Defense against Microarchitecture Side-Channel Attacks through Runtime Detection, Isolation and Prevention

by

Arsalan Javeed

July, 2022

A thesis submitted to the
Faculty of Engineering and Natural Sciences
of the
Sabanci University, Türkiye
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Committee in charge:

[redacted]

The thesis of Arsalan Javeed, titled *Defense against Microarchitecture Side-Channel Attacks through Runtime Detection, Isolation and Prevention*, is approved by:

Supervisor: [redacted]

Signature: ______________________________

Co-supervisor: [redacted]

Signature: ______________________________

Committee Member: [redacted]

Signature: ______________________________

Committee Member: [redacted]

Signature: ______________________________

Committee Member: [redacted]

Signature: ______________________________

Committee Member: [redacted]

Signature: ______________________________

Signed page is on file

Abstract

DEFENSE AGAINST MICROARCHITECTURE SIDE-CHANNEL ATTACKS THROUGH RUNTIME DETECTION, ISOLATION AND PREVENTION

ARSALAN JAVEED

Computer Science and Engineering, PhD Dissertation, July 2022

Thesis Advisors: Assoc. Prof. Cemal Yilmaz, Prof. Erkay Savas

Keywords: side-channel, microarchitecture, timing, attacks, systematic-mapping, defense, countermeasures

Over the course of recent years, microarchitectural side-channel attacks emerged as one of the most novel and thought-provoking attacks to exfiltrate information from a computing hardware. They leverage the unintended artefacts produced as side-effects to computation, under certain architectural design choices and they prove difficult to be effectively mitigated without incurring significant performance penalties. Moreover, such attacks could operate across isolated processes, containers and virtual machines.

In this thesis, we focus on countermeasuring microarchitectural side-channel attacks on computing systems. We investigate the origins of such attacks, effectiveness of existing countermeasure approaches, and lessons that can be learned to build secure systems of future against these attacks. To this end, we perform a systematic mapping of existing literature from recent years under a classification scheme that we developed for this purpose, and provide sought-after answers from the curated set of primary studies through systematic mapping.

Furthermore, we present a novel approach called *Detector*$^+$ to detect, isolate and prevent microarchitecture timing-attacks at runtime. We observe that time measurement behavior of timing attacks differ from benign processes, as these attacks need to measure the execution times of typically quite short-running operations. Upon presence of suspicious time measurements, noise is introduced into the returned measurements to prevent the attacker from extracting meaningful information. Subsequently, the timing measurements are analyzed at runtime to pinpoint malicious processes. We demonstrate the effectiveness of our approach and its incurred negligible performance overhead both in the standalone server environment as well as virtualized cloud environment. Lastly, we discuss some potential avenues for future research in this area of computer and cybersecurity.

Özet

MIKROMIMARI YAN KANAL SALDIRILARINA KARŞI DINAMIK TESPIT, İZOLASYON VE ÖNLEME TABANLI SAVUNMA

ARSALAN JAVEED

Bilgisayar Bilimi ve Mühendisliği, Doktora Tezi, Temmuz 2022

Tez Danışmanları: Doç. Prof. Cemal Yilmaz, Prof. Erkay Savaş

Anahtar kelimeler: yan kanal, mikromimari, zamanlama, saldırılar, sistematik haritalama, savunma, karşı önlemler

Son yıllarda, mikromimari yan kanal saldırıları, bir bilgi işlem donanımından bilgi sızdırmak için en yeni ve merak uyandıran saldırılardan biri olarak ortaya çıkmıştır. Bu saldırılar, belirli mimari tasarımlar altında, hesaplamaların yan etkileri olarak üretilen istenmeyen yapay olgulardan yararlanırlar ve ciddi performans ek yüklerine maruz kalmadan etkin bir şekilde engellenmeleri genellikle zordur. Ayrıca, bu tür saldırılar birbirlerinden yalıtılmış işlemler, sanal kaplar ve sanal makineler arasında da çalışabilirler.

Bu tez bilgisayar sistemlerine yönelik mikromimari yan kanal saldırılarına karşı alınabilecek önlemler üzerine yoğunlaşmaktadır. Bu tür saldırıların kökenleri, mevcut karşı önlem yaklaşımlarının etkinliği ve bu saldırılara karşı geleceğin güvenli sistemlerini oluşturmak için öğrenilebilecek dersler araştırılmaktadır. Bu amaç için tez çalışmaları kapsamında geliştirilen bir sınıflandırma şeması kullanılarak son yıllardaki literatürün sistematik bir haritası çıkarılmakta ve bu haritalama kullanılarak yine tez çalışmaları kapsamında belirlenen araştırma soruları için elde edilen cevaplar sunulmaktadır.

Bütün bu çalışmaların yanında mikromimari zamanlama saldırılarını, sistemlerin çalışmaları ile eş zamanlı olarak tespit eden, izole eden ve önleyen Detector+ adlı yeni bir yaklaşım sunulmaktadır. Sunulan yaklaşımın temeli, bahse konu saldırıların genellikle çok kısa süren işlemlerin çalışma sürelerini ölçtükten dolayı zararsız yazılımlara göre daha farklı bir zaman ölçüm karakteristiği gösterdiği gözlemine dayanmaktadır. Şüpheli zaman ölçümlerinin varlığı durumunda saldırganın anlamlı bilgiler elde etmesini önlemek için döndürülen ölçümlere gürültü eklenir. Ayrıca, şüpheli zamanlama ölçümleri kötücül işlemlerin sistemlerin çalışmalarıyla eş zamanlı olarak saptanması için analiz edilir. Sunulan yaklaşımın etkinliği ve sistemler üzerine getirdiği ihmal edilebilir düzeydeki ek çalışma yükü maliyeti hem bağımsız sunucu ortamlarında hem de sanallaştırılmış bulut ortamlarında yapılan deneysel çalışmalarla gösterilmiştir. Bu tez çalışması kapsamında siber güvenlik alanında gelecekteki araştırmalara ışık tutabilecek nitelikte potansiyel araştırma konuları da tartışılmaktadır.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Assoc. Prof. Cemal Yilmaz and co-advisor Prof. Erkay Savas, for their continuous support, advice and continual guidance during my journey of doctoral studies and research. I would acknowledge and deeply admire their immense knowledge, motivation, mentorship and patience, and I feel fortunate to have worked with and learned from them.

Beside my advisors, I would like to thank the remainder of my thesis committee: Assoc. Prof. Kamer Kaya, Asst. Prof. Onur Durahim, Prof. Husnu Yenigun, and Assoc. Prof. Hasan Sozer; for their encouragement, insightful comments and questions.

Also, I would like to acknowledge my friends: Ahmed Nouman, Faseeh Ahmed, Momina Rizwan, Imran Rizvi, Timur Gedik and Esaam Jamil; who were with me during some of the tough times and encouraged me through the thick and thin of this journey. Moreover, I would also convey my thanks and gratitude to the extremely supportive administrative, academic and healthcare staff at Sabanci University.

Finally, I must express my immense gratitude to my mother Najma Yasmin, and my family for their consistent support and continuous encouragement throughout these years of my doctoral studies.

Table of Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

Since the mid-nineties, side-channel based crypt-analysis emerged as a new research area and gained increasing interest over the years, as a means to either potentially break or compromise the strength of a cryptographic primitive to decipher secret information. Since the inception of modern cryptographic standards and protocols, although rigorous mathematical analysis ensured their promised strength which rendered classical brute-force based approaches almost worthless. However, the adversaries discovered new ways to undermine the strength of aforementioned cryptographic primitives, by leveraging weaknesses originated from their physical implementation on a computational hardware.

In a typical setting, an adversary aims to recover the secret parameters in use during cryptographic computation, through leveraging some form of implementation-specific characteristics, attributed to the underlying hardware by carrying out a physical attack. Depending upon adversary's abilities such attacks can be invasive or non-invasive in nature. Invasive attacks would involve efforts to directly-access critical components of the target hardware, through means such as unpackaging a chip, sniffing traffic on a data bus and so on. On the contrary, non-invasive attacks would leverage some form of externally available information, which would had been unintentionally emitted. Kocher [Koc96] describes the act of deriving secret values undergoing in a computation, through leveraging unintentionally emitted influences on the surrounding environment as side-channel attacks. Kocher performed a side-channel attack involving timing differences in execution among different parts of a victim algorithm, such attacks are now known as *timing side-channel attacks*. Over the following years, researchers demonstrated various types of side-channel attacks targeting sensitive computations; through leveraging a wide-range of artefacts unintentionally emanated as by-product of a computation. Such artefacts span across power consumption [RD20], electromagnetic emissions [SLKS19], acoustic emissions [CBRY20], and many more.

A distinct to side-channel but relevant class of attacks is referred as fault attacks. In contrast, an adversary tries to manipulate or derail flow of computation on a device to either weaken its security mechanisms or to reveal the secrets. The attack dynamics involve manipulating the sanity of environment through inducing a fault. Such faults could involve voltage glitching [KGT18], clock perturbations [ADN$^+$10], temperature changes [HS13] and so on. The underlying notion in such attack attempts is to push the hardware to operate on the boundaries of operating specifications, in efforts which would lead to weaken its security posture.

Modern computer architecture is deeply rooted into its target microarchitecture and instruction set architecture(ISA). Microarchitecture refers to the number of viable ways in which a given ISA could be implemented at microelectronics level for a specific microprocessor. Similarly, a specific ISA can also be implemented in a variety of microarchitectures. Such design variations are driven by factors as design cost, optimizations and target usecase. These aforementioned entanglements when in play although could meet design goals, however inadvertently on occasion lead up to security vulnerabilities when discovered, could be leveraged by an adversary. Furthermore, practical security attacks from recent years revealed that, exploitable information leakage is not attributed to its algorithmic level, but in many cases is caused as a side-effect to microarchitecture optimization. Such optimizations are performed in lieu to the nature of data being processed, locality and how frequently it is accessed. The end effect of aforementioned optimizations through a side-channel can help a motivated adversary to infer the content of data, to which it has no access rights. Based upon the partial knowledge developed from such inferred data can uncover the secret values of interest [AGYS20].

A specific class of side-channel attacks which are rooted into exploitation of vulnerabilities arisen from microarchitectural optimizations are referred as *microarchitecture side-channel attacks*. Over the past years, this class of attacks emerged as some of the most notorious and potent side-channel attacks. Some of the notable examples among these attacks are: Meltdown [LSG$^+$18], Flush+Reload [YF14], Prime+Probe [LYG$^+$15] and many more. One of the challenges faced when combating these attacks revolve around developing countermeasure approaches which can effectively mitigate these attacks and would impose least amount of impact on system's performance. Often security and performance are orthogonal to each other and a solution that could provide security but affects system's performance is least desirable from practical standpoint.

In this dissertation, we focus on the subject of defending computer systems against microarchitecture side-channel attacks by demonstrating two countermeasure approaches we developed, as well as we investigate into the research arena of these attacks. Specifically, we carried out a systematic mapping study of existing literature and researched answers into aspects spanning across: the origins of microarchitectural leakage; ways through which attack corridors can be created; existing countermeasure approaches; and lastly what steps could be taken to architect computing systems resilient to these attacks.

During our investigation to develop generic countermeasures to combat, we observed that all these attacks share a common trait, that their attack dynamics rely heavily on very fine-grained time measurement behavior (of the order of hundred of clock cycles). We leverage this behavioral trait and utilize it to detect, isolate and prevent ongoing timing based microarchitectural side-channel attacks. Furthermore, doing so our proposed countermeasure approaches demonstrated low-enough overhead on-average $\leqslant 1.5\%$, whereas detected and prevented ongoing attacks upto $\approx 99\%$ accuracy as soon as the attack was initiated. Furthermore, not only we developed the presented countermeasures for stand-alone operating systems, we also targeted virtualized environments, which are an essential part of cloud based services, as multiple concurrent operating systems are executed side-by-side over shared hardware.

In the following sections, we will outline our main contributions in this dissertation as well as present an outline of the organization for the remainder of this dissertation.

1.1 Main Contributions

Since we are interested in protecting and strengthening security posture of computing systems against microarchitecture side-channel attacks, we put forward four core research questions. The research questions were aimed to investigate and find answers from existing state-of-art literature into aspects: such as, origins of microarchitectural information leakage; ways through which attack corridors can be created; existing countermeasure approaches; and lastly what steps could be taken during the system design phase to cater for microarchitectural security of future. To scientifically answer the proposed research questions we performed a systematic mapping study of existing literature. In this regard, we searched across three main scientific databases, using tailored keyword searches in the context of aforementioned research questions and retrieved more than 350 published peer-reviewed articles, dated until Sep 2021. Next we went through article screening and selection process, where we followed the guidelines of a systematic mapping protocol and elected 84 articles to serve as primary studies under a classification criteria. This work was carried out in collaboration with Cemal Yilmaz and Erkay Savas and detailed in Chapter 3 of this thesis. This work is due to be submitted as a journal article.

Next we continued and studied the behavior of microarchitecture side-channel attacks and focused on their time measurement behavior. We observed, in essence they rely on very fine-grained time reading behavior during their attack cycles. They do so to measure the elapsed time between two microarchitectural operations of interest, such as if some data resides in cache memory or not and so on. We instrumented our experiment platform and analyzed the time reading behavior of five such attacks, in contrast to a suite of benign applications. We found the observed behavior be significantly different among attacks and benign processes and leveraged this towards developing a generic countermeasure approach, we named *Detector*$^{+}$, which turned into a journal publication [JYS21]. This work had been carried out in collaboration with Cemal Yilmaz and Erkay Savas. This work in included as Chapter 4 of this thesis.

1.2 Outline

This dissertation is organized as follows: Chapter 2 provides relevant background information to equip reader with necessary information for the succeeding chapters. Chapter 3 presents our work with regards to systematic mapping study, which serves to provide both the state-of-art existing literature, as well as provides directions in microarchitecture security research. Chapter 4 presents our work with regards to a generic countermeasure approach *Detector*$^{+}$ to detect, prevent and isolate timing attacks. Chapter 5 lastly, discusses the directions for future work and draws conclusions.

CHAPTER 2

Background

This chapter will present the necessary background to understand and discuss microarchitectural side-channel attacks. These attacks target microarchitectural elements, therefore we will discuss how microprocessors are organized.

2.1 Basics of Side-channel Attacks

In a side-channel attack, an adversary is a party with a malicious intention, whose aim to penetrate the defenses of an otherwise secure system by leveraging and exploiting unintentional information leakage from the system as a side-effect to computations being carried out. Such leakage could be in any number of forms: such as, power consumption [RD20], electro-magnetic emissions [SLKS19], acoustic emissions [CBRY20], and many more. The end goal of an adversary is to steal private information, to whom he has no access rights to do.

In microarchitecture side-channel attack, the adversary steals secret information by exploiting observable information leakage from microarchitectural elements. The attack activity unfolds as follows: a certain target application, processes secret information on the processor. The secret information could be a cryptographic key being used to encrypt data. Due to the nature of computation (in this case cryptography, being performed by the target application) certain runtime behavior is being repeatedly demonstrated, such as frequent memory accesses to fetch data. This behavior in turn manifests into reveal some recurring characteristic, such as first the data is being looked into the cache, if it is not found then memory access is made and vice-versa. By identifying the interaction between inputs, runtime behavior and characteristics of computation; the adversary can learn about the nature and structure of secrets [LZJZ21].

Applications can implicitly reveal important details to an adversary, about internal processing of secret input by its coded logic in a number of ways. Among these ways, secret dependent control-, and data-flow are attributed to its software implementation, which could be remedied by revised implementation. However, among other aforementioned ways, an adversary can deduce important information about the internals of the application through observing the state changes in shared hardware components, which are directly being affected by the execution of application. As hardware component sharing leads often to contention and manifests itself into timings. Notable statistical differences in timings, in relation to application activity, can reveals important details about how certain application processes data. Timing based microarchitectural side-channel attacks thus, remain quite relevant and widespread.

2.2 Differentiating Side and Covert Channels

A covert channel is a communication channel used to transfer secret information between a sending and receiving party, such as two processes on a system. Covert channel often established intentionally among malicious parties involved, as a consequence to a security exploitation. These channels conceal themselves by avoiding usage of legit data transfer mechanisms provided on a system, as a result they succeed at bypassing access control policies and remain undetected by conventional monitoring mechanisms in place. In contrast, a side-channel resembles a covert channel but the sender does not intentionally communicates information to receiver, rather information is leaked as a side-effect to the way hardware is utilized, or leverages some flaw of the software implementation [Sze].

Broadly speaking, side- and covert-channels can generally leverage either timing-based or access-based dynamics to infer secret information. Timing-based channels generally measure timings of short running operations (e.g. time taken to read data from memory vs cache) and subsequently derive secret information. In contrast, access-based channels rely on some directly accessible information to infer secret information (e.g. observing the frequency of cache miss/hit to derive sparsity of data accesses for a process).

2.3 Primer on Microprocessor Organization and Architecture

Since the dawn of early microprocessors in the early 70s, each subsequent generation went through innovative architectural changes and delivered enhanced execution performance. Partly these performance enhancements are attributed to refinements in semiconductor process technology and the revolutionary enhancements in the microarchitecture itself.

The instruction set architecture (ISA) of processor refers to its instruction set semantics, registers and programming interface visible to the programmer. The ISA is generally well maintained and often enhanced from one generation to the next. However, the microarchitecture refers to the actual realization of processor's ISA in the silicon down towards the microelectronics level. The microarchitecture could change from one generation to the next while keeping the same ISA. In a similar regard, one microarchitecture could support multiple different ISA.

Modern microprocessors have enjoyed increasing clock speeds to deliver increased execution performance year after year, until the last decade, when designers realized further increase in clock speed was not viable due to limitations of semiconductor process technology. However, the designers came up with a myriad of alternative ways and microarchitecture optimizations to deliver even further performance despite the staled clock speeds. Among notable enhancements, designers leveraged sophisticated pipelining, instruction stream optimization, and multi-core architecture.

Instruction pipelining is used to achieve enhanced execution performance of a microprocessor. Instructions are split into multiple stages in a pipeline, broadly speaking the instructions in a pipeline go through fetch, decode and execute stage. Modern microprocessors can process multiple instructions concurrently by providing multiple identical stages of pipeline to increase throughput, where multiple streams of instructions execute concurrently. Such pipelined designs are although quite sophisticated in nature and do require a lot of book keeping by the internal logic of processor nonetheless, they vastly increase overall throughput.

Multicore processors are a new trend in modern microprocessors. A single processor contains multiple execution cores in a single package. Each core shares some core private resources, such as cache and pipeline stages, as well as shared resources with other cores. Multicore processors deliver significant performance improvements as parallelizable workloads are distributed among available cores to be executed concurrently. However, if the workload is not parallelizable then multicore design would be of little use.

Out-of-order and speculative execution in contrast, are other performance enhancement mechanisms leveraged to optimize execution of instruction stream on modern processors. Out-of-order execution refers to executing instructions independently from each other in their order of execution to increase overall throughput. Those instructions that do not have pending data dependencies are executed immediately whereas, instructions with pending dependencies are scheduled for a later time when those dependencies are met. However, out-of-order execution remains hidden from the application point of view and everything appears as it has been executed sequentially. In a distinct regard, speculative execution refers to executing branches speculatively ahead of time, by guessing most probable branch. If the guesswork is correct then results are already available from executed branch, however, if not the executed branch is discarded and correct branch is taken in time.

2.4 Cache

The mismatch between the rate of improvement in processor clock speed against the latency of main memory manifested into a performance bottleneck. A performant CPU had to waste precious clockcycles while the data being fetched from the slower main memory.

Processor caches were introduced as remedy to this issue. Caches are small-sized fast buffers employed to conceal the latency between a fast processor and slow memory. In modern processors multiple cache hierarchies with each having multiple levels are provided for performance among other specialized purposes. A typical cache size is of the order of tens of KiB.

Cache being intermediary serves as gateway to all memory accesses. To serve a memory access its first looked into the cache, if found it is served from cache and referred as cache hit and vice versa a cache miss.

As a matter of course, caches being small in size are subject to constantly out of room for new data fetch requests. Room for new data requests is created through continual eviction of existing data to be subsequently replaced by newly fetched data from memory. This aforementioned eviction and replacement operation are performed under a replacement policy, which is usually vendor specific and whose details are often undocumented.

On modern systems, typically there exist multiple cache levels (L1, L2, often L3) arranged in cache-hierarchy comprising the cache-subsystem. L1 cache is split as instruction and data cache, followed by large shared L2 cache. On some systems L3 cache, also called last-level cache (LLC) is shared between all CPU cores. On modern Intel microarchitectures, LLC is inclusive in nature which contains all data within L1 and L2 caches.

2.5 Memory

Dynamic RAM (DRAM) serves as the main memory of a computer system, providing temporary data storage capacity of the order tens of GiB. However, a typical DRAM operates at a much slower clock speed than main processor and thus projects significantly higher latency. The underlying DRAM storage cells require periodic refreshing to maintain data integrity, which also contributes a fair share in overall latency. Intermediate caches and increasing the bandwidth of DRAM are some of the ways through which the latency is overcome and performance improved.

The integrated memory controller on the processor serves as intermediary to handle memory accesses to DRAM. When a data request is not found in the cache it is passed on to memory controller which first translates the presented virtual address to actual physical address and then passes on to DRAM. The DRAM internally is organized into channels, banks, rows and columns; whose information is kept into an internally maintained DRAM map.

A given physical address is translated into internal DRAM address through physical-to-DRAM addressing function, which utilizes the DRAM map. The request is served according to the bank and channel it corresponds. Often multiple requests are reordered and served concurrently for better performance in conjunction to the memory controller.

A given data request goes through the following steps to be served. First it is stored into a corresponding bank buffer for the request, where a scheduling algorithm will prioritize and subsequently arbitrates its bank access. From there it is passed on to channel scheduler for further arbitration and subsequent access to target channel, row and column. Once the data is located it is placed into row buffer to be served to memory controller. In this whole sequence of steps involving serving a data request, several sources of contention and timing variations exist which are exploited for microarchitectural side-channel attacks.

CHAPTER 3

Systematic Mapping Study

The manifestation of computer architecture roots into microarchitecture (abbrv. uarch) and instruction set architecture (ISA). Microarchitecture refers to the ways in which an ISA is implemented for a specific microprocessor. For reasons such as design, cost, optimizations and target application; a given ISA can be implemented through different uarchs. Microarchitecture for a given ISA, may differ in the ways through which constitutent components of a processor are interconnected and interoperated. With all these aforementioned complex entaglements in play; exploitable security vulnerabilities may arise which can be abused by a malicious adversary to affect the confidentiality and integrity of a computing system and may result in serious loss. In recent years, Meltdown[LSG$^+$18], Spectre[KHF$^+$19] been few of notable examples. Nonetheless, the security researchers have been involved in discovering newer expoitable flaws, developing countermeasures against existing flaws and researching the ways in which more secure systems can be built.

The literature on microarchitecture security research is although available and growing. However, efforts that overarch the individual works, to systematically gather and assemble in a broader context on various aspects of microarchitecture, are limited, somewhat outdated, yet still desired. The purpose of our presented work as a systematic mapping study is a step in this direction. A systematic mapping study provides an overview of a existing research through systematic classification of published literature on the topic, guided by the posed research questions under the constraints of mapping protocol. A prime finding from such studies are the underlying research trends which are emerging; and research spots which have gained considerable attention and viceversa. These trends can launch newer investigations and guide the direction of further research. In contrast, a literature survey aims to comprehensively index and thus, report the existing state of affairs on a said topic. Thus, both of these aforementioned types of studies differ in terms of their end-goals and their followed research process.

To guide our investigation, we've raised four research questions (RQs)(Section 3.2.1) to address security concerns about microarchitectures; such as, aspects leading up to leakage; the ways in which exploitable corridors being created; effectiveness of existing countermeasures and their applicability for zeroday attacks; and lessons which can be learnt to build more secure systems.

The main contributions of this mapping study are: i) broad overview of research efforts in the area of microarchitecture security; ii) a classification scheme that can be used to classify newer research in this area; iii) decent coverage of published research in this area, particularly in the last five years; iv) identification of research spots where most research emphasis had been and viceversa.

The target audience of this work are researchers and practioners who want to establish better overarching understanding of research efforts being carried out in this area.

The rest of this paper is structred as follows: Section 3.1 presents related work; Section 3.2 presents methodology that we have followed for this mapping study detailing search strings, inclusion and exclusion criteria, study selection process; Section 3.3 presents classification scheme that we employed and subsequently, the mapping results. Section 3.4 details the answer to research questions outlined in this study; Section 3.5 presents potential threats to validity to this study and possible directions for future work; and Section 3.6 concludes this paper.

3.1 Related Work

During our search of literature, we didn't find (to the best of our knowledge) any systematic mapping study in the area of microarchitecture security. However, we did come across a few surveys [KKOA12, Sze, GYCH18]; and individual works [ODK20, EPS10], addressing one specific aspect. Albeit, these works had either been somewhat dated, or lacked wider scope. Moreover, some of such works may either partly overlap with our presented work or provide additional supplementary material.

Kanuparthi et al. present a survey[KKOA12] dating till 2012. The survey scope was bound for embedded microprocessor security only. As the the proliferation of embedded systems was on the rise, so as emerging challanages to secure their vulnerable nature caused by limited resources. The authors reported some of the prominent attacks plaguing that arena during those times. Furthermore, the authors compared the tradeoffs among sought after countermeasure approaches levering integrity checks, data encryption and microarchitecture revisions.

Szefer et al. present a survey[Sze] on microarchitectural side and covert channels with existing defense proposals until 2017. Moreover, the prime emphasis of their survey been on timing based exploitations and to some degree access-based exploitations. In a similar regard, the authors did foresee role of prefetchers in microarchitectural exploitations and did theoretically presented a setting for such an attack, yet did not cited a published source, as the research was still in its infancy at that time.

In another survey[GYCH18], Ge et al. authored around the same year of 2018, has exclusive focus on timing based microarchitecture attacks and their countermeasure. The main motivation behind this survey had been to taxonomize aforementioned efforts. These attacks established their reputation as prime vectors for remote exploitation in cloud computing. Furthermore, the authors emphasize the need to secure cloud systems against this class of attacks, as our ever-increasing adoption and growing reliance on cloud computing will stay.

A fact to mention, a direct comparison with these works wouldn't be straightforward because of different research objectives and the methodology being employed. Last but not least, we believe the contituent studies employed in our work are fairly recent in the sense, half of them have been published in the last five years (since 2018).

3.2 Methodology

In this section, we will describe our methodology employed for this study and followed workflow.

We followed the guidelines for systematic mapping laid out by the authors of these [PVK15, K+07] and also adopted by relevant mapping studies such as [SF13, MTE+21, ZSG16]. The mapping process in a nutshell is comprised of these sequential steps [PVK15]: *Defining research questions*, *conducting the search*, *screening of papers*, *keywording of abstracts* – aka *classification* and *data extraction and mapping*. Although the aforementioned steps are sequential in nature, however often times the individual steps are iterative in nature, aimed to refine the end result of mapping process.

At the start of this work, our team carried out a planning phase, primarily to lay out the research questions; method to locate and appraise primary studies; craft a search strategy; and carry out curation of primary studies for this work.

3.2.1 Defining the research questions

We put together the following research questions, as a basis for this study.

Title, **Abstract**, **Author defined**, **All metadata**
"micro[-]?architecture", "attack", "vulnerability", "side[-]?channel", "origin[s]?", "defen[c|s]e", "counter[-]?measure[s]?", [0|zero]?[-]?day, "detect[ion]?","prevent[ion]?", "leakage", "predict", "secure", "system"

Table 3.1: Keywords used to form search strings. To avoid redundancy, we're adopting regex format to specify potential variations of a keyword.

- What are the aspects of microarchitectural artefacts which contributes to sensitive information leakage to compromise security and privacy?
- How some of the recent microarchitectural surfaces were crafted and turned into feasible attack corridors?
- How effective are the proposed countermeasures of microarchitectural sidechannel attacks and whether these countermeasures are generalizable? Can these generalized countermeasures can predict/prevent zero-day attacks?
- Given the published countermeasures, how secure a system we can build against micro-architectural sidechannel attacks and what lessons can we incorporate in this architectural process?

3.2.2 Conducting the search

Guided by our research questions, we defined an initial set of keywords as seed to search in Google Scholar to locate top relevant papers. We read these papers and utilized *snowballing technique* [Woh14] to find additional papers that we find relevant to our RQs. Snowballing refers to locating additional papers based on reference list, or citations of a given paper [MTE+21]. We also employed an online tool connected papers [con] to aid snowballing process. We came up with 11 core papers as a result of initial search, these papers were used to put together a set of keywords (listed in Table 3.1), which were then used as a basis for defining search strings. The validity of these keywords were tested by rediscovering the core papers they were driven from. We also eliminated some keywords that we found to be redundant, superflous or increased frequency of irrelevant search results.

To carry out actual search, we've utilized these standard databases as sources: *IEEEXplore*, *ACM Digital Library* and *Springer Link*. This decision was primarily influenced by the availability of database subscription that our campus had, the manpower of our team, and the fact that these sources are de-facto prime venues of scientific literature in computer science. Furthermore, we leveraged the advance search feature of these databases, whenever possible, to perform search queries. Advance search features enabled us to limit or expand the scope of queries in lieu to frequency of retreived results that were irrelevant. Based on keywords from Table 3.1 the search string was iteratively experimented with. Moreover, the internal mechanics of these databases were found to be different, given the fact that a query suitable for one database didn't find to be equally effective in another database. For example, we found out that searching for keywords in the abstract portion for IEEExplore was effective in contrast to ACM. Nonetheless, we used the same set of keywords, the specifics of advance search features (such as boolean operators) and filters to tailor the relevance of search results. For search results, whenever possible we sorted them in-order of relevance and picked top 100 results. Such was the case with ACM and SpringerLink. For IEEExplore we utilized to stop at 5 irrelevant papers in a row, as stopping criteria.

3.2.3 Screening and classification process

To aid the screening and study selection process, we tailored the following inclusion and exclusion criteria adopted by [K$^+$07, ZSG16, MTE$^+$21] described as:

Inclusion criteria

- a study must atleast answer one of the proposed RQs.
- a study must be dated until end of September 2021.
- during selection priority must be given to evaluation and validation research studies.
- a study must be peer-reviewed and published in English.

Exclusion criteria

- a study shouldn't be in one of these forms: patents, whitepapers, reports, thesis, tutorials and webpages.
- duplicate studies found by different search engine.
- inaccessible, irretreivable or irrelevant to the theme of our work.
- any article whose subject matter is around quantum computing.

The aforementioned criterion aided us to perform initial screening and sanitization of the search results that we retreived from databases searches.

Our initial search effort resulted in 379 articles from the three databases as: 171 from IEEExplore, 100 from ACM, and 108 from SpringerLink. However, after the title and abstract screening, we performed content scanning for articles, where we felt the abstracts were either short to be properly meaningful or lacked enough clarity, that was unable to zero-in on the investigated problem; and actual contribution being made. Furthermore, we also performed content scanning of sections from the text body, where we felt that individual clauses of inclusion and exclusion criteria was difficult to apply. Moreover, where deemed necessary the authors of this study discussed and compared their concensus on the studies to be included. For classification we settled for 84 articles to serve as our primary studies, the list of which is available in Table A.1. The classification scheme is described in Section 3.3 which guided us towards final data extraction and mapping of results. Ultimately, the primary studies also served as the basis for answering our RQs which was the main goal of this mapping study.

3.3 Results

Systematic mapping studies aim to provide an overview of a research arena through classification of published literature on the subject topic. In order to aid discussion and present results of our study, we first describe the classification scheme, which we used to categorize the primary studies, and then present the mapping results in the light of aforementioned scheme.

3.3.1 Classification Scheme

The classification scheme that we employed broadly classifies the studies in to seven categories: *research type*, *main contribution*, *hardware platform*, *instruction set architecture (ISA)*, *leakage vector*, and *leakage component*. In the following, we describe each of these categories and subcategories.

Research Type

This category describes different research approaches as outlined by Wieringa et al. [WMMR06]. As earlier mapping studies [ZSG16, MTE$^+$21, RRB18] used Wieringa's classification scheme, we chose to follow the same footsteps. We associated each of our primary studies with one of these research approaches: *evaluation*, *validation*, and *solution proposal*. Based on a typical exclusion criteria of mapping studies [RRB18, MTE$^+$21], we disregarded *philosophical, opinion*, and *personal experience* papers; and focused on opting primarily for *evaluation* and *validation* research. Aformentioned research approaches aim for demonstrating the practical usage of a technique backed by real data, relevant experiments, and empirical-evaluation.

Main Contribution

Categorization based on contribution type in mapping studies, has been employed previously by Zein et al. [ZSG16] and Shahrokhni et al. [SF13]; which classifies each of the primary studies to one of the following specific type of contribution: *framework, tool, metric, approach*, and *criterion*. A *framework* is a detailed method which has a broad scope and tend to focus on more than one research questions or areas. Whereas, an *approach* has a narrower scope and tend to have a more specific goal addressing a single research question. Similarly, a *tool* is an implementation realization of at least one approach, aimed to demonstrate the applicability to the practitioners. Primary studies, where core emphasis was on tool demonstration, are classified in this category. Whereas, a *metric* is an empirical measure to quantitatively describe a variable of interest. In contrast, a *criterion* outlines a strict method by which a certain quantitative or qualitative attribute of an observatory aspect can be judged upon.

Hardware Platform

This category represents the nature of physical hardware considered by a primary study upon which either the main subject matter was presented and/or evaluated upon. We opted for the following subcategories: *commodity, mobile, embedded/IoT, cloud*, and *any*; based upon explicit mentioning of the platform-type discussed by a particular primary study. *Commodity* hardware refers to general purpose computing platforms such as workstations for daily usage. Whereas, *mobile* hardware refers to typically handheld portable devices, which are primarily meant for communication and serve limited computing needs of user. Smartphones are prime example of this category. Similarly, *embedded/IoT* devices refer to small microprocessor equipped hardware, with optional networking ability and occasionally connected to internet; designed to perform a dedicated function. The software stack running on such devices usually has a small memory-footprint and optimized to consume electric power efficiently. Biomedical health monitoring devices, and infotainment systems are some of common examples in this category. Similarly, *cloud* platform refers to group of server machines that are networked together and connected to internet, located typically in a data-center environment. The users are given remote access over the internet to utilize these machines for their application usecases. In contrast, virtualization refers to the use of the hardware functionality through software emulation of underlying hardware-subsystems. Last but not least, *any* refers to either an unspecified platform, or the specification of platform does not matter. Those primary studies, which present a generic approach, criteria, or empirical metric; fall under this category; as their subject-matter is agnostic to a specific hardware type.

Evaluation Methodology

This category refers to the methods adapted in the primary studies to conduct their experimental evaluation of their presented approach. In this regard, we have observed, following five types of approaches: pure *software* implementation; employment of *simulation* tools; evaluation performed on *real* hardware; evaluation performed on *FPGA* synthesized hardware; and *other* refers, where the aforementioned categories do not matter. For instance, studies presenting criteria or an empirical metric belong under *other*. It is to be noted here, primary studies often do not exclusively base their evaluations on one categorization and occasionally fall under more than one such categorization. For instance, a study carrying out its evaluation through simulation, and followed by hardware implementation.

Instruction Set Architecture

Instruction set architecture (ISA) refers to the syntactic and semantic realization employed at machine level; to which the compiled software is assembled for execution. In this regard, we found the following different types of ISAs, subject-mattered in our primary studies: *Intel*(x86,x86_64), *ARM*, *RISC*, Nvidia-*Cuda*, *MIPS*; and *other* – refers when the subject-matter of a study covers more-than-one ISA; not-any-particular ISA; or a lesser known ISA(such as Texas Instruments MSP430).

Leakage Vector

Leakage Vector refers to the side-channel means through which the sensitive information leaks or emanates. In this regard, we consider the following main leakage vectors: *access patterns*, *contention*, *sharing*, *dependence*, *duplication*, *EM*, *execution*, *fault*, *interruption*, *power*, *speculation*, *state*, and *timing*. Access patterns refer to the recurrent pattern of a hardware resource access, through which high level behavior of the requesting entity can be inferred. This can later be used as a ladder step to either assist or rely for further malicious exploitation. In contrast, contention refers to at least two entities contending to get access to a shared resource, in lieu to their functional demands. Similarly, sharing refers to serving, a provided functionality by a non-dedicated resource, fulfilling needs of requesting entities. Whereas, *dependence* refers to an essential provision of a resource as a prerequisite to an operation being successfully carried out. In contrast, duplication refers to creating a duplicated copy of some requested resource for a requesting entity to fulfil its need. Moreover, EM refers to electromagnetic emanations which can be observed at a distance without physical proximity. These emanations can be utilized to infer the underlying state and ongoing activities of the system radiating them. Similarly power refers to the electrical power being consumed in relation to hardware demand. Whereas, execution refers to the act of instruction execution on a processor. Fault refers to means through, which the sanity of internal state required for correct computation is disturbed, hence leading to discarding of intermediate computation . In contrast, interruption refers to an event, which needs to be served immediately and cause temporary suspension of lower-priority execution happening at time of interruption. Speculation refers to an act of out-of-order execution of instruction stream ahead-of-time based on predictable path which could either be utilized or discarded later on, dictated by the actual path computation demands. Similarly, state refers to an existing internal state of a resource which would lead into one of the possible future states. Lastly, timing refers to the elapsing of time interval between two microarchitectural events of interest.

Leaking Component

This refers to the microarchitectural functional unit or subsystem which is the source of side-channel leakage using one or more leak vectors. The constituents for this subcategory, that we relied to organize the primary studies are: *buffers*, *caches*, *instructions*, *interconnects*, *memory*, *microprocessing elements*, *tee*(trusted execution environment) and *timers*. Essential among these subcategories have been described in Section 3.4.1

3.3.2 Mapping Results

In the following subsections, we will present the quantitative and qualitative results of our systematic mapping study with respect to the classification scheme described in Section 3.3.1.

Publications Timeline

We curated a total of 84 primary studies(Table A.1), Figure 3.1 illustrate their publication timeline over the years from 2006-2021. The figure shows that research in the subject area of this mapping study becomes visible in year 2006 and gained gradual attention over the years until 2017. From 2017 onward, this research area gained more and increasing attention to date, as evident from sharp increase in the number of publications. Moreover, Figure 3.1 also illustrates that the pool of primary studies we assembled is fairly recent, as 76% (i.e. 64 out of 84) were published in the last 5 years (i.e. 2017-2021). It is to be noted regarding year 2021, the studies were included until the end of September, so we do not have a complete number for this year. However, based on linear regression estimates, 19 more publications were anticipated for the year 2021.

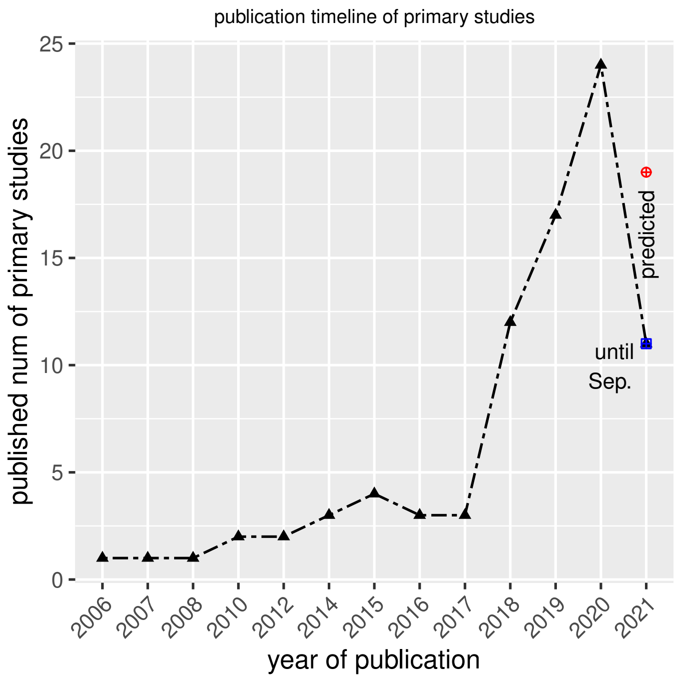

Figure 3.1: publication timeline of primary studies. The timeline spans from 2006 to Sep-2021. The predicted number for 2021 is marked by the red circle

Research Spots

Figure 3.2 highlights the research spots as matrix of bubbles, addressed by our set of primary studies. The size of the bubble is proportional to the volume of studies at the intersection of each pair of leakage vector and leaking component being involved.

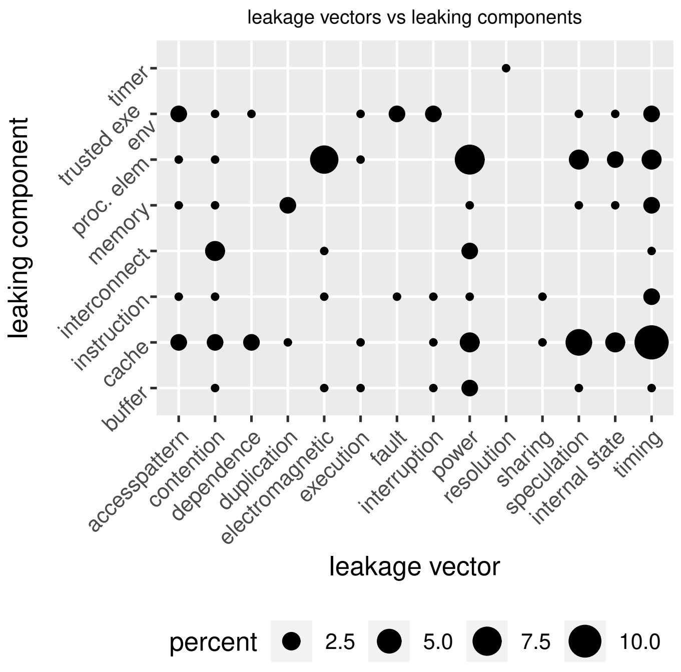

Figure 3.2: Research spots for leakage vector vs leaking components

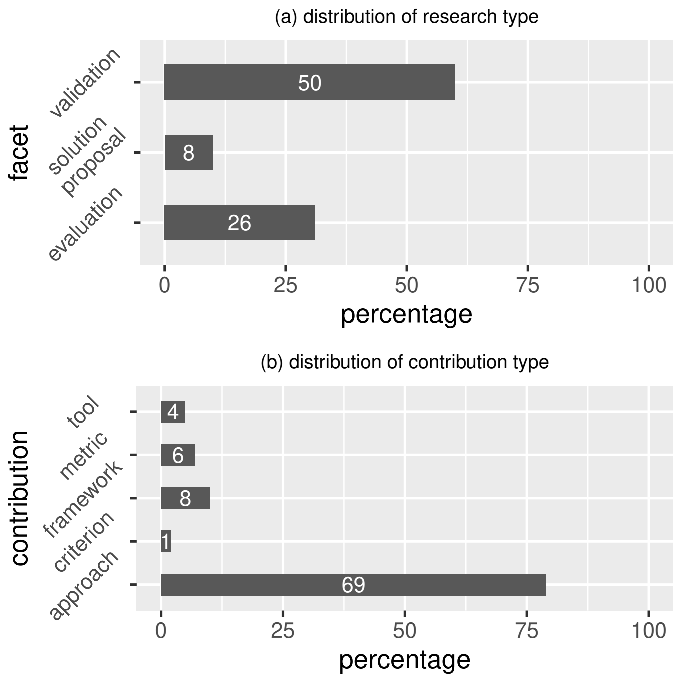

Figure 3.3: percent-wise contribution of primary studies wrt to (a) research facet, and (b) contribution made. The bar length in the plots indicate the percentage, and the integer value on each bar indicate the actual count.

Moreover, the figure also reveals the hot and cold spots, where research has focused and vice versa. The cold spots, maybe due to the fact that perhaps actionable exploitations and their remediations have not been observed yet. Nonetheless, it is to be kept in mind that the set of primary studies are primarily assembled based on a strict criteria designed to address research questions. If a study violates any of the contraint of the selection criteria, it is simply dropped. In this regard, the areas which are underlooked, perhaps had been affected by contraints of selection criteria. Nonetheless, we tried to carefully select and vet the primary studies which ultimately comprised the final pool. Moreover, it is out of the scope of systematic mapping to exhaustively assemble all published literature on the subject, because this matter is goal of a research survey. Nonetheless, the figure illustrates the research characteristics of the primary studies. Among notable patterns in Figure 3.2, we observe *contention* and *timing* as leakage vectors have been involved across almost all(7 out of 8) leaking components. In a similar manner, *cache* and *trusted exe. env* as leaking components have been involved across most(10- and 9- out of 14) of the leakage vectors.

Research Facet and Contribution Type

We classify the primary studies(Table A.1) along axes of research facet, main contribution, among other subcategories of study classification scheme (Section 3.3.1). Figure 3.3 illustrates both of these classifications percent-wise and the actual number of studies in that category. We see that most of the primary studies belong to *validation* research (50 out of 84), followed by *evaluation* research (26 out of 84) and lastly, *solution proposal* (8 out of 84). It is to be noted that, *solution proposal* are customarily [MTE^{+}21], excluded as part of study selection criteria, because by nature they lack considerable evaluation. . However, we opted to flex on this practice and in the final iteration of our study selection process, we included a small number of 8 solution proposal studies. Since, we found their presented ideas to be interesting and relevant in the scope of our research questions.

Along the same lines, we observe from Figure 3.3(b) that 69% of primary studies are actually contributing a proposed approach backed by their evaluations. However, remaining 31% may or may not exclusively contribute *tool* (4 out of 84), *metric* (6 out of 84), *framework* (8 out of 84) and criterion (1 out of 84). It is to be noted that not necessarily a study exclusively contribute to one particular categorization, but rather on occasion contribute to more than one categorization at the same time.

Target Platform and Instruction Set Architecture

In terms of target platform hardware(Figure 3.4(a)) and instruction set architecture of cpu(Figure 3.4(b)), we see that all of our primary studies have considered them all, albeit to various degrees as shown through percent-wise proportion in Figure 3.4. However, we do observe that commodity hardware platform and Intel x86* ISA has the largest share overall, which is to be expected because of their proliferation worldwide. Nonetheless, we see contribution from other ISAs and platforms too, which reveals an important fact that microarchitectural exploitations and vulnerabilities are not characteristic to one specific platform and ISA, but rather are being applicable to the whole spectrum of computing hardware.

Evaluation Methodology

The percent-wise distribution of types of evaluation methodology employed in the primary studies is shown in Figure 3.4(c). It can be seen that 52% and 25% of primary studies opted to evaluate on real hardware and FPGA respectively. In contrast, 21% and 15% opted for simulation, software implementation based evaluations. Lastly, a meager 3% are categorized under other, where aforementioned categories do not apply (for instance, studies which propose criterion).

3.4 Discussion

In this section, we discuss our results with respect to the research questions. Furthermore, we compare our findings with previous works to contrast similarities and discuss

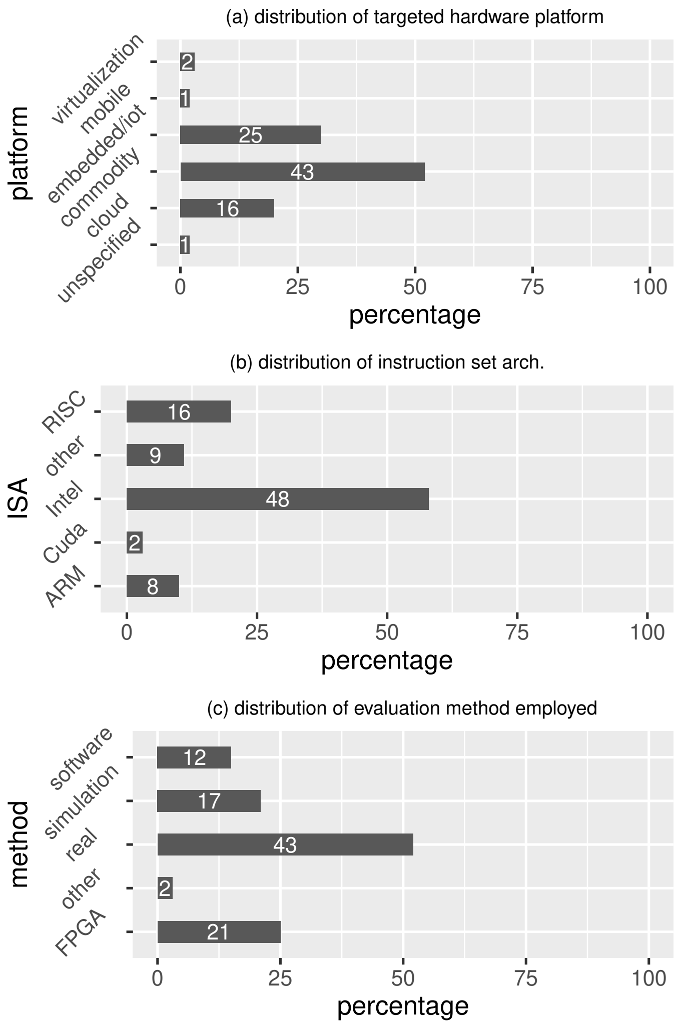

Figure 3.4: percent-wise contribution of primary studies wrt to (a) target hardware platform and (b) instruction set architecture. The bar length in the plots indicate the percentage, and the integer value on each bar indicate the actual count.

the limitations of study and directions for future research.

3.4.1 What are the aspects of microarchitectural artefacts which contributes to sensitive information leakage to compromise security and privacy?

Undesired information leakage from microarchitecture often originates due to spatio-physical implementation of the cpu, the design optimizations, employed trade-offs, and temporal resource allocation among various functional units, as program execution happens. Effective resource allocation and optimal scheduling of functional units can deliver desired performance but may not deliver perfect security. From the existing security research, diverse aspects of microarchitecture had been explored to discover security vulnerabilities; often involving one or more components being engineered into an information leakage state meant to be exploited for sensitive information exfiltration.

The data extraction from the primary studies(Table A.1) in this mapping study, identified the following artefacts compromising security. We will provide a high-level overview of each in the following paragraphs. Whereas, more focused and technically inclined, security specific account will be covered in Section 3.4.2. The presented content in this RQ should serve as primer for a starter audience.

Caches are small memories, meant to bridge the data-access latency between slow but large physical memory and fast CPU(s), by buffering frequently accessed data. On modern systems, typically there exist multiple cache levels (L1, L2, often L3) arranged in cache-hierarchy comprising the cache-subsystem. L1 cache is split as instruction and data cache, followed by large shared L2 cache. On some systems L3 cache, also called last-level cache (LLC) is shared between all CPU cores. On modern Intel microarchitectures, LLC is inclusive in nature[S18] which contains all data within L1 and L2 caches.

Concurrent execution of multiple processes has an influence on the contents of shared and private caches in a multicore environment. Moreover, hardware scheduling and resource allocation policy determines the assignment of cores to executing processes concurrently. Processes executing on the same core can access the core private caches as well as shared caches. However processes even executing on different cores can also influence content of private caches, belonging to other cores, through the propagation of side-effects from shared LLC cache to all cores[S13],[S18]. Thus, code execution and data access, ripple through the cache hierarchy in ways, to which an adversary can exploit.

The memory controller handles memory accesses to DRAM when data requested by CPU is not available in cache. Memory controller serves a data request, by translating the given physical-address into an internally maintained DRAM map of channel, bank, row and column information. Furthermore, the memory controller arbitrates internally among concurrent memory accesses through scheduling and buffering policies. Depending upon the choice of these internal policies, they exhibit an effect on timing variations and frequency of conflicts during bank- and row- address resolutions. Certain combinations of these policies result in exploitable timing differences, which an adversary can leverage [S21].

Memory deduplication is a mechanism to save physical memory space by keeping one copy of duplicate memory pages as long as the pages are unmodified, if so, the modified page(s) is isolated and kept as a separate independent page. Memory deduplication is often deployed as a cost-saving measure in the context of several virtual machines hosted on a single machine. However, studies revealed that memory deduplication can be feasibly exploited in a cross-vm covert-channel attack [S33].

Generally any microarchitecture essentially comprise on specialized processing elements generally referred as *functional units*; and intemediate temporary storage elements, referred as *registers* and *buffers*. Instructions of a program execute in a pipeline and typically go through instruction- *fetch*, *decode*, *execute*, *memory* and *writeback* steps. During each stage of this pipelined execution, the intermediate results are stored into pipeline-buffers and arguments and configuration is provided through intermediate registers. The contents within buffers and order of register allocation have significant impact on the internal state of the cpu core and could be exploited to engineer an internal state such as, intervals of transient execution could be prolonged to increase effectiveness of attacks reliant on transient execution [S58]. Furthermore, under favorable internal states, nefarable power and electromagnetic side-channel attacks [S54] could be carried out.

In recent years, a paradigm in secure computing has emerged themed around its reliance upon hardware based *Trusted Execution Environment* (TEE), realized in software through enclave programming [SYG$^+$]. CPU vendors provided TEE environments through microarchitectural extensions for instance; ARM provided ARM-TrustZone [NMB$^+$16] and similarly, Intel prior to 2015 provided TPM+TXT [WR09] and later Software Guard Extensions (SGX) [MAA$^+$16].

Intel SGX creates an isolated execution environment called as *enclave*; which provides a protected, secure and isolated sandbox to code segments in a user process. In such aforementioned code segments, sensitive computations are carried out, which rely on some secret information such as cryptographic attestation. The notion here being to protect this user process from supervisory software such as OS or hypervisor, which is considered untrusted. However, in recent years various microarchitectural attacks have been demonstrated to break the security provided by SGX, despite strong isolation guarantees of this mechanism [S78], [S77].

Modern microarchitectures provide an enhanced type of program execution, termed as *Speculative Execution*. In a nutshell, processors can speculatively fetch and execute a stream of instructions, ahead of time and precompute the outcome. If program actually proceedes along this aforementioned path the precomputed results are already available, thus, improving overall throughput. However if not, aforementioned outcomes are discarded, and the stream of instructions is simply retired an execution state is rolled-back. Regardless, of the sanity and integrity of execution state, speculative execution leaves behind trails of temporary microarchitectural state changes, such as entries in the cache and/or usage of execution units. Existing research revealed a number of side-channel attacks exploiting aforementioned state changes to exfiltrate sensitive information, which otherwise was not possible [S71], [S8].

Prefetching is among few of the employed methods besides speculative execution and branch prediction; to enhance performance, in modern CPUs. Hardware prefetchers are those microarchitectural units that perform the task of prefetching and handle associated aspects. In a nutshell, prefetchers increase the cache hit rate through speculatively prefetching data that would potentially be requested in future. For this task, patterns of memory access requests from CPU are observed and speculative data fetches are issued. Prefetchers exist in a hierarchy among all levels of cache-to-memory, such as L3-L2, L2-L1. Details of inner workings of prefetchers are usually proprietary and vendor specific that are subject to change as microarchitecture evolves. Broadly speaking prefetchers are categorized into *next line prefetchers* and *stride prefetchers*; the former simply prefetches the next line for a current cache-line in use, while the latter learns patterns of memory access to establish if there happens to be stride between data accesses. Recent research revealed that prefetchers can be exploited to establish a high-bandwidth covert-channel across processes, without relying on cache memory (whose role is typically essential in the attack dynamics), which makes such channels much more stealtheir and possess the ability to transmit information in the order of several tens of KBps [S38].

PCIe (Peripheral Component Interconnect *express*) is the de-facto protocol through which CPU and high-speed peripheral devices(GPU, NIC, etc.) are connected. In server and data-center settings, ever increasing number of peripheral devices are needed to be connected on a single machine. However, Intel CPUs offer fixed and limited number of PCIe interfaces, which is inadequate to connect many devices to available interfaces. However, this inadequacy is resolved through employing PCIe switches thus, allowing desired number of devices to be connected. However as a side effect at times congestion and contention do happen when multiple devices are contending for bandwidth. Although, such contentions do affect performance of connected devices, nonetheless, they also threaten the security and privacy of data transfers. If suitably equipped, an adversary can leverage PCI contention to his advantage and exfiltrate or infer sensitive information through this channel. In a recent study [S53], an adversary has been demonstrated to be able to directly access contents of victim's memory in a cloud based environment through aforementioned channel.

Modern CPUs are equipped with a range of onboard electronic modules to ensure operational reliability despite changing conditions in the surrounding environment. These modules are meant to manage power, regulate voltages, protect against excessive temperature, sense current draw and so on. However, these modules are not devoid of leakage, through which an adversary can snoop onto the inner workings of a CPU core. Typically in adversarial settings, power and electromagnetic emanations are leveraged [S39] to either infer or affect the inner state of CPU core.

In the preceding account, we observe that artefacts leading up to compromised security and privacy are not limited solely to one specific aspect of microarchitecture. However, they span across physical components such as cache, memory, registers and interconnects; to name a few. Nonetheless, they also span across executional mechanics of speculation, prefetching and so on. We also observe that, aforementioned artefacts spread across contention, timing and access behaviors; last but not least, interestingly, the onboard electronics contributes to artefact emanations in the power consumptions and electromagnetic emissions.

3.4.2 How some of the recent microarchitectural surfaces were crafted and turned into feasible attack corridors?

In the preceding RQ (Section 3.4.1), we presented a highlevel overview of the artefacts emanating from microarchitectural intricacies which ultimately are leveraged by adversaries to compromise security and privacy of computing system. In this RQ, we intend to cover the ways through which aforementioned artefacts are utilized to feasibly craft attack corridors. Based on the reportings and systematic classification from our primary studies(Table A.1), we subgrouped the studies on leaking component basis and will briefly summarize to highlight, how attack corridors had been crafted. Towards the end of this RQ, we will present notable mentions from our primary studies, which in our subjective opinion stand out from other works and present a notable research contribution.

Cache subsystem

From our curated set of primary studies, microarchitectural cache-attacks appear as a frequent theme. In the following, we will briefly summarize and highlight the aspects through which this involvement was observed.

Fernando et al. [S70] audits strength of cache mapping functions through their proposed framework and thus, reveal several vulnerabilities. The study emphasizes that sophisticated mapping functions, obfuscating address mapping to cache sets are needed. The authors, pointed out a malicious process can significantly infer portion of encryption key, utilized by a victim process, through indirect observation of cache sets being accessed. However, this approach requires, in-advance knowledge about how addresses are mapped onto cache sets by the hardware. This cache set mapping is performed through specific cache mapping functions, the details of which are often proprietary[YGL+15]. Weak cache mapping functions reveal a subset of the address bits that can be inferred by malicious processes. In this regard, one-to-one mapping functions were found to be weakest, which can reveal all n bits of set-index. Moreover, larger memory footprint also leaks a significant portion of cache tag. Thus, information-theoretic security of mapping function is absolutely essential.

In contrast, caches are found to be leaking, on-going data accesses of concurrently executing processes, through fine-grained timing observations. In such a setting, a malicious process manipulates the cache intentionally to either transmit information to a co-collaborating process, or to infer secret data being accessed by a victim process. In such settings, cache access patterns of the spy process happen to conflict with in-cache data being processed by a victim process. Such conflicts lead to data evictions from cache lines, followed by data loads for affected process. These events of evictions followed by loads, are observable in varying access latency which can be captured through fine-grained time measurements, made by a different process executing concurrently. To serve malicious intentions, such time measurements can be leveraged to pinpoint the cache lines being in current use by a victim process, which may be processing sensitive data. This information leakage through timing channels is a prevalent threat as caches expose a large attack surface and they are difficult to mitigate, since malicious processes merely modulate the timing behavior and do not leave any other malicious traces. This aspect has been of particular focus in some of the primary studies such as [S67], [S15] discussed in Section 3.4.3 of this work.

Memory subsystem

Semal et al. [S21] present two novel microarchitectural covert channel attacks demonstrating a vulnerability surrounding scheduler of memory controller. These attacks are particularly potent in the context of cloud based virtualization environment. The underpinnings of the attack dynamics involve a malicious pair of sender and receiver processes sharing regions of DRAM banks. The sender process continuously create intentional memory-bank conflicts to encode a covert message; this as a side-effect making the channel scheduler in memory controller to modulate latency, intended for receiving process. The receiver retrieves covert message by continuously performing uncached memory accesses in a pre-determined memory bank.

Lindemann et al. [S33] revealed that memory deduplication mechanism can be exploited, and demonstrated that a malicious virtual machine can identify software configurations of co-located virtual machines sharing physical memory of host. The attacker first establishes the knowledge about the exact subset of memory pages that would uniquely identify, broadly an application and specifically its version. This subset of memory pages is referred as application signature. The attacker overwrites the signature pages believed to be present in victim VM. The attacker silently waits for deduplication to take place, and afterwards rewrites the signature pages and measures the time needed. The time measurements serve as an indicator about the pages that've been overwritten during deduplication and infers from this information, if that specific application and its corresponding version is present on the victim machine. Similarly, Gulmezoglu et al. [S44] have relied on deduplication mechanism as a pre-step to carry out their proposed cross-VM attack on AES implementation.

Semal et al. [S64] cited a memory order buffer attack resulted as a side-effect of write-after-read hazards, also known as 4k-aliasing. The said hazard causes load/store bandwidth to drop, which can be leveraged to create a covert channel between two hyperthreads of a CPU core. In essence, the sender either fills or empties a 4k long buffer, meanwhile the receiver continuously probes load addresses on page-aligned boundaries. The probing operations reveal pattern of fills manifested by the sender.

Shi et al. [S7] have assessed the implications of unencrypted memory addresses while fetching data, in the context of secure processors. While the actual data may be encrypted but not necessarily the actual address being presented on the system bus, connecting secure processor and memory. The data stays in encrypted state within the memory, but once it is being processed, its decrypted on-the-fly. However, speculative execution poses a challenge; as it can potentially enables an adversary to snoop on this otherwise encrypted data. During speculative execution, this encrypted data is fetched ahead of the time and being decrypted for subsequent processing. An adversary can issue, a plain memory address of choice, and thus infer about the contents of encrypted data, cashing-in upon this speculative decryption. Thus, the ability of unencrypted data being snooped, in relation to knowledge of its memory address, defeats the main purpose of a secure processor. The authors propose remedies to this issue by proposing a number of alternate designs of a speculative pipeline for secure processors, integrating secure decryption with integrity verification. Furthermore, the authors evaluate the performance and security trade-offs of each of those designs.

Microarchitectural Buffers

Barenghi et al. [S59] have investigated the extent of microarchitectural information leakage attributed to execution pipeline buffers, and found out that significant information leakage was observed in the power traces, obtained during compute-intensive program execution. The observed leakage was uncorrelated to data-dependencies among instructions but rather the order of register allocation for program loads/stores. Moreover, minor changes in the order of register allocation potentially lead to exploitable vulnerabilities. In particular, Issue and Execution state (IS/EX); Execution and Writeback (EX/WB) ; ALU output buffers and Memory Data Register were main contributors of leakage which tended to get highly influenced from the order of register allocation. In [S54] the authors also found the role of pipeline buffers and associated functional units to cause power-based information leakage of various degrees.

Schluter et al. [S71] have also focused on the role of buffers in information leakage. The study investigates role of line-fill buffers (LFBs) in harboring microarchitectural data sampling attacks. Such attacks exfiltrate in-flight data of concurrent processes executing on logical cores during speculative execution in the event of page faults. Moreover, the study cites the role of store buffers, fill buffers and bus configuration registers as culprits of leakage.

Kim et al. [S58] have exploited return stack buffer (RSB) to lengthen the window of transient execution and leverage this to create a more potent variant of Meltdown [LSG$^+$18] attack. Their proposed exploitation technique enables establishment of covert channel without requiring a context switch and provides a better tolerance to noise based countermeasures.

Trusted Execution Environments (TEE)

In our curated set of primary studies TEE exploitations were another prominent theme. In the following we will briefly summarize those studies.

Gysenlik et al. [S82] show, in a novel attack, that legacy features for backward compatibility of x86 instruction set can be leveraged to compromise security of 32-bit SGX enclave. The attack abuses x86 segmentation unit to reveal enclaved memory accesses at granularity of page-level and in more favorable conditions to even byte-level. In essence, their presented attack loads the segmentation unit registers with an engineered configuration, which will ultimately lead to either a general protection fault or an ordinary page fault. The pattern of page faults reveals secret-information dependent memory accesses inside the enclave. From this information structure of sensitive code doing secret processing is inferred and control-flow can be spied upon. This study is notable in this regard to be first, exposing avenues for newer attacks exploiting legacy x86 features, which will remain part of Intel CPUs for backward compatibility in the foreseeable future.

Moghimi et al. [S78] demonstrate, secret information retrieval from a SGX enclave. Their attack exploits false dependence of memory read-after-write to serialize access to specific 4-byte memory blocks of a victim process, performing data retrieval from enclave-memory while executing inside an enclave. This serialization results in an observable latency of, otherwise a constant-time code implementation. Analysis of these patterns of latency, reveals the sequences of table lookup, which serves as crucial step to exfiltrate the secret cryptographic key.

Schwarz et al. [S77] present a practical SGX enclave based malware. Their work details an attack from a compromised malicious SGX enclave targeting co-located enclaves. In their work, the malicious code exploits SGX enclave protection features to conceal itself from operating system and thus thwarting discovery while remaining stealthy. The malicious enclave can either attack other co-located enclaves, or can launch attack on secure docker containers on the machine. The underpinnings of this attack utilize Prime+Probe [LYG+15] technique to determine cache-access patterns through observing latency and consequently leveraging them to retreive an RSA private key from a victim.

Skarlatos et al. [S32] target privacy of SGX enclaves, through microarchitecture replay attacks. Such attacks rely on the hardware support to rollback and re-execute instructions under certain preconditions in a quasi-transient state. The attack details an SGX adversary, which exfiltrates secret information from enclave-private memory of a victim through making it, repeatedly replay on page-faults. The repeated replays enable the adversary to break the privacy of the enclave and learn about secret enclave data to the maximal extent possible. Their approach positions itself as a potent technique, which can function effectively, even in the presence of considerable noise, utilized, as a means to countermeasure.

Lang et al. [S72] point out tactics of a malicious adversary having a complete control over OS, through which traffic to-and-from the outside world to a secure enclave can be influenced. One potential way is being the ability to frequently interrupt and preempt execution inside enclaves from outside. Such repeated interruptions could potentially land an enclave into such meta states, through which information can be exfiltrated. Moreover, in such a setting the adversary being untrusted OS can block, delay, replay, and modify all communications issued from outside towards an enclave that would assist establishment of a weakened inner state feasible for carrying out attack.

Prefetchers

Up until recent past [Sze], microarchitectural attacks targeting prefetchers, in a practical setting, were non-existent, however recently researchers have finally exploited prefetchers after empirical investigation of their inner workings and leveraged this knowledge to craft successful attacks.

Patrick et al. [S38] present a microarchitectural covert channel attack targeting hardware prefetcher on modern Intel CPUs. The presented attack establishes a bi-directional, high-bandwidth covert channel, which is stealthier and can avoid detection. The attack employs stride prefetcher to differentiate between accesses to data blocks: when they had to be retrieved from memory versus if already been prefetched into the cache. Timing differences serve to make this distinction. While running concurrently, spy process intentionally engineer sequences of prime and evict operations from L3 cache to perturb the thus-so-far learnt sequences by stride prefetchers. The degree of perturbation is used to encode bits of covert message while consequently is decoded by receiver process, through monitoring prefetch behavior.

Prefetcher had been used to build a covert channel in the aforementioned study. However, in the case of cache side channel attacks, prefetching unintentionally serves to hinder attack's effectiveness. Prefetching speculatively brings data into cache line, which weakens the attacker's ability to distinguish whether cache line had been fetched on demand by the victim or had been speculatively brought for victim by prefetcher. Wang et al. [S46] pointed out this shortcoming and presented a work-around solution in order to enhance effectiveness of cache side channel attacks, thus making them more potent. The main challenge they addressed is to understand and reliably model the uncertainty in prefetching patterns, which originates due to undisclosed proprietary details of the inner-workings of prefetchers by CPU vendors. In their presented work, the authors reverse engineered these inner-workings for Intel CPUs and developed a statistical description of their behavior. This description is later leveraged to strategically craft and place probes to build enhanced versions of cache side channel attacks. In their work, they have show cased Flush+Reload [YF14] attack, which is more potent and equipped with ability to effectively minimize the disruptive behavior caused by prefetcher.

PCIe

Tan et al. [S53] uncovered this issue and presented attacks, through which adversary can learn sensitive information spanning from keystroke timings, visited webpages to machine learning models being used on a GPU; inferred from patterns of PCIe contention. Such attacks are particularly relevant in the settings of cloud computing and data-centers where co-resident virtual machines can snoop on each other. The underpinnings of their presented attack assume a pair of peripheral devices connected to same PCIe switch, one serves the adversary process while another serves the victim process. The adversary is interested to learn distinguishable patterns in IO latency routed through PCIe switch, and these patterns are post-processed through a supervised learning approach to infer victim's state. The adversary can intentionally choke PCIe switch to a degree that causes these patterns to emerge, as a side-effect to intermediate buffering of data transfers for reliability.

Performance Counters & GPUs

Graphical Processing Units (GPUs) became one of the must-have component of modern computer systems, meant to provide enhanced capabilities and performance for graphical workloads. However, unlike originally intended their sole purpose did not remain to process graphics, but they were leveraged to aid processing of data-intensive workloads. The GPU vendors aided this ability through providing access to multitude of inner APIs and on-board registers which aid development and profiling. However, under specific scenarios GPU, as a resource, when shared between multiple concurrent workloads, enables a spy application to snoop on the behavior of a victim.

Naghibijouybari et al. [S35] have presented a number of attacks focused on GPUs to showcase exfiltration of keystroke timings, website fingerprinting; and inferral of inner parameters of neural network. All these aforementioned attacks have been achieved through misusing low-level GPU APIs for malicious purposes; monitoring and thus pattern-mining the graphical memory allocation statistics; and monitoring values inside performance counters to infer upon state of GPU activity.

Performance counter are special onboard registers, which can be accessed for profiling the current state of a GPU or CPU. Naghibijouybari et al. [S35] have demonstrated the information leakage through GPU performance counters to a spy process, to infer the graphics workload being rendered for a victim process. This information was enough to fingerprint websites. In such setting, a victim process such as web browser renders the web graphics through GPU; the workload being rendered leaves a trail of deviated values in the performance counters. The spy process employs machine learning to classify these trails and can identify with perfect accuracy which websites are being visited by victim. Moreover, in a similar setting, the interleavings between spy, victim, and other processes due to GPU scheduling influence the values in performance counter, which enables spy process to infer scheduling order of workloads as well as infer keystroke timings of a victim process during its IO sessions.

Dutta et al. [S13] describe the opportunities for microarchitectural attacks among GPU to CPU and viceversa. The authors argue that such GPU-CPU cross-component attacks are novel and could enable an attacker with new capabilities. Moreover, any counter-measuring research will face unique challanges in these contexts. The authors describe two ways to exploit contention arising from shared components among GPU-CPU. To this end, they demonstrate attacks aimed to establish covert channels exploiting, shared LLC among Intel CPU and integrated GPU; and the ring-bus interconnect among GPU-CPU. In the first attack scenario, the malicious processes run on CPU and GPU go through handshake routine and synchronize themselves from there the actual attack involves a pre-agreed upon LLC cache set being Primed+Probed as a way to encode-and-transmit information. Similarly, the second attack employ contention to encode-and-transmit such that if both GPU-CPU simultaneously accessed ring-bus the contention observed is significantly higher than, if each of these components accessed one after another. Furthermore, the authors describe ways to further make such attacks more capable by employing GPU parallelism to enhance covert channel bandwidth.

Onboard Electronics

Sehatbakhsh et al. [S39] demonstrated microarchitectural security vulnerability through targeting onboard electronics of CPU core. In their work, the authors focused on voltage regulator module (VRM), which serves to stabilize voltage variations to CPU core, in presence of either power fluctuations or workload variations. Moreover, a CPU can minimize its power consumption through switching itself into one of the operating power states, via software configuration of VRM. The authors observed that, electromagnetic emanations from VRM were directly correlated with operating state of the CPU. Leveraging this behavior, a malicious adversary could establish a covert communication channel, through manipulating a VRM by switching among possible operating states and hence transmitting secret information, from victim to a remote system. In their work, the authors showcased keystroke logging through this approach.

Notable Mentions

For this RQ, we elected papers, which were categorized under *threat* during our study selection process. Moreover, we considered the papers published in the last three years, as we consider them to be relevantly recent. Also, we further narrowed down our selection to papers, which we felt standout in the way of their attack dynamics. It is to be noted, that by no means we intend to provide a detailed account of each study in this section, yet we highlight the aspects at a high level, which is suitable to answer the current RQ. Next, we present those papers.

Chowdurry et al. [S8] leverage this observation that conditional branch executions in a speculative path have an affect on the continuously maintained history of branch predictors. Regardless, whether such speculation leads ultimately to abortion; the branch history is never restored back to its original state. Thus, the internal state of branch predictor is permanently affected. An adversary can either passively observe or actively modulates this speculatively affected history, as means of covert communication with a malicious peer process. The information encoding is done through predicting outcome of a conditional branch, and predicting the branch target's address.

Trippel et al. [S9] leverage formal methodology to discover vulnerabilities under speculative execution and then, synthesize the corresponding exploit programs. From a given specification of microarchitecture represented as a special graph form; the approach reasons about the orderings and interleavings of the hardware execution events, in relation to a program execution. This reasoning, mines for indicative behaviors leading up to the exploit scenarios. Once an exploitable pattern is found, several programs can be synthesized in automated manner, exploiting the vulnerability in potentially feasible ways. The approach showcase its unrivaled ability by discovering several novel and known-existing vulnerabilities under pure speculative execution. Moreover, notable timing and cache based vulnerabilities have been uncovered through same approach .

Saeshwar et al. [S36] demonstrate a feasible cache covert channel that although work on the principle of cache-line flush instruction, yet is flush-less in nature. Attacks relying on cache-line flush instruction were known to be some of fastest ones. However, their inherent reliance on flush instruction requires sender and receiver in tight synchronization to maintain low error-rates. The authors conjuncture that even higher data rates could be delivered, if asynchronous coupling between sender and receiver could be established. They observe, sequentially accessing a large-enough address space (comparable to size of LLC) implicitly triggers cache-thrashing operation (i.e. where previously accessed entries from LLC are automatically evicted to create room for newer requests). Leveraging this property, a malicious sender, if consistently stays ahead, and sequentially access successive addresses in a large array; and a receiver, who slightly falls behind to observe the incurred LLC-misses; covert information can be encoded and thus exchanged. However, in reality speculative execution and prefetchers can cause major disruptions by polluting cache to this aforementioned scheme. Thus, a coarse synchronization between sender and receiver, of the order of once every few thousand accesses is required as means of addressing this limitation.

Schwarz et al. [S79] present NetSpectre; a network-enabled variant of Spectre attack [KHF$^+$19], capable of mounting the attack remotely. Typical Spectre requires local code execution targeted at cache on a victim machine. However, failure to achieve so, keeps the victim secure. The authors overcame the fundamental reliance of attack dynamics on cache eviction. The authors had previously researched on vulnerability of AVX-instructions (i.e. vector operation instructions for x86). Leveraging their existing understanding, they successfully circumvented the reliance on cache eviction. In a nutshell, AVX operations do exhibit operation-dependent timing differences, which can be exploited to create a timing-channel. Although the data exfiltration rate is of the order of few bits per hour, nonetheless the presented approach demonstrates a fundamental paradigm shift in microarchitectural attacks.

Han et al. [S23] introduce exploitation of a special-function MEE (memory encryption engine) shared cache, which is part of Intel SGX enclaves. In particular, MEE cache frequently keeps portion of integrity tree data structure, meant to serve integrity and freshness of enclave-private data. The exploitation relies on this key observation that, for each enclave-private data access, number of MEE cache access varies. This number is directly influenced by the state of the aforementioned tree structure. The attack approach adopts the foot steps of a typical Prime+Probe methodology, systematically forcing integrity tree updates through MEE cache. The pattern of updates are leveraged for covert channel communication in a cross-enclave setting.

3.4.3 How effective are the proposed countermeasures of microarchitectural side-channel attacks and whether these countermeasures are generalizable? Can these generalized countermeasures can predict/prevent zero-day attacks?

We identified the studies from our list of primary studies(Table A.1), in which the primary focus has been presenting a *countermeasure* approach to address a microarchitectural threat. In this regard, we found out that such studies discussed the scope and effectiveness of their presented approach to various degrees. To answer our current RQ, we chose to include a subset of those countermeasure-presenting studies; in which the effectiveness and incurred overhead had been evaluated and thus, discussed by their authors. In this regard, we chose not to include those studies, in which the approach had either been presented but performance overhead has not have been evaluated. In particular, we chose not to include studies, categorized as solution proposals, as they lacked or partially addressed aforementioned evaluations.

We observed that some primary studies in conjunction to numerical reporting also qualitatively state in adjectives, the degree of the effectiveness and performance overhead of their approach; which serves as convenient take-away for the reader. Such as *highly effective approach and negligible performance overhead*. On the contrary, other studies only relied on numerical reporting, such as percentage. However, such inconsistent reporting style, forced us to follow the following categorization scheme, utilized to group studies in order to aid answering this RQ. We opted to categorize the reported *effectiveness* to countermeasure in two levels alongside their notations as: highly-effective (H_C), and simply an effective (E_C) countermeasure. Similarly *performance-overhead* in three levels: low (L_o), moderate (M_o), and high (H_o) overhead.

We assigned H_C label to those studies where the authors described their approach's effectiveness in superlative adjectives such as (quite-, highly-, extremely- etc.) or the numerical reporting between 85-100%; and E_C for vice versa. Similarly, the performance-overhead reporting under, adjectives such as (negligible-, low-, practically-zero etc.) or numerical reporting between 0-5% has been assigned L_o; comparative degrees of adjectives such as *some-, tuneable-, moderate- etc.* or numerical reportings between 6-20% been assigned M_o; and for remaining cases H_o has been assigned.

Table 3.2 presents the study IDs selected for this RQ, study categorization, effectiveness-, and the degree of incurred overhead-; of the proposed approach in that study.

In the following, we will present the answer to the individual parts of this RQ.

Table 3.2: Categorization of primary studies presenting countermeasure approaches.

study id	effective	overhead	zeroday
S40	H_c	L_o	
S48	H_c	L_o	
S6	H_c	L_o	✓
S12	H_c	L_o	
S67	H_c	L_o	
S31	H_c	M_o	
S15	H_c	M_o	✓
S41	H_c	M_o	
S45	H_c	M_o	
S51	H_c	M_o	
S43	H_c	M_o	
S72	H_c	H_o	✓
S75	E_c	L_o	
S56	E_c	M_o	
S27	E_c	L_o	
S52	E_c	L_o	
S47	E_c	L_o	✓
S24	E_c	H_o	
S55	E_c	M_o	

Effectiveness and generalizability of proposed countermeasures

The studies selected (Table 3.2) to answer this RQ comprise 22%(19 out of 84) from our list of primary studies (Table A.1). Regarding effectiveness, 63%(12 out of 19) of studies have been classified highly-effective and remaining 36% as effective. Regarding, incurred overhead, 47%(9 out of 19) studies had reported low, 42% as moderate, and remaining 10% as high overhead. Ideally, a countermeasure demonstrating high effectiveness with negligible overhead is desirable, however, it may not always be feasible. In the following we will briefly outline the countermeasure approaches from the individual studies in Table 3.2.

In S67, Fang et al. present a countermeasure approach addressing cache based covert timing channels, which tend to be an essential vehicle in large number of microarchitectural attacks. The proposed approach selectively monitor specific regions of interest of cache(s) being participant in suspicious behavior. The cache-miss patterns of aforementioned regions are recorded and constantly analyzed; if needed the size of monitored regions can be adaptively increased or decreased. Any significant statistical deviations from predetermined normal behavior, raising alarm in those patterns triggers mitigating action by the proposed approach.

Panda et al. S12 present an approach to defend against last-level cache eviction based attacks. Such eviction based strategies open up avenues for cross-core attacks and had been prominent in the recent years. The proposed approach leverages back-invalidation-hits triggered prefetching behavior, which, as a positive side-effect, provides defense against aforementioned class of attacks. Back-invalidation-hits refers to the behavior, when a specific cache block(s) in the shared inclusive LLC, causes data *invalidation* at the private cache(s) of a CPU core. Moreover, this may also cause the prefetcher to preemptively bring-in newer data within affected regions of cache hierarchy. The proposed approach had been evaluated to be significantly effective and incurred negligible overheads.

Yuce et al. S6 address fault attacks, which are particularly relevant in the settings of embedded/IoT systems. In such attacks, the adversary engineers a controlled-fault in the microprocessor through careful manipulation of the operating conditions, such that the execution gets into a state, from which the secret information can be exfiltrated. In the proposed countermeasure the operating state of microprocessor is constantly monitored through various sensors and any fault-injection attempts triggers the mitigative actions. The microprocessor maintains a hardware checkpoint of critical system-state on per-clock basis. In the event of a fault-injection attempt the checkpoint is frozen and subsequently a secure trap handler is initiated in the software, which ultimately recovers the past known-good state and rolls-back to it; thus, thwarting attack attempt.

Kumar et al. S40 propose an approach to enhance security of crypto-processors against power and electromagnetic emissions, which could potentially be harvested remotely and leveraged to weaken and thus, compromise its security. The authors argue that contemporary approaches had secured, against attacks aiming to infer cryptographic key through time-domain analysis of harvested emissions, yet they remain prone to frequency-domain analysis. Time- and frequency-domain analysis are mathematical techniques from signal processing, used to quantitatively study the signal properties, such as time-varying behavior in relation to its composition of constituent frequencies. The authors propose a hardware design enhancement in, low-dropout regulator (LDO) module employing principles of non-linear digital control within its control-loop, leveraging controlled randomization. This in-effect provides resistance against both in frequency- and time-domain analysis. This proposed design enhancement has enabled AES and RSA modules in a crypto-processor to remarkably deter against aforementioned attacks.

Mane et al. S48 propose approach to systematically eliminate exploitable side channel leakage to enhance resilience of block ciphers(AES, DES, so on) in the context of embedded processors. The authors argue that aforementioned leakage roots from data-dependent electromagnetic signal transitions which manifest from data-dependent processing at microarchitectural level. The authors observed that aforementioned leakage can be effectively mitigated at the level of implementation had Dual-rail Precharge

Logic (DPL) been employed. DPL can be acheived as; every data bit and logic operation, should be stored and processed simultaneously alongside its complementary form. The authors prototype and then evaluate a soft-core CPU synthesized on FPGA, with a custom instruction set and optimized memory organization following the principles of DPA, and report promising results.

Zhang et al. S31 present a countermeasure approach to harden Intel SGX enclaves, from leaking secrets, against exploitations carried out through engineering controlled page-faults. In such exploitation attempts, the adversary relies on deterministic memory access patterns against a provided known input. The proposed countermeasure is based on emulating a secure memory subsystem leveraging an enhanced ORAM (oblivious RAM [PR10]) protocol to load code and data into a pair of virtual caches. These caches periodically shuffle and re-organize their contents, which in-effect poses an effective barrier for the adversary leveraging controlled page-faults. The proposed approach exhibits a tuneable performance overhead thus, enabling the system policy to balance the trade-off between security and performance.

Guo et al. S15 present a countermeasure approach to cache-timing attacks in the context of speculative execution; as intermediate transient states occasionally lead to vulnerable cache states, from which secret information exfiltration is possible through analyzing patterns in access latencies. The proposed approach, relies on symbolic execution to systematically explore the state space of a program during speculative execution at conditional branches, to analyze the side-effects on potential paths. The cache side effects along each such path are accumulated and leveraged to create leak predicates, which subsequently are utilized to perform cache-behavior analysis using constraint-solving. Paths leading up to sensitive states, potentially leaking exploitable information are thus known in advance and avoided through tailoring program execution.

Vougioukas et al. S41 focus on mitigating exploits relying on hardware branch predictors, which are crucial for a high-performing system. However, they had been part of security exploitations, forcing deliberate context switches to inadvertently alter inner state to influence the branch predictions. As a result, an adversary can alter/perturb the instruction flow of another context, unrelated to current execution context, thus, opening up to a exploitable state. In the presented approach, the authors propose branch-retention buffer as a novel mechanism to maintain isolated contexts during context switching, thus, preventing vulnerable history alteration in the presence of malicious attempts.

Kiriansky et al. S45 present an approach based on minimal hardware modification to defend against a broad class of cache-state based attacks, involving the ones relying on timing and speculative execution. The proposed approach utilizes strong isolation achieved through protection domains. These domains segregate the cache-hit/miss metadata information and line replacement as well as cache update policies. Based on the security requirements, the domains can be either adaptively granularized costing performance, and/or nested together to establish a more performant yet coarse grained policy. The authors furthermore argue that proposed countermeasure is inherently robust against speculative execution attacks, such as Spectre.

He et al. S51 address mitigation for flush-based cache attacks in the context of embedded/IoT systems based on ARM CPU cores. The authors highlight the potential security threat demonstrated by Spectre attacks leveraging flush mechanism, leading to potential exfiltration of sensitive information during speculative execution. The proposed countermeasure provides a secure flush operation, encapsulating the native flush instruction and mechanism to monitor its invocations. The invocations are constantly logged to monitor, and any sequence of suspiciously close flush invocations are used to zero-in on the malicious process, and attempts to flush are blocked; thus, safe guarding the system.

Pham et al. S43 address, mitigation for timing-channel based information exfiltration in the context of embedded/IoT systems based on the idea of program diversification. In the proposed approach, a tailored LLVM compiler infrastructure is used to generate custom instructions for security sensitive program regions. LLVM is an extendable compiler framework aimed to support program analysis and transformations for arbitrary software[LA04]. Aforementioned custom instructions exhibit diverse timing characteristics, each time they are executed. However, hardware support is required for such custom instructions. The authors showcase their proposed approach by synthesizing a soft-core CPU synthesized on FPGA. In effect, the presented approach effectively delivers mitigation of timing-channels up to 80%.

Lang et al. S72 present an approach to mitigate access/trace-driven cache-attacks on secure enclaves, provided by Intel SGX. The mitigative approach leverages multiple concurrent threads of execution inside the enclave to monopolize the whole CPU during security-critical computations. These threads either perform enclave related or dummy computations to keep all cores occupied, which as a result starves any potential adversary attempting to carryout an attack due to CPU unavailability. In effect, although the proposed method is effective, however, it has considerable performance overheads.

Wu et al. S55 highlight the role of Simultaneous Multithreading (SMT) in contention based attacks. They argue that experimental evidence reveals, SMT provides broader attack surface as it exposes more microarchitectural components per-core than cross-core. Moreover, known protection mechanisms against such attacks incur high performance overheads and often remain partially effective, against newer attack variants. To this end, they propose Partial-SMT, a mechanism that alters the core-scheduling policy to protect security critical programs against SMT contention based attack vectors. The core allocation policy works in conjunction to a tailored threading library, such that security-critical programs can be either scheduled exclusively to a dedicated core. Moreover, thread placement of other concurrent applications across remaining cores are done in such a way to minimize the opportunities of SMT contention based exploitation.

Payer et al. S75, presents an effective proof-of-concept countermeasure system for wide range of microarchitectural attacks, based on concept of intrusion detection. The proposed system, named HexPADS, employs a plugin based architecture, which can be equipped to provide defense against presently known and future attacks by developing and installing an attack specific plugin. The whole system leverages the dynamic statistics from hardware performance counters on per-program basis and monitors deviations from known normal behavior profiles. On a system-wide level the presented approach correlates profiles from multiple processes to establish if multiple processes are involved in an ongoing attack. In the evaluations, the author utilized cache-performance metrics to demonstrate detection of rowhammer, CAIN, and other cache based attacks effectively with acceptable performance overheads.

Busi et al. S56 addressed the design guidelines for a provably secure architectural extensions for small microprocessors, such as enclaved execution. Such small microprocessors stay particularly relevant in the arena of embedded/IoT systems, and remain prone to interrupt-based attacks. Certain use cases demand such small microprocessors be equipped with isolation mechanisms such as enclaves. However, existing research has demonstrated insecure interruption of execution inside enclaves can break its security and isolation. The authors propose that full abstraction of such a processor be used as formal criterion, and be instantiated to the concrete case of extending the architecture that supports enclaved execution which is provably secure for interruptibility. Moreover, the authors present guidelines to realize such a processor and how to lay out the criterion. Furthermore, the effectiveness of presented approach has been demonstrated by synthesizing an open source enclave-enabled microprocessor.

Chen et al. S47 similarly propose an approach to address the exploitable surface of Intel SGX enclaves to speculative execution attacks and untimely interruptions. The study argues that to date Intel had been unable to fully mitigate and fix the aforementioned vulnerable enclaves and they remain to stay exploitable in the foreseeable future. The approach presents a software based compiler-assisted tool for enclave executables, by embedding code regions protecting the secrets, through attestation, sealing and unsealing. This ensures that confidentiality of enclave memory remains intact and not to be compromised in lieu to exploitation attempts during speculative execution. The approach also relies on temporary disabling of enclave interrupts to carry out sensitive computation in interrupt-free execution windows. Moreover, the approach addresses an exploitable weakness in the authenticity flow of OS-to-enclave through authenticating the delegating process during entry, interruptable and resumption phases of communications, from enclave-to-OS and vice versa.

Liu et al. S27 addressed the vulnerability of I-cache (instruction cache) to leak secret information in cloud based environments across different virtual machines. In such exploitations, the sensitive information is leaked through I-cache due to secret dependent execution paths the cipher takes. The authors argue that these attacks pose a serious threat and evaluate the degree of suitability of various randomized-mapping schemes as countermeasure approach to the I-cache. Given a set of randomized mapping schemes, the presented approach leverages machine learning to build a classification matrix to quantitatively characterize the strength of a given scheme against a subject I-cache attack. Although, no scheme delivered a silver-bullet mitigation, however, the presented approach enables the system to craft an adaptive solution to provide an effective defense, taking in consideration the trade-off in performance vs overhead.

Hsaio et al. S52 also address the vulnerability of guest virtual machines to cross-VM cache attacks, in cloud environment. The authors propose an approach to instrument the hypervisor itself; based on Intel VT-x extension in order to monitor memory allocation and access behavior of resident guest VMs, at runtime. The approach proposes a novel hardware-assisted MMU (memory management unit) redirection mechanism, which transparently intercepts any memory accesses made on behalf of the guest VM within the hypervisor memory space, to be monitored and profiled on-demand. Such an approach requires modification neither within the guest VM, nor within the host VM; however mere instrumenting the hypervisor is adequate. Depending upon the configurable trade-off between level of security vs. performance use case sensitive defense posture can be delivered.

Belleville et al. S24, propose a preventive countermeasure approach to harden programs against microarchitectural side-channel attacks. The approach utilizes code polymorphism as a mechanism to add unpredictablity and variable behavior in sensitive code segments for execution. Furthermore, the approach can be configured to strengthen the program through lightweight specialized code generation at runtime on top of static code optimization during compilation. The results confirm that programs treated with aforementioned approach present strong security and meet acceptable performance overhead.

Prediction/Prevention of zero-day attacks

Yuce et al. S6 approach's envisage to tailor itself and prevent zeroday fault attacks on the microarchitecture of embedded/IoT processors. Fault injection remain an effective vector of attack to perturb the ongoing computation, such as cryptography. Such perturbation in effect derails the sanity of internal state of processor which in effect can open up potential sidechannels for adversarial exploitation. The presented approach leverages three essential elements of fault detection, critical-state checkpointing and rolling-back to last known good state of computation in the event of encountering a fault. The authors argue that adopting the core principles of their approach in a microarchitectural design would ensure effective resilience against a range of fault attacks including clock glitches, voltage starvation and EM pulses on the hardware side. Similarly, on the software side, a set of custom instruction when placed inside sensitive regions of code would safely take the execution from exception handling back to normal execution in a way to minimally pollute the internal state of microarchitectural elements of a processor. Thus, avoiding any potentially expoitable state from which attack can be carried out.

Guo et al. S15 present an approach to detect in-advance and thus avoid potential program paths, leading to potential timing channel based exploitation of CPU caches. The authors argue that, static program analysis may establish a program is secure against aforementioned exploitation under normal execution. However, it may open up potentially exploitative paths under speculative execution, thus rendering static analysis based guarantees incomplete. The authors present a preventative approach leveraging state-space exploration through symbolic execution, tailored for speculative program execution. The approach discovers the potential paths that could be exploited, and avoiding such paths during speculative or normal execution would guarantee safety against aforementioned timing attacks, even being zero-day.

Lang et al. S72 present an approach to safeguard Intel SGX enclaves against all known access- and trace-driven cache attacks. The authors argue that to date, these aformentioned enclaves remain vulnerable to those attacks and still remain vulnerable in foreseeable future, and thus demands a mitigative solution. The presented approach targets to monopolize the whole CPU, during security-critical computations inside enclaves, through spawning a number of dummy SGX threads. This whole group of threads is closely monitored against voilations of exclusive scheduling, abrupt termination and enclave exit events. This mechanism in all-together ensures that a potential adversary stays starved during secure computations and thus the said enclaves remain protected against existing and such foreseeable zero-day attacks.

Chen et al. S47 also present an approach to safeguard Intel SGX enclaves against speculative execution attacks, such as Foreshadow[VBMW$^+$18] and similar. The authors claim that to-date Intel has been unable to fully patch SGX vulnerabilities and future zero-day attacks remain a concern, thus the need remains for practical countermeasures. To this end, they present a software based solution in conjunction to assistance from otherwise, untrusted operating system. The approach augments to further harden sensitive code regions of the program employing enclave-supported secure computation. Moreover, external interrupts are temporarily disabled during windows of such computation inside enclaves, as they remain one of the prime enabler for such attacks. The approach, establishes a secure environment even during speculative execution and thus, curbing down the possibility of carrying out successful aforementioned attacks, zero-day or otherwise.

In this section the presented studies discussed the following general themes of countermeasures. Cache focused countermeasures remain a prominent and recurrent theme, as they are being frequently exploited in the underlying attack dynamics, regardless of the hardware type; be it workstation-, cloud- computing, or smaller embedded/IoT hardware. Moreover, cache exploitation even in the speculative execution flows is more troublesome as any potential countermeasure is not without penalizing system's performance. Another recurring theme that we observed is securing trusted-computing mechanisms such as Intel SGX. Despite, CPU vendor's security assurances, researchers have been able to circumvent and compromise the tight security mechanisms of these enclaves. In this regard, countermeasure approaches have presented several mitigation mechanisms but they incur performance overheads; unless the CPU vendor makes revisions and updates to the microarchitecture, trusted computing mechanism will deliver little faith in their security guarantees to the end users. Last but not least, a mix of software-only, custom-hardware assisted software based countermeasure approach mechanisms have been adopted by the authors.

3.4.4 Given the published countermeasures, how secure a system we can build against micro-architectural side-channel attacks and what lessons we can incorporate in this system-design process?

To guide the security-oriented architectural process, we've identified the relevant primary studies for this RQ and grouped them in the following categories: *Metrics, Modeling and Assessment* – groups those studies which can guide the architectural process to appropriately model, quantatively assess the security posture of addressed microarchitectural component(s); *Designflow and Synthesis* – groups those studies which would aid a hardware designer to integrate security evaluation techniques to assess the security posture during the design phase; *Verification* – groups those studies which would help the designer to throughly vet and verify the intended security posture of designed component against any unintended other side-effects and potential bugs; and lastly *Miscellaneous* – groups those studies that does not belong to the aforementioned categories but are relevant in the scope of this RQ.

Metrics, Modeling and Assessment

He et al. S30 focus on security assessment of caches, as they can leak sensitive data in unexpected exploitable ways, through microarchitectural channels. Existing research proposes a number of secure cache architectures to address security flaws. However, the authors argue that absence of reliable method for assessing the strength of these architectures remain unavailable. To this end, the authors propose a probabilistic information flow graph (PIFG) based approach to model the interactions between a given cache architecture, a malicious program, and a victim program. The PIFG model is leveraged to compute a quantitative measure for a cache's attack resilience. The proposed metric had been thoroughly evaluated against nine different cache architectures and proved its worth. Lack of such a proposed metric leaves the researchers to rely on simulations or hardware instrumentation based approaches to assess a cache security.

Deng et al. S26 propose an approach to determine and thoroughly explore, if a given cache architecture is prone to timing-channel vulnerabilities. Their approach is based on bounded model checking employing computation tree logic. Vulnerabilities to attacks are represented as logic formulas which model the potential execution paths. The utility of their approach suggests its strengths, as it was able to detect 28 types of cache attacks including 8 new types.

Callan et al. S5 propose a metric termed SAVAT to quantitatively measure the side-channel signal, created by single-instruction difference among execution of otherwise, two identical programs. This metric is significant in the sense of its granular applicability to the level of single instruction execution; which involves a large variety of microarchitectural and electronic activity. The metric can be utilized to pinpoint the vulnerable aspects of a microarchitecture under the influence of program execution. This would help designers to make informed decisions about mitigating the overall side-channel vulnerability of their design.

Yilmaz et al. S42 developed a metric to quantitatively measure, as to how much information could be transmitted by the execution of a particular sequence of instructions on a CPU. The metric is significant, in regards to aiding design of programs, and hardware, which would minimize inadvertant side-channel leakage. Moreover, leveraging this metric would help identify the vulnerable portions of programs involved in such leakage.

Design flow and synthesis

Barenghi et al. S76 proposes strengthening of existing FPGA design flow with integrated side channel security mechanisms. FPGAs, being hardware programmable, among other types of hardware SoC (system on chip) realizations, are used to synthesize soft core processors. The study argues that an earlier security-oriented feedback provided during design phase, to the hardware designer, aids to develop and thus deliver a security-hardened post implementation. This proposed technique provides precise contribution and action of events in microarchitectural components leading upto a leaking state. The experimental evaluation validate this idea as mitigative means to address power consumption side-channel leakage, leveraged by adversary to exfiltrate AES key being processed inside a IoT SoC. The lesson to be learnt underpins, augmenting design flow that provide an evaluation of security with respect to a particular class of attacks, is certainly beneficial and design tools should aid designer to factor in security of intended chip architectures.

Arsath et al. S54 propose module-wise microarchitectural security audit of a given processor design. The notion behind the proposed approach is to ultimately establish a methodology that universally safeguards all sensitive applications executing on said processor. Moreover, such a ground-up design approach leads to minimal performance degradation, power and area overheads. As a use case, the authors focused on power side-channel resilience of an open source RISC CPU. The CPU is first analyzed against set of benchmark applications and its overall power consumption, correlated to degree of information leakage, is obtained. Then, the approach takes in this data and the source RTL(Register-transfer Level) of design to analyze module-wise leakage such as register banks, pipeline buffers, execute stages and so on. The degree of leakage is found to be in direct-proportion to module's vulnerability for exploitation. Such weakness in design originates as a byproduct of automated translations in EDA(Electronic design automation) to the module specifications. Although such translations are functionally correct, nonetheless they are prone to security flaws; which are to be mitigated through redesign, considering its degree of leakage. Moreover, the authors advocate that data path obfuscation, significantly helps effectively reduce information leakage.

Lee et al. S49 exclusively highlight the relevance of ECC (elliptic curve cryptography) for portable applications such as data exchange over wireless channels in IoT. Generally ECC is computationally intensive, so dedicated hardware is provided for sufficient performance, yet its resilience to power attacks is often questionable. IoT and embedded hardware are further constrained and thus often mathematically limited flavor of ECC is employed, such as single finite field. The authors present a hardware-efficient and secure design to support a robust version of ECC, which is also attack resilient. The proposed solution relies on single-chip heterogeneous dual-processing-element (dual-PE) architecture capable of providing various types of PEs leveraging full pipelining. Moreover, the authors advocate aforementioned scheme can further benefit from a two-level memory hierarchy with a local memory synchronization scheme.

Bache et al. S60 propose a methodology towards timing and power side-channel resistant cryptographic processor based on ARX-cryptography. ARX (addition, rotation, and xor) algorithms only utilized these elemental operations. Adequate utilization of ARX based computation ensures ample confusion and diffusion properties. However, naive ARX based implementations are prone to power analysis attacks, yet they are robust against timing and other classes of attacks and often deemed desirable in IoT and smart card applications. The authors address aforementioned limitations and propose a novel application-specific processor design based around ARX theory and provides security against timing and power attacks through protecting data paths with a custom boolean masking scheme. Furthermore, the processor specific instruction set follows principles of *Threshold Implementation* for provable security. Moreover, the approach delivers moderate costs comparable to more sophisticated and protected hardware implementations.

Kiaei et al. S61 advocate for a software synthesis technique in time-sensitive embedded applications. Although precise execution time is not of strict importance, yet it is sufficient to meet real-time deadline. Within certain timing bounds, these applications can become agnostic to timing channel leakage. However, the program timings need to be data-independent and precise. The authors, describe a parallel synchronous software model, such that it leverages *N* concurrent threads executing on a processor with fixed word length. Each thread is treated as single-bit synchronous state machine. The resulting software support fine-grained parallel execution for each of the *N* threads, which run without contention. This approach eliminates the thread scheduling problem on one hand, and does not require customized instruction scheduling, and neither does architecture modifications to processor; as existing programs are simply transformed by this technique . The implementation of this design is achieved through describing a thread in HDL(hardware description language) semantics of a single-bit synchronous machine, which is subsequently utilized under logic synthesis and ultimately leads to executable code generation.

Oh et al. S37 discuss hardware-based defense for mitigating access-based side-channel attacks under speculative execution, targeted at Intel SGX enclaves. The authors discuss that state-of-art mitigation is based on ORAM (Oblivious RAM) protocol, which although adequate, yet incurs significant overhead. The proposed solution is based on PCI plug-in hardware based on FPGA, synthesized with their approach. The external nature of this approach provides a trusted storage service, in a completely isolated and secure environment. The storage service on one hand securely keeps SGX authentication secrets as well as verifies the authenticity and authorization for connection establishment between software residing within enclave and FPGA. The authors advocate from empirical results that the FPGA based solution outperformed three state-of-art ORAM approaches and could be adapted in real life. Moreover, the presented approach could serve as a beacon to adopt a synthesized hardware based security solution, where software based solutions are simply too prohibitive on overall performance.

Barenghi et al S59 argue that changes in CPU microarchitecture and ISA manifest into side-channel leakage behaviors. Even innocuous changes, such as as register allocation order leads to an exploitable vulnerability. Thus, microarchitectural features of target CPU should be kept in consideration, while assessing the side-channel leakage behavior of a software implementation. Moreover, the authors discovered that contention inside the pipeline buffers, caused critical information leakage, which could never had been spotted through mere static analysis of program's assembly.

Vougiokas et al. S41 provide design lesson on attack resistant and performant branch predictors. The authors argue that existing approaches in this case, sacrifice performance gains, acheived by branch prediction over side-channel security. To remain secure, existing approaches often liberally, rely on forcing *flush* operations, clearing internal prediction state. To this end, the authors introduce a quantitative metric termed transient-state prediction accuracy(TIPA). TIPA measure should serve to rank side-channel free branch predictor designs for future. The motivation behind TIPA lies in the idea that flushing branch predictor during transient state affects, the performance far worse than flushing during steady-state. For future, the designers should focus on security, evaluated against performance degradation during transient-state flushing.

Verification

Eldib et al. S29 describe an approach to formally assess the degree of security provided by a countermeasure technique. As a use case, the study picked power leakage side-channel, which is attributed to microarchitectural activity in CPU. Data-masking techniques are, although effective remediation to this problem; as they obfuscate through statistically de-correlate leaked information in contrast to actual underlying data . However, it is difficult to assess the degree of effectiveness of one such technique over another. Utilizing formal verification for security assessment is argued as a viable choice. The presented approach leverages series of quantifier-free first-order logic formulas to be given into an SMT(satisfiability modulo theory)-based verification algorithm; keeping in view to correctly encode statistical nature of the input in the problem representation. The presented empirical evaluation suggest practicality, effectiveness, and the scalability of this approach in the aforementioned setting.

Colvin et al. S17 argue that speculative execution makes it difficult to formally reason about security of software systems. Such a void has left software systems to fall victim of Spectre and Meltdown attacks, which are attributed to speculative execution. The role of cache, which also becomes a surface of secret value exfiltration, in all these attacks is even a difficult problem to be formally reasoned about. To address these problems the authors present an abstract high level semantic framework to formalize speculative execution and its side effects. High level nature of proposed semantics enable to model and study side effects of speculative executions on components, such as caches. This framework is found to be effective in discovering sources of information leaks caused by speculative execution. Using this methodology, vulnerabilities like Spectre had been discovered early-on during security assessment. Another study S22 deals with the same aforementioned problem and follow a similar approach. The authors outline the methodology to transform a realistic model of speculative execution into abstract one, which is further simplified and refined under formal verification, using a standard model checker. The approach is found to be feasible, practical and been demonstrated on a pipelined RISC processor.

Miscellaneous

Regazzoni et al. S4 shed light on emerging paradigm of *approximate computing*. This is an architectural paradigm where limited and controlled errors are tolerated during computation. Attributed to approximation faster, smaller, cost effective, and less power consuming hardware circuits can be built. However, more research needs to be carried out in this area, as this area is still in its infancy. Resistance to fault injection attacks and resilience to side-channel attacks are among the security benefits, that approximate computing can deliver.

Grimsdal et al. S69 explore the feasibility of adopting microkernels to thwart microarchitectural attacks. In their work, three microkernels have been evaluated, all of which provided strict process isolation mechanisms. The authors conjuncture that strict process isolation is effective against limiting the effectiveness of microarchitectural attacks such as Meltdown and Spectre.

Nabeel et al. S11 advocate for the need to new processor designs that natively provide support for data privacy through cryptography. They argue that microarchitectural optimizations in the recent years paved the ways for microarchitectural vulnerabilities that caused attacks such as Meltdown and Spectre. They present their findings with regards to a secure co-processor design, aimed to deliver privacy through leveraging homomorphic encryption in the data paths. They chose to integrate their design as a co-processor linked through communication bus to main processor. Depending on data privacy settings the main processor can opt to delegate sensitive computation to this co-processor to guarantee data security.

Mao et al. S57 advocate for on-board reconfigurable hardware architectures, capable of on-demand software configuration. They discuss the exploitative role of caches in various microarchitectural attacks. Software based mitigations had impact on the system's performance, however a hardware based solution would be flexible and circumvent such penalties. The approach advocates the usage of FPGAs for such demands. The authors, synthesized a soft-core processor on an FPGA, whose architecture and features can be reconfigured through software. In their use case, this feature is used to create a hardware attack detection module and deliver a tailored hardware based mitigation for cache timing attacks.

Seuscheck et al. S14 highlight that embedded/IoT systems remain most vulnerable to side-channel attacks. Masking scheme based mitigations although effective, yet remain prone to the unwanted correlation based leakage, caused by different registers on the same shared CPU. Software cryptographic implementations on embedded/IoT systems remain particular candidate for such exploitation. Arbitrary mapping of registers to intermediate values of software cipher can result in such leakage. The study empirically investigates and reveals feasible exfiltration of sensitive data through these means. To rectify the authors propose that compiler should sensibly map intermediate values to registers.

Architecting systems that are microarchitectural exploitation proof in both arenas of hardware and software remains an actively researched topic. However, the lessons to be incorporated are evolving as more exploitations are being discovered and remedied. However, acheiving a fine tradeoff between delivering performance and security together remain still difficult problem, as both of these facets are often orthogonal. The designers are often faced with the challanges to produce a design within available monetary budget and available timeline. However, as research is being carried out, intermediate lessons are being learnt, quantitative measures to evaluate security-oriented facets of designs are being introduced and we are hoping for a more mature and security-oriented design flows becoming the common norm.

3.5 Threats to validity and future work

Systematic mapping studies, although capture research focus and trends within the literature, they do not delve into the details and quality of reportings and findings from the primary studies. Although researchers tend to implicitly pick quality works yet, they are often limited by selection process and classification criteria. If desired otherwise, a focused literature review with narrower scope is recommended.

Moreover, systematic mapping studies are empirical in nature and therefore, threats to validity such as construct validity and internal validity could exist. To address, threats to validity surrounding selection, screening and classification process, we opted to follow footsteps of previous mapping studies as to avoid potential validity issue; yet we chose to flex and adopt those footsteps, tailored to suit the focus and scope of our work. We also relied on more than one iteration(s) during selection, screening and classification process to minimize potential misjudgements and relied on team discussions to reach on mutual concensus. We also relied on three standard scientific databases, due to the fact they are well-known and considered standard, however, we do not neglect the possibility that inclusion of more research databases had potentially further improved the quality of our reportings or expanded the selection of primary studies. Due to limited manpower and resources available at our expendature, we did not had the opportunity to undertake multiple detailed reviews of complete set of papers. However, it is our firm belief that our selected pool of primary studies represent large and diverse aspects of microarchitecture security research, and does present an overall accurate and precise picture.

We belief that our adopted classification scheme is suitable and fits the focus of this work. Moreoever, aforementioned scheme is highly reusable and can be applied to capture new research trends by doing a similar similar study in the future. Last but not the least, inclusion of grey literature outside academia such as: whitepapers, patents, webpages, technical reports and so on; could be valuable and reveal emerging themes rather quickly, in this arena of cybersecurity. However, such an effort is manually demanding and resource intensive which would not be practical.

3.6 Concluding remarks

Our this systematic mapping study, presents a review of state of knowledge from the curated set of primary studies in the field of microarchitecture sidechannel security offenses, defences and measures to improve the security posture of computing systems. Directed by the review protocols, we relied on three standard databases for scientific literature and located 379 articles which further underwent through screening within the protocol constraints and identified 84 articles as primary studies for this work. The classification scheme utilized seven classes for the task to locate the culprit components and vectors responsible for microarchitectural exploits. Furthermore, we also curated answers for our target research questions from the primary studies. We questioned and answered the following aspects such as: aspects of microarchitectural leakage; how attack corridors been created; degree to which existing countermeasures are effective; and how to architect systems with improved resilience against such issues. Moreover, we also identified the research spots where research emphasis had been present as well as the spots which may get further attention. Lastly, we observed that over the course of last five years there has been a linear growth in the number of publications, which directly reflect the importance and the attention of research being carried out in this arena of cybersecurity.

CHAPTER 4

Detector$^+$

Side channels enable an attacker to infer information about a secret by measuring and analyzing the information unintentionally leaked by a computing system, such as execution times and power consumption. Over the recent years, research on the *side channel attacks* has revealed numerous novel ways to exfiltrate secret information, which is otherwise proved to be challenging [ZAB07, Sze19, BWM17].

An important category of side channel attacks, which is the focus of this paper, is *timing attacks* [Ber05, OST06, Per05], such as Meltdown [LSG$^+$18], Evict+Reload [LGS$^+$16], Flush+Flush [GMWM16], Flush+Reload [YF14], and Prime+Probe [LYG$^+$15]. At a very high level, timing attacks exploit the differences between the execution times of certain operations. For example, the Meltdown attack [LSG$^+$18] leverages the observable time differences between fetching data from the cache and from the RAM memory to exfiltrate private data belonging to other processes. Similarly, cache-based timing attacks, including Evict+Reload, Flush+Flush, Flush+Reload, and Prime+Probe, analyze the time differences between cache and memory fetches to infer the secret keys processed by cryptographic applications [LGS$^+$16, GMWM16, YF14, LYG$^+$15].

In a series of previous works, we demonstrated that side channel attacks, which leverage information unintentionally leaked by the systems under attack, ironically leak information by themselves through the same or related channels, which can be analyzed to detect, isolate, and prevent ongoing attacks at runtime [KDYS19, AGYS20, CSY16].

More specifically, in [KDYS19], we have developed a number of approaches for detecting cache-based timing attacks, which operate by monitoring the contentions in L1 cache memory and emitting warnings about possible ongoing attacks when the contentions reach a "suspicious" level. In [CSY16], we have monitored the contentions in L3 cache memory to detect the same or similar types of attacks by identifying similarities in cache access patterns between processes and known attacks. And, in [AGYS20], we have monitored segmentation faults occurring at memory addresses that are close to each other to detect, isolate, and prevent the Meltdown attacks.

In all of these aforementioned works, we obtained quite promising results [KDYS19, CSY16, AGYS20]. One issue, however, was that we had to develop a different approach for each type of attack. In particular, we had to analyze each attack in isolation and figure out what needs to be monitored (e.g., L1/L3 cache memory accesses or segmentation faults), what type of data to be collected (e.g., L1/L3 miss ratio or addresses at which segmentation faults occur), how to analyze the collected data, and when and how to take the countermeasures to prevent the ongoing attacks.

We, therefore, believe that developing a specialized approach for detecting each different type of timing attacks may not be sustainable in the long run. One issue is that sophisticated strategies may need to be developed to ensure the interference-free deployment of multiple detection approaches, so that the systems can reliably be protected from different types of attacks. Similarly, the collective accuracy of the existing attack detection approaches [KDYS19, CSY16, AGYS20, WFS14, WSG$^+$20, GYCH19, WS12] in the presence of multiple different types of attacks carried out concurrently, is (to the best of our knowledge) yet to be evaluated. Another issue is that as the aforementioned strategy requires to analyze each timing attack in detail, it may not necessarily be suitable for zero-day attacks. For example, although the approaches proposed in [KDYS19, CSY16, AGYS20] has a chance of detecting previously unknown attacks that leverage the same shared resources (e.g., L1 and L3 cache memory) monitored by these approaches, an attack utilizing a completely different shared resource would render them useless.

In this work, we, therefore, develop a generic approach, called *Detector*$^+$ (named after Kleene plus), to detect, isolate, and prevent timing attacks. The proposed approach is based on a simple observation: All timing attacks need to measure time, but their timing behaviors differ from those of the benign applications, especially the ones running in the production environments. In this context, we define the *timing behavior* of a process as the "typical" durations, which are needed to be timed by the process. More specifically, we observe that the malicious processes, compared to benign processes, often need to measure the execution times of quite shorter-running operations, such as accessing a single memory location. Note that making a *time measurement*, i.e., measuring the execution time of an operation, requires two time readings: one before the operation and the other after the operation, such that the execution time is computed as the difference between these readings. And, when the duration to be timed, is short, these two time readings occur close to each other in time.

Detector$^+$, therefore, monitors the time readings at runtime on a per process basis. When the time difference between two consecutive readings is "suspiciously" low, Detector$^+$ marks the pair as a suspicious measurement and introduces noise into the actual measurement to prevent possible ongoing attacks. The sequence of suspicious time measurements are then analyzed by using a sliding window-based approach to pinpoint the malicious processes, so that appropriate actions can be taken in time. These countermeasures are, however, beyond the scope of this work.

To evaluate the proposed approach, we have conducted a series of experiments by using five well-known timing attacks (Meltdown [LSG$^+$18], Evict+Reload [LGS$^+$16], Flush+Flush [GMWM16], Flush+Reload [YF14], and Prime+Probe [LYG$^+$15]) together with a well-known suite of benign applications [pho19], representing the applications that are commonly encountered in production environments. To further evaluate Detector$^+$, we have also tested it on different variations of the aforementioned attacks, each of which represents a mechanism that an attacker can employ to become stealthier. In one type of variation, each attack was carried out concurrently by multiple processes. In another variation, multiple types of attacks were carried out concurrently. In all the experiments, Detector$^+$ detected all the malicious time measurements with almost a perfect accuracy, prevented all the attacks, and correctly pinpointed all the malicious processes involved in the attacks without any false positives after they have made a few time measurements with an average runtime overhead of 1.56%.

The remainder of paper is organized as follows: Section 4.1 provides background information on the timing attacks used in the paper; Section 4.2 presents the attacker model; Section 4.3 introduces the proposed approach; Section 4.4 presents the experiments; Section 4.5 evaluates the potential threats to validity; Section 4.6 discusses the countermeasures that can be taken against Detector$^+$; Section 4.8 presents related work; and Section 4.9 concludes with possible directions for future work.

4.1 Background

In this section, we present background information about the timing attacks used in the paper without any intention of discussing all the details. The interested reader can get more information about these attacks by following the citations provided.

4.1.1 Meltdown

In a Meltdown attack [LSG$^+$18], the malicious process aims to access memory locations that belong to other processes. To this end, the malicious process attempts to use the value of a byte, which it does not have any rights to access, as an index into an adversarial array, which is purposefully created by the malicious process. The attacker is aware of the fact the request will eventually fail with a runtime error, such as with a SIGSEGV signal representing a segmentation fault. Due to out-of-order execution, however, the error occurs after the indexed item in the adversarial array was brought to the cache memory, which was intentionally flushed by the malicious process before the access. Although, the value of the target byte is never returned to the malicious process, the microarchitectural state of the CPU has now changed, leaking information; the item, the index of which is the value of the target byte, is now in the cache memory. To figure out the index that was accessed, thus the value of the target byte, the malicious process then probes the cache by accessing the indices in its adversarial array and each access is timed. Since the cache was flushed by the malicious process before the access, the index that takes the shortest amount time to access would indicate that the respective item is actually fetched from the cache memory (rather than the RAM memory), which, in turn, reveals the value of the target byte. Other bytes can then be targeted (as needed) by using the same mechanism.

Note that there are different variations of the Meltdown attack [LSG$^+$18]. All of these variations, however, make the same type of time measurements. In particular, they all measure the time it takes to access a single memory location. Therefore, from the perspective of Detector$^+$, there is no difference between these variations. Consequently, we opted to use the original version of the attack (as described above) in this work without losing the generality.

4.1.2 Prime+Probe

The Prime+Probe attack [LYG$^+$15, OST06, WQAGK19, MAB$^+$18] has two steps; prime and probe. The malicious process carried out these steps one after another in a loop to exfiltrate information from a victim process. In the prime step, the malicious process fills the whole cache memory with its own data. It then spends an ample amount of idle time waiting for the victim process to utilize the cache. Then, in the probe step, the malicious process probes the data that it brought to the cache memory in the prime step to figure out the data (thus, the cache lines/sets) that was evicted from the cache. To this end, each memory access is timed and the ones that take longer than the others, indicate the ones that were evicted from the cache, presumably by the victim process. The malicious process then uses this information to infer a secret about the victim process (such as, the secret key processed by the victim [LYG$^+$15, OST06]).

4.1.3 Evict+Reload

The Evict+Reload attack [LGS$^+$16, GSM15] exfiltrates information from a victim process by figuring out how frequently the victim process uses different code segments in shared libraries. In the evict step, the malicious process evicts a portion of the shared library from the cache. As the victim process accesses the shared library the respective parts are brought to the cache. In the reload step, the malicious process accesses the portions that it evicted in the first step. Each access is timed. The accesses that take shorter time than others, indicate the parts of the shared library that was (presumably) accessed by the victim process.

4.1.4 Flush+Reload

The Flush+Reload attack [YF14, YB14] uses a special machine instruction, called clflush, to operate. In the flush step, the malicious process evicts the entries of interest from all levels of the cache hierarchy by using the aforementioned instruction. Then, in the reload step, the malicious process measures the time it takes to access the entries evicted in the first step. Shorter access times indicate the entries that were (presumably) brought to the cache by the victim process.

4.1.5 Flush+Flush

The Flush+Flush attack [GMWM16], as was the case with the Flush+Reload attack, leverages the clflush instruction. Unlike the Flush+Reload attack, however, this attack measures the execution time of the clflush instruction, rather than the execution time of a memory access. The rationale is that if the entry to be evicted is already in the cache, then clflush takes longer to execute as the entry needs to be removed from all levels of the cache hierarchy. Otherwise, clflush takes shorter time as there is nothing to be evicted.

4.2 Attacker Model

In this section, we present a number of definitions to model the timing attacks.

An attacker is a party, which controls either a single or group of user space processes on a target platform, with malicious intentions to exfiltrate sensitive information by snooping the private data manipulated by other processes.

A timing attacker is a type of attacker, who operates by utilizing the differences between the execution times of certain operations as an essential component of its attack mechanism.

The operations that need to be timed are short living operations. To measure the execution time of an operation, the timing attacker requires two time readings; one before the operation and another after the operation. Both readings are carried out by the same process executing on a machine where Detector^{+} is operational, such as on the same machine with the system under attack. Note that this does not prevent the attacker from using multiple processes in an attack. Furthermore, time measurements may need to be repeated both to factor out the noise and/or to exfiltrate more data.

The timing attacker has neither special privileges nor direct access to other processes' data. All the mechanisms for reading and/or measuring time is under the abstraction of a software layer, such as an operating system. The attacker can neither bypass these mechanisms nor tamper with the data collected by them.

Measuring the execution times of short-living operations requires quick consecutive time readings, resulting in a side channel per se, which can be monitored and used for detecting ongoing attacks.

Note that Proposition 4.2 is generic in the sense that regardless of the shared resources utilized in the attacks or the operations timed, as long as the attacks rely on quick consecutive time readings, it causes malicious processes exhibit a certain characteristic behavior, which can be used to distinguish them from benign processes. Furthermore, introducing noise into the suspicious time measurements can prevent the attacks or make it difficult for the attackers to correlate the observed behavior with the private information under attack. This, therefore, removes the need for developing specialized detection, isolation, and prevention mechanisms for different types of timing attacks, which requires the manual analysis of the shared resources leveraged, the operations used, and the specific mechanisms employed by the attacks. By the same token, Proposition 4.2 can also be used against zero-day timing attacks.

By Remark 2, there exists a detection methodology since the side-channel, which is unintentionally created by the timing attacker, cannot be eliminated.

This proposition is, indeed, strongly supported by our previous works [KDYS19, CSY16, AGYS20], where we developed specialized approaches for detecting the Meltdown and cache-based timing attacks, demonstrating that the side-channel attacks, in the process of leveraging the information leaked by victim systems, unintentionally leak information by themselves, which can be used for detecting, isolating, and preventing them.

4.3 Detector$^+$

Detector$^+$ monitors the time readings made by processes on a per process basis with the goal of identifying consecutive readings that are suspiciously too close to each other.

4.3.1 Approach

Algorithm 4.1 presents the method employed by Detector$^+$. Note that this algorithm is carried out every time a process attempts to read the time. Furthermore, as there may be

Algorithm 4.1: Detector$^+$

```
1  Input:
2    pid: PID of the process requesting to a time reading
3
4  curr_time ← readtime
5  pid.read_cnt++
6  if curr_time - pid.last_reading ≤delta_threshold then
7      pid.suspicious_reads++
8      if pid.read_cnt == window_size then
9          score ← pid.suspicious_reads / pid.read_cnt
10         if score ≥ warning_threshold then
11             emit a warning
12             pid.warning_cnt++
13             if pid.warning_cnt ≥ alarm_threshold then
14                 raise an alarm
15                 pid.warning_cnt ← 0
16             end
17         end
18         else
19             pid.read_cnt ← 0
20             pid.suspicious_reads ← 0
21             pid.warning_cnt ← 0
22         end
23     end
24     introduce noise
25     curr_time ← readtime
26 end
27 pid.last_reading ← curr_time
28 return curr_time
```

different ways for the processes to read the time depending on the underlying hardware and software platforms, we opt to provide the algorithm in a platform agnostic manner.

Every time a process requests a time reading, the time of the request is obtained (line 4) and compared to the last time the process requested a time reading (line 6). If the time difference between these two consecutive readings, which is, from now on, referred to as time delta (in short, *delta*), is lower than or equal to a predetermined threshold value, called *delta threshold* (*delta_threshold* in Algorithm 4.1), then the time delta (thus the pair of readings) is marked as suspicious (line 7).

The delta threshold parameter, being a hyper-parameter of Detector$^+$, needs to be set, such that the false positive rate as well as the false negative rate is minimized as much as possible. Note that any approach, which determines the delta threshold, such that the false positive rate is minimized, requires the presence of benign processes only for analysis. Whereas, the approaches based on minimizing the false negative rate, assume that the attacks are known a priori, such that time deltas can be collected from the malicious processes in a controlled manner for analysis. Therefore, in the presence of both the benign and malicious processes, the delta threshold can be chosen in a way that balances the false positive and false negative rates according to the needs. For this work, however, we followed the former approach by using a well known benchmarking suite of benign processes [pho19] to determine the delta threshold, without requiring any prior knowledge of the attacks. More specifically, we picked a threshold value, which results in a false positive rate of smaller than 0.01% (Section 4.4.2).

Once a suspicious time delta is detected, to thwart a possible ongoing attack, Detector$^+$ introduces noise into the time measurement (line 24), such that malicious process is prevented from extracting reliable information by analyzing the differences between the time measurements. Note that determining the best approach for introducing the noise is out of the scope of this paper. Consequently, we opted to use a simple, yet quite effective approach. In particular, we introduce a random amount of idle clock cycles (between 512 and 4352 clock cycles) by executing a random number of NOP (no-operations) instructions, causing slightly delayed time readings in the presence of suspicious time deltas. The results of our experiments strongly suggest that this approach can prevent the attacks from being successful or significantly reduce their success rates (Section 4.4.4).

Detector$^+$, not only detects and prevents the attacks, but also pinpoints the malicious processes carrying out the attacks. This is important as once the malicious processes are pinpointed, appropriate countermeasures can be taken to prevent the ongoing attacks or to reduce their harmful consequences.

To determine the malicious processes, we use a non-overlapping sliding window based approach, which can be configured with the help of 3 hyperparameters: *window_size*, *warning_threshold*, and *alarm_threshold*. The *window_size* parameter defines the size of the windows to be used for analysis, i.e., the number of consecutive time deltas in a window. Note that Detector$^+$ forms and analyzes the windows on a per process basis. If the ratio of the number of suspicious time deltas in a window exceeds a predetermined value (i.e., *warning_threshold*), a *warning* is emitted about the offending process (lines 10-12). And, if the warnings persist for a number of consecutive windows indicated by *alarm_threshold*, then an *alarm* is raised and the offending process is marked as suspicious (lines 13-15).

In the presence of an alarm, various countermeasures can be taken against the suspicious processes, such as terminating them, migrating them to different machines, or sandboxing them for further analysis. Such remediation strategies are, however, beyond the scope of this work.

4.3.2 Implementation

We have implemented Detector$^+$ in a Linux distribution, namely CentOS 7 with kernel v.3.10.0-957.5.1.el7.x86_64 and glibc v.2.19. In Linux (as is the case with the other modern operating systems), there are two ways of reading the time using the services provided by the operating system: by making a *system call* (in short, *syscall*) and by making a *virtual system call* (in short, *vsyscall*). Virtual system calls are an alternative mechanism provided by an operating system for a small number of frequently used system calls (such as the timing-related calls) to reduce the runtime overheads by avoiding context switches.

The operating system implements the vsyscalls by mapping a fixed-size (1024-byte) *virtual dynamic shared object* (*vdso*) into the address space of each process [vds11]. When a process requests a timing-related service of the operating system, the request is captured by the glibc library, which implements the core functionalities for the user-level processes. If vdso is enabled and a vsyscall is available for the requested service, glibc reroutes the request to the respective vsyscall, avoiding the context switch. Otherwise, the request is rerouted to a regular syscall, causing a context switch. Note that vdso can be enabled/disabled at will during boot time. And, regardless of whether the vdso is on or off, the user processes remain oblivious of the underlying dynamics.

Detector$^+$, therefore, instruments all the timing-related syscalls and vsyscalls with the probe given in Algorithm 4.1. For this study, we have modified the operating system kernel for the former and the wrappers provided by the glibc library for the latter.

Besides the syscalls and vsyscalls, the only mechanism (to the best of our knowledge) that one can employ to read the time, is to use the native `rdtsc` instruction (or its variations), which collectively will be referred to as *rdtsc* in the remainder of the paper.

One issue with rdtsc is that a process can use it to surreptitiously read the time without letting the operating system know. As this paper strongly suggests that being able to measure the time is an integral part of the timing attacks. For improved security, we are, therefore, a strong advocate of allowing accesses to the timing-related services only through privileged software/hardware entities, so that suspicious time measurements can be monitored and appropriate countermeasures can be taken. Indeed, the security threat exhibited by rdtsc (in particular, whether it should be banned from the user space or not) has been a topic of debate for a while [Oya19, Per05, cr009]. It is exactly for this reason that operating systems (with the help of the CPUs) provide facilities, which make rdtsc available only in ring 0, such that non-ring 0 accesses result in runtime failures. For example, in Linux, this is achieved by using the `prctl(PR_SET_TSC,...)` instruction [JLD$^+$17, cr009].

Note that, for Detector$^+$, it is not about disabling the user-level accesses to rdtsc, but about getting this instruction wrapped up by a privileged entity, so that all the accesses can be monitored and controlled. One way to achieve this could be to move the rdtsc accesses to a different protection ring, which has also been a topic of a debate [cr009]. For this study, we, therefore, instrumented all the binaries and source codes, such that Detector$^+$ is notified about every access to rdtsc by running the logic in Algorithm 4.1.

4.4 Experiments

To evaluate Detector$^+$, we have conducted a series of experiments. These experiments were specifically designed to address the following research questions: 1) Do timing attacks exhibit distinguishing timing behaviors? (Section 4.4.2); 2) Can the attackers be pinpointed? (Section 4.4.3); 3) Can the attacks be prevented? (Section 4.4.4); 4) Can the runtime overhead be kept at an acceptable level? (Section 4.4.5); and 5) Can the attack variations be detected? (Section 4.4.6).

4.4.1 Setup

Attack types. In the experiments, we have used 5 different timing attacks, namely Meltdown [LSG$^+$18], Evict+Reload[LGS$^+$16], Flush+Flush[GMWM16], Flush+Reload[YF14], and Prime+Probe[LYG$^+$15] (Section 4.1). We have chosen these attacks since they are well-known representatives of all the existing timing attacks and they have been used in many related works for evaluation purposes [LSG$^+$18, AGYS20, KDYS19, GSM15, YSG$^+$19, CVBS$^+$19, BHLY16, Yar16, WNQ$^+$19, ZXZ16, GES17].

Attack variations. We have also evaluated the proposed approach on two different attack variations (Section 4.4.6), both of which mimic some of the strategies that an attacker can use to become stealthier [AGYS20]. In one set of variations, each type of timing attack was carried out by multiple processes running concurrently (Section 4.4.6). In the other set of variations, multiple types of different timing attacks were carried out concurrently.

Benign processes. As the suite of benign processes, which we needed to evaluate whether the timing behaviors of the attacks differ from those of the benign processes as well as to measure the runtime overhead of the proposed approach, we used the Phoronix benchmarking suite [pho19]. We chose this suite because it represents a wide spectrum of scenarios, which are commonplace in today's production environments, including cryptographic and numerical computations, audio and video encoding, various CPU-, memory-, network- and disk-bound computations, and database operations. Furthermore, the Phoronix benchmarks have also been used in related works to evaluate preventive countermeasures against timing attacks [GES17, AGYS20].

Table 4.1: Phoronix benchmarks used in the experiments.

benchmark	ID	version	app cnt	sub-benchmarks	delta cnt	false positives (%)	overhead (%)
Audio Encoding	1	1.0.1	2	2	1.07 (+04)	0.2244	0.02
Cryptography	2	1.1.0	5	11	3.58 (+04)	0.137	1.55
Compression	3	1.0.1	5	5	6.94 (+04)	0.2002	1.43
Compilation	4	1.2.1	6	6	1.86 (+05)	0.0162	1.09
Video Encoding	5	1.1.0	3	3	4.10 (+05)	0.0784	0.86
Molecular Biology	6	1.0.2	1	1	1.26 (+06)	0.0198	0.004
Memory	7	1.1.0	5	12	2.57 (+06)	0.0002	2.00
Workstation	8	1.1.0	9	11	2.71 (+06)	0.0001	1.73
Multicore	9	1.2.0	16	17	3.36 (+06)	0.005	0.72
Gaming	10	1.0.1	2	5	1.65 (+07)	0.0003	0.73
Machine Learning	11	1.2.0	2	2	2.07 (+07)	0.0175	1.37
Network	12	1.1.0	2	6	2.15 (+07)	0	6.75
Database	13	1.1.0	2	5	2.33 (+07)	0.4851	0
Scientific Computing	14	1.0.0	2	9	3.09 (+07)	4.1563	1.42
Desktop Graphics	15	1.2.0	2	5	4.03 (+07)	0.0028	0.29
Disk	16	1.3.0	5	17	7.75 (+07)	0.2474	3.42
Server-Memory	17	1.1.3	30	56	1.60 (+08)	0.0019	1.01
Kernel	18	1.1.0	13	25	1.76 (+08)	0.1067	2.28
Server-CPU	19	1.0.0	24	36	2.28 (+08)	0.0135	1.38
Server	20	1.2.1	8	25	2.65 (+08)	2.4356	1.72
Overall	-	-	144	259	1.07 (+09)	0.007	1.56

Table 4.1 presents information about the Phoronix benchmarks that we were able to run on our computing platforms. The first five columns in this table, depict the name, ID, and the version of the benchmark as well as the the number of different applications and sub-benchmarks involved, respectively. The remaining columns of this table will be discussed later in Sections 4.4.2 and 4.4.5.

Operational framework. All the experiments were carried out on an E5630 Intel Xeon system equipped with 32 GB of RAM, 32 KB of L1, 256 KB of L2 and 12288 KB of L3 cache memory running CentOS v7 operating system with kernel v3.10.0-957.5.1.el7.x86_64.

4.4.2 Study 1: Do timing attacks exhibit distinguishing timing behaviors?

Our first research question was whether the timing behaviors of the attacks differ from those of the benign processes. If this is not the case, then Detector$^+$ will certainly fail to fulfill our claims. Note that, in this context, the *timing behavior* of a process indicates the "typical" durations, which are needed to be timed by the process.

To address this research question, we carried out a series of experiments to determine the delta thresholds for each timing mechanism, i.e., by using the system calls, virtual system calls, or the rdtsc instructions. We do this because different timing mechanisms can impose different amount of measurement noise. Note that, since Detector$^+$ is aware of the actual timing mechanism being used for each time measurement (as the instrumentation code specified in Algorithm 4.1 is inserted on a per timing mechanism basis), a different delta threshold can be used for each mechanism to determine the suspicious time deltas.

For this study, we opted to determine the delta thresholds based on the distributions of the time deltas obtained from the benign processes (Section 4.4.1), avoiding the need for knowing the attacks a priori. To this end, we had to run the Phoronix benchmarks, which took about 2 weeks of computation time, resulting in more than one billion (1.07 e09) time deltas (Table 4.1).

For this study, we needed to log all the deltas. Note that the aforementioned logging step is something, which is required only for carrying out the study, so that we can report our findings by performing a detailed analysis in an offline manner. That is, Detector$^+$ does not log the time deltas for later processing as it simply computes the time deltas and determines whether they are suspicious or not by making simple comparisons at runtime (Algorithm 4.1), requiring a constant amount of memory per process to operate.

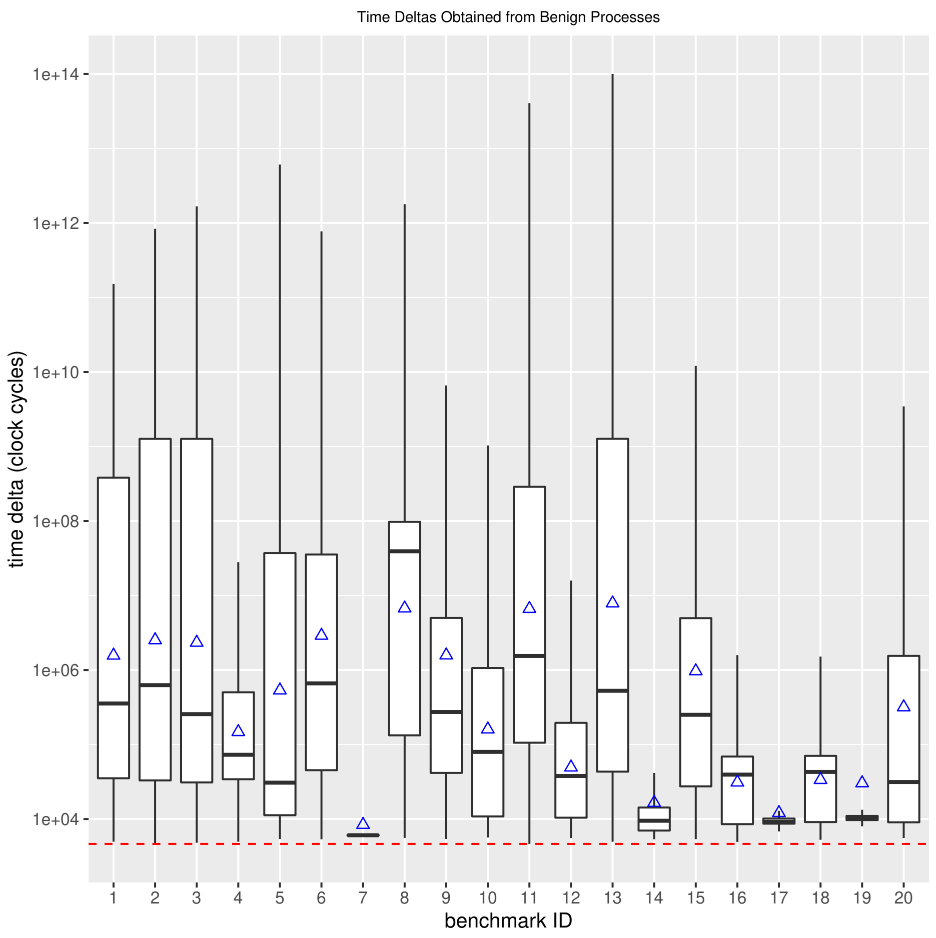

Figure 4.1: Distributions of the time deltas obtained from benign processes. The triangles depict the average time deltas and the line indicates the delta threshold of 4635.

It turned out that, for the computing platforms we used in the study, the most reliable and scalable way of collecting the data, was to carry the logging task in the OS kernel. We, therefore, disabled vdso and forwarded all the timing calls to the system calls. This, indeed, enabled us to compute a lower bound on the detection accuracy of the malicious timing deltas since, among all the different timing mechanisms, using the system calls was the one that introduced the highest level of noise in the measurements due to the context switches required. And, the more the measurement noise, the more the relative differences tend to diminish between the malicious and benign time measurements, thus making it more difficult to differentiate malicious deltas from the benign ones.

Figure 4.1 visualizes the time deltas we obtained from the benign processes (in clock cycles). Each box in the figure represents the distribution of time deltas obtained from a particular benchmark. The top and bottom bars of a box depict the first and third quartile of the distribution, respectively, i.e., half of the time deltas fall into the box. Furthermore, the median and mean values are represented by the middle bars and the triangle symbols associated with the boxes. The information about the actual

Algorithm 4.2: Pseudo-Attack Loop

```
1 ...
2 $start\_time \leftarrow readtime$
3 // malicious intent goes here
4 $end\_time \leftarrow readtime$
5 ...
```

benchmarks used in this study as well as the total number of time deltas observed in each benchmark can be found in Table 4.1.

Using the maximum false positive rate of 0.01% (see Section 4.3 for more discussion), we determined 4635 clock cycles as the delta threshold to be used (depicted by the dashed line in Figure 4.1). With this threshold, among all of the 1.07 e09 time deltas obtained from benign processes, only two of them (both of which occurred in benchmark ID=19) were, indeed, incorrectly marked as suspicious, resulting in an actual false positive rate of $\approx$ 0% (Table 4.1).

After determining the delta threshold using the benign processes, we used it on the time deltas obtained from the timing attacks to determine the false negative rate. To this end, we ran each attack about 30 seconds, which turns out to be sufficient amount of time for these attacks to succeed, and repeated the experiments 3 times. In total, we have collected about 188 million (18.84 e07) time deltas from the attacks.

We first observed that the time deltas obtained from malicious processes tend to come from multiple (typically 2) distributions. An in-depth analysis revealed that this is because the malicious processes typically make their measurements in a loop, such that after every time measurement, some computations are performed, for example, to store the measurements and/or to analyze them.

Algorithm 4.2 illustrates this phenomenon. At every iteration of the attack loop, a time measurement is made by using two time readings; one at line 2 and the other at line 4. Assuming, for example, that the loop iterates two times, four time readings would be intercepted by Detector$^+$: t_1, t_2, t_3, and t_4, where t_1 and t_2; and t_3 and t_4 correspond to the time readings at lines 2 and 4 in the first and second iteration of the loop, respectively. Thus, three time deltas are computed: $\Delta_1 = t_2 - t_1$, $\Delta_2 = t_3 - t_2$, and $\Delta_3 = t_4 - t_3$. Note, however, that only two of these deltas correspond to the actual time measurements that the attacker carries out: Δ_1 and Δ_3, each of which represents a measurement from line 2 to line 4. The other time delta, namely Δ_2, is an artifact of the monitoring process as the time deltas are computed without the knowledge of the contexts the processes may be in. More specifically, Δ_2 represents a time measurement from line 4 to line 2, which does not correspond to an actual time measurement made by the attacker.

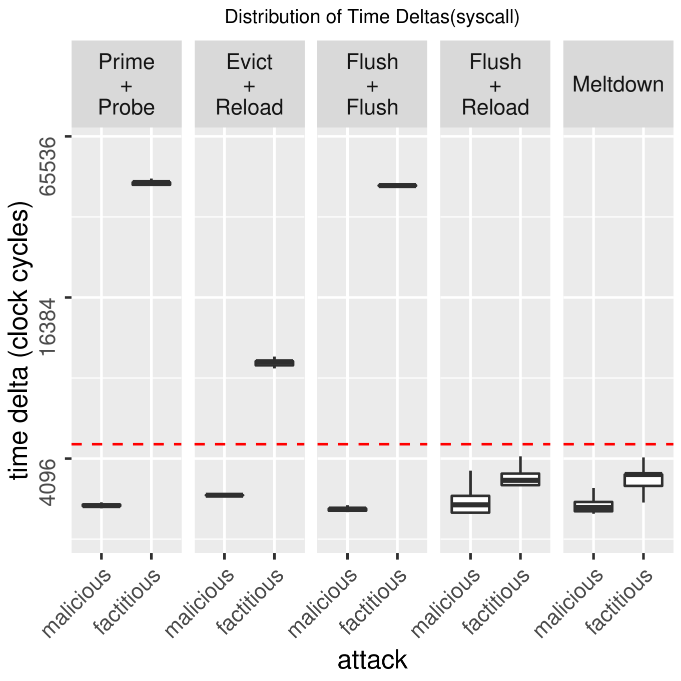

Figure 4.2: The distributions of the malicious and factitious time deltas obtained from different attacks when the system calls are used as the timing mechanism. The dashed line indicates the delta threshold of 4635 clock cycles.

Consequently, the time deltas obtained from such an attack loop would tend to come from two distributions; one representing the malicious measurements from line 2 to line 4 and the other representing the factitious measurements from line 4 to line 2. In the remainder of the paper, the time deltas that correspond to the actual time measurements that the attacker makes with malicious intentions, such as Δ_1 and Δ_3, are referred to as *malicious time deltas* (*malicious deltas*, for short). And, all the other time deltas obtained from an attack will be referred to as *factitious time deltas* (*factitious deltas*, for short).

Figure 4.2 demonstrates an advent of this phenomenon in our experiments by visualizing the distributions of the malicious and factitious time deltas obtained from different attacks. For a given attack, the two distributions were significantly different from each other. Note that, for this work, we are primarily concerned with the malicious deltas.

Applying the selected delta threshold of 4635 (depicted by the dashed line in Figure 4.2), which roughly aligned with the 99.99th quantile of the malicious deltas obtained from the attacks, we obtained a false negative rate of 0.0009%. That is, only 0.0009% of the malicious deltas were above the threshold.

All told, when the time deltas obtained from both the benign and malicious processes were used, we obtained an almost perfect accuracy in detecting malicious deltas (i.e., an accuracy of $\approx$ 100% with a false positive rate of $\approx$ 0% and a false negative rate of 0.0009%), strongly supporting our hypothesis that timing attacks exhibit distinguish-

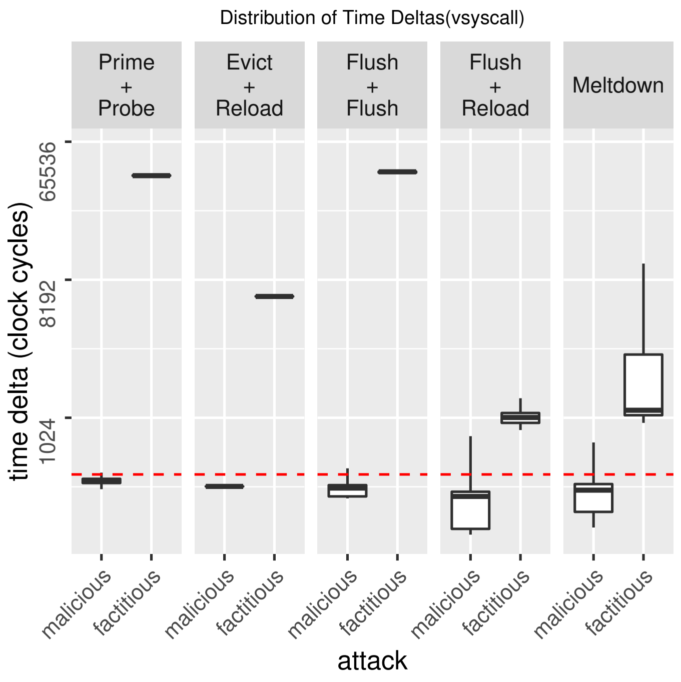

Figure 4.3: The distributions of the malicious and factitious time deltas obtained from different attacks when the virtual system calls are used as the timing mechanism. The dashed line indicates the delta threshold of 436 clock cycles.

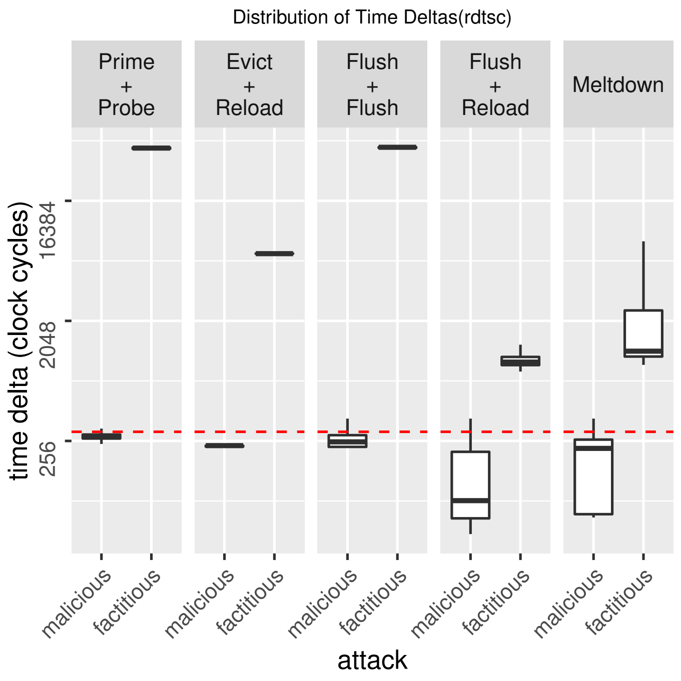

Figure 4.4: The distributions of the malicious and factitious time deltas obtained from different attacks when the rdtsc instructions are used as the timing mechanism. The dashed line indicates the delta threshold of 300 clock cycles.

able timing behaviors.

Repeating the experiments with the remaining types of timing mechanisms (e.g., vsyscalls and rdtsc) and focusing on the 99.99th quantile, resulted in the delta thresholds of 436 and 300 for vsyscall and rdtsc, respectively. The distributions of the malicious and factitious deltas obtained with these timing mechanisms can further be found in Figures 4.3 and 4.4. Expectedly, as the noise introduced by measurement mechanism decreased, the delta threshold decreased. Note that to carry out this study, we made the attacks to use the timing mechanism of choice (i.e., syscall, vsyscall, or rdtsc). We did this because our ultimate goal was to reason about the delta thresholds, had the attacks used different timing mechanisms. Therefore, whether the attacks succeeded with the alternative timing mechanisms was irrelevant for us in this regard.

Note that the factitious deltas are just a concept we introduced to demonstrate that the time deltas obtained from the malicious processes tend to come from multiple distributions. In the next section (Section 4.4.3), we use this phenomenon to discuss the factors that need to be taken into account when setting the warning threshold. Consequently, we do not make any claims that factitious deltas should be larger than the malicious deltas or that factitious deltas should be above the delta threshold. Detector^{+} is based rather on the hypothesis that the malicious time deltas tend to be smaller than the time deltas observed in the benign processes. Therefore, the proposed approach would work as long as the malicious time deltas stay below the delta threshold and the time deltas obtained from the benign processes stay above it. From this perspective, factitious deltas, when smaller than the delta threshold, can even help identify the attacks.

From our experiments, we established the minimal delta threshold for attack detection for each of the three time measurement methods. All in all, the delta threshold ranges from 300, 436 to 4635 clockcycles from rdtsc, vsyscall to syscall based time measurement. In practical terms, a system equipped with delta threshold of 4635 will detect any timing attack whether the attacker relies on syscall, vsyscall or rdtsc based time measurement. An attacker might try to circumvent lower delta thresholds of 300 and 435 through executing fixed number of NOPs during malicious measurement in order to stay above these thresholds of 300 and 436. However, using delta threshold of 4635 would counter any such tactic. Since our experimentation affirms that iff attacker tries to evade aforementioned threshold value, the intrinsic system noise would be enough to hamper the effectiveness of such malicious time measurement (See Section 4.6). To date, we did not see any practical evidence of such a real world attacker adapting to our approach and trying to evade *Detector*$^{+}$ in adaptive manner. However, such future possibility remains and for such case an adaptive delta threshold based strategy will be needed.

4.4.3 Study 2: Can the attackers be pinpointed?

After determining the delta thresholds, we focused on our second research question: Can suspicious time deltas be used to pinpoint the malicious processes?

Note that the sooner the malicious processes are determined, the better it is in terms of carrying out the appropriate countermeasures on time to prevent the ongoing attacks or to reduce their harmful consequences. To this end, we aimed to reduce the values of our hyper-parameters, i.e., *window_size*, *warning_threshold*, and *alarm_threshold*, as much as possible (Section 4.3).

We observed that, with $window\_size = 4$, $warning\_threshold = 0.5$, and $alarm\_threshold = 1$, which will be referred to as the *default configuration* in the remainder of the paper, the first warning as well as the first alarm for each malicious process was emitted after the process made a total of 5 time readings (i.e., 4 time deltas). That is, all the malicious processes were correctly pinpointed after the first window of time deltas. And, this was done without emitting neither a false warning nor a false alarm for any of the benign processes.

A related configuration, which produced slightly earlier warnings for the malicious processes at the expense of two false warnings, but no false alarms for the benign processes, was $window\_size = 2$, $warning\_threshold = 0.5$, and $alarm\_threshold = 2$. With these hyper-parameters, for each malicious process, the first warning was emitted after the process made 3 time readings (i.e., after the first window of 2 time deltas) and the first alarm was emitted after a total of 5 time readings (i.e., after the second window of 2 time deltas).

We have also observed that some care must be taken when setting the warning threshold above 0.5. This is because (as discussed in Section 4.4.2) the time deltas obtained from malicious processes tend to come from at least two different distributions, one of which represents the malicious measurements. Therefore, setting the warning threshold above 0.5 may prevent the detection of the malicious processes.

On the other hand, setting the warning threshold lower than 0.5 may increase the number of false warnings and false alarms. This issue can, however, be addressed by increasing the window size and/or the alarm threshold. For example, using a warning threshold of lower than 0.01 (i.e., less than one suspicious delta per 100 deltas) still perfectly pinpoints all the malicious processes in our experiments without having any false warnings or alarms, when the window size is set above 100. Or, setting the warning threshold to 0.01 with a window size of 100 requires an alarm threshold of at least 2 to avoid any false alarms.

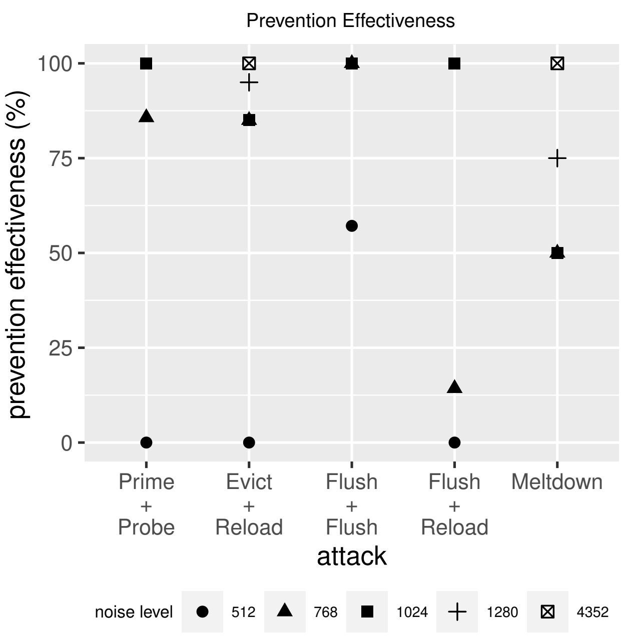

Figure 4.5: Prevention effectiveness obtained under different noise levels.

4.4.4 Study 3: Can the attacks be prevented?

Our next question was then whether the attacks can be prevented. To this end, in the presence of a suspicious time delta, we have introduced noise in the respective measurement by executing a random number of NOP instructions (each of which takes one clock cycle to execute), hampering the usability of the measurements made by the attacker (Section 4.3).

In particular, we experimented with the following noise levels: 512, 768, 1024, 1280, and 4352. For a given noise level n, we determined the actual noise to be introduced, i.e., the number of NOPs to be executed, by randomly picking a number between $[256, n]$. Note that the lower bound in the aforementioned range ensures that at least a minimum number of NOPs are guaranteed to be executed.

For each attack, starting from the lowest noise level, we ran the attack 20 times and computed the *prevention effectiveness* obtained by the selected noise level as the percentage of the experiments, in which the attack failed to operate. To this end, we used the oracles that came with the source code distributions of the attacks (Section 4.1). More specifically, each attack had a mechanism, indicating if the attack was successfully carried out. We kept on increasing the noise level until we had a perfect prevention effectiveness, i.e., until all the attacks were prevented under the given noise level.

Figure 4.5 presents the results we obtained. We first observed that the proposed approach prevented all the attacks with perfect effectiveness (i.e., with 100% prevention effectiveness). Although the noise level required varied from one attack to another, using a sufficiently large noise level (in our case, 4352) prevented all the attacks.

Note that an important feature of Detector$^+$ is that since the noise is introduced for each suspicious delta, Detector$^+$ does not wait until a warning or an alarm is emitted for a process before acting on it. That is, attacks can still be prevented even if no warnings or no alarms are raised.

4.4.5 Study 4: Can the runtime overhead be kept at an acceptable level?

After observing that Detector$^+$ detected, isolated, and prevented all the attacks, we evaluated its runtime overhead. Since the security-related approaches, such as Detector$^+$, target the fielded instances of software systems, excessive runtime overheads are generally not acceptable.

To carry out the study, we have opted to measure the runtime overhead of the proposed approach by enabling vdso and forwarding all the timing calls to vsyscalls, which also allowed us to mimic the scenarios where rdtsc is wrapped with a fast mechanism (Section 4.3.2). This, therefore, enabled us to compute an upper bound on the runtime overhead of the proposed approach since the vsyscalls introduce significantly less overhead compared to the syscalls (Section 4.4.2), making the overhead introduced by Detector$^+$ more apparent.

We have then executed the Phoronix benchmarks between 9 to 15 times (depending on the amount of time required for each benchmark) on both the original operating system and the operating system instrumented with Detector$^+$. This took us to run a total of 259 sub-benchmarks involving 144 applications for about 5 months nonstop.

Each benchmark was designed to report its own performance measurements. We used these measurements to compute the runtime overheads as follows: $((P' - P)/P) * 100$, where P and P' are the performance measurements obtained from the original and the instrumented system, respectively. The lower the overhead, the better the proposed approach is.

Table 4.1 presents the overheads we obtained. We observed that the overall runtime overhead of the proposed approach was 1.56%, on average. Indeed, for 52% (137 out of 259) of the sub-benchmarks, Detector$^+$ introduced virtually no overheads. And, for most of the remaining sub-benchmarks the overhead was close to the average overhead of 1.56% (Table 4.1). One exception was the network benchmarks (benchmark ID=12), where we observed an average overhead of 6.75%. Interestingly enough, no suspicious deltas were detected for this benchmark. That is, no action, except for comparing the observed time deltas with the threshold value, was needed. Furthermore, in terms of the duration of this benchmark as well as the total number of time deltas observed in it, we could not identified any singularities either. That is, there were other benchmarks with similar durations and similar number of time deltas, but resulting in virtually no overheads. An in-depth analysis then revealed that the performance of this benchmark is greatly affected by the network traffic present in the underlying platforms. We indeed observed that even the performance measurements obtained from the different repetitions of the same sub-benchmarks running on the same platform varied between 2.2% and 5.5%, possibly explaining the singularity in the overheads.

4.4.6 Study 5: Can the attack variations be detected?

We have then evaluated Detector$^+$ under different attack variations. Each attack variation mimicked a mechanism that can be used by an attacker to stay stealthier.

Carrying out an attack by using multiple processes. We first focused on the variations, in which the same attack is carried out concurrently by multiple processes. Note that this can potentially reduce the suspiciousness of individual processes by distributing the malicious activities over multiple processes, each of which can, for example, target a different part of the secret information.

To carry out the study, we executed each attack by using $p = 2, 5,$ and 10 concurrent processes and monitored the time deltas using the default configuration of Detector$^+$ (i.e., $window\_size = 4$, $warning\_threshold = 0.5$, and $alarm\_threshold = 1$). We observed that, except for 2 processes, all the malicious processes were pinpointed after the first window; the first warning and the first alarm for these processes were emitted after they had made 5 time readings (i.e., after 4 time deltas). And, the two aforementioned malicious processes were pinpointed after the second window; the first alarm as well as the first warning for these processes were emitted after they had made 9 time readings. One of these processes was a Flush+Flush process when $p = 2$ and the other was a Prime+Probe process when $p = 5$. It turned out this happened because the first few malicious time deltas obtained from these processes were unexpectedly high. We believe that this happened due to the noise introduced by spawning multiple processes at around the same times. Note that such noise also makes it difficult (if not impossible) for the attacker to extract information from the measurements (see Section 4.6 for more information).

Carrying out different attacks concurrently. Last but not least, we have evaluated Detector$^+$ on attack scenarios, in which different types of attacks were carried out concurrently. Note that this type of variation is different than the previous type of variation because in the former a single type of attack is carried out at a time by using multiple concurrent processes.

To carry out the study, we executed all of the 5 attacks concurrently and used the default Detector$^+$ configuration (i.e., $window\_size = 4$, $warning\_threshold = 0.5$, and *alarm_threshold* = 1) for analysis. Furthermore, we have repeated the experiments 6 times. All of the malicious processes used in these experiments were pinpointed after the first window, i.e., after they have made 5 time readings (i.e., 4 time deltas).

4.5 Threats to validity

One external threat concerns the representativeness of the timing attacks used in the study, namely Meltdown [LSG$^+$18], Evict+Reload [LGS$^+$16], Flush+Flush [GMWM16], Flush+Reload [YF14], and Prime+Probe [LYG$^+$15]. However, all of these attacks are well-known attacks and they have all been used in related works [LSG$^+$18, GSM15, YSG$^+$19, Yar16, WNQ$^+$19, AGYS20]. Furthermore, as the base attacks, we have used the publicly available implementations of these attacks together with the facilities provided by these implementations to determine whether the attacks were successful or not. We have also evaluated Detector$^+$ on different variations of these attacks by carrying out the attacks using multiple processes and by carrying out multiple different types of attacks concurrently, mimicking some of the mechanisms that attackers may employ to stay stealthier.

Another threat concerns the representativeness of the benign processes used in the study. In the experiments, we have used the Phoronix benchmarks as the suite of benign processes [pho19]. Phoronix, indeed, provides a variety of software systems, which are commonly encountered in production environments, including network, database, and machine learning systems; cryptographic, scientific computing, audio/video processing, and graphics applications; and a wide spectrum of IO-/CPU-bounded applications for workstations and servers(Table 4.1). Phoronix has also been used in related works [SB13, DSM$^+$08, Lou19, AGYS20].

4.6 Countermeasures against Detector$^+$

Detector$^+$, as is the case with other detection frameworks, can (and will) naturally become the subject of attack techniques that aim to find innovative ways of avoiding detection by Detector$^+$. In this section, we discuss some of the potential countermeasures that can be taken against Detector$^+$ as well as possible mitigation strategies that Detector$^+$ can employ against them.

Detector$^+$ operates by monitoring and analyzing how processes perform time measurements and their patterns thereof. An attacker may, therefore, attempt to surreptitiously measure the time and/or temper with the measurements made by Detector$^+$. Except rdtsc, this, however, requires elevated privileges (which is against the premises of the timing attacks) as the aforementioned time measurement mechanisms use either system calls or virtual system calls.

Regarding rdtsc, as the results of the studies we carried out in this paper strongly suggest that monitoring and analyzing timing behaviors can be used as a countermeasure against timing attacks, we are a strong advocate of allowing accesses to rdtsc (and, for the same reason, to all timing related services as well as to hardware performance counters/events [Gui11, MBDH99]) only through privileged software/hardware entities, so that suspicious time measurements can be monitored and appropriate countermeasures can be taken. Further discussion on this can be found in Section 4.3.2.

An attacker may also increase the time gap between the consecutive pairs of time readings in an attempt to stay stealthy. If the resulting time deltas stay above the delta threshold, then they will not be marked as suspicious. One way this could be done is to measure the execution time of a series of operations, rather than a single operation. For example, in the cache-based timing attacks, rather than measuring the time it takes to access a single memory location to determine whether the respected data is in the cache, one could measure the time required for accessing multiple memory locations. This, however, makes the attacks more complicated to implement as more involved analyses are required to factor out the ambiguity in the measurements.

Another way could be to intentionally introduce deterministic amount of noise into the measurements, which can then be factored out by the attacker after the measurements. For example, during a time measurement, in addition to accessing a memory location of interest, the attacker can carry out a fixed number of NOP instructions to stay above the delta threshold. However, the more *intentional* noise introduced by the attacker, the more *unintentional* noise is generated by the underlying platform (e.g., operating system), which makes the analysis complicated (if not impossible at all). This is because there is no guarantee as to how these benign instructions will be executed. For example, during the execution of these instructions, thus in the middle of a time measurement, the operating system may take the control of the CPU from the malicious process and give it to another process, which would inadvertently introduce a great deal of system noise.

To evaluate this conjecture, we carried out a study where we introduced a fixed number of NOPs into the time measurements made by the Meltdown attack. The noise was then subtracted from the actual time readings before they are analyzed by Meltdown. We experimented with different levels of intentional noise. For each noise level (i.e., the number of NOPs introduced into each time measurement), we used the reliability oracle that comes with the source code distribution of Meltdown to measure the accuracy of the attack. The accuracy in this context is simply computed by the percentage of the attempts, in which the target memory location is read successfully. Thus, the higher

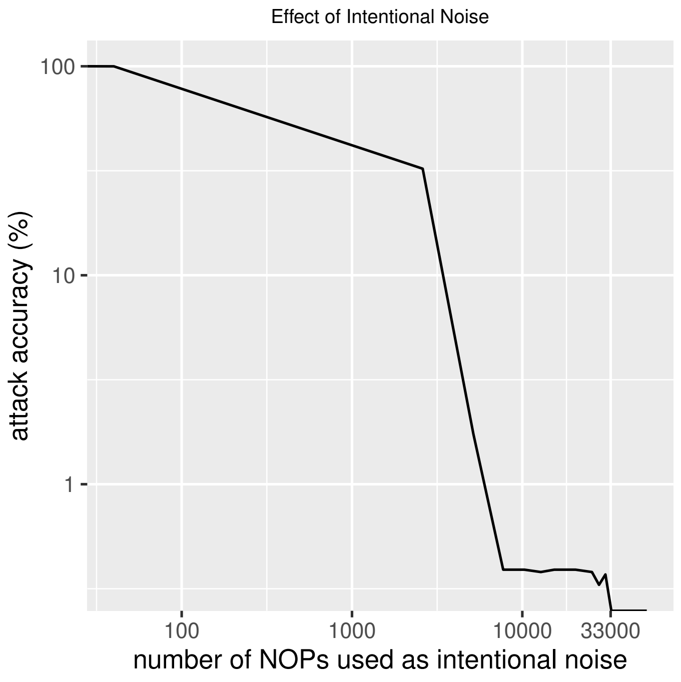

Figure 4.6: The effect of the intentional noise introduced into the time measurements on the accuracy of the Meltdown attack.

the accuracy, the better the attack is.

Figure 4.6 presents the results we obtained. As the level of intentional noise is increased, which the attacker needs in order to stay above the delta threshold, the accuracy of the attack is diminished and eventually reached 0% where not even a single byte of information was exfiltrated, supporting our hypothesis.

Note that to account for this phenomenon the attacker would need more measurements (which increases the amount of time required by the attack) and/or more sophisticated analyses, thus reducing the chance of success. Nevertheless, the hyperparameters offered by Detector$^+$ can be used as a defense mechanism for these countermeasures. For example, one can increase the delta threshold, which makes the longer time deltas also look suspicious. This, however, may reduce the accuracy in isolating the malicious processes. To cope with this, the window size as well as the warning and alarm thresholds can be increased. For example, in the presence of intentionally introduced noise in the Meltdown attack (Figure 4.6), we increased the delta threshold to 7555, such that the accuracy of the attack is kept under 1%. With this new threshold, we observed that having window size $= 30000$, warning threshold$= 0.5$, and alarm threshold $= 50$, pinpointed all the malicious processes without emitting any false alarms for the benign processes. Note that although increasing the values of these hyper-parameters may increase the detection times of the malicious processes, since Detector$^+$ does not wait until a process is marked as malicious before acting on it (i.e., in the presence of suspicious time deltas, the time measurements are thwarted regardless of whether the respective processes are marked as malicious or not), the chance of success for the attacks will still be reduced (if not prevented at all). After all, a complementary way of reducing false alarms is to explicitly mark the trusted processes (such as, the system processes), so that they can be excluded from the analysis, which, in turn, can also help reduce the values of the hyper-parameters.

In summary, Detector$^+$ utilizes a strong feature set, that proves to be highly generic and extremely effective against detecting and eliminating a wide range of cache-based timing attacks by itself and/or in conjunction with other detection techniques.

4.7 HyperDetector

We extended *Detector*$^+$ approach to operate at the level of hypervisors and thus, named *HyperDetector* [UJYSng]. This itself is a generic approach, to detect, isolate and prevent ongoing timing attacks, albeit at the level of hypervisor, providing protection to tenant virtual machines. HyperDetector, uses a hardware extension for virtualization and intercepts rdtsc instructions, such that consecutive pairs of time readings which are suspiciously close to each other can be detected. Upon detection of potentially malicious time measurements, noise is introduced in to the time measurements to prevent ongoing attacks. Furthermore, the sequence of time measurements are analyzed at runtime through a sliding window based approach to determine the malicious process. In our experiments, HyperDetector performed remarkably well and detected all malicious processes and snubbed the success rate of attacks from 99% to $\approx$ 0%. Meanwhile the exhibited runtime overhead was 1.14%.

4.8 Related work

Information exfiltration from cryptographic algorithms employing timing-channels had been extensively studied [Ber05, OST06, Per05, AS08] in the past. These attacks often leverage a timing-channel being established as a side effect to cache-contentions. Typically, the attacker constantly monitors a set of cache lines to determine if they have been accessed by another process; either through continual eviction of a cache line or continuously filling a small portion of cache accessible to attacker. In both of the mentioned schemes, statistically notable change in the access latency enables discovery of accessed cache lines by a victim process.

Timing attacks have been demonstrated to circumvent sandboxes and even kernel space hardening (such as ASLR) [HWH13, ZJRR12, WXW14, OKSK15, MIE17]. One mitigation strategy extensively highlighted in the literature is to ensure that critical parts of the systems, which process secret information, should remain agnostic of observable sequences of cache accesses [OST06]. This, for example, can prevent the influence of plain and cipher text parameters in cryptographic applications from being deterministically observed. Bernstein [Ber05] suggested that, one way of mitigation must involve constant time implementation of the cipher. However, this remain a very difficult problem [Ber05].

Cock et al. [CGMH14] claim that although storage based side-channels can be discovered and prevented through formal analysis, as is the case with seL4 [KEH$^+$09] micro-kernel, nonetheless such an approach does not extend to timing-channels. Moreover, they [CGMH14] advocate, although in-theory a constant-time implementation approach can guarantee absence of a timing-channel; in reality, it remains prone to the CPU architecture type, as was the case with an OpenSSL vulnerability which had been rectified on x86 by such a fix, while remained ineffective on ARM. Furthermore, Coppens et al. [CVDBDS09] uncovered that the timing behavior of cryptographic software is directly dependent upon its utilization of variable-latency instructions in the code. To remediate, a compiler based solution is presented, which removes control- and data-flow dependence on the secret key. Although the presented approach had been effective for addressing the main problem nonetheless, it incurred significant performance overhead. Along the similar lines, Rodrigues et al. [RQPA16] developed a more effective compile-time tool, discovering timing-channel vulnerabilities manifested in the implicit control flow of the actual implementation of cryptographic code. Their presented approach successfully revealed a known vulnerability in OpenSSL (v1.0.1e), use casing the effectiveness of their tool. Approaches for developing security-aware and attack-resistant cache architectures have also been studied in literature [LL14, Pag05, DFS20, DJL$^+$12, WL07, LWML16, HWS$^+$19].

Mitigating timing-channels for adversaries by coarsification of high resolution clock sources has also been explored in the literature. Hu et al. [Hu92] present timestamp fuzzing technique for VAX systems in classical literature. They present that reducing the resolution of high resolution clock sources reduces the bandwidth of a timing-channel. To this end, they present a set of techniques which involve adding random amount of noise to all system wide clock sources, essentially limiting the resolution of timestamps. However, since such a technique would introduce noise to all time measurements, it would also deny the legitimate applications of high-resolution measurements. This would also inadvertently penalize the system throughput, as co-ordination of the hardware level operations depend upon high resolution timers. Martin et al. [MDS12] present a similar technique, that selectively limits the resolution of the rdtsc instructions. Our work is different from this work that we are concerned with not only the hardware timekeeping instructions (such as, rdtsc), but also the software timekeeping mechanisms (such as syscalls and vsyscalls). Furthermore, the aforementioned work requires some hardware modifications (as it proposed a new machine instruction) and does not automatically determine the malicious processes. It simply changes the granularity of the rdtsc instructions for some pre-determined (i.e., given) processes. Our work, however, is implemented at the software level and pinpoints the malicious processes, so that they cannot further harm the system by, for example, intentionally making time measurements to hurt the performance of the system.

Language based predictive mitigation for timing-channels remain another topic of research, quantizing bounds over security of information flow. Zhang et al. [ZAM12] propose well-typed programs can only leak a bounded amount of information through a timing-channel. To this end, they propose a mitigation language employing predictive timing to ensure strict bounds on the execution time of programs. In contrast, Li et al. [LKO$^+$14, LTO$^+$11] present a hardware focused approach by developing statically verifiable hardware description language to ensure information-flow security. Porter et al. [PBR$^+$14] present an augmentative solution to retrofit an existing OS, based on decentralized information flow control, which remains promising in providing end-to-end security guarantees. In the aforementioned approach, the security policies need to be specified by labeling data prior to its processing in context-sensitive security methods. Nonetheless, such an approach remains partially effective for the prevention of timing-channels and can result in, considerable performance overhead. These aforementioned preventive approaches may lack generality to cater for wider spectrum of timing-channel attacks, yet strengthen the defensive posture of a system for their respective category of attacks, albeit at the cost of performance. Such approaches, however, may lack prevention against zero-day attacks. We, therefore, believe that for sensitive and security first application scenarios, the aforementioned approaches can complement Detector$^+$ and provide stronger defense mechanisms.

Reactive, dynamic, and attack specific mitigative approaches have also been presented in literature. Kulah et al [KDYS19] present an anomaly based detection approach to prevent Prime+Probe, Flush+Reload, and Flush+Flush attacks. Their presented approach monitors contentions occurring in shared resources and associates them with processes, such that any suspicious level of contentions lead to the issue of a warning and subsequently a preventive action to stop an ongoing attack. Akyildiz et al [AGYS20] present an approach tailored to detect and prevent the Meltdown attack by monitoring segmentation faults occurring at memory addresses close to each other. In the presence of a segmentation fault, the aforementioned approach flushes the cache hierarchy as a reactive measure to prevent possible information leakage. The sequence of segmentation faults are then analyzed to pinpoint the malicious processes.

Nomani et al. [NS15] suggest leveraging information available from hardware performance counters to learn and predict upcoming execution phases of programs. A program scheduler equipped with this information can adaptively schedule programs such that an on-going attack can either be thwarted to a reasonable extent. Zhang et al. [ZR13] present an approach to mitigate Prime+Probe attack by relying on frequent intentional cache-cleanings which in-effect obfuscates usable information in the timing data to render it useless for an attacker. The aforementioned approaches lack a general mechanism to prevent all timing-channel attacks, as they're tailored to the specifics of the certain attacks. In contrast, Detector$^+$ is a generic approach, agnostic to the specifics used in the timing attacks and can detect an attack as long as the attack demonstrates fine-grained time reading behavior.

4.9 Concluding Remarks

In this work, we have presented a novel approach, called Detector$^+$, for detecting, isolating, and preventing timing-based side channel attacks at runtime.

The proposed approach is based on a simple observation that the way, timing attacks perform their time measurements and their patterns thereof differ from those of the benign processes. In particular, timing attacks need to measure the execution times of typically quite short-running operations. Detector$^+$, therefore, monitors time readings made by processes and mark consecutive pairs of readings that are close to each other in time as suspicious. In the presence of a suspicious time measurement, a random amount of noise is introduced in the measurement to prevent the attacker from extracting information by using the measurement. The sequence of suspicious measurements is then analyzed at runtime by using a sliding window-based approach to determine the malicious processes.

We evaluated the proposed approach by conducting a series of experiments. In these experiments, we used five well-known timing attacks together with a well-known suite of benign applications, representing the applications that are commonly encountered in production environments. To further evaluate the proposed approach in the presence of stealthier attacks, we have also tested Detector$^+$ with different variations of the timing attacks. In all the experiments, Detector$^+$ detected all the malicious time measurements with almost a perfect accuracy, prevented all the attacks, and correctly pinpointed all the malicious processes involved in the attacks without any false positives after they have made a few time measurements with an average runtime overhead of 1.56%.

We believe that this line of research is novel and interesting. Therefore, we continue working in this field. In particular, we are interested in understanding the fundamental mechanisms involved in side channel attacks, identifying the commonalities between these mechanisms, and developing low-overhead approaches to detect, isolate, and prevent not only the known attacks, but also the zero-day attacks at runtime.

CHAPTER 5

Future Work and Concluding Remarks

In this dissertation, we aimed to defend computing systems against microarchitectural side-channel attacks. Developing effective countermeasure approaches is although possible. However, the challenge lies in delivering low to negligible performance overhead of these countermeasures as security and performance are often orthogonal to each other. Evidence suggests, new attacks and more potent variants of existing attacks will continue to emerge and thus, broad spectrum countermeasures are desired and will remain in demand.

From a microprocessor's design standpoint, microarchitectural side-channel security had to be fundamentally incorporated into the design process. Existing microarchitecture side-channel attacks exploit the performance and optimization features of existing designs. Furthermore, no matter what, motivated researchers and exploit developers will continue to find new vulnerabilities even when the CPU vendor guarantees new security mechanisms been introduced. Such was the case with Intel's trusted execution environment, where researchers discovered vulnerabilities and show cased them to snoop-on secure computation within enclaves. White-hat research must continue to discover new vulnerabilities both to thoroughly vet the strengths of newer microarchitectures, as this not only establishes confidence into the level of provided security, as well as aides to find newer security vulnerabilities and drive research efforts to address them.

Moreover, regardless of the CPU vendor, or the underlying ISA, or the target platform; researchers revealed that current generation of CPU's are microarchitecturally prone to side-channel attacks in a myriad of ways. This fact is worrisome, although from research standpoint we do hope that future designs would further harden microarchitectural security posture. Nonetheless, current generation of computing systems will continue to operate in the years to come and will remain exploit prone.

We began our research by asking four fundamental questions about the microarchitecture side-channel security research. We performed a systematic mapping study to find answers to aforementioned questions under guidelines of a strict mapping protocol. We searched existing, peer-reviewed literature from three major scientific research databases. We located and reviewed 379 articles at first and appraised 84 articles to serve as primary studies, after rigorous screening process. We developed a study classification scheme to be used for this context. This developed classification scheme not only served our needs, it will be usable even in the future to carry out similar research efforts. Among notable observations from our mapping study, we observed that past five years show a linear growth in the number of publications, suggesting the importance and attention of research being carried out in this arena of computer and cybersecurity.

From the gist of our systematic mapping study, existing research demonstrates that artefacts leading up to compromised security and privacy are not limited solely to one specific aspect of microarchitecture. They span across executional mechanics of speculation, prefetching and out-of-order execution on one hand; on the other, they span across physical components such as cache, memory, registers, interconnects and so on. We also observe that, aforementioned artefacts span across contention, timing and access behaviors. Last but not least, interestingly the onboard electronics contributes to artefact emanations in the power consumption and electromagnetic emissions. Furthermore, regardless physical, or virtualized/emulated hardware, software based microarchitecture side-channel attacks remain to exist, cause harm and are becoming more and more lucrative to computer system exploitations.

To our end, we further undertook the task of constructing and demonstrating an effective, and efficient countermeasure approach we named *Detector*$^+$ targeting microarchitectural timing attacks. *Detector*$^+$ detected, isolated and prevented ongoing timing attacks at runtime with negligible overhead. On its core, we leveraged the following observation that timing attacks perform very fine-grained time measurements and reveal patterns quite different than benign processes. We monitor all processes system wide and observe their time reading behavior, and any consecutive pair of readings that are close to each other are marked suspicious. At first we inject random amount of noise to thwart attacker from extracting useful information from said time measurement. Then, sequence of suspicious time measurements are analyzed at runtime and culprit processes carrying out malicious activities are pinpointed, subsequently a mitigative action is taken and the computer system is thus safeguarded.

An extension termed *HyperDetector* to aforementioned work was later carried out, where we extended the underlying approach to defend operating systems running in virtualized environments such as cloud. For this purpose, we implemented the

the core logic of *Detector*$^+$ into a kvm hypervisor and we monitored the time reading behavior of all processes running inside the guest operating systems. We also obtained promising results that showed negligible performance overhead and successfully kept all the tenant virtual machines defended against timing attacks.

The potential avenues of research to defend against microarchitecture side-channel attacks could focus on countermeasures delivering negligible overheads. Hybrid solutions in the form of pluggable hardware security modules and accompanying software would be a step in this direction, such as FPGAs based solutions. FPGAs being hardware programmable could be employed to synthesize soft-core processors and/or any subsystem. Synthesized hardware would deliver performance where a software based solution is prohibitive. Such a design is flexible in nature and whose synthesized hardware could be reconfigured via a software update. Another research avenue could be, automated auditing and patching of exploitable binaries to microarchitecture side-channels at runtime. In a similar regard, approaches to formally reason and fix, side-channel security of programs during speculative execution are needed. Adaptive speculative execution strategies ensuring secure execution of vulnerable programs can be researched.

Bibliography

[ABG] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New Results on Instruction Cache Attacks. volume 6225, pages 110–124.

[ADN$^+$10] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When clocks fail: On critical paths and clock faults. In *International conference on smart card research and advanced applications*, pages 182–193. Springer, 2010.

[AGS] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. volume 4887, pages 185–203.

[AGYS20] Taha Atahan Akyildiz, Can Berk Guzgeren, Cemal Yilmaz, and Erkay Savas. Meltdowndetector: A runtime approach for detecting meltdown attacks. *Future Generation Computer Systems*, 112:136–147, 2020.

[AKFGBR] Muhammad Arsath K F, Vinod Ganesan, Rahul Bodduna, and Chester Rebeiro. PARAM: A Microprocessor Hardened for Power Side-Channel Attack Resistance. pages 23–34.

[AS] Onur Acıiçmez and Werner Schindler. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL. volume 4964, pages 256–273.

[AS08] Onur Acıiçmez and Werner Schindler. A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl. In *Cryptographers' Track at the RSA Conference*, pages 256–273. Springer, 2008.

[BBF$^+$] Alessandro Barenghi, Matteo Brevi, William Fornaciari, Gerardo Pelosi, and Davide Zoni. Integrating Side Channel Security in the FPGA Hardware Design Flow. volume 12244, pages 275–290.

[BCHC] Nicolas Belleville, Damien Couroussé, Karine Heydemann, and Henri-Pierre Charles. Automated Software Protection for the Masses Against Side-Channel Attacks. 15(4):1–27.

[Ber05] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[BHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload–a cache attack on the bliss lattice-based signature scheme. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 323–345. Springer, 2016.

[BJV] Roderick Bloem, Swen Jacobs, and Yakir Vizel. Efficient Information-Flow Verification Under Speculative Execution. volume 11781, pages 499–514.

[BNB$^+$] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Muhlberg, and Frank Piessens. Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors. pages 262–276.

[BP] Alessandro Barenghi and Gerardo Pelosi. Side-channel security of superscalar CPUs : Evaluating the Impact of Micro-architectural Features. pages 1–6.

[BSMG] Florian Bache, Tobias Schneider, Amir Moradi, and Tim Giineysu. SPARX — A side-channel protected processor for ARX-based cryptography. pages 990–995.

[BVIB] Ali Galip Bayrak, Nikola Velickovic, Paolo Ienne, and Wayne Burleson. An architecture-independent instruction shuffler to protect against side-channel attacks. 8(4):1–19.

[BWM17] Johann Betz, Dirk Westhoff, and Günter Müller. Survey on covert channels in virtual machines and cloud computing. *Transactions on Emerging Telecommunications Technologies*, 28(6):e3134, 2017.

[CBRY20] Peng Cheng, Ibrahim Ethem Bagci, Utz Roedig, and Jeff Yan. Sonarsnoop: Active acoustic side-channel attacks. *International Journal of Information Security*, 19(2):213–228, 2020.

[CCFV] Gianpiero Cabodi, Paolo Camurati, Fabrizio Finocchiaro, and Danilo Vendraminetto. Model Checking Speculation-Dependent Security Properties: Abstracting and Reducing Processor Models for Sound and Complete Verification. volume 11445, pages 462–479.

[CGMH14] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on sel4. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 570–581, 2014.

[CLH] Valence Cristiani, Maxime Lecomte, and Thomas Hiscock. A Bit-Level Approach to Side Channel Based Disassembling. volume 11833, pages 143–158.

[CLZZ] Guoxing Chen, Mengyuan Li, Fengwei Zhang, and Yinqian Zhang. Defeating Speculative-Execution Attacks on SGX with HyperRace. pages 1–8.

[con] Connected Papers | Find and explore academic papers.

[cr009] Time-stamp counter disabling oddities in the linux kernel. `https://blog.cr0.org/2009/05/time-stamp-counter-disabling-oddities.html`, 2009. Accessed: (2021-01-16).

[CSY16] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.

[CVBS$^+$19] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, 2019.

[CVDBDS09] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE Symposium on Security and Privacy*, pages 45–60. IEEE, 2009.

[CW] Robert J. Colvin and Kirsten Winter. An Abstract Semantics of Speculative Execution for Reasoning About Security Vulnerabilities. volume 12233, pages 323–341.

[CY] Patrick Cronin and Chengmo Yang. A Fetching Tale: Covert Communication with the Hardware Prefetcher. pages 101–110.

[CZP] Robert Callan, Alenka Zajic, and Milos Prvulovic. A Practical Methodology for Measuring the Side-Channel Signal Available to the Attacker for Instruction-Level Events. pages 242–254.

[DFS20] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. Hybcache: Hybrid side-channel-resilient caches for trusted execution environments. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 451–468, 2020.

[DJL$^+$12] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–21, 2012.

[DM] Guillaume Didier and Clémentine Maurice. Calibration Done Right: Noiseless Flush+Flush Attacks. volume 12756, pages 278–298.

[DNAG$^+$] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Leaky Buddies: Cross-Component Covert Channels on Integrated CPU-GPU Systems. pages 972–984.

[DSM$^+$08] Todd Deshane, Zachary Shepherd, Jeanna Matthews, Muli Ben-Yehuda, Amit Shah, and Balaji Rao. Quantitative comparison of xen and kvm. *Xen Summit, Boston, MA, USA*, pages 1–2, 2008.

[DXS] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. Cache timing side-channel vulnerability checking with computation tree logic. pages 1–8.

[EPS10] Antti Evesti and Susanna Pantsar-Syväniemi. Towards micro architecture for security adaptation. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 181–188, 2010.

[ERAGP] Dmitry Evtyushkin, Ryan Riley, Nael CSE and ECE Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. pages 693–707.

[EWS] Hassan Eldib, Chao Wang, and Patrick Schaumont. Formal Verification of Software Countermeasures against Side-Channel Attacks. 24(2):1–24.

[FDY$^+$] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Cache-Zoomer: On-demand High-resolution Cache Monitoring for Security. 4(3):180–195.

[FL] Adi Fuchs and Ruby B. Lee. Disruptive prefetching: Impact on side-channel attacks and cache designs. pages 1–12.

[GCL$^+$] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. SpecuSym: Speculative symbolic execution for cache timing leak detection. pages 1235–1247.

[GES17] Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. Cache-based application detection in the cloud using machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 288–300, 2017.

[GII$^+$] Berk Gulmezoglu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross-VM Cache Attacks on AES. 2(3):211–222.

[GKDG] Navyata Gattu, Mohammad Nasim Imtiaz Khan, Asmit De, and Swaroop Ghosh. Power side channel attack analysis and detection. pages 1–7.

[GLV$^+$] Gunnar Grimsdal, Patrik Lundgren, Christian Vestlund, Felipe Boeira, and Mikael Asplund. Can Microkernels Mitigate Microarchitectural Attacks? volume 11875, pages 238–253.

[GMWM] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. volume 9721, pages 279–299.

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[GOPT] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications. volume 5984, pages 176–192.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, 2015.

[Gui11] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2(11), 2011.

[GVBPS] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution. volume 10953, pages 44–60.

[GYCH18] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018.

[GYCH19] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: the missing os abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[HK] Youngkwang Han and John Kim. A Novel Covert Channel Attack Using Memory Encryption Engine Cache. pages 1–6.

[HL] Zecheng He and Ruby B. Lee. How secure is your cache against side-channel attacks? pages 341–353.

[HLRP] Carl-Daniel Hailfinger, Kerstin Lemke-Rust, and Christof Paar. CCCiCC: A Cross-Core Cache-Independent Covert Channel on AMD Family 15h CPUs. volume 11833, pages 159–175.

[HMG$^+$] Min He, Cunqing Ma, Jingquan Ge, Neng Gao, and Chenyang Tu. Flush-Detector: More Secure API Resistant to Flush-Based Spectre Attacks on ARM Cortex-A9. pages 1–6.

[HS13] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 219–235. Springer, 2013.

[HSC] Shun-Wen Hsiao, Yeali S. Sun, and Meng Chang Chen. Hardware-Assisted MMU Redirection for In-Guest Monitoring and API Profiling. 15:2402–2416.

[Hu92] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992.

[HWH13] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[HWS$^+$19] Austin Harris, Shijia Wei, Prateek Sahu, Pranav Kumar, Todd Austin, and Mohit Tiwari. Cyclone: Detecting contention-based cache information leaks through cyclic interference. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 57–72, 2019.

[ICLY] Md Hafizul Islam Chowdhuryy, Hang Liu, and Fan Yao. BranchSpec: Information Leakage Attacks Exploiting Speculative Branch Instruction Executions. pages 529–536.

[JLD$^+$17] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 377–390, 2017.

[JYS21] Arsalan Javeed, Cemal Yilmaz, and Erkay Savas. Detector+: An approach for detecting, isolating, and preventing timing attacks. *Computers Security*, 110:102454, 2021.

[K$^+$07] Staffs Keele et al. Guidelines for performing systematic literature reviews in software engineering. Technical report, Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007.

[KDYS19] Yusuf Kulah, Berkay Dincer, Cemal Yilmaz, and Erkay Savas. Spydetector: An approach for detecting side-channel attacks at runtime. *International Journal of Information Security*, 18(4):393–422, 2019.

[KEH$^+$09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[KGT18] Jonas Krautter, Dennis RE Gnad, and Mehdi B Tahoori. Fpgahammer: Remote voltage fault attacks on shared fpgas, suitable for dfa on aes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 44–68, 2018.

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

[KKOA12] Arun K Kanuparthi, Ramesh Karri, Gaston Ormazabal, and Sateesh K Addepalli. A survey of microarchitecture support for embedded processor security. In *2012 IEEE Computer Society Annual Symposium on VLSI*, pages 368–373. IEEE, 2012.

[KLA$^+$] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. pages 974–987.

[KLS$^+$] Raghavan Kumar, Xiaosen Liu, Vikram Suresh, Harish K. Krishnamurthy, Sudhir Satpathy, Mark A. Anders, Himanshu Kaul, Krishnan Ravichandran, Vivek De, and Sanu K. Mathew. A Time-/Frequency-Domain Side-Channel Attack Resistant AES-128 and RSA-4K Crypto-Processor in 14-nm CMOS. 56(4):1141–1151.

[Koc96] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[KSa] Pantea Kiaei and Patrick Schaumont. Synthesis of Parallel Synchronous Software. 13(1):17–20.

[KSb] Taehyun Kim and Youngjoo Shin. Reinforcing Meltdown Attack by Using a Return Stack Buffer. 7:186065–186077.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[LCCL] Jen-Wei Lee, Szu-Chi Chung, Hsie-Chia Chang, and Chen-Yi Lee. Efficient Power-Analysis-Resistant Dual-Field Elliptic Curve Cryptographic Processor Using Heterogeneous Dual-Processing-Element Architecture. 22(1):49–61.

[LF] Jens Lindemann and Mathias Fischer. On the detection of applications in co-resident virtual machines via a memory deduplication side-channel. 18(4):31–46.

[LGS+16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016.

[LKO+14] Xun Li, Vineeth Kashyap, Jason K Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T Chong. Sapper: A language for hardware-level security policy enforcement. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 97–112, 2014.

[LL14] Fangfei Liu and Ruby B Lee. Random fill cache architecture. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 203–215. IEEE, 2014.

[LLW+] Fan Lang, Huorong Li, Wei Wang, Jingqiang Lin, Fengwei Zhang, Wuqiong Pan, and Qiongxiao Wang. E-SGX: Effective Cache Side-Channel Protection for Intel SGX on Untrusted OS. volume 12612, pages 221–243.

[LLWR] Bozhi Liu, Roman Lysecky, and Janet Meiling Wang-Roveda. Composable Template Attacks Using Templates for Individual Architectural Components. pages 1–8.

[Lou19] Michail Loukeris. Efficient computing in a safe environment. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1208–1210, 2019.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[LTO$^+$11] Xun Li, Mohit Tiwari, Jason K Oberg, Vineeth Kashyap, Frederic T Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: a hardware description language for secure information flow. *ACM Sigplan Notices*, 46(6):109–120, 2011.

[LWL] Fangfei Liu, Hao Wu, and Ruby B. Lee. Can randomized mapping secure instruction caches from side-channel attacks? pages 1–8.

[LWML16] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, 2016.

[LYG$^+$15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[LZJZ21] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Computing Surveys (CSUR)*, 54(6):1–37, 2021.

[MAA$^+$16] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 1–9. 2016.

[MAB$^+$18] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Rao Naveed Bin Rais, Vianney Lapotre, and Guy Gogniat. Run-time detection of prime+ probe side-channel attack on aes encryption algorithm. In *2018 Global Information Infrastructure and Networking Symposium (GIIS)*, pages 1–5. IEEE, 2018.

[MBDH99] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710. Citeseer, 1999.

[MDS12] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 118–129. IEEE, 2012.

[MGK$^+$] Fernando Mosquera, Nagendra Gulur, Krishna Kavi, Gayatri Mehta, and Hua Sun. CHASM: Security Evaluation of Cache Mapping Schemes. volume 12471, pages 245–261.

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017.

[MMN] Yuxiao Mao, Vincent Migliore, and Vincent Nicomette. REHAD: Using Low-Frequency Reconfigurable Hardware for Cache Side-Channel Attacks Detection. pages 704–709.

[MNHF] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. volume 9148, pages 46–64.

[MTE$^+$21] Per Håkon Meland, Shukun Tokas, Gencer Erdogan, Karin Bernsmed, and Aida Omerovic. A systematic mapping study on cyber security indicator data. *Electronics*, 10(9):1092, 2021.

[MTS] Suvarna Mane, Mostafa Taha, and Patrick Schaumont. Efficient and side-channel-secure block cipher implementation with custom instructions on FPGA. pages 20–25.

[MWES] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations. 47(4):538–570.

[NAC$^+$] Mohammed Nabeel, Mohammed Ashraf, Eduardo Chielle, Nektarios G. Tsoutsos, and Michail Maniatakos. CoPHEE: Co-processor for Partially Homomorphic Encrypted Execution. pages 131–140.

[NBL$^+$] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. Validation of Abstract Side-Channel Models for Computer Architectures. volume 12224, pages 225–248.

[NMB$^+$16] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451. IEEE, 2016.

[NNQAG] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered Insecure: GPU Side Channel Attacks are Practical. pages 2139–2153.

[NS15] Junaid Nomani and Jakub Szefer. Predicting program phases and defending against side-channel attacks using hardware performance counters. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–4, 2015.

[OAP$^+$] Hyunyoung Oh, Adil Ahmad, Seonghyun Park, Byoungyoung Lee, and Yunheung Paek. TRUSTORE: Side-Channel Resistant Storage for SGX using Intel Hybrid CPU-FPGA. pages 1903–1918.

[ODK20] Hamza Omar, Brandon D'Agostino, and Omer Khan. Optimus: A security-centric dynamic hardware partitioning scheme for processors that prevent microarchitecture state attacks. *IEEE Transactions on Computers*, 69(11):1558–1570, 2020.

[OKSK15] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418, 2015.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[Oya19] Yoshihiro Oyama. How does malware use rdtsc? a study on operations executed by malware with cpu cycle measurement. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 197–218. Springer, 2019.

[Pag05] D Page. Partitioned cache architecture as a ėide-channel defence mechanism. 2005.

[Pan] Biswabandan Panda. Fooling the Sense of Cross-Core Last-Level Cache Eviction Based Attacker by Prefetching Common Sense. pages 138–150.

[Pay] Mathias Payer. HexPADS: A Platform to Detect "Stealth" Attacks. volume 9639, pages 138–154.

[PBR$^+$14] Donald E Porter, Michael D Bond, Indrajit Roy, Kathryn S McKinley, and Emmett Witchel. Practical fine-grained information flow control using laminar. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(1):1–51, 2014.

[Per05] Colin Percival. Cache missing for fun and profit, 2005.

[PFB$^+$] Thinh Hung Pham, Alexander Fell, Arnab Kumar Biswas, Siew-Kei Lam, and Nandeesha Veeranna. CIDPro: Custom Instructions for Dynamic Program Diversification. pages 224–2245.

[pho19] Phoronix test suite. `https://www.phoronix-test-suite.com/`, 2019. Accessed: (2021-01-16).

[PKA$^+$] Naman Kamleshbhai Patel, Prashanth Krishnamurthy, Hussam Amrouch, Jorg Henkel, Michael Shamouilian, Ramesh Karri, and Farshad Khorrami. Towards a New Thermal Monitoring Based Framework for Embedded CPS Device Security. pages 1–1.

[PMF$^+$] Thinh Hung Pham, Ben Marshall, Alexander Fell, Siew-Kei Lam, and Daniel Page. XDIVINSA: eXtended DIVersifying INStruction Agent to Mitigate Power Side-Channel Leakage. pages 179–186.

[PR10] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Annual cryptology conference*, pages 502–519. Springer, 2010.

[PVK15] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and software technology*, 64:1–18, 2015.

[RD20] Mark Randolph and William Diehl. Power side-channel attack analysis: A review of 20 years of study for the layman. *Cryptography*, 4(2):15, 2020.

[RP] Francesco Regazzoni and Ilia Polian. Side Channel Attacks vs Approximate Computing. pages 321–326.

[RQPA16] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 110–120, 2016.

[RRB18] Ali Ahmadian Ramaki, Abbas Rasoolzadegan, and Abbas Ghaemi Bafghi. A systematic mapping study on intrusion alert analysis in intrusion detection systems. *ACM Computing Surveys (CSUR)*, 51(3):1–41, 2018.

[SB13] S Sharath and Anirban Basu. Performance of eucalyptus and openstack clouds on futuregrid. *International Journal of Computer Applications*, 80(13), 2013.

[SF13] Ali Shahrokni and Robert Feldt. A systematic review of software robustness. *Information and Software Technology*, 55(1):1–17, 2013.

[SFQ] Gururaj Saileshwar, Christopher W. Fletcher, and Moinuddin Qureshi. Streamline: A fast, flushless cache covert-channel attack by enabling asynchronous collusion. pages 1077–1090.

[SL] Weidong Shi and Hsien-Hsin S. Lee. Authentication Control Point and Its Implications For Secure Processor Design. pages 103–112.

[SLKS19] Asanka Sayakkara, Nhien-An Le-Khac, and Mark Scanlon. A survey of electromagnetic side-channel attacks and discussion on their case-progressing potential for digital forensics. *Digital Investigation*, 29:43–54, 2019.

[SLR] Till Schlüter and Kerstin Lemke-Rust. Differential Analysis and Fingerprinting of ZombieLoads on Block Ciphers. volume 12609, pages 151–165.

[SMAKa] Benjamin Semal, Konstantinos Markantonakis, Raja Naeem Akram, and Jan Kalbantner. Leaky Controller: Cross-VM Memory Controller Covert Channel on Multi-core Systems. volume 580, pages 3–16.

[SMAKb] Benjamin Semal, Konstantinos Markantonakis, Raja Naeem Akram, and Jan Kalbantner. A Study on Microarchitectural Covert Channel Vulnerabilities in Infrastructure-as-a-Service. volume 12418, pages 360–377.

[SMGM] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. volume 10322, pages 247–267.

[SR] Hermann Seuschek and Stefan Rass. Side-Channel Leakage Models for RISC Instruction Set Architectures from Empirical Data. pages 423–430.

[SSL$^+$] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. volume 11735, pages 279–299.

[SWG$^+$] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Abusing Intel SGX to conceal cache attacks. 3(1):2.

[SYG$^+$] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. MicroScope: Enabling microarchitectural replay attacks. pages 318–331.

[SYZPa] Nader Sehatbakhsh, Baki Berkay Yilmaz, Alenka Zajic, and Milos Prvulovic. EMSim: A Microarchitecture-Level Simulation Tool for Modeling Electromagnetic Side-Channel Signals. pages 71–85.

[SYZPb] Nader Sehatbakhsh, Baki Berkay Yilmaz, Alenka Zajic, and Milos Prvulovic. A New Side-Channel Vulnerability on Modern Computers by Exploiting Electromagnetic Emanations from the Power Management Unit. pages 123–138.

[Sze] Jakub Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security*, 3(3):219–234.

[Sze19] Jakub Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security*, 3(3):219–234, 2019.

[TLM] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests. pages 947–960.

[TWZL] Mingtian Tan, Junpeng Wan, Zhe Zhou, and Zhou Li. Invisible Probe: Timing Attacks with PCIe Congestion Side-channel. pages 322–338.

[UJYSng] Musa Sadik Unal, Arsalan Javeed, Cemal Yilmaz, and Erkay Savas. Hyperdetector: Detecting, isolating, and mitigating timing attacks in virtualized environments. In *21st International Conference on Cryptology and Network Security Abu Dhabi, UAE*. IEEE, 2022 (forthcoming).

[VBMW$^+$18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[vds11] On vsyscalls and the vdso. `https://lwn.net/Articles/446528/`, 2011. Accessed: (2021-01-16).

[VNS$^+$] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M. Al-Hashimi, and Geoff V. Merrett. BRB: Mitigating Branch Predictor Side-Channels. pages 466–477.

[WFS14] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. Timing channel protection for a shared memory controller. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 225–236. IEEE, 2014.

[WHZ+] Xiaohui Wu, Yeping He, Qiming Zhou, Hengtai Ma, Liang He, Wenhao Wang, and Liheng Chen. Partial-SMT: Core-Scheduling Protection Against SMT Contention-Based Attacks. pages 378–385.

[WL07] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505, 2007.

[WMAG+] Chong Wang, Nasro Min-Allah, Bei Guan, Yu-Qi Lin, Jing-Zheng Wu, and Yong-Ji Wang. An Efficient Approach for Mitigating Covert Storage Channel Attacks in Virtual Machines by the Anti-Detection Criterion. 34(6):1351–1365.

[WMMR06] Roel Wieringa, Neil Maiden, Nancy Mead, and Colette Rolland. Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. *Requirements engineering*, 11(1):102–107, 2006.

[WNQ+19] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael B Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries. In *NDSS*, 2019.

[Woh14] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pages 1–10, 2014.

[WPJG] Zihao Wang, Shuanghe Peng, Wenbin Jiang, and Xinyue Guo. Defeating Hardware Prefetchers in Flush+Reload Side-Channel Attack. 9:21251–21257.

[WQAGK19] Daimeng Wang, Zhiyun Qian, Nael Abu-Ghazaleh, and Srikanth V Krishnamurthy. Papp: Prefetcher-aware prime and probe side-channel attack. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[WR09] Rafal Wojtczuk and Joanna Rutkowska. Attacking intel trusted execution technology. *Black Hat DC*, 2009:1–6, 2009.

[WS12] Yao Wang and G Edward Suh. Efficient timing channel protection for on-chip networks. In *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, pages 142–151. IEEE, 2012.

[WSG+20] Nils Wistoff, Moritz Schneider, Frank K Gürkaynak, Luca Benini, and Gernot Heiser. Prevention of microarchitectural covert channels on an open-source 64-bit risc-v core. *arXiv preprint arXiv:2005.02193*, 2020.

[WXW14] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud. *IEEE/ACM Transactions on Networking*, 23(2):603–615, 2014.

[WZ] Xin Wang and Wei Zhang. Cracking Randomized Coalescing Techniques with An Efficient Profiling-Based Side-Channel Attack to GPU. pages 1–8.

[Yar16] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit. *Retrieved from School of Computer Science Adelaide: http://cs. adelaide. edu. au/yval/Mastik*, 16, 2016.

[YB14] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack. *IACR Cryptol. ePrint Arch.*, 2014:140, 2014.

[YCPZ] Baki Berkay Yilmaz, Robert L. Callan, Milos Prvulovic, and Alenka Zajic. Capacity of the EM Covert/Side-Channel Created by the Execution of Instructions in a Processor. 13(3):605–620.

[YDG$^+$] Bilgiday Yuce, Chinmay Deshpande, Marjan Ghodrati, Abhishek Bendre, Leyla Nazhandali, and Patrick Schaumont. An Secure Exception Mode for Fault-Attack-Resistant Processing. 16(3):388–401.

[YF14] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[YGL$^+$15] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B Lee, and Gernot Heiser. Mapping the intel last-level cache. *Cryptology ePrint Archive*, 2015.

[YSG$^+$19] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 888–904. IEEE, 2019.

[ZAB07] Sebastian Zander, Grenville Armitage, and Philip Branch. A survey of covert channels and countermeasures in computer network protocols. *IEEE Communications Surveys & Tutorials*, 9(3):44–57, 2007.

[ZAM12] Danfeng Zhang, Aslan Askarov, and Andrew C Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 99–110, 2012.

[ZBPF] Davide Zoni, Alessandro Barenghi, Gerardo Pelosi, and William Fornaciari. A Comprehensive Side-Channel Information Leakage Analysis of an In-Order RISC CPU Microarchitecture. 23(5):1–30.

[ZJRR12] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316, 2012.

[ZR13] Yinqian Zhang and Michael K Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 827–838, 2013.

[ZSG16] Samer Zein, Norsaremah Salleh, and John Grundy. A systematic mapping study of mobile application testing techniques. *Journal of Systems and Software*, 117:334–356, 2016.

[ZSY$^+$] Pan Zhang, Chengyu Song, Heng Yin, Deqing Zou, Elaine Shi, and Hai Jin. Klotski: Efficient Obfuscated Execution against Controlled-Channel Attacks. pages 1263–1276.

[ZXZ16] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-oriented flush-reload side channels on arm and their implications for android devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 858–870, 2016.

APPENDIX A

Primary Studies

A.1 List of primary studies

Study ID	Title
S1	A Comprehensive Side-Channel Information Leakage Analysis of an In-Order RISC CPU Microarchitecture [ZBPF]
S2	An Architecture-Independent Instruction Shuffler to Protect against Side-Channel Attacks [BVIB]
S3	Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs [FL]
S4	Side Channel Attacks vs Approximate Computing [RP]
S5	A Practical Methodology for Measuring the Side-Channel Signal Available to the Attacker for Instruction-Level Events [CZP]
S6	A Secure Exception Mode for Fault-Attack-Resistant Processing [YDG+]
S7	Authentication Control Point and Its Implications For Secure Processor Design [SL]
S8	BranchSpec: Information Leakage Attacks Exploiting Speculative Branch Instruction Executions [ICLY]
S9	CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests [TLM]
S10	Composable Template Attacks Using Templates for Individual Architectural Components [LLWR]
S11	CoPHEE: Co-processor for Partially Homomorphic Encrypted Execution [NAC+]
S12	Fooling the Sense of Cross-Core Last-Level Cache Eviction Based Attacker by Prefetching Common Sense [Pan]
S13	Leaky Buddies: Cross-Component Covert Channels on Integrated CPU-GPU Systems [DNAG+]
S14	Side-Channel Leakage Models for RISC Instruction Set Architectures from Empirical Data [SR]
S15	SPECUSYM: Speculative Symbolic Execution for Cache Timing Leak Detection [GCL+]
S16	XDIVINSA: eXtended DIVersifying INStruction Agent to Mitigate Power Side-Channel Leakage [PMF+]
S17	An Abstract Semantics of Speculative Execution for Reasoning About Security Vulnerabilities [CW]
S18	C5: Cross-Cores Cache Covert Channel [MNHF]
S19	CCCiCC: A Cross-Core Cache-Independent Covert Channel on AMD Family 15h CPUs [HLRP]
S20	Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript [SMGM]
S21	Leaky Controller: Cross-VM Memory Controller Covert Channel on Multi-core Systems [SMAKa]
S22	Model Checking Speculation-Dependent Security Properties: Abstracting and Reducing Processor Models for Sound and Complete Verification [CCFV]
S23	A Novel Covert Channel Attack Using Memory Encryption Engine Cache [HK]

S24	Automated Software Protection for the Masses Against Side-Channel Attacks [BCHC]
S25	BranchScope: A New Side-Channel Attack on Directional Branch Predictor [ERAGP]
S26	Cache Timing Side-Channel Vulnerability Checking with Computation Tree Logic [DXS]
S27	Can Randomized Mapping Secure Instruction Caches from Side-Channel Attacks? [LWL]
S28	Cracking Randomized Coalescing Techniques with An Efficient Profiling-Based Side-Channel Attack to GPU [WZ]
S29	Formal Verification of Software Countermeasures against Side-Channel Attacks [EWS]
S30	How Secure is Your Cache against Side-Channel Attacks? [HL]
S31	Klotski: Efficient Obfuscated Execution against Controlled-Channel Attacks [ZSY+]
S32	MicroScope: Enabling Microarchitectural Replay Attacks [SYG+]
S33	On the Detection of Applications in Co-Resident Virtual Machines via a Memory Deduplication Side-Channel [LF]
S34	Power Side Channel Attack Analysis and Detection [GKDG]
S35	Rendered Insecure: GPU Side Channel Attacks Are Practical [NNQAG]
S36	Streamline: A Fast Flushless Cache Covert-Channel Attack by Enabling Asynchronous Collusion [SFQ]
S37	TRUSTORE: Side-Channel Resistant Storage for SGX Using Intel Hybrid CPU-FPGA [OAP+]
S38	A Fetching Tale: Covert Communication with the Hardware Prefetcher [CY]
S39	A New Side-Channel Vulnerability on Modern Computers by Exploiting Electromagnetic Emanations from the Power Management Unit [SYZPb]
S40	A Time-/Frequency-Domain Side-Channel Attack Resistant AES-128 and RSA-4K Crypto-Processor in 14-nm CMOS [KLS+]
S41	BRB: Mitigating Branch Predictor Side-Channels. [VNS+]
S42	Capacity of the EM Covert/Side-Channel Created by the Execution of Instructions in a Processor [YCPZ]
S43	CIDPro: Custom Instructions for Dynamic Program Diversification [PFB+]
S44	Cross-VM Cache Attacks on AES [GII+]
S45	DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors [KLA+]
S46	Defeating Hardware Prefetchers in Flush+Reload Side-Channel Attack [WPJG]
S47	Defeating Speculative-Execution Attacks on SGX with HyperRace [CLZZ]
S48	Efficient and side-channel-secure block cipher implementation with custom instructions on FPGA [MTS]
S49	Efficient Power-Analysis-Resistant Dual-Field Elliptic Curve Cryptographic Processor Using Heterogeneous Dual-Processing-Element Architecture [LCCL]

S50	EMSim: A Microarchitecture-Level Simulation Tool for Modeling Electromagnetic Side-Channel Signals [SYZPa]
S51	Flush-Detector: More Secure API Resistant to Flush-Based Spectre Attacks on ARM Cortex-A9 [HMG+]
S52	Hardware-Assisted MMU Redirection for In-Guest Monitoring and API Profiling [HSC]
S53	Invisible Probe: Timing Attacks with PCIe Congestion Side-channel [TWZL]
S54	PARAM: A Microprocessor Hardened for Power Side-Channel Attack Resistance [AKFGBR]
S55	Partial-SMT: Core-Scheduling Protection Against SMT Contention-Based Attacks [WHZ+]
S56	Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors [BNB+]
S57	REHAD: Using Low-Frequency Reconfigurable Hardware for Cache Side-Channel Attacks Detection [MMN]
S58	Reinforcing Meltdown Attack by Using a Return Stack Buffer [KSb]
S59	Side-channel security of superscalar CPUs : Evaluating the Impact of Micro-architectural Features [BP]
S60	SPARX — A side-channel protected processor for ARX-based cryptography [BSMG]
S61	Synthesis of Parallel Synchronous Software [KSa]
S62	Towards a New Thermal Monitoring Based Framework for Embedded CPS Device Security [PKA+]
S63	A Bit-Level Approach to Side Channel Based Disassembling [CLH]
S64	A Study on Microarchitectural Covert Channel Vulnerabilities in Infrastructure-as-a-Service [SMAKb]
S65	A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL [AS]
S66	An Efficient Approach for Mitigating Covert Storage Channel Attacks in Virtual Machines by the Anti-Detection Criterion [WMAG+]
S67	Cache-Zoomer: On-demand High-resolution Cache Monitoring for Security [FDY+]
S68	Calibration Done Right: Noiseless Flush+Flush Attacks [DM]
S69	Can Microkernels Mitigate Microarchitectural Attacks? [GLV+]
S70	CHASM: Security Evaluation of Cache Mapping Schemes [MGK+]
S71	Differential Analysis and Fingerprinting of ZombieLoads on Block Ciphers [SLR]
S72	E-SGX: Effective Cache Side-Channel Protection for Intel SGX on Untrusted OS [LLW+]
S73	Efficient Information-Flow Verification Under Speculative Execution [BJV]
S74	Flush+Flush: A Fast and Stealthy Cache Attack [GMWM]
S75	HexPADS: A Platform to Detect “Stealth” Attacks [Pay]
S76	Integrating Side Channel Security in the FPGA Hardware Design Flow [BBF+]
S77	Malware Guard Extension: abusing Intel SGX to conceal cache attacks [SWG+]

S78	MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations [MWES]
S79	NetSpectre: Read Arbitrary Memory over Network [SSL$^+$]
S80	New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures [AGS]
S81	New Results on Instruction Cache Attacks [ABG]
S82	Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution [GVBPS]
S83	Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications [GOPT]
S84	Validation of Abstract Side-Channel Models for Computer Architectures [NBL$^+$]

Table A.1: curated articles to serve as primary studies