technique.
- O: The node executes or tries to execute the input provided.
- R: The output behavior of the node is observed, which can be an expected output value or just different timing behavior.
- E: In this phase, when the system does not crash and present a different behavior, the attacker knows that a valid input was revealed. Next, he/she analyzes what function has been uncovered by exploring the same parameter under different circumstances.

Attacks: In [23], the author presents a tool called Sandsifter. It audits ×86 processors for hidden instructions and hardware bugs by systematically generating machine code to search through a

processor's instruction set and monitoring execution for anomalies. This attack injects random inputs to force the processor into abnormal conditions. In [24], the attacker creates microcode updates to two AMD processors (K8 and K10) and observes the output. The output is represented by the register values and memory locations. The underlying idea is to generate distinct behaviors between the original and the patched macroinstruction execution. More precisely, the patch contains a microcode instruction that constantly crashes on execution.

Group IP-NODE-MAN: This group refers to attacks that force specific behaviors to infer design-related information about the node. Unlike the IP-NODE-INJ group, this group creates only expected and valid inputs. Hence, it falls into the manipulation category. The objective here is to define a feature to observe and provide the input that maximizes its actuation.

Attack Formula: The steps of the proposed attack formula can be described as follows:

S: In the setup phase, the attacker defines which feature to highlight during the attack. Examples are schedule algorithms, privilege definitions, or task arbitration.
T: The attacker creates valid inputs that can highlight the target feature and apply it to the node.
O: The node executes the provided input.
R: The execution of the task will reveal the strategy or algorithm behind the design. Multiple attempts can be made to identify these differences.
E: In this phase, the attacker analyzes the outputs to infer what known algorithm matches the observed result. Finally, the attacker understands important design-related information.

Attack: The last attack proposed in [25] runs specific benchmarks in the GPU to infer design-related information. Each benchmark aims to emphasize a specific GPU characteristic, hence revealing how it works. Because this attack runs expected programs, but carefully chooses those that might reveal design secrets, this attack is classified into the manipulation category.

Group IP-NODE-OBS: This group contains attacks that mainly observe the system output when regular input is provided. Unlike the IP-NODE-MAN group, this group does not create particular inputs to manipulate the system to reveal more information. The attacks in this group focus on running typical applications or simply accessing system components and observing their behavior, output values, or timing.

Attack Formula: The steps of the proposed attack formula can be described as follows:

S: In the setup phase, the attacker defines which component to explore.
T: The attacker then creates many different inputs and applies them to the system. In this step, the attacker can use random input generation, simple trial and error, or brute force (trying all possibilities).
O: The node executes the provided input.
R: The execution of the task will reveal the strategy or algorithm behind the design. Multiple attempts can be made to identify these differences.
E: In this phase, the attacker analyzes the outputs to infer what known algorithm matches with the observed result. Finally, the attacker understands important design-related information.

Attack: In [26], the attacker aims to discover which special registers can trigger hidden ×86 instructions. The registers that can perform such operations are the global configuration registers, known as MSR registers in the ×86 architecture. The first step in this attack performs a timing analysis by accessing all available MSR registers to understand which one contains a unique behavior. Because most MSR registers trigger similar functions in the system, their access times are also similar. Only different MSRs will have different access times.

6.2 Memory

Existing attacks on memory design information fall only into the manipulation category, defined here as group IP-MEM-MAN.

Group IP-MEM-MAN: The attacks in this group create specific programs to run in the system and to collect running information to infer memory characteristics. Attacks in this group can target cache memories and DRAMs. Most attacks target features like physical-to-logical address mapping, cache line size, and replacement policies, as well as physical characteristics like access latency.

Attack Formula: The steps of the proposed attack formula can be described as follows:

S: A set of algorithms is prepared to highlight the target characteristics.
T: The algorithm is run in the system.
O: The natural execution of the algorithm creates covert channels with different latencies or different behavior (that can be observable).
R: Retrieve: Attackers access system monitors like high-performance counters or use their own measurement means (e.g., a timer) to collect side-channel information.
E: The attackers process the obtained information to infer the feature value. The attack is repeated to guarantee that the findings are highly provable. Fine-tuning the set of algorithms to optimize new runs is also possible.

Attacks: There are several research efforts that infer cache properties using measurement-based analysis [27–29]. These solutions use the performance counters available in current platforms to infer properties of the cache hierarchy. Recent works that infer DRAM properties [30–32] focus on address mapping, like the virtual-to-physical memory allocation scheme, by first inferring the mapping between virtual address bits and physical bank bits for the Intel Xeon processor using latency-based analysis. Another example uses a latency-based analysis to identify channel, rank, and bank bit mapping between virtual and physical addresses.

6.3 Communication

Only manipulation attacks have so far been proposed to perform reverse engineering on the communication structure. This group, IP-COMM-MAN, is described below.

Group IP-COMM-MAN: This group refers to attacks that inject multiple packets to infer the configuration used inside the communication system.

Attack Formula: The steps of the proposed attack formula can be described as follows:

S: The attacker requests the packets.
T: The system responds to the request and sends the packets to the network.
O: In this step, flooding is performed. This can be done by requesting too many packets, triggering packets with wrong paths, triggering packets that intend to create a deadlock, or triggering packets that cannot reach the target.
R: Flooding causes the victim to lack resources.
E: The availability of the victim's resources is checked in this step. If there is a lack, this means that the **denial of service** (**DoS**) attack has been performed successfully.

Attack: The attack presented in [33] uses an algorithm to uncover the details of the communication structure by simply manipulating packets. Figure 5 illustrates the steps of IP attacks. Clearly, some attacks have steps in common. For instance, "IP-NODE-MAN" and "IP-NODE-OBS" have similar Operate, Retrieve, and Evaluate steps. Obviously, this kind of analysis and revelation of detailed information on attack structure can lead to building a better mitigation strategy. In the next sections, the same analysis is reported for Functionality and Data attacks.

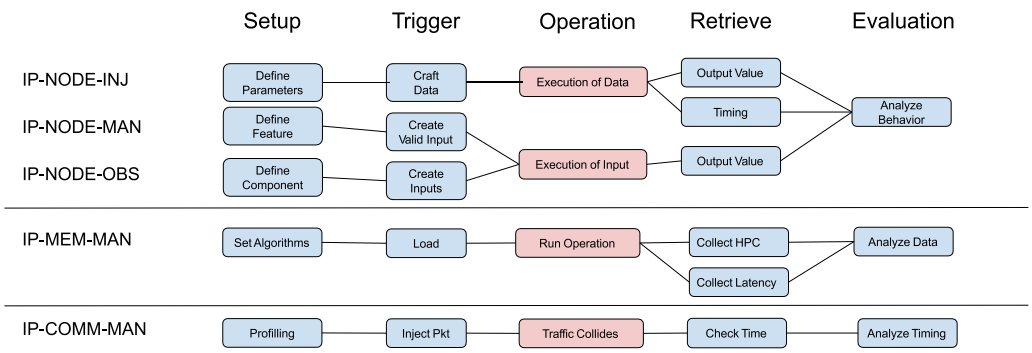

Fig. 5. Attack phases of IP attacks and the associated groups.

7 FUNCTIONALITY ATTACKS

Depending on the target component, the attack behavior will differ, and therefore these behaviors were identified as follows. Figure 4 represents existing attacks belonging to this attack type under the "Functionality attack" category.

7.1 Node

Functionality attacks on processing elements aim to subvert the program's control flow from its normal course and to force programs to act in the manner that attackers wish. In the attacks observed in the state of the art that focus on processing elements, three attack patterns were identified: buffer overflows, reuse of existing code, and speculative overflows. Each pattern is described in the following paragraphs.

Group FUNC-NODE-INJ: The first group refers to the attack known as buffer overflow. Buffers are memory storage regions that temporarily hold data while it is being transferred from one location to another. A buffer overflow (or buffer overrun) occurs when the volume of data exceeds the storage capacity of the memory buffer. As a result, the program that is attempting to write data to the buffer overwrites adjacent memory locations. Attacks that exploit memory errors such as buffer overflows constitute the largest class of attacks reported by organizations such as the CERT Coordination Center [94] and pose a serious threat to computing infrastructure. In the literature, various kinds of buffer overflow attacks ([35] and [34]) are described as follows:

(i) Buffer overflow that overwrites the return address.
(ii) Buffer overflow that overwrites the frame pointer.
(iii) Buffer overflow that overwrites the function pointer.
(iv) Buffer overflow that overwrites the dynamic linker tables.

However, the authors believe that these attacks can be identified in one main group because their attack algorithms are almost the same and they differ only in the Trigger phase.

Attack Formula: The attack phases are as follows:

S: The attacker abusea inputs functions w.r.t. type or size; for instance, if the buffer size is set to 500 characters, the attacker inserts 510 characters to overrun the buffer (stack).
T: The attacker then overwrites the sensitive area of code and points it to an exploit payload to gain control over the program. This can cause the program to behave unpredictably and generate incorrect results, memory access errors, or crashes. Note that this phase is the only phase where the mentioned attacks slightly differ.

O: This step redirects execution to another attacker's code, giving privileges. For example, an attacker can overwrite a pointer (an object that points to another area in memory) and point it to an exploit payload to gain control over the program.
R: Attacker's code execution
E: Checking privileges

Attacks: [35] and [34] exhaustively studied the various kinds of buffer overflow attacks. [36] demonstrated security attacks that exploit unsafe functions to overflow stack buffers and methods to detect and handle such attacks. The approach described in [38] argues that defenses such as **control-flow integrity** (**CFI**) do not stop control-flow hijacking attacks, which occur by buffer overflows (return address overwritten). It demonstrates a mitigation strategy for such a case. [42] exploited different protection techniques for stack-based buffer overflows and consequently presented four strategies to bypass these protections. [37] presented an address obfuscation strategy to prevent these kinds of attacks. [43] studied different buffer overflow attacks and a compile-time mitigation method for address randomization to stop attacks. [45] demonstrated a code injection attack that resulted from a buffer overflow and a mitigation strategy by **instruction set randomization** (**ISR**). [41] introduced a mitigation technique called SmashGuard, which is a hardware solution to prevent manipulation by buffer overflow attack of function return addresses. [40] provided a solution to detect code reuse attacks on ARM mobile devices. [44] described buffer overflow attacks (Stack Smashing) in the UNIX operating system. PACMAN is introduced in [100] that manipulates the ARM pointer authentication mechanism to hijack the control flow and corrupt the memory. Finally, [39] introduced another mitigation technique called StackGuard to prevent buffer-overflow attacks.

Group FUNC-NODE-MAN: This group consists of attacks that reuse existing code, also known as return-oriented programming attacks or return-into-LIBC attacks. This type of attack is used when a system mitigates the possibility of injecting code into memory to be executed. The most widely used code injection protection method in operating systems is "W⊕X" [95], which prevents writable data in memory from being executable at the same time. As a response, attackers reuse existing code to perform valid actions on the system. The standard C library, LIBC, is the most common target because it is placed near the kernel code and provides useful functions like system calls. However, in principle, any available code, either from the program's text segment or from a library to which it links, could be used. Consequently, the attacker can identify useful sequences in the code that, when put together, create the malicious functionality. Each sequence can perform a specific function like addition of two registers, load constant, load memory data, store, and so forth. Note that it is important that each code sequence ends with a return call. The attacker then needs only to manipulate the stack to call each sequence (i.e., jump to the address of the first instruction of the sequence) in a specific order to achieve the desired functionality. Stack manipulation can use the buffer overflow attacks previously mentioned as Group A.

Attack formula: In this group, the steps of the formula can be described as follows:

S: In the setup phase, the attacker dumps the memory content and identifies useful code sequences. Thereafter, the attacker builds a "program" ordering the sequence of addresses that he must call.
T: Attackers manipulate a function call, which is typically performed through buffer overflow attacks.
O: In this step, the processor executes the instructions of each sequence (i.e., valid code already in the system).

R: There are many exploitation methods for this attack because the attacker defines the algorithm to be executed. In this case, the example of a privilege escalation of the attacker's user will be examined.
E: Verify the level of privilege in the system.

Attacks: [46] describes this kind of attack and a mitigation procedure to prevent control-flow hijacking attacks and code-injection attacks. The work introduces KAISER as an algorithm that enforces strict kernel and user space isolation such that the hardware does not hold any information about kernel addresses while running in user mode.

7.2 Memory Elements

The main series of functional attacks on memory is typically derived from a single family of attacks called Rowhammer. Rowhammer is a security exploit that takes advantage of an unintended and undesirable side effect in **dynamic random-access memory** (**DRAM**), in which memory cells interact electrically between themselves by leaking their charges, possibly changing the contents of nearby memory rows that were not addressed in the original memory access. These attacks repeatedly perform accesses to certain memory rows with high frequency to degrade their internal charging capacitance. This operation is defined as "hammering," and that is why such a threat is called Rowhammer. There are three possible attack patterns: flush-based [48], eviction-based [49], and remote-based [52]. They are described in detail in the following paragraphs.

Group FUNC-MEM-MAN-I: Attacks in the first group perform "hammering" through a flush operation. Flush operations can erase a cache line or region. By continuously forcing flush operations, the same memory rows must be accessed to retrieve the missing information.

Attack Formula: The steps of the proposed attack formula can be described as follows:

S: In the setup phase, the attacker accesses a memory row and erases cache lines through a Flush instruction.
T: The attacker then accesses the next memory row and flushes this line as well.
O: This phase represents a Flush. It is applied to determine whether the data from the previous step have changed. This phase helps the Evaluate phase to gather information related to identifying the cache timing.
R: This phase is performed by a Reload instruction.
E: In this phase, the timing behavior is analyzed. If, during the Trigger phase, the victim accesses the memory line and flushes it, the line will need to be brought from memory, and the reload operation will take a longer time. If, on the other hand, the victim has not flushed the memory line, the line will be available in the cache, and the reload will take a significantly shorter time.

Attacks: The known attacks belonging to this group are as follows. [48] introduced a flush-based hammering technique called SGX Bomb, which can lock down a processor. DRAMMER, the hammering attack described in [53], relies on the predictable memory reuse patterns of standard physical memory allocators as implemented on Android/ARM. [51] presents DRAMA attacks, a class of attacks that exploit the DRAM row buffer, which is shared in multi-processor systems.

Group FUNC-MEM-MAN-II: Attacks in the second group perform "hammering" by causing evictions from cache memory. This can be done by accessing data that align with the same line in the cache. As a result, the victim address in the DRAM must be continually accessed due to evictions in the cache. Compared to the flush-based attack, this approach is much less efficient because multiple DRAM accesses are required to successfully evict a cache line.

Attack Formula: The steps of the proposed attack formula can be described as follows:

S: In the setup phase, the attacker accesses a memory row and erases a cache line through an Eviction instruction.
T: The attacker then accesses the next memory row and evicts this line as well.
O: This step represents Eviction instructions. They are applied to determine whether data from the previous step have changed. This phase helps the Evaluate phase to gather the information related to identifying the bit flip.
R: This phase is performed by Flush.
E: This phase checks whether there has been a bit flip after all four steps above.

Attacks: [49] describes an eviction-based hammering attack that causes a bit flip by hammering only one location in memory. [47] demonstrated that an attack called Rowhammer.js can be forced into fast cache eviction to trigger the Rowhammer bug with only regular memory accesses.

Group FUNC-MEM-MAN-III: Attacks in the third group are executed remotely. Note that in some studies, this concept is extended to **remote direct-memory access** (**RDMA**). In other terms, it is applied on a server; i.e., it continuously receives packets from the sender and writes to RDMA. In this group, the Operate phase is done by Flush, the Trigger phase by a Reload instruction, and the Evaluate phase by analyzing cache timing behavior.

Attack Formula: The steps of the proposed attack formula can be described as follows:

S: In the setup phase, the attacker accesses a memory row, which can be done by a read or a write instruction.
T: The attacker then accesses the next memory row.
O: In this step, a Flush instruction is executed. This phase is done by the victim and leads the attacker to gain access to information.
R: A Reload is applied to determine whether the data from the previous step have changed or not. This phase helps the Evaluate phase to gather information related to measuring the time to load the data.
E: In this phase, timing behavior is analyzed.

Attacks: Throwhammer [52] and Nethammer [50] are two examples of this group of attacks. These attacks can remotely perform hammering on servers by continuously receiving packets from senders and writing them to RDMA.

7.3 Communication

This subsection focuses on functionality attacks on the communication structure. By observing existing attacks, only one group could be defined, which falls into the injection category.

Group FUNC-COMM-INJ: DoS attacks are exhaustively used in the state of the art when the vulnerability of a communication protocol is assessed. This group of attacks generally disrupts the system by overloading resources. For example, a malicious application that generates packets at a high injection rate can produce this attack. In some cases, the attack can overload the communication infrastructure.

Attack Formula: The steps of the proposed attack formula can be described as follows: This attack generally attempts to flood the system by sending a massive number of packets. Note that different flooding algorithms have been demonstrated by researchers; for instance, in some studies the flooding is performed by assigning a wrong path to the packet. This means that the packet introduces erroneous paths to the network, with the aim to trap it into a dead end. These

packets use paths that intentionally disrespect the deadlock-free rules of the routing technique and are intended to create deadlocks in the network. In some studies, these packets cannot reach their targets and circulate indefinitely in the network, causing a waste of bandwidth, latency, and power [57]. Although these algorithms use different approaches to flood the network, the authors believe that they can be divided into similar phases. The following paragraphs describe the proposed five phases for this group.

S: This step is performed by the attacker and requests packets.
T: This step involves replying to the request and sending the packets to the network.
O: In this step, flooding is performed. This can be done by requesting too many packets, triggering packets with wrong paths, triggering packets that intend to create deadlock, or triggering packets that cannot reach the target.
R: Due to the flooding, the victim experiences a lack of resources.
E: In this step, resource availability for the victim is checked. If there is a lack, this means that the DoS attack has been performed successfully.

Attacks: The method in [60] showed that a thread running on an implementation of an SMT processor can suffer from DoS through a malicious thread. This work proposes a number of algorithms to counter such an attack. Some affect the core scheduling algorithm, and others simply attempt to identify an activity that would affect threads sharing the same processor core. The proposed attack in [55] performs packet inspection and injects faults to create a DoS attack. The faults injected are used to trigger a response from error correction code schemes and cause repeated retransmission to starve the victim of network resources and create deadlocks that can lead to failure of a single application or an entire chip. [62] describes a mitigation approach for a DoS diagnosis scheme that detects DoS attacks based on performance degradation of sensitive flows. The proposed method using latency metrics can be leveraged to detect a DoS attack and also to locate the attack source. [56] identifies a DoS attack and proposes a way to protect communication and computation against such attacks by creating continuous secure zones at runtime. These zones are isolated areas in the system that prevent traffic flows from crossing their boundaries and thus provide a space where an application can be executed securely. [57] presents a monitoring system for NoC-based architectures. Its goal is to detect DoS attacks carried out against information in the system. The method in [63] is another DoS mitigation technique that creates security zones according to application security requirements. The approach proposed in [61] can detect DoS attacks on NoC based on evaluating runtime latency. This evaluation makes it possible to monitor the trustworthiness of the NoC throughout the chip lifetime. [33] introduces DoS attacks on the NoCs used in SoC design. These attacks have different implementations, such as full ASIC or full FPGA implementations, with corresponding mitigation strategies. The method proposed in [58] presents a mitigation strategy based on isolation of secure zones that responds to current DoS strategies. [54] proposes a hardware mechanism to secure data transactions between NoC routers. The security mechanism can detect and prevent DoS attacks that aim to degrade system performance. A firewall is demonstrated in [59] for a hardware-based NoC, which performs rule-checking of memory requests at segment level. This firewall aims to protect the NoC against DoS attacks on ARM and Spidergon STNoC.

Figure 6 shows the steps of Functionality attacks and illustrates which types have similar structure. For instance, “FUNC-NODE-INJ” and “IP-NODE-MAN-I” have similar Retrieve steps, and “Func-MEM-MAN-II” and “FUNC-MEM-MAN-III” have similar Trigger and Evaluate steps. As said earlier, this kind of analysis that reveals detailed information on attack structure can lead to building a better mitigation strategy. In the next section, the same analysis is reported for Data attacks.

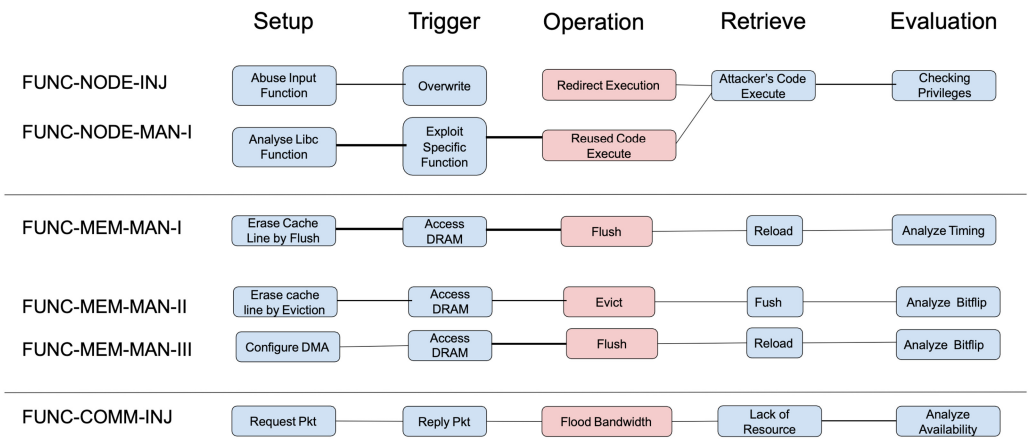

Fig. 6. Attack phases of functionality attacks and groups.

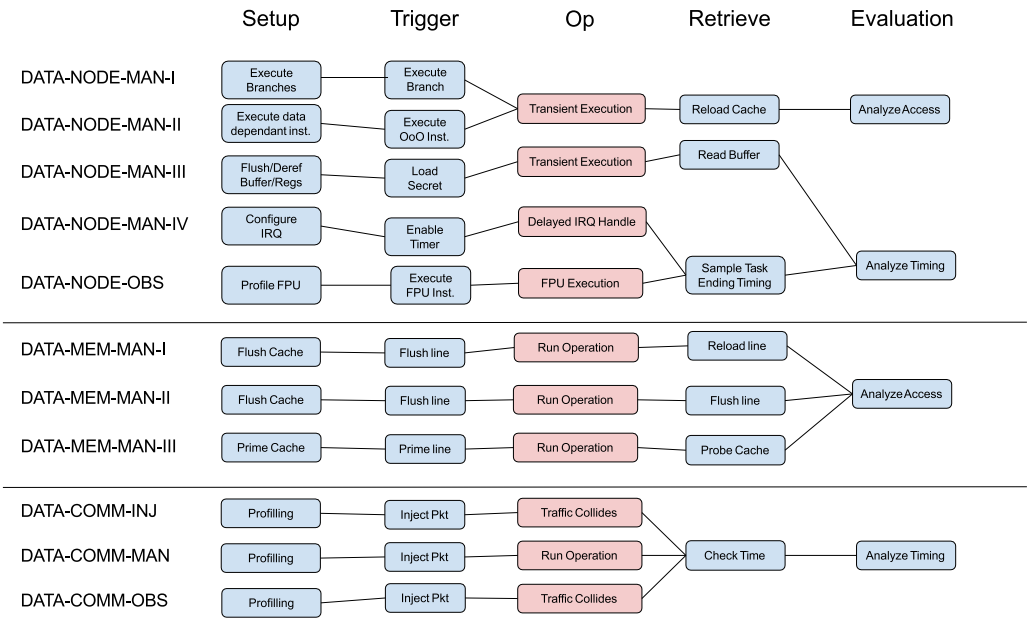

Fig. 7. Attack phases of data attacks and groups.

8 DATA ATTACKS

This section describes various attacks that are mainly performed on data. Figure 4 presents the references for existing Data ArchA under the "Data" branch. An overall view of this section is presented in Figure 7.

8.1 Processing Elements

In the attacks observed in the state of the art that focus on processing elements, seven attack patterns have been identified. Each pattern is described according to the proposed attack phase model and is depicted in Figure 7. In the following discussion, each group is described in more detail.

Group DATA-NODE-MAN-I: The first group consists of attacks on the **branch prediction unit** (**BPU**) of modern processors. In summary, they exploit the speculative behavior of such components to force illegal instructions to execute. Although the processor can detect such mistakes caused by speculation and undo the illegal operation, the processor or the system might have changed. If the attack has been carefully designed, the attacker can hide the stolen information in such different states of the system. The most typical example is to use the cache memory, where access to an address will permanently change the cache state.

Attack Formula: The following list presents the phases of this attack:

S: In the setup phase, the attacker performs many executions, forcing the branches to take the same result always (e.g., taken). As a result, the BPU will be manipulated to speculate to such a condition.

T: The attacker then executes an illegal instruction, now that the BPU has been conditioned to execute it speculatively.

O: In this step, the processor executes the operations of the illegal instruction, and before it finishes, undoes all operations related to exception handling. Such an executed instruction that was later undone is also referred to as a transient execution.

R: The illegal instruction performs an operation reflected in a change to a specific cache address. This address was defined by one bit of sensitive data. The target bit can be used in a shift operation that may point to zero when the bit is zero, or to address 64 when the bit is 1 (assuming a shift by 6).

E: The attacker identifies whether address zero or address 64 has a different state. The used address will present a hit behavior.

Attacks: The approach described in [69] represents a Spectre attack, which is one of the best-known BPU attacks. This attack involves inducing a victim to speculatively perform operations that would not occur during correct program execution and that leak the victim's confidential information through a side channel to the adversary. Netspectre is described in [70] and follows the same idea as Spectre, but over a network. The attack presented in [65] is another type of these attacks, called BranchScope, where the attacker infers the direction of an arbitrary conditional branch instruction in a victim program by manipulating the shared directional branch predictor. Finally, Spectre attack is performed on ARM CPUs in [98], contrary to the literature in which the main focus of Spectre family attack is on ×86 CPUs.

Group DATA-NODE-MAN-II: The second group of attacks consists of out-of-order executions. Any modern processor rearranges the order of instructions to be executed to add a gap between instructions with data dependency. Such filling instructions must be independent of the current flow. Consequently, those filling instructions are executed out of order, and in the last part of the pipeline, a module reorganizes the outputs (i.e., puts them again in order). However, a common vulnerability inside most processors is that illegal operations are only checked in the last part when the reordering happens. This can cause a transient execution. With this understatement, this kind of attack organizes a program where the independent instruction performs an illegal operation with sensitive data. Such an operation typically accesses the cache, modifying its state. Hence, even with the undo performed by the processor, the cache state has already been changed. Finally, the attacker needs to reread the cache and verify at what address the state has changed.

Attack Formula: The phases of this attack are presented below:

S: In the setup phase, the attacker runs several instructions with data dependency.

T: The attacker then executes an illegal instruction independent of the previous instructions. The out-of-order unit will prioritize this independent instruction.

O: In this step, the processor executes the operations of the illegal instruction, and before it finishes, undoes all operations related to exception handling. Such an executed instruction that was undone is also referred to as a transient execution.
R: The illegal instruction performs an operation reflected in a change at a specific cache address. This address was defined by one bit of sensitive data. The target bit can be used in a shift operation which may point to zero when the bit is zero, or to address 64 when the bit is 1 (assuming a shift by 6).
E: The attacker identifies whether address zero or address 64 has a different state. The address used will present a hit behavior.

Attacks: One of the well-known attacks belonging to this group is Meltdown [66]. This is a hardware-based attack that works on different Intel microarchitectures and exploits the side effects of out-of-order execution on processors to read arbitrary kernel-memory locations, including personal data and passwords. In [16], two new versions of Meltdown were proposed: Meltdown-PK (Protection Key Bypass) on Intel, and Meltdown-BND (Bounds Check Bypass) on Intel. In contrast, the approach in [64] presents Foreshadow, a software-only micro-architectural attack that dismantles the security objectives of current SGX implementations.

Group DATA-NODE-MAN-III: This type of attack targets the internal storage elements of the processor. Both registers and buffers can be exploited by these attacks, which are mainly used to load and store memory data. Some examples are the line-fill buffers, load ports, and **store buffers** (**SBs**). SBs are internal buffers used to track pending stores and in-flight data in optimizations such as store-to-load forwarding. Some modern processors enforce strong memory ordering, where load and store instructions that refer to the same physical address cannot be executed out of order. However, because address translation is a slow process, the physical address might not be available yet. The processor performs memory disambiguation to predict whether particular load and store instructions refer to the same physical address. This process enables the processor to execute load and store instructions out of order in a speculative manner. As a micro-optimization, if the load and store instructions are ambiguous, the processor can speculatively store-to-load forward the data from the store buffer to the load buffer. On the other hand, **line fill buffers** (**LFBs**) are internal buffers that the CPU uses to keep track of outstanding memory requests and perform a number of optimizations such as merging multiple in-flight stores. Sometimes, data may already be available in the LFBs, and as a micro-optimization, the CPU can speculatively load these data (similar optimizations are also performed on, e.g., store buffers). In both cases, modern CPUs that implement aggressive speculative execution may speculate without any awareness of the virtual or physical addresses involved [72].

Attack Formula: The following list presents the phases of this attack:

S: In this step, the victim, as part of normal execution, loads or stores some secret data. The secret data can be a piece of leaked data from inactive code obtained by forcing cache evictions.
T: In Trigger, the attacker also performs a load such that the processor speculatively uses data from the LFBs rather than valid data.
O: When the processor eventually detects the incorrect speculative load from the previous step, it discards any and all modifications to registers or memory and restarts execution with the right value.
R: Because traces of the speculatively executed load still exist at the microarchitectural level, it is possible to observe the leaked data using a simple (Flush+Reload). Thus, the attacker reloads the data (memory line), together with the information related to measuring the time to load it.
E: In the Evaluate phase, the timing behavior is analyzed.

Attacks: RIDL is an attack of this type introduced in [72]. RIDL attacks are implemented from linear execution with no invalid page faults, eliminating the need for exception suppression mechanisms and enabling system-wide attacks from arbitrary unprivileged code (including JavaScript in the browser). Fallout is another attack presented in [68]. Fallout is a transient execution attack that leaks information from a previously unexplored store buffer. An unprivileged user process can exploit Fallout to reconstruct privileged information recently written by the kernel.

Group DATA-NODE-MAN-IV: This group consists of attacks that exploit interruption handling to monitor the time of sensitive operations indirectly. When a high-priority interruption occurs, the processor finishes the current execution and changes its context to interruption handling. However, depending on the operation that has been executed inside the processor, the interruption handling will have different execution times. These time differences result from a delayed process to change the context, which correlates with the target operation. Consequently, an attacker can sample all these timings and perform an analytical approach to infer a secret.

Attack Formula: The phases of this attack are presented below.

S: Configure an interruption component to periodically activate.
T: Enable the component responsible for interrupting the processor. Typically, a timer is used in this step.
O: The processor receives the IRQ signal, finishes the current instruction execution, changes the processor context, and executes the IRQ handling function. Such steps may result in different timings due to the instruction and data being operated upon.
R: Save the time to finish the IRQ handling.
E: After several attempts, all saved time data can be analyzed to infer sensitive information. A computation based on correlation can be applied here.

Attacks: Only one attack has been published on this domain. The authors in [71] present Nemesis, an attack that abuses the CPU's interrupt mechanism to leak instruction timings from CPU executions. Basically, the approach uses a timer to create an interruption-based attack.

Group DATA-NODE-OBS: This identified group targets co-processors or hardware accelerators. Commonly, specialized hardware components are designed to obtain maximum performance and therefore often present different timing behavior depending on the input data. Regarding processing elements, the most common attack targets **floating-point units** (**FPUs**).

Attack Formula: The phases of this attack are presented below.

S: Profile the hardware component. In the case of FPU, random inputs are applied, and the different time behavior is saved.
T: The processor is manipulated to compute sensitive data using the target component (e.g., FPU).
O: The processor performs operations in the target hardware component using sensitive data. This results in different execution time due to hardware optimization.
R: The time to operate is saved.
E: The saved data are analyzed using the profile as a base.

Attacks: A benchmark has been developed in [73] to measure the timing variability of floating-point operations and report on the results. The approach uses floating-point data timing variability to demonstrate practical attacks on the security of the Firefox browser (versions 23 through 27) and the Fuzz private database. In [74], an attack called LazyFPU that exploits the ×87 FPU is presented. This attack enables an adversary to recover the FPU and the SIMD register sets of arbitrary processes. The attack works on processors that transiently execute FPU or SIMD instructions that follow an instruction generating the fault, indicating the first use of FPU or SIMD instructions.

8.2 Memory Elements

In this section, various attacks based on manipulation of cache memory are described.

Group DATA-MEM-MAN-I: The first group consists of the Flush-and-Reload family, which relies on sharing pages between attacker and victim processes. With shared pages, the attacker can ensure that a specific memory line is evicted from the whole cache hierarchy. The spy uses this capability to monitor access to the memory line.

Attack Formula: The phases of this attack are presented below.

- S: In the setup phase, the attacker gains memory access to monitor the memory, and the memory line is flushed from the cache hierarchy.
- T: The attacker waits to allow the victim time to access the memory line before the next phase.
- O: This step represents an action that is taken by the victim and leads the attacker to obtain the desired information.
- R: In the Retrieve phase, the attacker reloads the memory line, together with the information related to measuring the time to load it.
- E: In the Evaluate phase, the timing behavior is analyzed. If during the Trigger phase (waiting), the victim accesses the memory line that is available in the cache, the reload operation takes a short time. If, on the other hand, the victim has not accessed the memory line, the line must be brought in from memory, and the reload takes significantly longer. From this observation, the attacker can find out whether the victim has gained access to the cache.

Attacks: A Flush+Reload attack was presented for the first time in [78]. In this work, the attack was studied on a single-core processor and exploited the ability of the adversary process to evict data from the physical memory pages it shares with the victim process from the CPU cache (e.g., via the instruction clflush). In [75], another kind of Flush+Reload attack is presented. The difference is that, unlike most other attacks, this one can make use of side-channel information from almost all observed executions. This means that it obtains private key recovery by observing a relatively small number of executions. The approach in [85] uses the Flush+Reload attack as a primitive and extends it by leveraging within an automaton-driven strategy for tracing a victim's execution. This attack was executed between tenants on commercial **Platform-as-a-Service** (**PaaS**) clouds. [79] instead demonstrated Flush+Reload cache attacks on a virtual machine. Finally, [20, 76, 77, 82] implemented Flush+Reload attacks in three steps and proposed a detection strategy for such attacks.

Group DATA-MEM-MAN-II: There exists another cache attack technique called Flush+Flush. This attack relies only on the difference in timing of the flush instruction between cached and non-cached memory accesses. In contrast to other cache attacks, it does not perform any memory accesses. Instead, it builds upon the observation that the flush instruction leaks information on the state of the cache.

Attack Formula: The phases of this attack are presented below:

- S: In the setup phase, the attacker gains access to memory to monitor it, and the memory line is flushed from the cache hierarchy.
- T: The attacker then waits to allow the victim time to access the memory line before the next phase.
- O: An action is performed by the victim, which leads the attacker to obtain the desired information.
- R: In the Retrieve phase, the attacker flushes the memory line, together with the information related to measuring the time to load it.
- E: In the Evaluate phase, the timing behavior is analyzed. The attacker measures the execution time of the flush instruction. Based on the execution time, the attacker decides whether the

memory line has been cached. Because the attacker does not load the memory line into the cache, this reveals whether some other process has loaded it. At the same time, clflush evicts the memory line from the cache for the next loop round of the attack. At the end of an attack round, the program optionally yields times to lower system utilization and waits for a second process to perform memory accesses.

Attacks: In [80], a Flush+Flush attack was demonstrated. This attack is applicable in multi-core and virtualized environments if read-only shared memory with the victim process can be acquired. The authors in [82] performed a Flush+Flush attack on an open source cache memory [96] and demonstrated a mitigation strategy for this attack based on address randomization.

Group DATA-MEM-MAN-III: Evict+Time and Prime+Probe are other attacks that extract time measurement information by manipulating the state of the cache before each encryption and observing the execution time of the subsequent encryption.

Attack Formula: The phases of this attack are presented below:

S: In the setup phase, the spy runs a spy process that monitors the victim's cache usage. Then the spy fills one or more cache sets with its own code or data (this is called priming the cache).

T: The spy waits while the victim executes and utilizes the cache.

O: This step represents an action that is taken by the victim and leads the attacker to obtain the desired information.

R: Retrieve: The spy continues execution (probing the cache) and measures the time to load each primed set of data or code.

E: In the Evaluate phase, the timing behavior is analyzed. If the victim has accessed some cache sets, it will have evicted some of the spy's lines, which the spy observes as increased memory access latency for those lines. As the Probe phase accesses the cache, it doubles as a Prime phase for subsequent observations.

Attacks: One of the most general techniques for Evict+Time and Prime+Probe is presented in [81], which describes their execution in a formal format. For instance, in Evict+Time, the attack extracts time measurement information by manipulating the state of the cache before each encryption and observing the execution time of the subsequent encryption. The article assumes the ability to trigger an encryption and to know when it has begun and ended. It also assumes knowledge of the memory address of each lookup table, and hence of the cache sets to which the table is mapped. In [83], the measurement method is the same as in the [81] Prime+Probe attack, but the analytical theory is based on the misalignment of AES lookup tables over the L1 data cache and points out a way to detect the lookup tables resident in memory. [84] presents another Prime+Probe cache side-channel attack that can prime physical addresses. These addresses are translated from the virtual addresses used by a virtual machine. The time to access these addresses is then measured and will vary according to where the data are located. If they are in the CPU cache, the time will be less than if they are in main memory. The attack described in this article was implemented in a server machine that was comparable to cloud environment servers. The technique described in [77] is again based on Prime+Probe as described in [81], but it is performed on the last-level caches on a virtual machine.

8.3 Communication Elements

This subsection describes attacks on the communication structure with the aim of retrieving sensitive data. Basically, this type of attack targets the latency or injection throughput to identify the features of sensitive traffic. Examples of features are communication affinity (i.e., the nodes with which a specific node communicates), communication data rates, size of messages, and so forth.

With this information, an attacker can model communication behavior and infer the type of application that is running, which nodes in the system are important, and the role of each node. However, there are different methods to accomplish these attacks, by creating the conditions (injection category), by manipulating the system to obtain the desired conditions (manipulation category), or by simply observing normal system behavior (observation category). These three main groups of attacks are described below.

Group DATA-COMM-INJ: The first group consists of attacks where the message is crafted to force communication collisions. As a result of these collisions, the attacker can infer and extract sensitive information. In this group, this paper focuses on attacks that modify message priorities. Different priorities can change communication behavior by creating back-pressure on the buffers inside the routers. As a result, the attacker can do fine-tuning to observe sensitive traffic behavior.

Attack Formula: The phases of this attack are presented below:

- S: In the setup phase, the attacker installs itself into a node and starts injecting packets with different priorities to understand normal behavior.
- T: The attacker then chooses the appropriate priority to create the expected collision.
- O: In this step, the system exchanges messages under various priorities. The target traffic is somehow affected or affects the attacker's message.
- R: Retrieve: In the Retrieve phase, the attacker measures and analyzes each injection to observe drops in performance. There is also an attack where the attacker observes a rise in performance when successful.
- E: The various timing information collected is correlated with the setup used to attack. Based on the delay in sending packets, the priority used, and the periodicity of target traffic, important features can be extracted.

Attacks: In [86], two attacks that manipulate the priority of packets are described. The first one is called indirect congestion and forces collisions in the NoC when the attacker injects high-priority packets. The second attack considers an attacker with no privilege to create high-priority packets. In this case, the attacker forces a condition called backpressure and observes when packet injection becomes faster due to release of buffers in the path.

Group DATA-COMM-MAN: The second group consists of attacks that use the communication structure as a means to improve other architectural attacks that focus on nodes or memories. In this group, the attacker exploits the communication structure to identify the right instant to perform some malicious operation. As a result, it improves classical attacks by several orders of magnitude.

Attack Formula: The phases of this attack are presented below:

- S: In the setup phase, the attacker injects random packets (i.e., random size and random destination) to understand normal latency when injecting a packet into the communication structure. In addition, in this phase, the attacker can install malware in another node to observe other traffic that corresponds to target nodes or memories.
- T: The attacker then asks for service from the target node.
- O: The target node uses the communication structure to accomplish its tasks (e.g., retrieve data from shared memory).
- R: In the Retrieve phase, the attacker injects messages in such a way that they collide with the target node paths.
- E: In the Evaluate phase, the differences in timing behavior show the presence of sensitive traffic. As a result, the attacker can infer in which part of the algorithm the target node is. Next, the attacker can reuse known methodologies to perform node timing attacks or memory access attacks.

Attacks: From this group, one example is Earthquake [87], an attack that uses the NoC to optimize the Differential Cache attack from Bogdanov. Results have shown that when Earthquake was used, the original cache attack became even more powerful because the original required high efficiency to work. [89] and [88] presented methodologies to use the NoC to improve cache access attacks like Prime+Probe. MeshUp, which is introduced in [99], exploits the timing difference caused by CPU mesh interconnection contrary to common side-channel attacks that rely on the timing difference between miss and hit. Finally, [90] demonstrated that the same methodology used for NoCs can be used in bus-based systems as well.

Group DATA-COMM-OBS: The last group refers to pure communication timing attacks, where the attacker only observes system behavior and retrieves information.

Attack Formula: The phases of this attack are presented below.

S: In the setup phase, the attacker injects random packets (i.e., random size and random destination) to understand the normal latency of injecting a packet into the communication structure. In addition, in this phase, the attacker can install malware in another node to observe other traffic.
T: The attacker directly observes the traffic in the system.
O: The system nodes and memories use the communication structure to accomplish their tasks.
R: In the Retrieve phase, the attacker injects messages that will collide with system traffic.
E: Evaluate: In the Evaluate phase, the latency to deliver a message is analyzed. If the time takes more than the normal measured during the setup phase, the attacker considers that this is sensitive traffic. The amount of latency added, the duration of this behavior, and the instance in time can reveal important information like message size, data rate, and application type or behavior. The attacker can replicate itself in the system to a different node to obtain better efficiency in inferring system details.

Attacks: [92, 93] presented the first methodologies regarding Network-on-Chip timing attacks. They showed how injection latency could be explored as a leakage in this system. Later, [91] detailed their methodologies by applying a practical use case. In addition, the authors in [91] extended the NoC timing attack to a distributed timing attack, where other infected nodes could help to create controlled congestion in the network, improving the attacker's observation capabilities. In these attacks, the objective is to observe communication behavior to infer sensitive traffic information.

9 DISCUSSIONS

This section discusses the importance and benefits of having such a taxonomy and an attack model. As described, the proposed taxonomy has been created by exhaustively studying all architectural attacks (called ArchA in this survey). Subsequently, it was observed that all ArchA can be generally categorized based on three main metrics: What, Where, and How. "What" basically refers to the objective of the attacks, which can be classified as *obtaining intellectual property information*, *modifying functionality*, and *stealing data*. "Where" refers to the locations where attackers perform the attacks. In this survey, these locations are identified as *Processing elements*, *Memories*, and *Communication components*. "How" describes the method by which attackers execute attacks. These methods are classified as *Injection*, *Manipulation*, and *Observation*.

The proposed taxonomy exceeds the state of the art in several aspects: (i) by analyzing all different objectives, unlike the work in [8], which is only focused on attacks that steal data; (ii) by considering different possible locations, unlike the approaches in [9, 10, 12], which are focused only on caches; and (iii) by introducing a new concept that classifies ArchA based on the method

used to execute the attacks. To the best of the authors' knowledge, this classification is introduced for the first time in this work.

On the other hand, this survey introduced an innovative attack model called STORE, which identifies five main phases for all ArchA executions. Accordingly, STORE can be described as follows: a Setup phase for attack preparation, a Trigger phase consisting of actions to activate vulnerability, an Operate phase for execution and exposing vulnerabilities, a Retrieve phase for collecting essential information for attacks, and finally an Evaluate phase for analyzing the gathered information to attain the final attack objective. The proposed attack model is essential because the same attacks in the state of the art are introduced with different numbers of phases. This means that authors have different views when analyzing attacks. For instance, [17] divided the Spectre attack into three steps, whereas the approaches in [22] and [16] categorized it into two and five phases, respectively. Another example is that the authors of [14] and [97] divided Flush+Reload into three phases, but their definitions of these phases were completely different. However, STORE analyzed all existing ArchA from one point of view and in consistent phases.

In summary, having such a complete and unified taxonomy and attack model leads to greater ease in elaborating countermeasures. Moreover, it contributes to developing more accurate security tools and predictions of possible attacks.

10 CONCLUSIONS

This article has presented a novel taxonomy for ArchA that uses three metrics, What, Where, and How, to analyze each attack type. These metrics together with the proposed taxonomy go beyond the current state of the art by categorizing all existing ArchA into unified groups. This article furthermore introduced an appropriate attack model consisting of five phases: Setup, Trigger, Operate Retrieve, and Evaluate, to formally describe the sequence of actions in any ArchA. In a manner similar to the taxonomy problem, most attack models only cover a subset of attacks, and they are mainly focused on caches. In addition, it should be stressed that having such a clear taxonomy and a unified attack model leads to a better understating of attack behavior, thus facilitating development of a system that can mitigate such attacks.

REFERENCES

[1] Fortune Business Insights. 2022. Cyber Security Market Size. Retrieved from https://www.fortunebusinessinsights.com/industry-reports/cyber-security-market-101165

[2] Cybersecurity and Infrastructure security agency. 2021. 5G Security and Resilience. Retrieved in 2022 from https://www.cisa.gov/5g.

[3] Alvaro A. Cárdenas, Tanya Roosta, Gelareh Taban, and Shankar Sastry. 2008. Cyber security basic defenses and attack trends. *Homeland Security Technology Challenges* (2008), 73–101.

[4] Julian Jang-Jaccard and Surya Nepal. 2014. A survey of emerging threats in cybersecurity. *J. Comput. System Sci.* 80, 5 (2014), 973–993. DOI: https://doi.org/10.1016/j.jcss.2014.02.005

[5] Vijay Tiwari and Dwivedi. 2016. Analysis of cyber attack vectors. In *Proceedings of the 2016 International Conference on Computing, Communication and Automation (ICCCA'16)*. IEEE. DOI: https://doi.org/10.1109/CCAA.2016.7813791

[6] Yoohwan Kim, Wing Cheong Lau, Mooi Choo Chuah, and H. J. Chao. 2006. PacketScore: A statistics-based packet filtering scheme against distributed denial-of-service attacks. *IEEE Trans. Dependable Secure Comput.* 3, 2 (2006), 141–155. DOI: https://doi.org/10.1109/TDSC.2006.25

[7] Huanyu Wang, Henian Li, Fahim Rahman, Mark M. Tehranipoor, and Farimah Farahmandi. 2022. SoFI: Security property-driven vulnerability assessments of ICs against fault-injection attacks. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 41, 3 (2022), 452–465. DOI: https://doi.org/10.1109/TCAD.2021.3063998

[8] Andrey Bogdanov, Thomas Eisenbarth, Christof Paar, and Malte Wienecke. 2010. Differential cache-collision timing attacks on AES with applications to embedded CPUs. In *Topics in Cryptology (CT-RSA'10)*, Josef Pieprzyk (Ed.). Springer, Berlin, 235–251.

[9] Anne Canteaut, Cédric Lauradoux, and André Seznec. 2006. *Understanding Cache Attacks*. Research Report RR-5881. INRIA. https://hal.inria.fr/inria-00071387.

[10] Tianwei Zhang and Ruby Lee. 2014. *Secure Cache Modeling for Measuring Side-Channel Leakage*. Technical Report, Princeton University.
[11] 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. *IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 1–19. DOI: 10.1109/SP.2019.00002
[12] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.* 8, 1 (2018), 1–27.
[13] Yangdi Lyu and Prabhat Mishra. 2018. A survey of side-channel attacks on caches and countermeasures. *J. Hardware Syst. Sec.* 2, 1 (2018), 33–50.
[14] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2019. *Analysis of Secure Caches using a Three-Step Model for Timing-Based Attacks*. Cryptology ePrint Archive, Report 2019/167. Retrieved from https://eprint.iacr.org/2019/167.
[15] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX*.
[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 *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*. ACM.
[17] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre attacks: Exploiting speculative execution. In *IEEE SP*.
[18] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: The case of AES. In *CT-RSA 2006*. Springer, 1–20.
[19] David Budgen and Pearl Brereton. 2006. Performing systematic literature reviews in software engineering. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*. Association for Computing Machinery, New York, NY, 1051–1052. DOI: https://doi.org/10.1145/1134285.1134500
[20] Cezar Reinbrecht, Said Hamdioui, Mottaqiallah Taouil, Behrad Niazmand, Tara Ghasempouri, Jaan Raik, and Johanna Sepülveda. 2020. LiD-CAT: A lightweight detector for cache attacks. In *Proceedings of the 2020 IEEE European Test Symposium (ETS'20)*, 1–6. DOI: https://doi.org/10.1109/ETS48528.2020.9131603
[21] Daniel J. Bernstein. 2005. Cache Timing Attacks on AES. Retrieved December 12, 2016.
[22] Joseph Bonneau and Ilya Mironov. 2006. Cache-collision timing attacks against AES. In *CHES*. 201–215.
[23] Christopher Domas. 2017. Breaking the x86 ISA. Retrieved in 2022 from https://www.blackhat.com/docs/us-17/thursday/us-17-Domas-Breaking-The-x86-ISA.pdf
[24] Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison, Robert Gawlik, Christof Paar, and Thorsten Holz. 2017. Reverse engineering ×86 processor microcode. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*, 1163–1180.
[25] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS'10)*. IEEE, 235–246. DOI: https://doi.org/10.1109/ISPASS.2010.5452013
[26] Christopher Domas. 2018. GOD MODE unlocked: Hardware backdoors in x86 CPUs. Retrieved in 2022 from https://i.blackhat.com/us-18/Thu-August-9/us-18-Domas-God-Mode-Unlocked-Hardware-Backdoors-In-x86-CPUs.pdf
[27] Andreas Abel and Jan Reineke. 2013. Measurement-based modeling of the cache replacement policy. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*. IEEE, 65–74. DOI: https://doi.org/10.1109/RTAS.2013.6531080
[28] C. L. Coleman and J. W. Davidson. 2001. Automatic memory hierarchy characterization. In *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'01)*. IEEE, New York, NY, 103–110. DOI: https://doi.org/10.1109/ISPASS.2001.990684
[29] Jack Dongarra, Shirley Moore, Philip Mucci, Keith Seymour, and Haihang You. 2004. Accurate cache and TLB characterization using hardware counters. In *Computational Science (ICCS'04)*, Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra (Eds.). Springer, Berlin, Heidelberg, 432–439.
[30] Mohamed Hassan, Anirudh M. Kaushik, and Hiren Patel. 2015. Reverse-engineering embedded memory controllers through latency-based analysis. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 297–306. DOI: https://doi.org/10.1109/RTAS.2015.7108453
[31] Mohamed Hassan, Anirudh M. Kaushik, and Hiren Patel. 2018. Exposing implementation details of embedded DRAM memory controllers through latency-based analysis. *ACM Trans. Embed. Comput. Syst.* 17, 5 (Oct. 2018), Article 90, 25 pages. DOI: https://doi.org/10.1145/3274281
[32] Matthias Jung, Carl C. Rheinlä, Christian Weis, and Norbert Wehn. 2016. Reverse engineering of DRAMs: Row hammer with crosshair. In *Proceedings of the 2nd International Symposium on Memory Systems*. Association for Computing Machinery, New York, NY, 471–476. 9781450343053.

[33] S. Evain and J.-P. Diguet. 2005. From NoC security analysis to design solutions. In *Proceedings of the IEEE Workshop on Signal Processing Systems Design and Implementation, 2005*, 166–171. DOI: https://doi.org/10.1109/SIPS.2005.1579858
[34] One Aleph. 1996. Smashing the stack for fun and profit. Retrieved in 2022 from http://www.shmoo.com/phrack/Phrack49/p49-14.
[35] Steven Alexander. 2005. Defeating compiler-level buffer overflow protection. The USENIX Magazine; login (2005). Retrieved in 2022 from https://www.usenix.org/publications/login/june-2005-volume-30-number-3/defeating-compiler-level-buffer-overflow-protection
[36] Arash Baratloo, Navjot Singh, and Timothy K. Tsai. 2000. Transparent run-time defense against stack-smashing attacks. In *Proceedings of the USENIX Annual Technical Conference, General Track*, 251–262.
[37] Sandeep Bhatkar, Daniel C. DuVarney, and Ron Sekar. 2003. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *PRoceedings of the USENIX Security Symposium, Vol. 12*, 291–301.
[38] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceeding of the USENIX Security Symposium*, 161–176.
[39] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium, Vol. 98*. 63–78.
[40] Yongje Lee, Ingoo Heo, Dongil Hwang, Kyungmin Kim, and Yunheung Paek. 2015. Towards a practical solution to detect code reuse attacks on ARM mobile devices. In *Proceedings of the 4th Workshop on Hardware and Architectural Support for Security and Privacy (HASP'15)*. Association for Computing Machinery, New York, NY, Article 3, 8. DOI: https://doi.org/10.1145/2768566.2768569
[41] Hilmi Ozdoganoglu, T. N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. 2006. SmashGuard: A hardware solution to prevent security attacks on the function return address. *IEEE Trans. Comput.* 55, 10 (2006), 1271–1285.
[42] Gerardo Richarte. 2002. Four different tricks to bypass stackshield and stackguard protection. Retrieved in 2022 from https://www.cs.purdue.edu/homes/xyzhang/spring07/Papers/defeat-stackguard.pdf
[43] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and Communications Security*. ACM, 298–307.
[44] Nathan P. Smith. 1997. Stack smashing vulnerabilities in the UNIX operating system. Retrieved in 2022 from https://www.zdnet.com/article/stackclash-vulnerabilities-rip-root-holes-in-linux-systems/
[45] Ana Nora Sovarel, David Evans, and Nathanael Paul. 2005. Where's the FEEB? The effectiveness of instruction set randomization. In *Proceedings of the USENIX Security Symposium*, Vol. 10.
[46] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is dead: Long live KASLR. In *Proceedings of the International Symposium on Engineering Secure Software and Systems*. Springer, 161–176.
[47] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer. js: A remote software-induced fault attack in javascript. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 300–321.
[48] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking down the processor via Rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. ACM, 5.
[49] 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 SIGARCH Computer Architecture News*, Vol. 42. IEEE Press, IEEE, 361–372.
[50] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. 2020. Nethammer: Inducing rowhammer faults through network requests. *IEEE European Symposium on Security and Privacy Workshops (EuroSPW)*, Genoa, Italy. 710–719, DOI: 10.1109/EuroSPW51379.2020.00102
[51] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *Proceedings of the USENIX Security Symposium*, 565–581.
[52] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Throwhammer: Rowhammer attacks over the network and defenses. In *Proceedings of the 2018 USENIX Annual Technical Conference*. USENIX Association.
[53] 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*. ACM, 1675–1689.
[54] A. Ben Achballah, S. Ben Othman, and S. Ben Saoud. 2017. Toward on hardware firewalling of networks-on-chip based systems. In *Proceedings of the 2017 International Conference on Advanced Systems and Electric Technologies (IC_ASET'17)*. IEEE, 7–13. DOI: https://doi.org/10.1109/ASET.2017.7983658

[55] T. Boraten and A. K. Kodi. 2016. Mitigation of denial of service attack with hardware trojans in NoC architectures. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*. IEEE, 1091–1100. DOI: https://doi.org/10.1109/IPDPS.2016.59

[56] L. L. Caimi, V. Fochi, E. Wachter, D. Munhoz, and F. G. Moraes. 2017. Activation of secure zones in many-core systems with dynamic rerouting. In *Proceedings of the 2017 IEEE International Symposium on Circuits and Systems (ISCAS'17)*. IEEE, 1–4. DOI: https://doi.org/10.1109/ISCAS.2017.8050256

[57] Leandro Fiorin, Gianluca Palermo, and Cristina Silvano. 2008. A security monitoring service for NoCs. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'08)*. ACM, New York, NY, 197–202. DOI: https://doi.org/10.1145/1450135.1450180

[58] Leandro Fiorin, Cristina Silvano, and Mariagiovanna Sami. 2007. Security aspects in networks-on-chips: Overview and proposals for secure implementations. In *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD'07)*. IEEE, 539–542.

[59] M. D. Grammatikakis, K. Papadimitriou, P. Petrakis, A. Papagrigoriou, G. Kornaros, I. Christoforakis, O. Tomoutzoglou, G. Tsamis, and M. Coppola. 2015. Security in MPSoCs: A NoC firewall and an evaluation framework. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 34, 8 (2015), 1344–1357. DOI: https://doi.org/10.1109/TCAD.2015.2448684

[60] Dirk Grunwald and Soraya Ghiasi. 2002. Microarchitectural denial of service: Insuring microarchitectural fairness. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 35)*. IEEE Computer Society Press, 409–418. 0-7695-1859-1.

[61] J. S. Rajesh, Dean Michael Ancajas, Koushik Chakraborty, and Sanghamitra Roy. 2015. Runtime detection of a bandwidth denial attack from a rogue network-on-chip. In *Proceedings of the 9th International Symposium on Networks-on-Chip*. ACM, 8.

[62] J. Sepúlveda, M. Strum, and W. J. Chau. 2010. A hybrid switching approach for NoC-based systems to avoid denial-of-service SoC attacks. In *Proceedings of the 16th Iberchip Workshop (IWS'10)*. 23–25.

[63] J. Sepúlveda, D. Flórez, and G. Gogniat. 2015. Efficient and flexible NoC-based group communication for secure MPSoCs. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, Riviera Maya, Mexico, 1–6. DOI: https://doi.org/10.1109/ReConFig.2015.7393301

[64] 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 18)*. USENIX Association, Baltimore, MD, 991–1008. 978-1-939133-04-5 https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[65] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A new side-channel attack on directional branch predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(seriesASPLOS'18)*. ACM, New York, NY, 693–707. DOI: https://doi.org/10.1145/3173162.3173204

[66] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. DOI: https://doi.org/10.48550/ARXIV.1801.01207

[67] Giorgi Maisuradze and Christian Rossow. 2018. Speculose: Analyzing the security implications of speculative execution in CPUs. *CoRR* abs/1801.04084 (2018), 10–30.

[68] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Berk Sunar, Frank Piessens, and Yuval Yarom. 2019. Fallout: Reading kernel writes from user space. (2019).

[69] Koruyeh Esmaeil Mohammadian, Khasawneh Khaled N., Song Chengyu, and Abu-Ghazaleh Nael. 2018. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies WOOT 18 (seriesWOOT'18)*. USENIX Association, Baltimore, 10–30.

[70] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2018. Netspectre: Read arbitrary memory over network. *arXiv preprint arXiv:1807.10535* (2018).

[71] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Canada) *(seriesCCS'18)*. Association for Computing Machinery, 178–195. DOI: https://doi.org/10.1145/3243734.3243822

[72] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue in-flight data load. In *S&P*. IEEE, CA.

[73] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On subnormal floating point and abnormal timing. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, San Jose, CA, 623–639.

[74] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480* (2018).
[75] Naomi Benger, Joop Pol, Nigel P. Smart, and Yuval Yarom. 2014. Ooh Aah... Just a Little Bit: A Small Amount of Side Channel Can Go a Long Way. In *CHES*. New York, NY, 75–92.
[76] Tara Ghasempouri, Jaan Raik, Kolin Paul, Cezar Reinbrecht, Said Hamdioui, and Mottaqiallah. 2021. Verifying cache architecture vulnerabilities using a formal security verification flow. *Microelectron. Reliab.* 119, (2021), 114085. DOI : https://doi.org/10.1016/j.microrel.2021.114085
[77] Tara Ghasempouri, Jaan Raik, Kolin Paul, Cezar Reinbrecht, Said Hamdioui, and Mottaqiallah Taouil. 2020. A security verification template to assess cache architecture vulnerabilities. In *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. 1–6. DOI : https://doi.org/10.1109/DDECS50862.2020.9095707
[78] D. Gullasch, E. Bangerter, and S. Krenn. 2011. Cache games - bringing access-based cache attacks on AES to practice. In *IEEE SP*. 490–505.
[79] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. *Wait a Minute! A fast, Cross-VM Attack on AES*. Springer International Publishing, 299–319.
[80] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. 2015. Last-level cache side-channel attacks are practical. In *IEEE SP*. San Jose, CA, 605–622.
[81] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and countermeasures: The case of AES. In *Topics in Cryptology - CT-RSA 2006*. Springer, 1–20.
[82] Ameer Shalabi, Tara Ghasempouri, Peeter Ellervee, and Jaan Raik. 2020. SCAAT: Secure Cache Alternative Address Table for mitigating cache logical side-channel attacks. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*. IEEE, Kranj, Slovenia, 213–217. DOI : https://doi.org/10.1109/DSD51259.2020.00043
[83] Z. Xinjie, W. Tao, M. Dong, Z. Yuanyuan, and L. Zhaoyang. 2008. Robust first two rounds access driven Cache timing attack on AES. In *CSSE, Vol. 3*. Hubei, China, 785–788.
[84] Younis A. Younis, Kashif Kifayat, Qi Shi, and Bob Askwith. 2015. A new prime and probe cache side-channel attack for cloud computing. In *CSIT*, 1718–1724.
[85] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-tenant side-channel attacks in PaaS clouds. In *ACM SIGSAC*. Scottsdale, Arizona, 990–1003.
[86] B. Forlin, C. Reinbrecht, and J. Sepúlveda. 2019. Attacking real-time MPSoCs: Preemptive NoCs are vulnerable. In *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, Cuzco, Peru, 204–209.
[87] C. Reinbrecht, B. Forlin, A. Zankl, and J. Sepú. 2018. Earthquake – A NoC-based optimized differential cache-collision attack for MPSoCs. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 648–653. DOI : https://doi.org/10.23919/DATE.2018.8342090
[88] C. Reinbrecht, A. Susin, L. Bossuet, G. Sigl, and J. Sepú. 2016. Side channel attack on NoC-based MPSoCs are practical: NoC Prime+Probe attack. In *2016 29th Symposium on Integrated Circuits and Systems Design (SBCCI)*, 1–6. DOI : https://doi.org/10.1109/SBCCI.2016.7724051
[89] Cezar Reinbrecht, Altamiro Susin, Lilian Bossuet, Georg Sigl, and Johanna Sepúlveda. 2017. Timing attack on NoC-based systems: Prime+Probe attack and NoC-based protection. *Microprocess. Microsys.* 52, (2017), 556–565. DOI : https://doi.org/10.1016/j.micpro.2016.12.010
[90] J. Sepú, M. Gross, A. Zankl, and G. Sigl. 2017. Exploiting bus communication to improve cache attacks on systems-on-chips. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 284–289. DOI : https://doi.org/10.1109/ISVLSI.2017.57
[91] C. Reinbrecht, A. Susin, L. Bossuet, and J. Sepulveda. 2016. Gossip NoC - Avoiding timing side-channel attacks through traffic management. In *ISVLSI 16*. IEEE, Pittsburgh, 601–606.
[92] J. Sepulveda, J. Diguet, M. Strum, and G. Gogniat. 2015. NoC-based protection for SoC time-driven attacks. *Embedded Systems Letters, IEEE* 7, 1 (2015), 7–10.
[93] W. Yao and E. Suh. 2012. Efficient timing channel protection for on-chip networks. In *NOCS*. Lyngby, Denmark, 142–151.
[94] CERT Coordination Center 2015. Software Engineering Institute, Carnegie Mellon University. Retrieved from https://www.sei.cmu.edu/about/divisions/cert/index.cfm.
[95] Yefeng Ruan, Sivapriya Kalyanasundaram, and Xukai Zou. 2016. Survey of return-oriented programming defense mechanisms. *Secur. Commun. Netw.* 9, 10 (2016), 1247–1265.
[96] Chair of VLSI Design, Diagnostics and Architecture. 2016. *PoC - Pile of Cores*. Technische Universitä. https://github.com/VLSI-EDA/PoC.
[97] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A fast and stealthy cache attack. In *13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.
[98] Hetterich Lorenz and Schwarz Michael. 2022. Detection of intrusions and malware, and vulnerability assessment. In *19th International Conference*, DIMVA, Cagliari, Italy.

[99] Wan Junpeng, Bi Yanxiang, Zhou Zhe, and Li Zhou. 2022. MeshUp: Stateless cache side-channel attack on CPU mesh. In *IEEE Symposium on Security and Privacy (SP)*.

[100] Ravichandran Joseph, Na Weon Taek, Lang Jay, and Yan Mengjia. 2022. PACMAN: Attacking ARM pointer authentication with speculative execution. In *IProceedings of the 49th Annual International Symposium on Computer Architecture*.

Received 30 June 2022; revised 22 April 2023; accepted 31 May 2023