Universidade do Minho
Escola de Engenharia

João Pedro Antunes Gomes da Silva Reis

Cache-based Timing Side-channels in Partitioning Hypervisors

Dezembro de 2019

Universidade do Minho
Escola de Engenharia

João Pedro Antunes Gomes da Silva Reis

Cache-based Timing Side-channels in Partitioning Hypervisors

Dissertação de Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do

Professor Doutor Sandro Pinto

Dezembro de 2019

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Agradecimentos

As minhas primeiras palavras de apreço são dirigidas ao meu orientador Professor Doutor Sandro Pinto, pela partilha de conhecimento e sugestões ao longo da dissertação. De igual forma, agradeço ao Mestre José Martins pela prontidão e celeridade com que acorreu às minhas dúvidas. A disponibilidade que sempre demonstraram no decorrer da dissertação e o informalismo que demarcou a nossa relação, tornaram a realização da dissertação um projeto proveitoso e aprazível.

A todos os meus companheiros do laboratório *Embedded Systems Research Group* que me acompanharam durante a dissertação, agradeço os momentos importantes de companheirismo vividos durante o ano.

Em último, mas não menos importante, agradeço à minha família, especialmente aos meus pais e ao meu irmão, por me terem dado apoio incondicional e me terem alegrado nos momentos de maior angústia.

Um obrigado a todos!

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Abstract

Cache-based Timing Side-channels in Partitioning Hypervisors

In recent years, the automotive industry has seen a technology complexity increase to comply with computing innovations such as autonomous driving, connectivity and mobility. As such, the need to reduce this complexity without compromising the intended metrics is imperative.

The advent of hypervisors in the automotive domain presents a solution to reduce the complexity of the systems by enabling software portability and isolation between virtual machines (VMs).

Although virtualization creates the illusion of strict isolation and exclusive resource access, the convergence of critical and non-critical systems into shared chips presents a security problem. This shared hardware has microarchitectural features that can be exploited through their temporal behavior, creating sensitive data leakage channels between co-located VMs. In mixed-criticality systems, the exploitation of these channels can lead to safety issues on systems with real-time constraints compromising the whole system.

The implemented side-channel attacks demonstrated well-defined channels, across two real-time partitioning hypervisors in mixed-criticality systems, that enable the inference of a co-located VM's cache activity. Furthermore, these channels have proven to be mitigated using cache coloring as a countermeasure, thus increasing the determinism of the system in detriment of average performance.

From a safety perspective, this dissertation emphasizes the need to weigh the tradeoffs of the trending architectural features that target performance over predictability and determinism.

Keywords: Automotive Industry, Mixed-criticality Systems, Side-channel Attacks, Virtualization

Resumo

Cache-based Timing Side-channels in Partitioning Hypervisors

Nos últimos anos, a indústria automotiva tem sido objeto de um crescendo na sua complexidade tecnológica de maneira a manter-se a par das mais recentes inovações de computação. Sendo assim, a necessidade de reduzir a complexidade sem comprometer as métricas pretendidas é imperativa.

O advento dos hipervisores na indústria automotiva apresenta uma solução para a redução da complexidade dos sistemas, possiblitando a portabilidade do software e o isolamento entre *virtual vachines* (VMs).

Embora a virtualização crie a ilusão de isolamento e acesso exclusivo a recursos, a convergência de sistemas críticos e não-críticos em *chips* partilhados representa um problema de segurança. O hardware partilhado tem características microarquiteturais que podem ser exploradas através do seu comportamento temporal, criando canais de fuga de informação crítica entre VMs adjacentes. Em sistemas de criticalidade mista, a exploração destes canais pode comprometer sistemas com limitações de tempo real.

Os ataques *side-channel* implementados revelam canais bem definidos que possibilitam a inferência da atividade de cache de VMs situadas no mesmo processador. Além disso, esses canais provaram serem passíveis de ser mitigados usando *cache coloring* como estratégia de mitigação, aumentando assim o determinismo do sistema em detrimento da sua performance.

De uma perspetiva da segurança, esta dissertação enfatiza a necessidade de pesar os *tradeoffs* das tendências arquiteturais que priorizam a performance e secundarizam o determinismo e previsibilidade do sistema.

Palavras-chave: Ataques *Side-channel*, Indústria Automotiva, Sistemas de Criticalidade Mista, Virtualização

Contents

List of Figures

List of Tables

List of Listings

Glossary

ABI	Application Binary Interface
ACP	Accelerator Coherency Port
ACTLR	Auxiliary Control Register
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
APU	Application Processor Unit
ASIC	Aplication Specific Integratedd Circuit
AXI	Advanced Extensible Interface
CE	Consumer Electronics
CPU	Central Processing Unit
DCU	Domain Control Unit
DMA	Direct Memory Access
DMAC	Direct Memory Access Controller
DoS	Denial-of-Service
DPR	Dynamic Partial Reconfiguration
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processing
E/E	Electrical/Electronic
ECU	Engine Control Unit
FIQ	Fast Interrupt Request
FPGA	Field Programmable Gate Array
GIC	Generic Interrupt Controller
GPIO	General-Purpose Input/Output

GPOS	General-Purpose Operating System
I/O	Input/Output
IaaS	Infrastructure-as-a-Service
IDE	Integrated Development Environment
IOMMU	Input/Output Memory Management Unit
IoT	Internet of Things
IPC	Inter-Partition Communication
IPI	Inter-Processor Interrupt
IRQ	Interrupt Request
ISR	Interruption Service Routines
IVI	In-Vehicle Infotainment
LLC	Last-level Cache
LR	Linker Register
LTZVisor	Lightweight TrustZone-assisted Hypervisor
MMU	Memory Management Unit
MPMU	Message Passing Management Unit
OCM	On-Chip Memory
OS	Operating System
PA	Physical Address
PFN	Page Frame Number
PIPT	Physically Indexed, Physically Tagged
PIVT	Physically Indexed, Virtually Tagged
PL	Programmable Logic
PS	Processing System
RAM	Random-Access Memory
ROM	Read-Only Memory
RTOS	Real-Time Operating System
SCTLR	System Control Register
SCU	Snoop Control Unit
SLCR	System-Level Control registers
SMC	Secure Monitor Call

SMT	Simultaneous Multithreading
SoC	System on-Chip
SP	Stack Pointer
SPSR	Saved Program Status Register
SRAM	Static Random-Access Memory
TCB	Trusted Computing Base
TZ	TrustZone
VIPT	Virtually Indexed, Physically Tagged
VIVT	Virtually Indexed, Virtually Tagged
VM	Virtual Machine
VMCB	Virtual Machine Control Block
VMM	Virtual Machine Monitor
XSDK	Xilinx Software Development Kit

1. Introduction

The embedded systems used to be relatively simple, single-purpose devices with very specific functionalities focused on meeting hardware constraints and real-time requirements, thus exhibiting low to moderate software complexity. Over the years, while still being resource-constrained and keeping their real-time behavior, they started trending towards general-purpose systems with much more functionalities that impose a growth in software complexity [2], as it has been the automotive industry's case [3].

With the rising use of consumer electronics, comes a rising expectancy of infotainment-wise performance of modern vehicles. As such, the automotive industry must bridge the gap between consumers' expectations for infotainment features and the current performance of today's automotive systems. To address these problems, virtualization is emerging as a key solution.

Among the many benefits virtualization brings in the automotive domain, software portability and isolation between VMs stand as the most important features in virtualized automotive systems. With virtualization, the automotive systems can be separated into critical and non-critical partitions, allowing the integration of components with different criticality levels onto modern multi-core System on-Chips (SoCs) [4]. This feature makes it possible for the firmware of one partition (e.g., non-critical) to be updated without affecting the other, thus reducing the overall time-to-market and manufacturing costs [5] [6]. At the lower level, resides a hypervisor to enforce isolation - thus, reliability and safety - between the guest operating systems.

The automotive industry is also dependent on virtualization solutions due to the amount of needed Electronic Control Units (ECUs), which, prior to virtualization application, would require a set of dedicated microcontrollers, and which are currently being consolidated into fewer platforms, named Domain Control Units (DCUs).

While it is true that the isolation of VMs from each other was and is one of the goals of virtualization, it is inappropriate to consider the latter as a technique for bringing security into a system [7]. This results from the fact that using VMs may not be as secure as running OSes on dedicated physical hardware each.

Consequently, extreme precautions need to be taken in order to not introduce security risks and attack vectors with virtualization, otherwise, the safety of modern vehicles can be compromised.

The security and safety of a vehicle are tightly coupled, as one metric compromises the other. If a mixed-criticality system is vulnerable to attacks, the execution of critical tasks, such as airbag deployment, can be delayed by populating a cache that is shared by two critically-different VMs through a non-critical domain [5]. As the cache is shared, the non-critical domain can populate the shared cache, with the result that the content stored by the critical domain can be evicted, hence provoking cache misses at the next access. This phenomenon may generate large and unpredictable interference across domains, thus breaking isolation by introducing a strong coupling of their timing properties.

The shared hardware has microarchitectural features, such as control speculation [8], shared caches [9][10][1] or interruption service routines (ISRs) [11] that can be exploited through their temporal behavior, creating sensitive data leakage channels between co-located VMs. These channels can occur at any degree of hardware sharing and concurrency involved, as long as there are at least two VMs within the same chip [1].

This thesis focuses on the open problem of microarchitectural timing channels, which exploit timing variations to leak information between co-located VMs. They are harder to deal with, partially because of the breadth of exploitable mechanisms, being the usual defence to strive for deterministic execution time via constant-time algorithms [9] [12], which degrades substantially the system's performance.

1.1 Goals

The goals to be achieved are summarized as follows:

1. Replicate the timing side-channel attacks which explore the microarchitecture of ARM-based platforms.

2. Show the effectiveness of these attacks when the targeted system's security is not appropriately secured by an hypervisor which provides isolation across VMs, by benchmarking the replicated attacks.

3. Propose different mitigation strategies which prove to be relevant across similar ARM-based platforms, ultimately bringing security into the system.

2. Background

In this chapter, it is provided the required preliminaries and discussed related work in the context of cache attacks.

The themes addressed in this chapter range from the detection of exploitable modern caches' properties, to the dissection of known countermeasures against cache attacks. The concept of virtualization and its prominence in the embedded systems' world are briefly explained to introduce the software (i.e., Jailhouse, Bao) that is leveraged to find channels and upgraded to mitigate them.

2.1 State of Art

Back in 1996, Kocher et al. [13] showed that it is possible to find factor RSA keys and break other cryptographic systems by measuring the execution time of private key operations. In 1999 Kocher et al. [14] introduced Simple Power Analysis (SPA) and Differential Power Analysis (DPA) where an attacker can extract cryptographic keys by studying the power consumption of a device.

Then, a different kind of side-channel focused on exploring the memory hierarchy, more specifically the caches, was introduced. Cache side-channel attacks exploit the different access times of memory addresses that are either held in the cache or the main memory. While the *Evict+Time* and *Prime+Probe* techniques by Osvik et al. [15] explicitly targeted cryptographic algorithms, Yarom et al. [16] introduced the *Flush+Reload* attack in 2014 that laid the foundation for new attack scenarios. The *Flush+Reload* attack allows an attacker to determine which specific parts of a shared library or a binary executable have been accessed by the victim with unprecedented high accuracy.

Although a major part of the known side-channel attacks targets the caches, they do not restrict to only one component of the microarchitecture. As Heiser et al. [1] demonstrated, these attacks occur across all hardware sharing levels (i.e., thread shared through system shared) and concurrency involved (i.e., full concurrency, time-sliced execution on a single core, or hardware threading). Heiser et al. [1] affirmed that the attacks that occur closer to the core tend to achieve higher severity because they have more precise information available, while those at lower levels (e.g. the bus interconnect) could be mostly used to cause interference, thus jeopardizing the determinism of the co-located cores.

In 2015, Heiser et al. [17] have used cross-core cache attacks that relied on the inclusiveness of Last-level Caches (LLCs). The problem is that only the x86 architectures and the recent ARMv8 architectures used inclusive LLCs. To the other architectures that didn't use inclusive LLCs, Irazoqui et al. [18] exploited cache coherence protocols to mount cross-core cache attacks on SoCs with non-inclusive shared LLCs.

In 2016, Moritz et al. [19] attacked cryptographic implementations and utilized microarchitectural timing side-channel attacks to infer sensitive information (e.g., to differentiate between entered letters and special keys on the keyboard, or measure the length of swipe and touch gestures) of ARM-based smartphones. This disclosed the immense threat that those attacks represent, since they can be mounted on millions of stock Android devices without the requirement of any privileges or permissions.

In 2018, Heiser et al. [1] summarised all microarchitectural attacks known to date, alongside existing mitigation strategies, and developed a taxonomy based on both the degree of hardware sharing and concurrency involved.

2.2 Virtualization

The increasing importance of isolation and security in modern systems (either for cloud servers which hold, more than ever, confidential and critical information, and more recently, embedded systems which are responsible for performing critical operations [20][21][22]), the underutilization of hardware, the increase in raw speed of processors over the decades (which makes the overhead of VMs more tolerable) and the preference of multicore processors over single-core processors [2] [7], paved the way to a natural adoption of the virtualization technology by the enterprise, cloud computing and embedded systems domains [23].

2.2.1 Definition

Virtualization in general terms refers to the act of creating or using a virtual version of a resource (e.g., computer hardware platforms, operating systems, storage devices or network resources) rather than the physical one [24].

A virtual machine (VM) is commonly defined as a software program that emulates the behavior of a separate computer system, being capable of performing tasks such as running applications and programs like a separate computer. The emulated system is called the *guest* system and the system on which it is being emulated is called the *host* system. The concept of emulation allows software or a peripheral designed for the guest system to be executed on the host system.

In virtualized systems, a single computer can run multiple VMs supporting a number of different operating systems (OSes) with all of them sharing the hardware resources. This contrasts with a conventional platform, where a single OS owns all hardware resources and no other OS can obtain them [25].

2.2.2 Advantages

The relevance of virtualization in embedded systems stems from the ability to address some of the new challenges posed by them.

There are 3 main advantages regarding the use of virtualization on modern machines:

1. Improving protection, as it allows Infrastructure-as-a-Service (IaaS) companies (e.g., Amazon Web Services) to protect users from each other while sharing the same server [25], and general-purpose systems from jeopardizing systems with real-time constraints under the same hardware resources [2].

   The protection is guaranteed as the access to memory between guests is not reachable by userland, because the accessed memory is perceived by the users as the physical one, instead of a virtual one which is then remapped to a physical address.

2. Managing software, which enables a developer to run multiple systems (some of them stable, and other, unstable releases) in one virtualized server without having a bunch of servers sitting around for the occasional use of tests [25], or providing architectural abstraction, as the same software architecture can be migrated essentially unchanged between a multicore and a (virtualized) single-core [2].

3. Managing hardware, by enabling the concurrent execution of an application OS (e.g., Linux) and a real-time OS (RTOS) on the same processor. This way, virtualization provides support for heterogeneous operating-system environments, as a way to address the conflicting requirements of high-level APIs for application programming, real-time performance, and legacy support [2].

Furthermore, the embedded systems can use the hardware management feature to meet their real-time constraints, which means: (i) managing the power consumption of a system by dynamically adding cores to an application domain which requires extra processor power, or (ii) removing processors and shutting down idle cores.

2.2.3 Disadvantages

The costs of using virtualization in a business context depend on the requirements to run virtual machines. The embedded systems environment distinguishes from the server/desktop environment by its resource-constrained nature. The embedded systems are, by their nature, highly integrated and this characteristic conflicts with the isolation factor that virtualization brings onto them. Hence, there needs to be an effort made to not compromise the functional requirements of the embedded system without neglecting the isolation between the subsystems that compose a modern embedded system [21][**?**][26].

The aforementioned advantages that made the virtualization a must on modern embedded systems, have some intrinsic mismatches with the embedded systems requirements that need to be addressed:

1. Scheduling, as both co-located VMs are treated as black boxes with universal priorities, neglecting the fact that within the real-time VMs there are tasks that should interleave with tasks of the other general-purpose VMs [2].

2. Software complexity problem, that can lead, among other problems, to performance deterioration. Virtualization introduces a layer of memory abstraction that increases cache contention, mostly related to the frequent invocation of the hypervisor's enter-and-exit operations [27], and an increase in time taken to do memory operations, as every operation has to look up in page table, which means it needs to access the page table and then the memory address.

One can conclude that the process of virtualization can be used to provide platform independence and a secure environment for execution but this comes at a cost. Hence, one must ensure that embedded hypervisors are used only when needed and the overhead and performance dip is well compensated by the use of caching or multicore environments [24].

2.2.4 Hypervisors

With the increasing importance of virtualization in embedded systems, comes the need to address the software layer that underlies the VMs. This section presents the definition and how hypervisors have evolved into different implementations that can be classified in many ways.

2.2.4.1 Definition

The software that supports VMs is called a virtual machine monitor (VMM) or hypervisor, and presents a code size much smaller than a traditional OS. The underlying hardware platform is called the host, and its resources are shared among the guest VMs. The hypervisor determines how to map virtual resources to physical resources: a physical resource may be time-shared, partitioned, or even emulated in software [4] [21].

It presents a software interface to guest software, it must isolate the state of guests from each other, it must protect itself from guest software, and it must ensure that the guest system only interacts with virtual resources (i.e. a conventional guest OS runs as a user-mode program on top of the hypervisor) [4][25].

The qualitative requirements are:

1. Guest software should behave on a VM exactly as if it were running on the native hardware (to improve emulation speed), except for performance-related behavior or limitations of fixed resources shared by multiple VMs [25]. This can be acquired by making the ISA of the VM the same as the host.

2. Guest software should not be able to change the allocation of real system resources directly [25].

To improve the performance of virtual machines, it is needed to:

1. Reduce the cost of processor virtualization [21].

2. Reduce interrupts overhead cost due to the virtualization [21].

3. Reduce interrupt cost by steering interrupts to the proper VM without invoking the hypervisor [21].

2.2.4.2 Classification

There are two types of hypervisors:

1. **Type-1** - it runs directly on the hosting hardware to control it and to handle guest operating systems [21] [26];

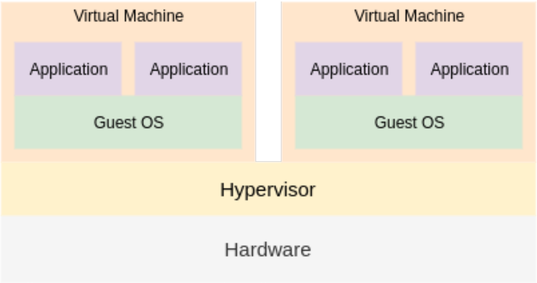


Figure 2.1: Illustration of a system virtualization stack with Type-1 hypervisor.

2. **Type-2** - hypervisor is provided as an extension to an operating system that is executed on the host while the guests run as tasks [21] [26].

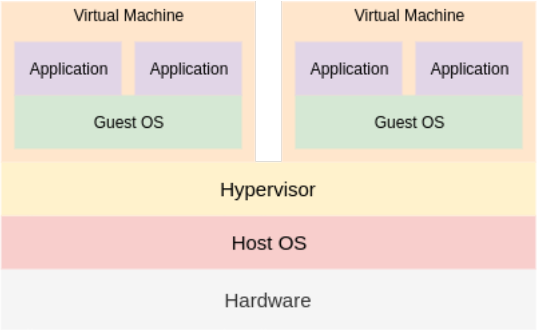


Figure 2.2: Illustration of a system virtualization stack with Type-2 hypervisor.

Another element of distinction comes from the API exposed by the host to the generic guest OS:

1. **Full Virtualization** - the guest executes transparently and without software modifications, while the hypervisor provides the API to emulate the underlying platform [26][28];

2. **Paravirtualization** - the guest is aware of the presence of virtualization. Thus it uses an API similar, but not identical, to that of the underlying hardware. This allows to create specific solutions and reduce the overhead [26][28].

2.3 Jailhouse

Jailhouse is a real-time, OS-agnostic partitioning hypervisor with a minimal code base that aims to minimize hypervisor activity and focus on isolation and resource partitioning. The system is divided into isolated domains that directly access physical resources, instead of recurring to resource virtualization and scheduling (e.g., Xen hypervisor) [29].

This static approach derives from Jailhouse's target domain, which is safety-critical industrial applications [30], and allows to:

1. **Provide average latencies and jitters similar to bare metal solutions** - Jailhouse's only task is to use virtualization techniques to isolate guests, but doesn't emulate any devices for them. Besides that, it does one-to-one resource assignment to separate resources between partitions which means that, if one partition has access to some I/O port, PCI device or any other resource, the other partition hasn't. These properties make the performance of the Jailhouse to be very close as if tasks run on bare metal [31].

2. **Ease potential certification processes** - Safety-critical industrial applications need to be certified according to numerous safety standards, and these standards give more strict requirements on systems with higher criticality. So, it is important to keep the complexity of the critically-high systems low, to ease the process of validation and certification, being this the main reason for the minimalistic code base of Jailhouse [30].

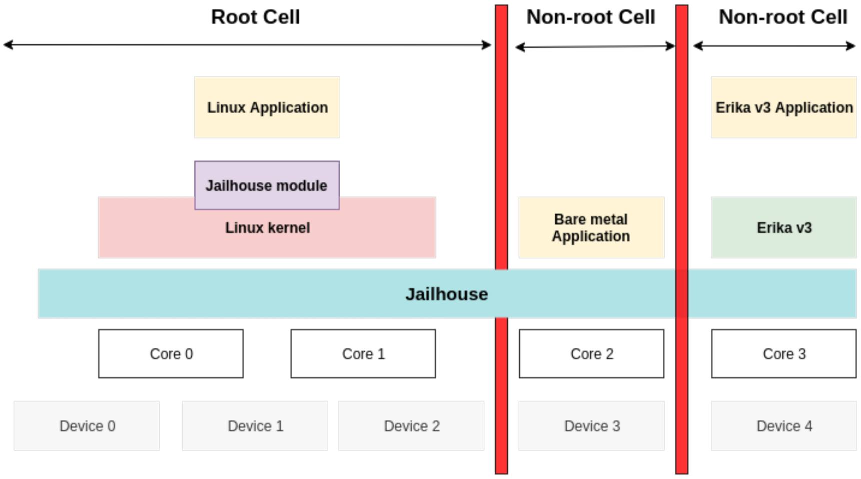


Figure 2.3: Illustration of Jailhouse's partitioning scheme. The root cell contains Linux, the Jailhouse module and two cores along with three devices. The first non-root cell (left) contains a bare-metal application and one core along with one device. The second non-root cell (right) contains Erika v3 and one core along with one device.

Regarding the classification of the hypervisor, it doesn't fit in the traditional classification (i.e., Type-1 or Type-2 hypervisor) because it runs on hardware like a bare-metal hypervisor, but needs to use Linux as bootloader to provide initialized hardware [29].

2.3.1 Terminology

Each partition (i.e., allocated physical hardware) is called a *cell*, while the software, or VM, that can only reach that subset of physical hardware (i.e., *cells*) is called the *inmate*.

The partitions are divided in two categories: (i) the *root cell*, and (ii) the *non-root cells*. The partition with Linux that bootstraps Jailhouse and from where other cells could be managed is called *root cell*. The other partitions that are added afterward, which may contain RTOSes or GPOSes, are called *non-root cells*.

2.3.2 Operation

This section describes the basic Jailhouse functionality, explains internal processes and also provides steps which should be done to enable and start the inmate in a cell.

2.3.2.1 Cell Configuration

The first concept that the user must be aware of is the configuration of the cells (root or non-root). The configuration is done statically before Jailhouse starts running, and it determines which hardware resources can be accessed by each cell.

The configurations are done by using .c files where parameters have to be assigned as fields of special C structures (defined in *cell-config.h* file). For the non-root cell, this setup looks like in Listing 2.1.

Listing 2.1: Non-root cell code example.

```
struct {
    /*The size of arrays there must correspond with the amount of
    fields of each type.*/
    struct jailhouse_cell_desc cell;
    __u64 cpus[1];
    struct jailhouse_memory mem_regions[3];
} __attribute__((packed)) config = {
    .cell = {
        .signature = JAILHOUSE_CELL_DESC_SIGNATURE,
        .revision = JAILHOUSE_CONFIG_REVISION,
        .name = "gic-demo",
        .flags = JAILHOUSE_CELL_PASSIVE_COMMREG,
        .cpu_set_size = sizeof(config.cpus),
        .num_memory_regions = ARRAY_SIZE(config.mem_regions),
        .num_irqchips = 0,
        .pio_bitmap_size = 0,
        .num_pci_devices = 0,

        .console = {
            .address = 0xff010000,
            .type = JAILHOUSE_CON_TYPE_XUARTPS,
            .flags = JAILHOUSE_CON_ACCESS_MMIO |
                 JAILHOUSE_CON_REGDIST_4,
        },
    },
    /*CPUs which are assigned to a cell.
    <n> bit set = core <n> will be used.*/
    .cpus = {
        0x8, /* e.g., core 3 is assigned*/
    },
    /*Here is setup which mem regions this cell
    could have access and with which rights (flags).*/
    .mem_regions = {
        /* UART */ {
            .phys_start = 0xff010000,
            .virt_start = 0xff010000,
            .size = 0x1000,
            .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                JAILHOUSE_MEM_IO | JAILHOUSE_MEM_ROOTSHARED,
        },
        /* RAM */ {
            .phys_start = 0x800600000,
            .virt_start = 0,
            .size = 0x00010000,
            .flags = JAILHOUSE_MEM_READ |
                     JAILHOUSE_MEM_WRITE ,
        },
        /* communication region */ {
            .virt_start = 0x80000000,
            .size = 0x00001000,
            .flags = JAILHOUSE_MEM_READ |
                     JAILHOUSE_MEM_WRITE,
        },
    }
};
```

2.3.2.2 Jailhouse Enabling

After configuring appropriately each cell, the user still needs to provide the reserved memory region to Jailhouse and non-root cells, appending *mem=* (which assigns the non-reserved memory space) kernel parameter on boot. The typical memory layout after reserving memory is depicted in Figure 2.4.

The values of *.phys_start* and *.size* in the header of root cell configuration, and the physical address values of the defined memory regions on non-root cell configuration must be within the reserved memory.

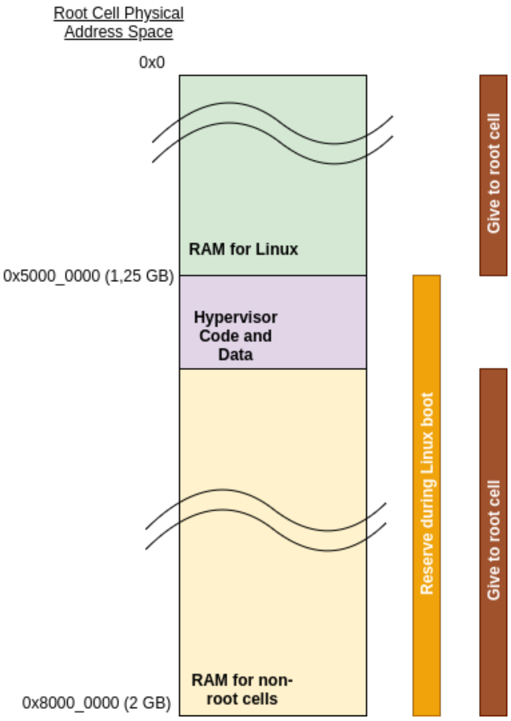


Figure 2.4: Typical RAM layout in *ZCU104 Evaluation kit*. The first 1280 MB are available for the root cell, while the rest of the memory is reserved for the hypervisor and the non-root cells.

The first step to enable Jailhouse is to load the *jailhouse.ko* module into the kernel, which enables */dev/jailhouse* in the system to enable the operation with Jailhouse user-space tools.

The second step is to execute jailhouse user-space program *jailhouse enable <path/to/cell/conf.cell>*. In this program, the driver remaps the reserved memory region to the kernel address space memory, so hypervisor could be accessed from the user-space. The driver also copies that binary at the start of this memory area and cell configuration right after it [30].

2.3.2.3 Activation Procedure

Jailhouse can only be enabled after the full boot of Linux using a kernel module. After Linux inserts the module into the kernel, the hypervisor takes control over all hardware resources and, according to a partition configuration file (available for each cell), it reassigns the hardware to the cells, lifting Linux into the state of a VM [29].

The deferred activation procedure of Jailhouse has the considerable practical advantage that the majority of hardware initialization is fully offloaded to Linux, and Jailhouse can entirely concentrate on managing virtualization extensions [29].

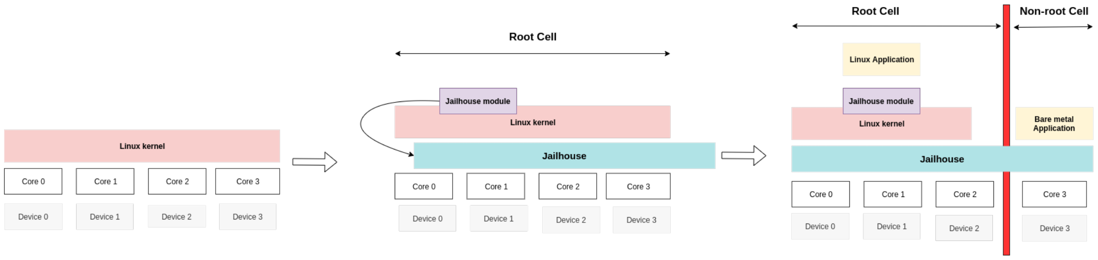


Figure 2.5: Example of the Jailhouse's activation procedure. After being initialized, Jailhouse reassigns hardware to Linux and a Bare metal application.

Although it follows a hardware partitioning philosophy, Jailhouse allows the sharing of memory between cells in a region called *ivshmem*, enabling inter-cell communication and communication from the guests to the hypervisor. Such concurrent access is, however, not arbitrated by Jailhouse and needs to be addressed appropriately by the guests, in the correspondent configuration files [29].

Since Jailhouse only remaps and reassigns resources, the ideal design concept is that it does not need to be active after setting up and starting all guests. It only intercepts in case of access violations (e.g., illegal access across partitions), excepting some circumstances that still require intervention by the VMM, such as (i) interrupt reinjection [29], and (ii) interception of non-virtualizable hardware resources (e.g., parts of the GIC on ARM) [29].

2.4 Bao

Bao is a real-time partitioning hypervisor that follows a similar approach to Jailhouse's approach concerning the preference of a minimal hypervisor activity that leverages hardware virtualization support to focus on isolation between guests and fault-containment over scheduling, as they target the same applications.

Bao was originally developed to serve as a base scaffold to research on, and deepen VM microarchitectural isolation. Nowadays, it supports Armv8-A and RISC-V architectures and is readily available through an open-source license.

Among the characteristics of it, Bao partitions and assigns resources at virtual machine (VM) instantiation time, and the virtual interrupts are directly mapped to physical ones, with one-to-one virtual-to-physical CPU mapping. Unlike other open-source hypervisors, Bao has no external dependencies, especially on privileged VMs running untrustable, large monoliths such as Linux, and as such, comprises a much smaller trusted computing base (TCB).

Regarding the classification of the hypervisor, it can be classified as a Type-1 hypervisor as it runs directly on top of the hosting hardware and, unlike Jailhouse, it loads firstly the hypervisor image which then allocates pages for the guests. This contrasts with the deferred activation procedure of Jailhouse, having the disadvantage of increasing the code size of the hypervisor since it needs to handle the hardware initialization. On the other hand, it is completely independent of Linux, which presents three main advantages: (i) as the hypervisor doesn't rely on Linux to boot, it has faster boot time than Jailhouse, (ii) it has a more universal behavior as it can host more operating systems aside Linux, and (iii) since the hypervisor is not controlled by a trusted partition (i.e., root cell), it has a much smaller trusted computing base (TCB), hampering the occurrence of attacks that could use Linux to jeopardize the hypervisor.

2.5 Caches

The steadfast development of the technology in the last decades has seen a divergence between the clock rate of processors and the latency of memory, creating a bottleneck between both memory and processor technologies that can be detrimental to the overall performance of a system [1]. Due to this divergence, the way that the memory is accessed needed to be changed for its latency to be reduced.

The memory hierarchy solution, sustained by a philosophy of using different memory technology layers that range from the fastest (but costliest) memory to the largest and cheapest memory, proved to be the solution that provides the best performance and narrows the bridge between memory and processor speed.

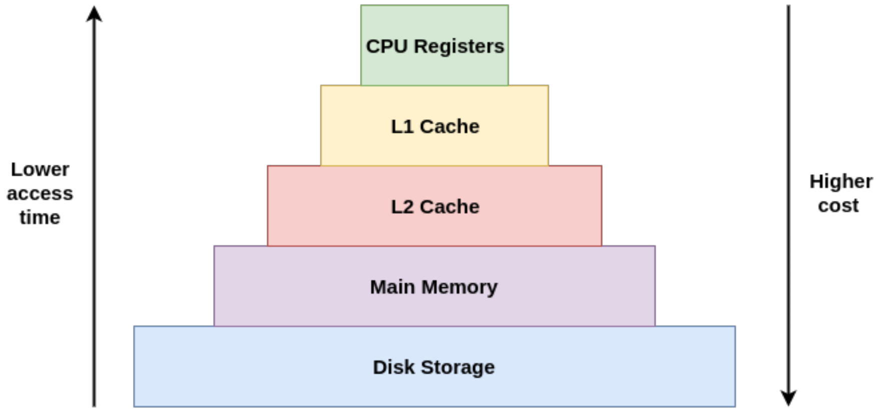


Figure 2.6: Illustration of the memory hierarchy with 2 cache levels. From the top to the bottom, the memory technology enlarges and gets slower.

2.5.1 Definition

A part of the memory hierarchy are the caches, a small quantity of fast (but expensive) memory that hides the latency of large and slow (but cheap) main memory, by buffering frequently used data [19].

It lays in the lowest layer of the memory hierarchy, being the first memory to be accessed in all memory requests. Its effectiveness relies critically on the *hit rate* (i.e., the fraction of requests that are satisfied from the cache). Due to the large difference in latency, a small decrease in *hit rate* leads to a much larger decrease in performance [1].

2.5.2 Cache Lines or Blocks

Caches are divided into lines. A cache line holds one aligned, power-of-two-sized block of adjacent bytes loaded from memory. If any byte needs to be replaced (i.e., evicted to make room for another), the entire line is reloaded. It is the minimum unit that may be cached [1].

On increasing the cache line size there is an increase in cache hits up to a certain extent. This results in performance enhancement based on the principle of locality [1]. However increasing block size beyond a certain point can also have a performance penalty because it will reduce the number of blocks which can fit in the cache, and whenever the block needs to be replaced, it will take much more time than a reduced block size. Not only this, each additional word will be less local and hence will be less used. Cumulatively, the probability of using newly fetched information will become less than the probability of reusing replaced.

[1] Principle of locality: principle for which caches rule, that states that a computer program tends to access same set of memory locations for a particular period

2.5.3 Cache Associativity

The associativity of the cache is another major property that must be taken into account when performing an attack as it can alter the way the address is divided (i.e., to appoint which address belongs to which line or set) or induct how many addresses must be accessed to completely evict a set. It refers to the way that each line can be replaced, if it is confined to a sole line (i.e., direct-mapped), if it can be replaced within a set (i.e., set-associative) or within any location (i.e., full associative). This choice of cache design can be seen as a tradeoff between complexity (and hence speed), and the rate of conflict misses [1].

2.5.3.1 Fully Associative

Ideally, any memory location could be placed in any cache line, thus the cache would always be used to its full capacity (i.e., misses occur only when there is no free space in the cache). However, this cache architecture requires that all lines are matched in parallel to check for a hit, which increases complexity and energy consumption, limiting speed. Such designs are therefore limited to small and local caches, such as TLBs [1].

2.5.3.2 Direct Mapped

The direct-mapped cache design is the opposite of the fully associative one. In this architecture, each memory location can be held by exactly one cache line, determined by the cache index function. If two memory locations that map to the same line are accessed, it will result in an eviction of the first one even though the cache may have unused lines, being the increase of miss rate the main disadvantage of this kind of design [1].

2.5.3.3 Set Associative

In this design, the cache is divided into small sets (usually of between 2 and 24 lines), within which addresses are matched in parallel, as for a fully-associative cache. The calculation of a set that an address maps to is given by the function of its address (just like the index is given in a direct-mapped design).

A cache with *N*-line sets is called *N*-way associative. Using this terminology, we can refer to direct-mapped and fully-associative caches, as 1-way and *N*-way associativity, respectively (where *N* is the number of lines in the cache).

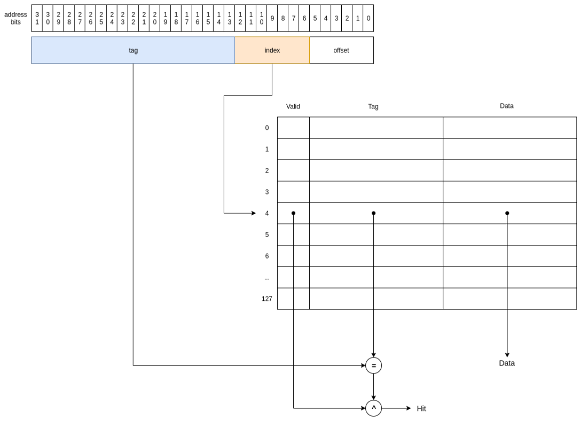


Figure 2.7: Direct mapped cache.

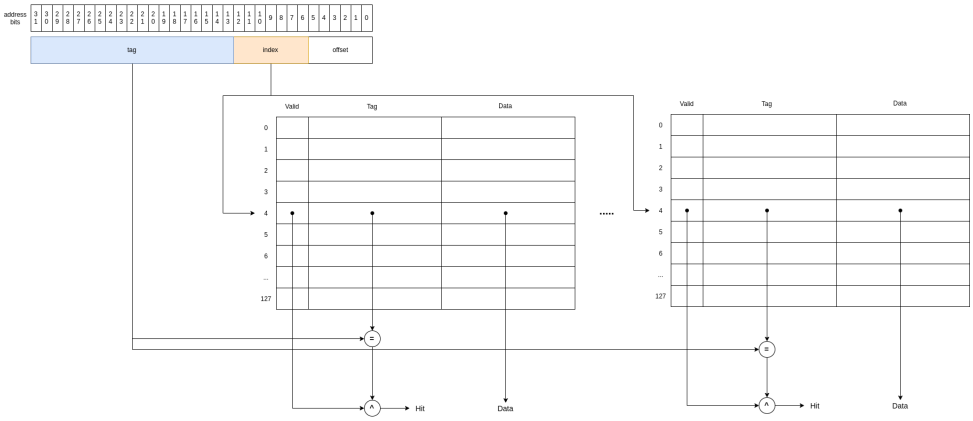


Figure 2.8: Set associative cache.

For both the direct-mapped and set-associative caches, the predictable map from address to line is exploited: (i) in attacks, to infer cache sets used by an algorithm under attack, and (ii) in cache coloring, to ensure that an attacker and its potential victim never share sets, and thus cannot conflict [1].

2.5.4 Cache Replacement Policies

When the cache is full and a cache miss occurs, buffered code and data must be evicted from the cache to make room for new cache entries. The heuristic that is used to decide which entry to evict is called replacement policy. The replacement policy has to decide which existing cache entry is least likely to be used in the near future, being implemented in hardware and not controlled by software [28].

There are numerous ways of predicting future cache accesses to optimize the cache resource contention. The most known replacement policies are listed as follows:

1. **Least Recently Used Replacement Policy** - The least recently used cache entry will be replaced.

2. **Round Robin Replacement Policy** - The replacement occurs starting from the first cache entry until the last one, independent of the temporal locality of the cache entries.

3. **Pseudo-random Replacement Policy** - A random cache entry will be selected and evicted based on a pseudo-random number generator.

There is a wide consensus that deterministic replacement policies like LRU or FIFO are to be preferred in the context of real-time systems. Nonetheless, increasingly more vendors implement random replacement policies to favor applications targeting average performance [32].

2.5.5 Cache Inclusion Policies

The inclusiveness property of a cache determines if all cache lines from the lower-level cache are also stored in the higher-level cache [17].

The term is used to describe the higher-level cache with regard to the lower-level cache, and the higher-level cache can be classified as follows:

1. **Inclusive Cache** - The higher-level cache contains all the data that is stored on the lower-level cache. This means a superset of the lower-level cache.

2. **Exclusive Cache** - The contents of the higher-level cache are not stored on the lower-level cache and vice versa. This architecture has the advantage of storing much more data in the cache subsystem than the inclusive one.

3. **Non-inclusive Cache** - Both higher-level and lower-level cache can store at the same time the same content (just like an inclusive cache); however, the data can be evicted from the higher-level cache and still reside in the lower-level cache (like an exclusive cache). Hence, accesses to L1 cache cannot be detected by monitoring the LLC.

If a word is read from the cache, the data in the cache will be identical to the one in the main memory. However, when the core executes a store instruction, a cache lookup to the address that is written to is performed. If a cache hit occurs, there are two possible policies:

1. **Write-back Policy** - Writes are performed on the cache and not to the main memory, making the content of the cache lines diverge on both. To mark the cache lines with the most recent data, an

associated dirty bit is used. If the bit is set, it means that the write updated the cache and not the main memory. If the replacement policy evicts a cache line where the dirty bit is set, the cache line is written out to the main memory.

2. **Write-through Policy** - Writes are performed to both the cache and the main memory which means that they are kept coherent. Since there are more writes to the main memory, this policy is slower than the write-back policy.

2.5.6 Cache Indexing

The cache can derive the index of a certain memory from its virtual or physical address, making the cache virtually indexed or physically indexed, respectively.

The multiple cache indexing possibilities are listed as follows:

1. **Virtually Indexed, Virtually Tagged (VIVT)** - The virtual address is used for both, the index and the tag. This method is faster in general because the caches do not require virtual to physical address translation before the cache lookup [28][33][34]. However, this can lead to a problem of redundancy in the cache in which the same physical address, that has been accessed by different cores (each one with their virtual address that maps to same physical address), is cached in different cache lines, reducing the performance.

2. **Physically Indexed, Physically Tagged (PIPT)** - The physical address is used for both, the index and the tag. This method is slower since the virtual address has to be looked up in the TLB. However, shared memory is only held once in the cache[35].

3. **Virtually Indexed, Physically Tagged (VIPT)** - The virtual address is used for the index, and the physical address is used for the tag. The advantage of this combination compared to PIPT is the lower latency since the index can be looked up in parallel to the TLB translation. However, the tag can not be compared until the physical address is available.

4. **Physically Indexed, Virtually Tagged (PIVT)** - The physical address is used for the index, and the virtual address is used for the tag. This combination has no benefit since the address needs to be translated, the virtual tag is not unique and shared memory still can be held more than once in the cache.

2.5.7 Cache Coherence

In modern systems where each core has its local cache and shared memory is implemented, it is common for the applications to run simultaneously on different cores working on the same memory, urging the need to certify that multiple cached copies of data that reside in different cores are updated according to the main memory. This leads to the emergence of cache coherence protocols.

The coherency can be easily jeopardized in systems where the memory is shared, as it is illustrated in Figure 2.9. In the first step, the core 0 accesses data *x*. Afterward, in step 2, the core 1 accesses the same data *x*. In step 3, core 0 sets *x* to a new value, which is *3* in the illustrated example. Then, in step 4, core 0 re-accesses the value of *x*, reading an obsolete value concerning the other core. This problem is extendable to DMA devices for example. Therefore, it is needed a coherence mechanism which can act in these situations.

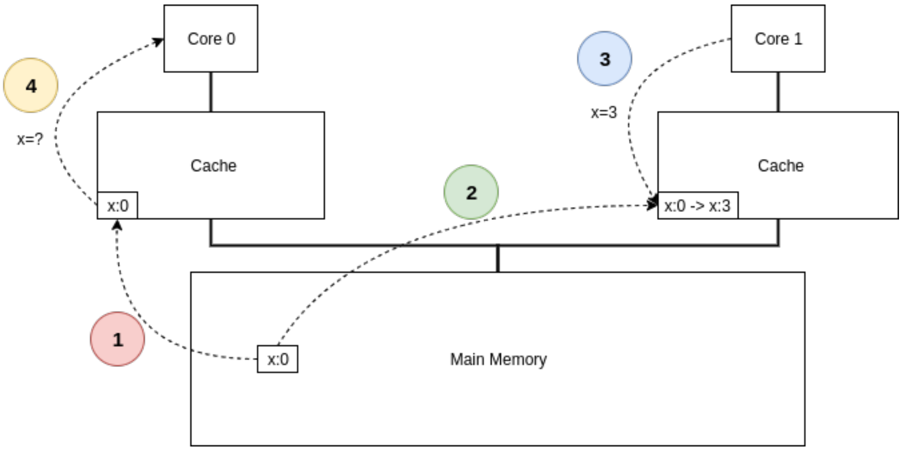


Figure 2.9: Illustration of a cache coherence problem.

There are three known mechanisms that tackle these coherency problems:

1. **Caching Disable** - An effective way of avoiding coherency problems is to disable the caching of memory. This is the costliest mechanism in terms of performance and power as all of the cache advantages are gone.

2. **Software-managed Coherency** - The software solution acts by cleaning dirty data and invalidating obsolete data to enable sharing with other processes each time a read or write of shared data is required.

3. **Hardware-managed Coherency** - The hardware solution is the most effective, and universal, as it enables coherency to software applications transparently. The most known hardware

implementation of a coherency keeper is the *Bus Snooping*. In this mechanism, the cache controller snoops on the bus, thereby monitoring occurring transactions and taking action if such transaction pertains to it (e.g., a write to a memory block of which it has its copy in the cache). The coherence is managed at the granularity of a cache line and, thus, either an entire cache line is valid or invalid.

2.6 Side and Covert Channels

In this chapter, both side and covert channels are presented, along with their respective countermeasures. Moreover, the known benchmarks used to evaluate the attacks are listed.

2.6.1 Definition

Heiser et al. [1] differentiated side channels from covert channels by reporting that ”side channels refer to the accidental leakage of sensitive data (e.g., an encryption key) by a trusted party, while covert channels are those exploited by a victim to deliberately leak information.”

Hence, side and covert channel attacks exploit a device’s hardware characteristics leakage (e.g., power dissipation, computation time or electromagnetic emission) to extract information about the processed data and, if used along statistical computations [36], use the data to deduce sensitive information (e.g., cryptographic keys, messages).

To observe both side or covert channels, not only a communication medium (i.e., shared hardware) is required but also an exploitation technique (e.g., *Prime+Probe*) to leverage the hardware’s contention [9].

2.6.2 Benchmarks

To measure how much information each channel can leak, there have been some terms used to describe the amount of transmitted information through a channel. This section lists the most commonly used terms, and reflects on how the benchmarks can be affected by the cache’s properties.

2.6.2.1 Channel Capacity or Bandwidth

The most known way of characterizing a channel is named capacity or bandwidth of a channel, and it represents, in bits per second, how much information can be transmitted through that channel. The more capacity a channel has, the more threatening it becomes to the targeted system [37].

Regarding the capacity of leaking information, it is known that collusion allows better utilization of the underlying hardware mechanism and hence covert channels tend to have much higher bandwidth than side channels based on the same mechanism (i.e., exploitation technique). The capacity of the covert channel is the upper bound of the corresponding side channel capacity [9].

A preponderant factor when obtaining the channel's capacity is the probing resolution of the LLC, which is not tied to the victim preemption, but is fundamentally limited only by the speed at which the attacker can perform the probe, in case of a *Prime+Probe* attack, referred in Section 2.6.3. This is much slower than for a local cache, for two reasons:

1. The LLC typically has higher associativity than the L1 cache (e.g., 12 to 24-way versus 4 to 8-way), hence more memory accesses are required to completely prime or probe a cache set [17].

2. The probe time increases due to the long access latency of the LLC (12 cycles more for *Cortex-A53* processor [38]). Even with all lines resident in the LLC, the attacker, when performing a probe of one LLC set, will still experience misses in the L1 and L2 caches, due to their lower associativity. Furthermore, a miss in the LLC will cause more than 150 cycles latency while a miss in the L1 or L2 cache has a latency of fewer than 40 cycles [17].

The slower the probing resolution, the less frequent will the cache's contents be observed, hence the channel capacity will decrease drastically.

2.6.2.2 Channel Matrix

A more recent way of visualizing the channels was introduced by Heiser et al. [9], which specifies the conditional probability of an observed output symbol (e.g. spy, or attacker, probing time) given an input symbol (e.g. cache lines accessed), by the use of a heat map, or a graph when visualizing low-capacity channels.

An example of a heat map that demonstrates the existence of a well-defined channel (i.e., noticeable horizontal variation) is illustrated in Figure 2.10, which has been used to characterize a channel on the instruction side of an L1 cache.

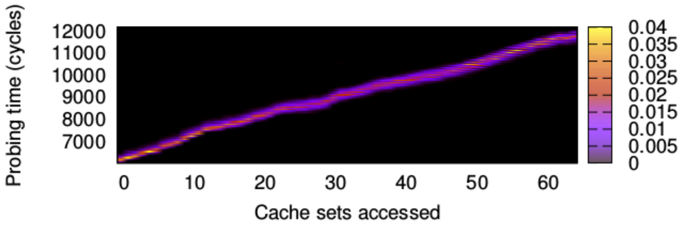


Figure 2.10: Channel matrix for the unmitigated L1 I-cache channel on Sandy Bridge platform [10].

In the absence of a channel, outputs are independent of inputs and the graph will show no horizontal variation, as it is illustrated in Figure 2.11.

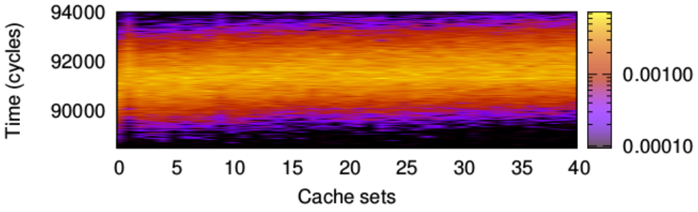


Figure 2.11: Channel matrix for the mitigated L1 I-cache channels on Hikey platform [10].

2.6.3 Attack Types

As mentioned in the definition of a side or covert channel, an attack needs a mechanism that explores a communication medium. The classification of the attacks is done taking into account which communication medium is leveraged: a) attacks that explore the cache, and b) attacks that explore real-time contention (e.g., buses or other resources present in the resource sharing hierarchy, described in Section 3.1.2).

2.6.3.1 Cache Exploitation Techniques

Prime+Probe

In the Prime+Probe technique, the attacker primes the cache by filling cache sets with its own data, then waits for the victim to replace some of the cache lines based on the input symbol it transmits. Lastly, the attacker probes the cache sets by measuring the access time to the previously cached data, thus measuring the victim's cache footprint. The output symbol is the total probing time of the attacker [10].

Although this technique doesn't rely on sharing memory, it can have some handicaps, such as lower resolution in comparison to other techniques because it can only target a cache set, and, due to the pseudo-random replacement policy, it might happen that the access to one congruent address evicts a previously accessed address from the attacker and thus it is possible that during the probing phase false positives occur [28].

This technique is privileged in cases where there is a resource-sharing case but without memory sharing between victim and attacker.

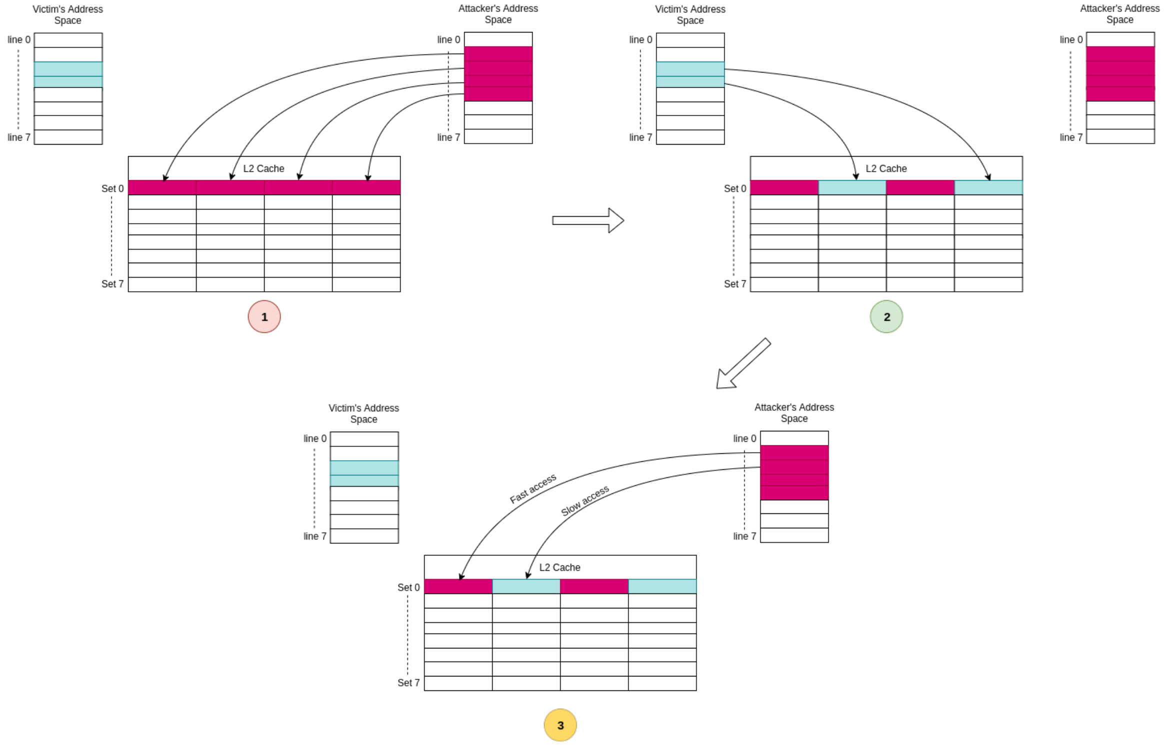


Figure 2.12: Illustration of the Prime+Probe attack by means of a 4-way (columns) cache with 8 sets (rows). In step 1, the attacker primes the cache filling all the ways from the same set. In step 2, the victim replaces some of the cache lines from the cache set. In step 3, the attacker probes the cache set accessing all cache lines. The cache lines changed by the victim have a slow access time, while the unchanged cache lines have a faster access time.

Flush+Reload

In Flush+Reload, the attacker first flushes a shared line of interest (by using dedicated instructions or by eviction through contention). Once the victim has executed, the attacker then reloads the evicted line by touching it, measuring the time taken. A fast reload indicates that the victim touched this line (reloading it), while a slow reload indicates that it didn't. The advantage of Flush+Reload over Prime+Probe is that the attacker can target a specific line, rather than just a cache set [1].

This technique relies on the existence of shared virtual memory (e.g., shared libraries or page deduplication), and the ability to flush by virtual address, bringing much more resolution (hence leakage capacity) to the attack as it can target cache lines instead of cache sets.

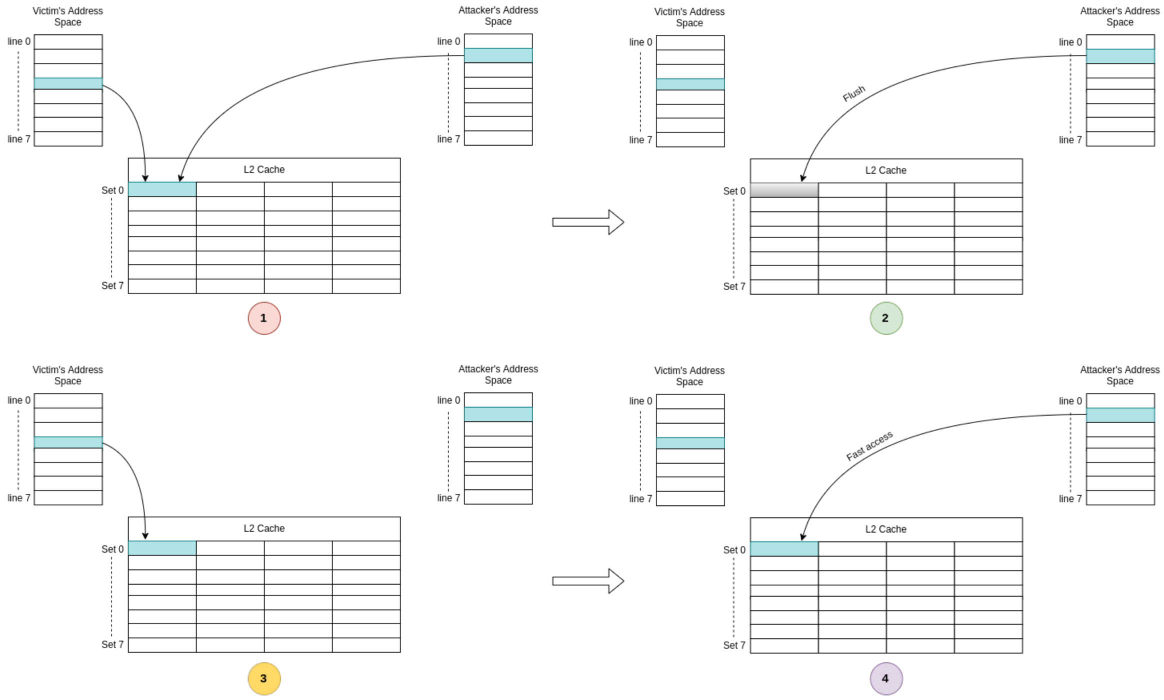


Figure 2.13: Illustration of the Flush+Reload attack by means of a 4-way (columns) cache with 8 sets (rows). In step 1, the attacker or the victim cache a shared line. In step 2, the attacker flushes the shared cache line. In step 3, the victim accesses the previously accessed cache line. In step 4, the attacker tries to reload the cache line. If the cache has been accessed by the victim, the attacker will experience a fast access time.

Evict+Time

This approach uses the targeted eviction of lines, together with overall execution time measurement. The attacker first causes the victim to run, preloading its working set, and establishing a baseline execution time. The attacker then evicts a line of interest and runs the victim again. A variation in execution time indicates that the line of interest was accessed [1].

Flush+Flush

The attack is basically the same as Flush+Reload. A binary or a shared object file is mapped into the address space of the attacker. An address is flushed from the cache, and the victim's program is scheduled. However, instead of the reloading step where the monitored address is accessed, it is flushed again causing no cache misses compared to Flush+Reload or Prime+Probe [28].

2.6.3.2 Exploiting real-time contention

These real-time attacks, also known as DoS attacks, target the systems by maliciously increasing the consumption of a shared resource. The attacks can occur wherever exists resource contention, including buses, by exhausting shared bus bandwidth with a large number of memory requests, as it was exemplified by Woo et al. [39], who on a machine with a single frontside bus, generated a DoS attack with L2 cache misses in a simulated environment.

2.6.4 Countermeasures

All of the published countermeasures, listed by a survey made by Heiser et al. [1], are summarized in this section.

2.6.4.1 Constant-time techniques

The most widely used countermeasure, primarily applied to microarchitectural attacks, is making the security operation time constant or random, regardless of the microarchitecture elements that are used. This approach comes with the disadvantage of deteriorating the performance of the system [36].

There are rudimentary techniques where the access to, presumed undetectable microarchitecture elements, depends on secret information. An example is the implementation of modular exponentiation in OpenSSL, which can access different memory addresses within a cache line, depending on the secret exponent. This countermeasure can be surpassed in processors that can leak information within a cache line (i.e., offset of a cache line), as it has been demonstrated by Bernstein et al. [40].

These techniques have been applied in debuggers and compilers. It is the case of *Valgrind* debugger, where Langley [41] modified it to trace the flow of secret information and warn if it is used in branches or as a memory index.

2.6.4.2 Injecting noise

This countermeasure focuses on introducing noise to the attacker's measurements making them essentially useless [1]. As it occurs with the constant-time techniques, this technique has the disadvantage of deteriorating the performance of the system.

Zhang et al. [42] introduced a bystander VM for injecting noise on the cross-VM L2 covert channel with a configurable workload. They found that as long as the bystander VMs only adjust their CPU time consumption, working sets and memory access rates, they impact the cross-VM covert channel's bandwidth.

Although noise injection can difficult the leak of information, it is inefficient for obtaining high security, because the amount of actual required noise increases dramatically with decreasing channel capacity.

This significantly degrades system performance, and makes it infeasible to reduce channel bandwidth by more than about two orders of magnitude [43].

2.6.4.3 Partitioning time

This countermeasure eliminates attacks which rely on either concurrent or consecutive access to shared hardware by either providing time-sliced exclusive access, or carefully managing the transition between time-slices (e.g., flushing the caches) [42]. Among the techniques that partition time to eliminate the attacks, the cache flushing and the kernel address space isolation techniques stand out.

Cache flushing

Zhang et al. [44] suggested flushing all local state, including BTB and TLB, and all levels of caches during VM switches in cloud computing when CPU switches security domains.

The lower and larger the cache level, the bigger the degradation of performance as the time to refill the caches is bigger, and the likelier a newly scheduled VM finds any data or instructions hot in the cache [45].

Kernel address space isolation

Gruss et al. [46] proposed isolating kernel form user address space by using separated page directories for each, so switching context between user and kernel spaces includes switching the page directory. This technique is designed to mitigate the timing attack on prefetch instructions.

2.6.4.4 Partitioning hardware

The partitioning hardware mechanism is only used in truly concurrent attacks that can only be prevented by partitioning hardware resources among competing threads or cores [1].

Cache Coloring

The cache coloring approach exploits set-associativity to partition caches in software. Besides improving determinism and predictability in detriment of average-case performance [32], it can be used to protect against cache timing channels.

Cache coloring implementations divide memory into colored memory pools and allocate memory from different pools to isolated security domains. As an example, Figure 2.14 shows how the same physical address (PA) is interpreted from 4 different points of view. At the top, the bits of a PA are seen from the perspective of an OS/Hypervisor as divided into physical frame number (PFN) and page offset (PO) bits. At the lower layer, the structure of the same PA from the point of view of the L1 and L2 caches is depicted, highlighting its color bits. Physical frames whose addresses diverge in any of these color bits are not mapped to the same cache set, and thus never conflict.

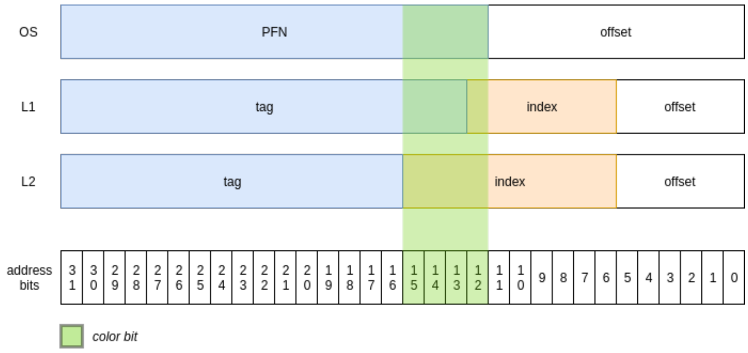


Figure 2.14: Disposition of colour bits in a 32-bits PA address from the point of view of an OS, L1 and L2 (PIPT) caches.

3. Analysis

The analysis begins by defining the experimental system and which processor is being targeted. Then, based on this dissertation's context, the study is done upon three main factors that determine the exploitation technique to be used, composing the attack's strategy.

After defining the attacker's strategy, all the challenges necessary to conduct the exploitation technique are addressed, and it is presented to each one of them, several solutions.

3.1 Attack Strategy

To know which kind of attack is the most effective, a whole attack's strategy needs to be defined. The most preponderant attributes for choosing the appropriate exploitation technique refer to (i) the concurrency level, (ii) the resource sharing level, and (iii) the memory sharing level.

3.1.1 Experimental System Definition

The targeted CPU within the *Ultrascale+ MPSoC* is the APU, a *Cortex-A53 MPCore* with 4 cores. Each core has a private L1 instruction cache and L1 data cache, both connected through a shared L2 cache inclusive on the instruction side and non-inclusive on the data side. Within the APU, three of the cores will host the non-critical OS (i.e., Linux), while the other core will be responsible for serving the critical tasks (i.e., ErikaRTOS).

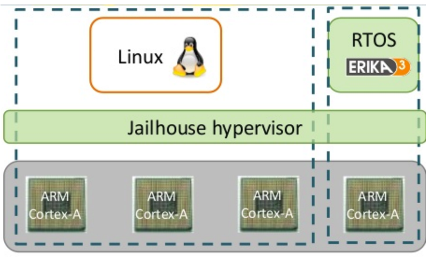


Figure 3.1: Mixed-criticality system using Jailhouse as hypervisor. The first 3 cores are assigned to Linux, and the other core is within Erika RTOS's domain.

3.1.2 Resources Sharing Level Definition

The extension at which the contended resources are shared defines their sharing degree. The lowest layer (i.e., interconnect) is shared between all processes in the system while the highest layer contains only thread shared resources in the same core (e.g., BTB, pipelines and functional units). The intermediate layers can be shared between packages (e.g., LLC) or cores (e.g., private caches). Those at higher levels tend to achieve higher severity by exploiting the higher-precision information available, while those at lower levels (e.g. the interconnect) tend to have lesser precision, being mostly DoS. Simultaneously, the lower the layer, the easier to protect against attacks.

In this case, the resource sharing level at which will the attack occur will be the package-shared one, as the proposed problem (i.e., extract information between different cores on the same chip) requires a cross-core attack.

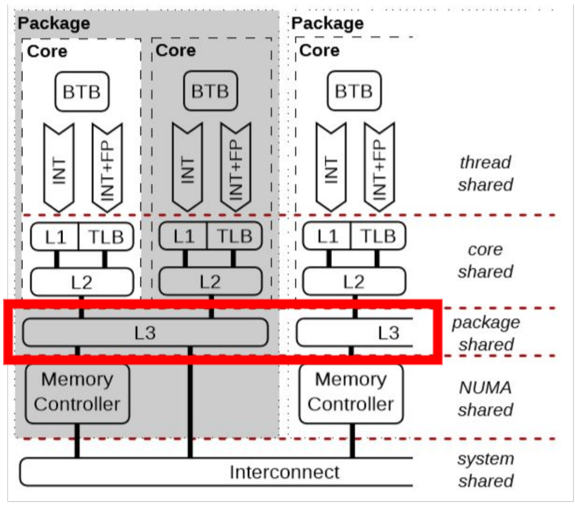


Figure 3.2: Contended resources in a hierarchical multicore system with 3 cache levels [1]. Within the red rectangle, it is represented the resource sharing level intended to be studied and replicated.

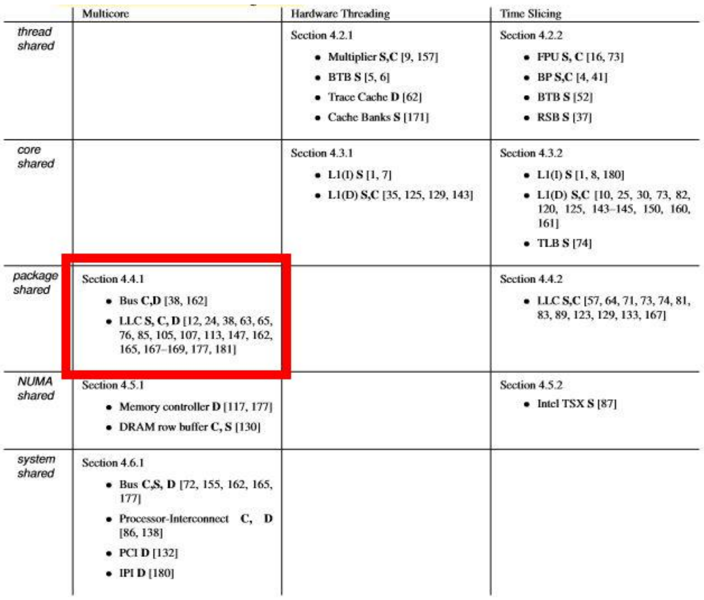
Due to the hypervisor's static partition hardware policy (i.e., AMP), there will be no memory shared between cores. As such, the attacker cannot rely on memory sharing techniques such as page deduplication (e.g., used in *Flush+Reload* or *Flush+Flush* exploitation techniques) to launch the attack.

3.1.3 Temporal Concurrency Level Definition

The degree of temporal concurrency at which will the attack occur varies with the level where the resource contention will be exploited. These degrees can be classified as (i) time-sliced execution on a single core, (ii) hardware threading (e.g. SMT), and (iii) full concurrency (i.e. multicore).

The attacks that occur at the lowest and highest sharing level require true concurrency, as the buses or the low-level components (e.g., L1, TLB) have relatively small states that are rapidly overwritten.

The chosen exploited resource (i.e., LLC), has a much larger persistent state than most of the resources on the other levels [1]. So, the attack can be exploited at a relatively coarse granularity, meaning that either time-sliced concurrency within a core, or concurrency between cores, are enough to deploy an attack on LLC, as it can be verified in Figure 3.3. Since the purpose of this dissertation is to deploy cross-core attacks, the attack will occur in a full concurrency context (i.e., multicore).



	Multicore	Hardware Threading	Time Slicing
thread shared		Section 4.2.1 • Multiplier **S,C** [9, 157] • BTB **S** [5, 6] • Trace Cache **D** [62] • Cache Banks **S** [171]	Section 4.2.2 • FPU **S, C** [16, 73] • BP **S,C** [4, 41] • BTB **S** [52] • RSB **S** [37]
core shared		Section 4.3.1 • L1(I) **S** [1, 7] • L1(D) **S,C** [35, 125, 129, 143]	Section 4.3.2 • L1(I) **S** [1, 8, 180] • L1(D) **S,C** [10, 25, 30, 73, 82, 120, 125, 143–145, 150, 160, 161] • TLB **S** [74]
package shared	Section 4.4.1 • Bus **C,D** [38, 162] • LLC **S, C, D** [12, 24, 38, 63, 65, 76, 85, 105, 107, 113, 147, 162, 165, 167–169, 177, 181]		Section 4.4.2 • LLC **S,C** [57, 64, 71, 73, 74, 81, 83, 89, 123, 129, 133, 167]
NUMA shared	Section 4.5.1 • Memory controller **D** [117, 177] • DRAM row buffer **C, S** [130]		Section 4.5.2 • Intel TSX **S** [87]
system shared	Section 4.6.1 • Bus **C,S, D** [72, 155, 162, 165, 177] • Processor-Interconnect **C, D** [86, 138] • PCI **D** [132] • IPI **D** [180]		

Figure 3.3: Table with known microarchitectural timing attacks. The horizontal axis represents the temporal concurrency level and the vertical axis represents resources sharing level [1]. Within the red rectangle, it is represented the type of attacks intended to be studied and replicated.

3.1.4 Exploitation Technique Definition

Based on the previously listed attributes, the exploitation technique that fulfills the requirements demanded by the system is a *Prime+Probe* attack on the *Ultrascale+*'s LLC (i.e., L2). The choice of the *Prime+Probe* technique is sustained by the fact that the technique doesn't rely on shared memory to successfully, find the co-located VM's cache activity at a cache set granularity.

3.2 Attack Challenges

Regarding the *Prime+Probe* attack, there are some challenges that need to be overcome to do an efficient attack. These challenges can be divided into two major actions that need to be undertaken:

1. Efficient eviction (i.e., how to construct an efficient eviction set during the prime step).

2. Precise timing (i.e., how to time correctly the probe step).

Each step of the *Prime+Probe* attack (i.e., prime step and probe step), has its own set of concerns that can range from the underlying properties of the cache (e.g., inclusiveness, replacement policy) to privileged accesses of Linux registers and files (e.g., */proc/pagemap* access, dedicated performance counter registers).

3.2.1 Prime Step Challenges

The prime step challenges can be synthesized as to (i) how to construct an eviction set, and (ii) how to evict efficiently a targeted LLC set by accessing cache lines. The concerns to take into account are as follows:

3.2.1.1 Shared LLC inclusiveness (L1-I inclusive, L1-D non-inclusive)

The inclusiveness of the LLC is exploited to evict lines that belong to the victim's private cache. By evicting lines from the LLC, the attacker can evict lines from the instruction side of L1, forcing this way that the following accessed cache lines are stored in LLC and doesn't occur any cache hit in L1. The inclusiveness can be exploited in two different ways:

1. **Direct Eviction -** This kind of eviction occurs by accessing the inclusive side of the local cache. The steps to evict directly an instruction from the adjacent core's local cache are as follows:

(a) Step 1: an instruction is allocated to the last-level cache and the instruction cache of one core.

(b) Step 2: a process accesses data or instruction that maps to the same cache set as the targeted cache set.

(c) Step 3: the process evicts the instructions from other core's instruction caches as well.

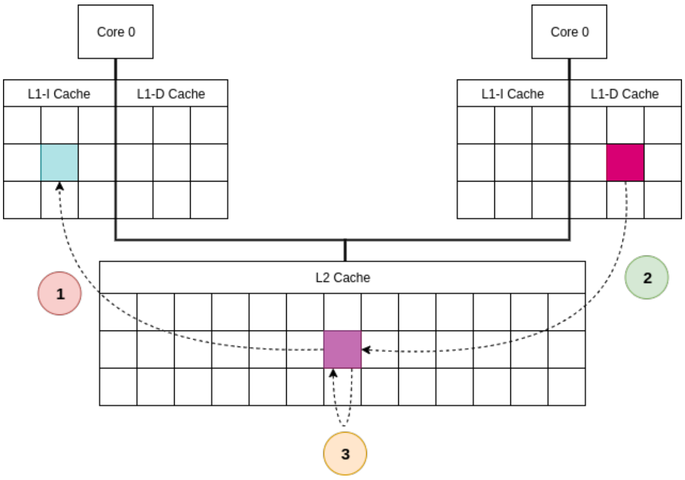


Figure 3.4: Cross-core instruction cache eviction through data accesses on a instruction- inclusive, data-non-inclusive cache.

2. **Indirect Eviction -** This kind of eviction occurs by accessing the non-inclusive side of the local cache. The steps to evict indirectly an instruction from the adjacent core's local cache are as follows:

   (a) Step 1: an instruction is allocated to the last-level cache and the instruction cache of one core.

   (b) Step 2: a process fills its core's data cache, thereby evicting cache lines into the last level cache.

   (c) Step 3: the process evicts the instructions from other core's instruction caches as well.

In the experimental setup that is intended to be used, both the attacker and the victim agree on a cache set to contend (explained further in Chapter 4). So, there is no need to pollute the L2 cache to evict lines from local caches. Therefore, the chosen method to evict the targeted cache set is the direct eviction way.

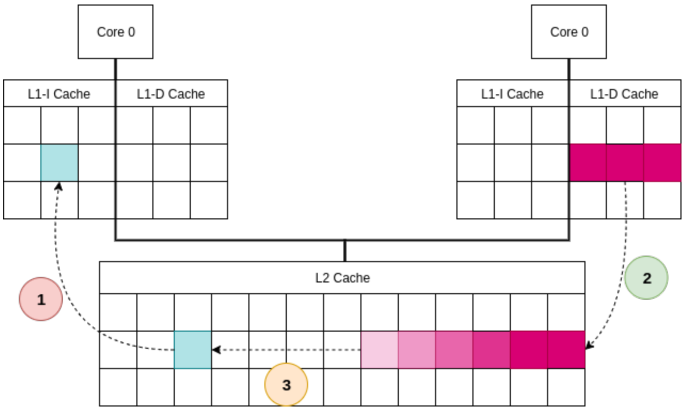


Figure 3.5: Cross-core instruction cache eviction through data accesses on a instruction- inclusive, data-non-inclusive cache.

Autolock Mechanism

There is an SoC-specific feature that can prevent the eviction from higher-level caches through lower-level caches. This feature is called AutoLock [47] and implements an indicator storage element that tracks which lines are stored in higher levels. The element can be realized with a set of indicators or a tag directory. If an indicator is set, the corresponding line is protected. This mechanism, therefore, prevents said performance penalties, because subsequent evictions in higher cache levels are prohibited.

3.2.1.2 Replacement policy (LRU policy on L1, pseudo-random policy on LLC)

During the prime step, the cache replacement policy plays an important role in the eviction efficiency of cache lines. During a *Prime+Probe* attack, when priming the set using an eviction set, due to the pseudo-random replacement policy, some of the attacker's addresses might be replaced with other attacker's addresses instead of replacing victim's addresses, causing a self eviction effect, which is named *cache trashing*. This problem is solved by using eviction strategies.

Eviction Strategy

The aforementioned attack techniques (see Section 2.6.3) like *Flush+Reload* and *Flush+Flush* use the unprivileged x86's flush instruction *clflush* to evict a cache line. But except for ARMv8-A CPUs, ARM processors do not have an unprivileged flush instruction, and therefore cache eviction must be used employing eviction strategies.

An eviction strategy accesses addresses from an eviction set in a specific access pattern and can ideally be used as a replacement for a flush instruction. The access pattern defines in which order addresses from the eviction set (i.e., a set of congruent addresses[1]) are accessed, including multiple accesses per

[1] Congruent Addresses: Addresses that map to the same cache set.

address.

The success of a cache eviction strategy is measured by testing whether the targeted memory address is not cached anymore over many experiments (i.e., average success rate). The three attributes that weigh the most in the success of the eviction of a set were enumerated by Maurice et al. [48] as: (i) the eviction set size (N), (ii) the number of different memory addresses (D), and (iii) the number of repeated accesses to the same address (A).

The eviction set size matters within the eviction strategy context because cache hits and cache misses only have impact to addresses that map to the same cache set. The number of different memory addresses that are accessed in a loop and the number of accesses to the same address also contribute to the effectiveness of the eviction strategy. It might occur that some replacement policies prefer to evict recently added cache lines over older ones, thus the repeated accesses are necessary to keep the lines in the cache.

Taking into account the most important attributes, the eviction strategy, also named by Irazoqui et al.[47] as sliding window eviction, can be described (in Algorithm 1) as a loop over an eviction set of size N, where only a subset of D addresses is accessed per round. A parameter A allows to make accesses overlap for repeated accesses.

Algorithm 1: Eviction loop using eviction set with N congruent addresses.

```
for $i = 0; i < N-D; i++$ do
    for $j = 0; j < A; j++$ do
        for $k = 0; k < D; k++$ do
            access (i+k)th address of eviction set;
        end for
    end for
end for
```

In order to acquire high-frequency measurements, the eviction has to be consequently fast. To find fast eviction strategies, Lipp et al. [48] used *Rowhammer* attack techniques to evaluate different eviction strategies and suggested the following table (illustrated in Table 3.1) for the *Cortex-A53 MPCore*.

N	A	D	Cycles	Eviction rate
-	-	-	767	100,00%
23	2	5	6209	100,00%
23	4	6	16912	100,00%
22	1	6	5101	99,99%
21	1	6	4275	99,93%

Table 3.1: Different eviction strategies on *Cortex-A53 MPCore* [28]. The first strategy is used recurring to the x86's *clflush* instruction, hence the absence of parameters.

3.2.1.3 Understand hidden mappings

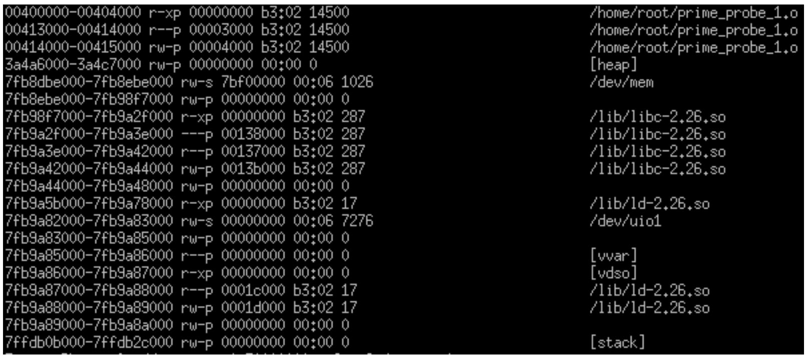
Another challenge that must be overcome is to know which physical addresses are being accessed while having only access to virtual addresses. If an eviction strategy is intended to be deployed, an eviction set must be constructed. Thus, to know which cache set a group of addresses map to, there are two ways: (i) know their physical address or, (ii) use huge pages.

1. **Find congruent addresses (privileged access) -** The first method, that requires privileged access to access the intended files, is to use operating-system services like */proc/<pid>/maps* or */proc/<pid>/pagemap*, to retrieve information on virtual and physical address mappings.

   The */proc/<pid>/maps* service offers information relative to regions of contiguous virtual memory, such as (i) the virtual addresses that bound the region in a process or thread, (ii) the permissions of the regions, and (iii) more importantly, the offset, if the region was mapped from a file, where the mapping begins(illustrated in Figure 3.6).

   On the other hand, */proc/<pid>/pagemap* offers information relative to virtual pages, letting a userspace process know, among other information, which physical frame each virtual page is mapped to. The page table, that has the virtual to guest's physical address translations, is accessible via this file.

   As the experimental setup comprises a hypervisor, an additional translation layer that separates the guest's physical address from the host's (i.e., hypervisor) physical address exists. In this case, the understanding of virtual to guest physical address translations isn't enough to address this problem (recall that the cache set index is derived directly from the physical address on ARM).



```
00400000-00404000 r-xp 00000000 b3:02 14500                               /home/root/prime_probe_1.o
00413000-00414000 r--p 00003000 b3:02 14500                               /home/root/prime_probe_1.o
00414000-00415000 rw-p 00004000 b3:02 14500                               /home/root/prime_probe_1.o
3a4a6000-3a4c7000 rw-p 00000000 00:00 0                                   [heap]
7fb8dbe000-7fb8ebe000 rw-s 7bf00000 00:06 1026                            /dev/mem
7fb8ebe000-7fb98f7000 rw-p 00000000 00:00 0
7fb98f7000-7fb9a2f000 r-xp 00000000 b3:02 287                             /lib/libc-2.26.so
7fb9a2f000-7fb9a3e000 ---p 00138000 b3:02 287                             /lib/libc-2.26.so
7fb9a3e000-7fb9a42000 r--p 00137000 b3:02 287                             /lib/libc-2.26.so
7fb9a42000-7fb9a44000 rw-p 0013b000 b3:02 287                             /lib/libc-2.26.so
7fb9a44000-7fb9a48000 rw-p 00000000 00:00 0
7fb9a5b000-7fb9a78000 r-xp 00000000 b3:02 17                              /lib/ld-2.26.so
7fb9a82000-7fb9a83000 rw-s 00000000 00:06 7276                            /dev/uio1
7fb9a83000-7fb9a85000 rw-p 00000000 00:00 0
7fb9a85000-7fb9a86000 r--p 00000000 00:00 0                               [vvar]
7fb9a86000-7fb9a87000 r-xp 00000000 00:00 0                               [vdso]
7fb9a87000-7fb9a88000 r--p 0001c000 b3:02 17                              /lib/ld-2.26.so
7fb9a88000-7fb9a89000 rw-p 0001d000 b3:02 17                              /lib/ld-2.26.so
7fb9a89000-7fb9a8a000 rw-p 00000000 00:00 0
7ffdb0b000-7ffdb2c000 rw-p 00000000 00:00 0                               [stack]
```

Figure 3.6: Example of */proc/self/maps* output when using a Linux application. The first column describes the starting and ending address of the contiguous region, the second column shows the permissions of the regions and the third column outputs the offset in the file where the mapping begins (if the memory was not mapped from a file, the value 0 is shown instead).

The second level translation problem is solved recurring to Jailhouse's configuration files, which allow to decide in which guest and host's physical addresses the memory regions reside, making it possible to establish a direct connection between the guest's physical address and the host's physical address. In Bao, this challenge is significantly more difficult to overcome, since there is no chance to assign physical addresses to memory regions. In this case, the only solution is to debug the hypervisor's code and find the translations. Once known the virtual to host's physical address translations, the virtual addresses can then be chosen to construct an eviction set based on their translation.

2. **Use huge pages (unprivileged access) -** The second method, that doesn't require any privileged access to any file, is to use huge TLB pages. The use of huge pages means the enlargement of virtual memory pages so that the page offset is large enough to cover all of the L2 set index bits.

   In the L2 cache, when using the default sized pages, the virtual-to-physical translation masks some of the bits that encode the cache set index. By not knowing the physical frame number, the attacker cannot determine the cache set that a memory address maps to.

   To resolve this uncertainty, huge pages (e.g., 2MB-sized pages) are used. Hence, when using huge pages, the L2 set index bits are preserved during virtual-to-physical address translation, allowing the attacker to determine the cache set index from the huge page offset.

   In *Cortex-A53 MPCore*'s case, if the default page size (i.e., 4KB-sized page) configuration is used, the L1 and L2 caches have their set index bits masked by the PFN. The L1 cache has the highest set index bit translated while the L2 cache has the set index bits [9:4] translated.

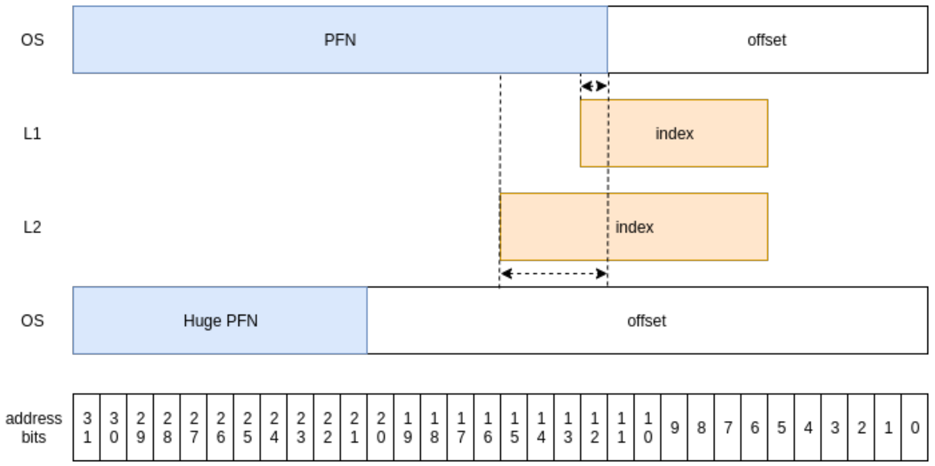


Figure 3.7: Effect of 2MB-sized huge pages on *Cortex-A53 MPCore*'s caches address translation. The 4KB-sized pages configuration (above) masks a portion of the caches set index bits, while the 2MB-sized pages configuration (below) doesn't mask any cache set index bit.

When the 2MB-sized pages are used, the page offset bits cover the set index bits of both L1 and L2 caches. Since the page offset bits aren't translated, the set index bits of L1 and L2 caches maintain their values during virtual-to-physical translation.

3.2.2 Probe Step Challenges

The main concern during the probe step is to time correctly the cache accesses. To have precise timing, the challenges to be taken into account described in the following sections.

3.2.2.1 Timing sources or dedicated performance counters to be used

There are 4 ways of obtaining an accurate timing of cache accesses that distinguishes cache hits from cache misses. They are listed below in descending order from timing accuracy:

1. **Performance counter registers (privileged access)** - ARMv8-A architectures provide one performance monitor register denoted as Performance Monitor Cycle Count Register (PMCCNTR) that counts processor cycles. While its measurements are fast and precise, the access to those performance counters is restricted to the kernel space by default. As root privileges are required to use these registers, this timing source is hardly accessible in serious attacks [28].
2. **perf syscall (unprivileged access)** - Linux kernel provides a powerful tool to instrument CPU performance counters and tracepoints [2], independently of the used hardware. The system call

[2] Tracepoints: instrumentation points placed at logical locations in code, such as for system calls, TCP/IP events, file system operations, etc.

perf_event_open is used to access such information from userspace; however, since this approach relies on a system call to acquire the cycle counter value, a latency overhead can be observed.

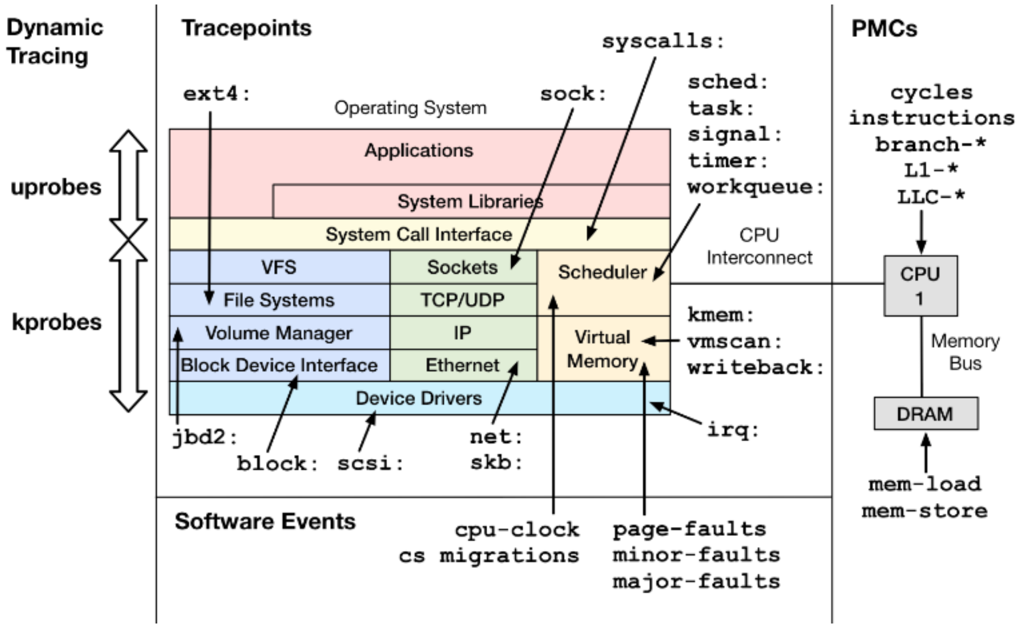


Figure 3.8: Map of Linux event sources used by perf tool. [49]

3. **POSIX function (unprivileged access)** - The *clock_gettime()* function retrieves the time of a clock that is passed as a parameter. Depending on the used clock, it allows obtaining timing information with a resolution in the range of microseconds to nanoseconds, which is still enough to distinguish cache hits and misses.

4. **Dedicated thread timer (unprivileged access)** - The least accurate method is done by an attacker which implements a thread running on a different core that increments a variable in a loop. The resolution of this threaded timing information is by far high enough to differentiate cache hits from cache misses.

3.2.2.2 Replacement policy (LRU policy on L1, pseudo-random policy on LLC)

During the probe step, the cache replacement policy plays an important role in the eviction efficiency of cache lines. During a *Prime+Probe* attack, when probing the targeted set, due to the pseudo-random replacement policy, the same *cache trashing* effect (occurred on prime step) can occur on the cache. This problem is solved by accessing in reverse order the cache lines from the set during the probe step and reducing the eviction set size [19].

3.2.2.3 Interaction with higher-level caches

Besides *cache trashing*, there is another noise variation associated with the memory hierarchy. Accessing memory from local caches is faster than reading from shared caches. So, local cache contents affect the cache probe time (i.e., the more the local cache contents have in common with the shared caches, the less will the probe time be) and introduce noise to its measurements. In *Cortex-A53 MPCore*, 4 accesses to the cache lines present in a L1-D's cache way have lower latency (i.e., 12 cycles for simple access via pointer) than 1 access to the L2 cache (i.e., 15 cycles). By measuring the total probe time, the attacker doesn't know if it accessed a L2 cache line or if it accessed 4 cache lines present in the local cache. This ambiguity in the probing time can infer multiple cases of the victim's cache activity to the attacker.

This interaction tends to have less effect when the associativity of lower-level caches is much higher than that of local cache since a local cache (e.g., L1) can only hold a small portion of the eviction set for the lower level cache (e.g., LLC). In *Cortex-A53 MPCore*, there can be a total variation of 48 cycles in the measured probing time. This variation is calculated by measuring the probing time difference between two extreme cases. The highest latency case is when all cache lines are present in L2 and no cache line is present in L1-D (i.e., 15x16 cycles). The lowest latency case is when 4 cache lines are present in L1-D and L2 (i.e., *Cortex-A53*'s L2 is non-inclusive on data-side), and the remaining 12 lines are present in L2 (i.e., 4x3 + 15x12 cycles).

3.2.2.4 Probing resolution

The probing resolution of the LLC is limited only by the speed at which the attacker can perform the probe. Probing an LLC is much slower than probing L1, which is tied to the reduced channel capacity of an LLC based covert channel. This is due to two main reasons:

1. The LLC typically has higher associativity than the L1 cache (e.g., 12 to 24-way versus 4 to 8-way), hence more memory accesses are required to completely prime or probe a cache set.
2. The probe time increases due to the long access latency of the LLC. To probe an LLC, the attacker has to experience cache misses on the higher-level caches.

4. Design

The last chapter allowed to define which exploitation technique is intended to be used based on three main factors: (i) the resource sharing level, (ii) the memory sharing level, and (iii) the temporal concurrency level. Once defined the exploitation technique, all the main challenges were addressed. Also, multiple alternatives to solve each challenge were presented, each one with its own advantages and disadvantages.

This chapter follows up the analysis by proposing algorithms that use the exploitation technique (see Section 2.6.3.1) to exploit the communication medium (i.e., L2), resulting in the observation of a channel. To observe channels, two algorithms were proposed and designed: (i) the first algorithm establishes the dependency of probing time with number of accessed lines, and (ii) the second algorithm, that leverages the principle established in the first algorithm to establish a communication between two partitioned cores.

4.1 Proposed Channels

After addressing the challenges of deploying an efficient *Prime+Probe* attack and defining the attack's strategy, it is possible to architect the channels which evidence the correlation between the victim accesses and the cache state. Notwithstanding, there is the need to recreate an idyllic environment where: (i) the attacker and the victim don't perform at the same time; (ii) the attacker and the victim can agree, beforehand, which addresses to access for the cache to be affected; and (iii) the attack can be done consecutively, without external interference, to get around the problem of the occurrence of false positives. By synchronizing both actors (much different from a realistic setting), the visualization of the channel gets much more evident, because there is a minimal cache interference between the probing and priming step, besides the interference of the victim.

4.1.1 Simple Channel

Osvik et al. [15] have shown that the time to access the addresses in the probing step is directly related to the number of ways that have been replaced by the victim. If the victim didn't replace any cache ways, the probe step will take minimal time. If the victim replaced more ways, the probe step will take longer. The algorithm to observe the channel is illustrated below in Algorithm 2.

Algorithm 2: Simple Prime+Probe attack

```
Input: Cache set s
Output: Probing time of cache set s
Prime: Occupy cache set s any data
Wait for victim to be scheduled
Probe: Re-access the data from the prime step in reverse order
if probe time < threshold then
    no victim access;
else
    victim access;
end if
```

4.1.1.1 Statechart Diagram

The behavior of the attacker and the victim during the attack execution are described by the following state machines, illustrated by Figure 4.6 and Figure 4.2, respectively.

1. **Attacker -** The states that the attacker takes throughout the attack, are solely focused on priming and probing the cache while the victim waits to access the cache. The attack starts when the attacker starts priming the cache and finishes when the prime and probe runs are done.

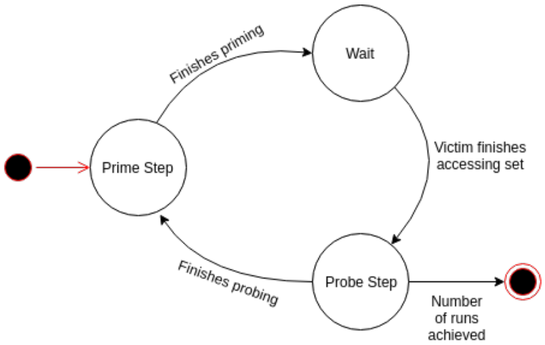


Figure 4.1: Attacker state machine during the attack that allows observing the Simple Channel.

2. **Victim -** On the other hand, the victim interleaves the execution time with the attacker, only accessing the cache when the attacker finishes priming the cache. As soon as the number of runs is achieved, it finishes accessing the lines and stops executing.

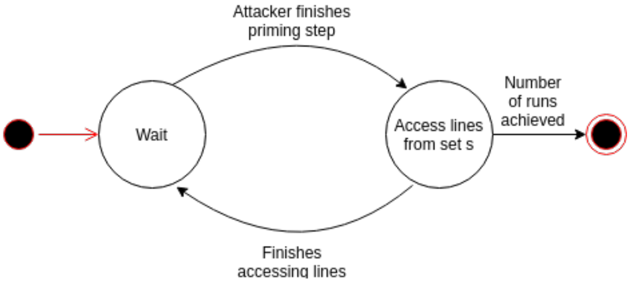


Figure 4.2: Victim state machine during the attack that allows to observe the Simple Channel.

4.1.1.2 Sequence Diagram

The sequence diagram that describes the synchronous events during one attack iteration of both actors is illustrated by Figure 4.3.

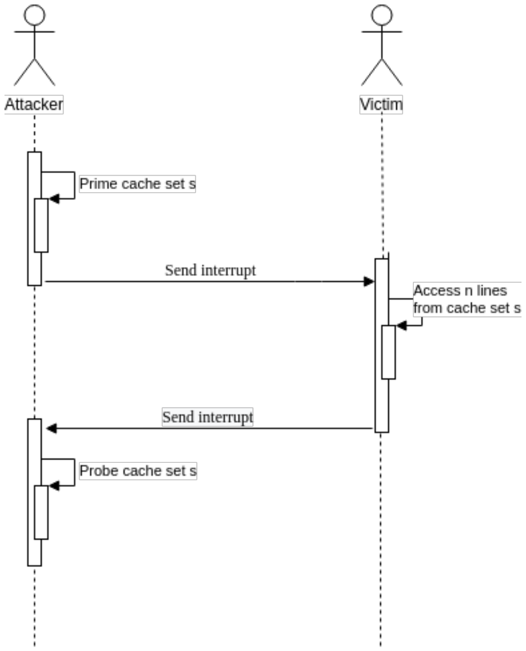


Figure 4.3: Sequence diagram of the attack that allows to observe the Simple Channel.

The attacker starts the attack by priming the cache set *s*. Then, it signals the victim to start accessing the cache sending an interrupt. After signaling the victim, the attacker halts its execution and waits for the victim to send back a signal. The victim proceeds to access a predefined number *n* of cache lines from cache set *s*. Then, it signals back the attacker sending another interrupt and stops executing. The attacker resumes the execution and completes the attack by probing the same cache set *s* that had been primed and accessed. After probing the cache set, the attacker may start another attack by repeating the sequence.

4.1.1.3 Flow Charts

The execution flows of the attacker and the victim are illustrated by Figure 4.4 and Figure 4.5, respectively.

1. **Attacker -** As aforementioned, to start the attack, the attacker and the victim need to agree beforehand on a cache set to contend. So, the first action the attacker needs to take before beginning the attack is to get the set index of the address that the victim will target. This is necessary because the attacker will need to target, when priming and probing the cache, a set of congruent cache lines[1] resorting to the obtained cache set index.

   Then, the next step for the attacker is to prime the cache. It obtains the congruent addresses and then evicts them through an eviction strategy that uses architecture-specific reading instructions.

   After the priming finishes, the attacker sends an interrupt to the victim and waits until the victim finishes accessing the cache. When the attacker receives the interrupt, it proceeds to probe the cache by accessing all the previously evicted lines (in the prime step) in reversed order. The access is done in reversed order to avoid self eviction caused by the pseudo-random policy of L2 cache.

   Then, accordingly to a predefined number of *Prime+Probe* runs, the attacker repeats the same procedure as many times as it is needed. This is done to prevent the observation of false positives, that may happen due to the evicting inefficacy of the eviction strategy.

   After completing all the attack iterations, the attacker calculates the average of the obtained probe times, comparing its value to a predefined threshold value (calculated by measuring the minimum amount of time an attacker takes to probe the cache). If the average value is below the threshold, it means that the victim didn't access the cache between the prime and probe phase, as every line accessed in the probe phase had already been there, resulting in multiple cache misses. If the average value is higher than the threshold, then at least one cache line of the agreed set has been accessed by the victim, resulting in one cache miss.

[1] Congruent addresses: addresses that map to the same cache set.

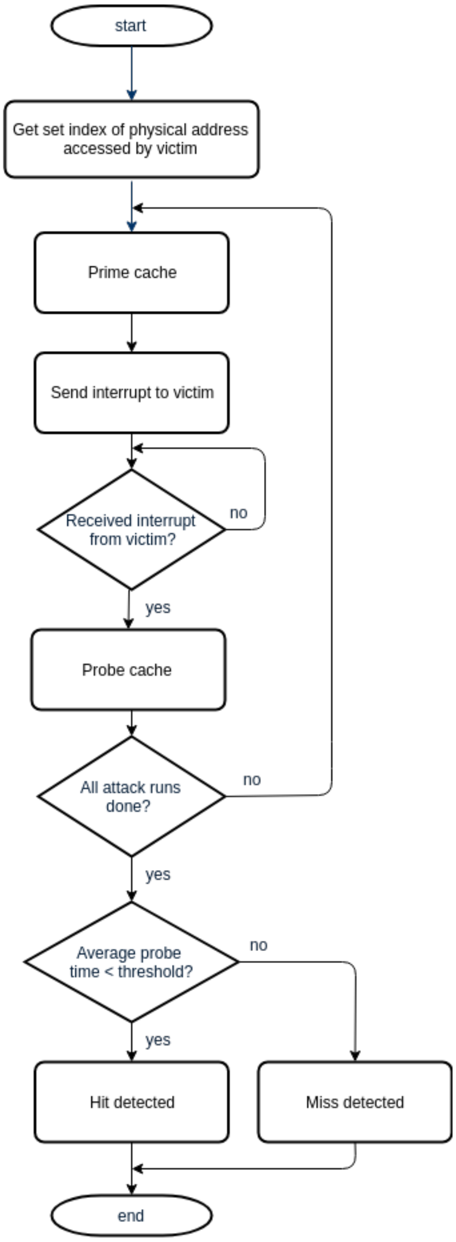


Figure 4.4: Attacker's flow chart during the attack that allows to observe the Simple Channel.

2. **Victim -** The victim's behavior is solely based on accessing the cache lines, with the caution of starting its activity when told to by the attacker.

   The victim waits indefinitely for an interrupt to be sent by the attacker. When the victim receives the interrupt, it proceeds to access a physical address that maps to the same cache set that has been accessed by the attacker.

   If the intended number of lines have been accessed, the victim sends back the interrupt to the attacker and ends the execution. If the victim pretends to access more lines, the victim changes the physical address to another physical address that maps to another line in the same cache set.

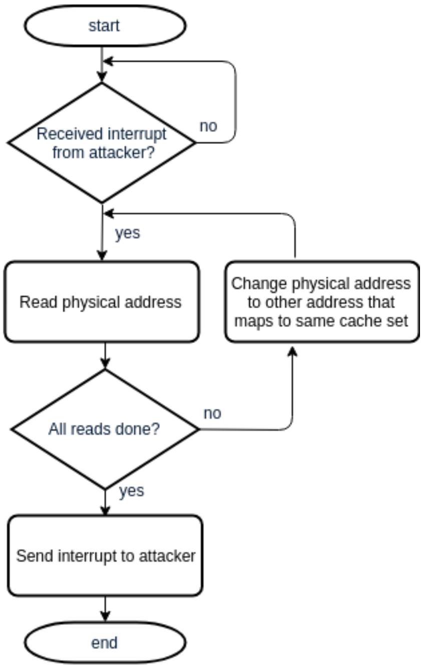

start
Received interrupt from attacker?
no
yes
Read physical address
Change physical address to other address that maps to same cache set
All reads done?
no
yes
Send interrupt to attacker
end

Figure 4.5: Victim's flow chart during the attack that allows to observe the Simple Channel.

4.1.2 Bits Transmission Channel

Heiser et al. [17] proposed a covert channel that uses the same principle of the first channel (i.e., the more the probing time, the more lines of a certain set have been accessed) as an indicator of cache set usage to establish communication between guests.

To observe the channel, the attacker and the victim need to agree on 2 cache sets and 2 cache lines, where line *0* maps to set *0*, and line *1* maps to set *1*. To send the value *1* or *0*, the victim continuously accesses line *1* or line *0*, respectively, for an amount of time, while the attacker primes and probes both cache sets that are mapped by the lines. The cache set that takes longer to probe is the one that was accessed by the victim between the prime and probe phases.

The algorithms used to observe the channel are illustrated below in Algorithm 3, for the victim, and Algorithm 4, for the attacker.

Algorithm 3: Covert channel protocol / Victim operations

```
Input: Cache lines 0 and 1
       D[N]: N bits to be transmitted
Output: Probing time of cache sets 0 and 1
for $i \leftarrow 0$ to N-1 do
    if D[i] to 1 then
        access line 1;
    else
        access line 0;
    end if
end for
```

Algorithm 4: Covert channel protocol / Attacker operations

```
for an amount of time do
    probe set 0 backwards;
    probe set 1 backwards;
end for
```

4.1.2.1 Statechart Diagram

The behavior of both the attacker and the victim during the attack execution are described with the correspondent state machine illustrated by Figure 4.7.

1. **Attacker -** The attacker focuses solely on priming and probing the cache interleaving with the victim. In contrast to the Simple Channel attack, where there was only one set to be contended, this channel requires two sets to be contended. So, the prime step of the attacker consists of priming two cache sets at the same time, and, after the victim accesses one of the sets, the attacker needs to probe both cache sets and compare them. This execution loop occurs until the number of needed prime and probe runs is achieved.

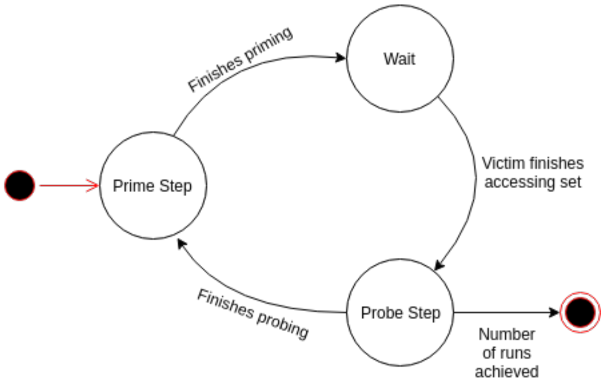


Figure 4.6: Attacker state machine during the attack that allows observing the Bits Transmission Channel.

2. **Victim -** In this channel, the victim accesses more than one set, alternating the sets being accessed between each iteration. The first set to be accessed is the set *0*, which is mapped by the cache line *0*. The victim waits for the attacker to prime that set, and then accesses the line *0* evicting a line from the cache set *0*. After accessing the set *0*, the victim waits for the attacker to probe the set *0* and proceed to prime the next set *1*, which is mapped by the cache line *1*. When the attacker finishes priming the set *1*, the victim repeats the sequence and accesses the cache line *1* to evict a line from cache set *1*. The attacker then probes the cache set *1* and compares the time that took to access each one of the sets.

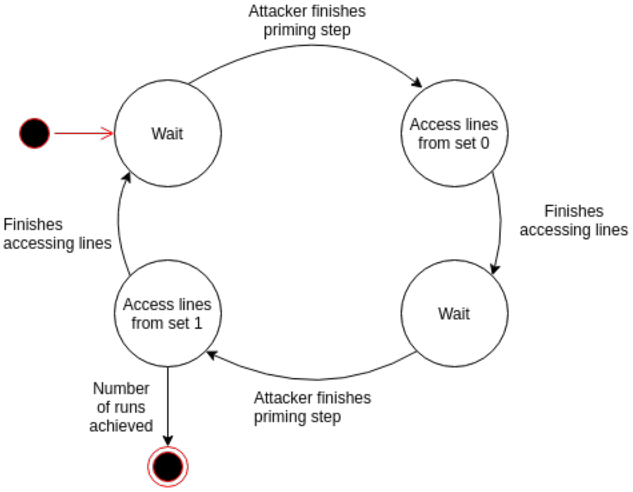


Figure 4.7: Victim state machine during the attack that allows observing the Bits Transmission Channel.

4.1.2.2 Sequence Diagram

The sequence performed during one attack iteration is described by the diagram illustrated in Figure 4.8.

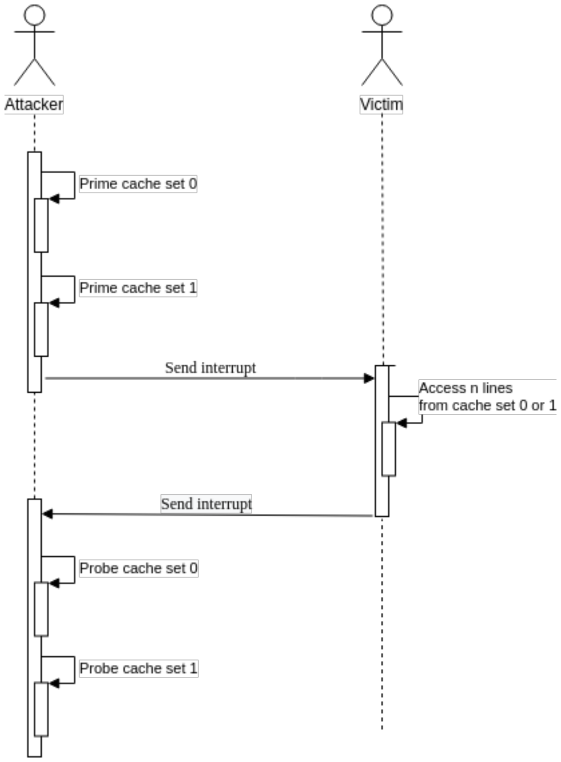


Figure 4.8: Sequence diagram of the attack that allows to observe the Bits Transmission Channel.

The attacker starts the attack by priming the cache set *0* and cache set *1*. After both cache sets have their content primed, the attacker signals the victim to access one of the cache sets and signal back the end of its execution. Likewise to the first channel, the synchronization between both actors is achieved using interrupt requests. Then, the attacker resumes its execution and completes the attack by probing both cache sets.

4.1.2.3 Flow Charts

The execution flows of the attacker and the victim are illustrated by Figure 4.9 and Figure 4.10, respectively.

1. **Attacker -** The attacker obtains the set index of the targeted physical address, enabling the attacker to find the congruent addresses.

Then, the attacker primes both cache sets *0* and *1*, and sends an interrupt to the victim signaling the end of the priming phase. The attacker waits until the victim finishes accessing one of the cache sets and probes both cache sets afterward. If the time taken to probe the cache set *0* is less than the probing time of the cache set *1*, it means that lesser lines of cache set *0* have been accessed (subsequently, evicted) by the victim. In this case, that means that the victim didn't access the cache set *0*, but accessed the cache set *1*. The same occurs for the case when the probing time of the cache set *1* is less than the cache set *0*.

After the victim transmits all of the bits (willingly) through the access of different cache sets within defined time intervals, the attacker then finishes its execution.

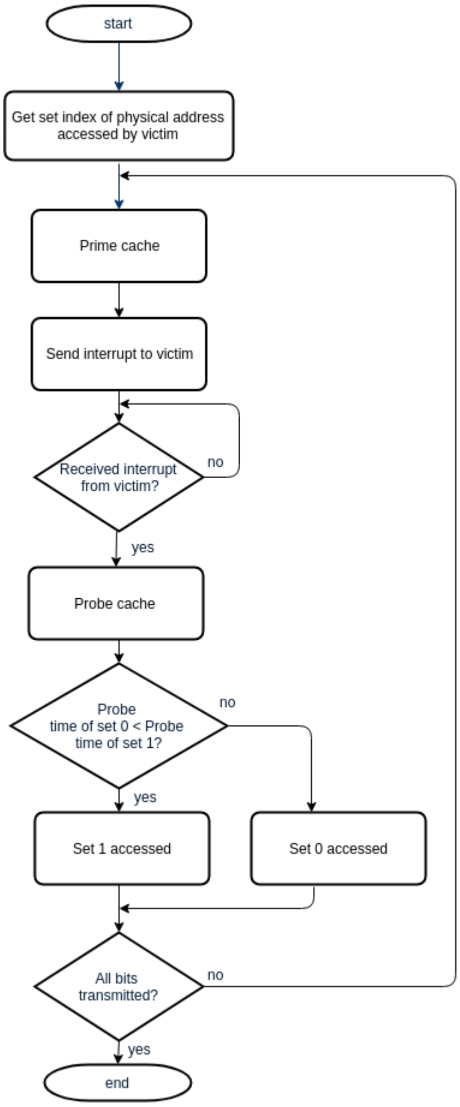

start
Get set index of physical address accessed by victim
Prime cache
Send interrupt to victim
Received interrupt from victim?
no
yes
Probe cache
Probe time of set 0 < Probe time of set 1?
no
yes
Set 1 accessed
Set 0 accessed
All bits transmitted?
no
yes
end

Figure 4.9: Attack's flow chart during the attack that allows to observe the Bits Transmission Channel.

2. **Victim -** Firstly, the victim waits for an interrupt from the attacker. When the victim receives the interrupts, it checks for the interrupt's occurrence. As the transmission is alternate, it is established that the even occurrences of interrupt are used to send bit *0*, while the odd occurrences are used to send bit *1*. So, an even occurrence means accessing a line that maps to cache set *0*, while an odd occurrence means accessing a line that maps to cache set *1*.

   When the access to either cache set *0* or cache set *1* is done, the victim might want to access more lines to increase the probing time of the attacker. This approach is preferable because the greater the probing time, the more noticeable is the victim's access to the attacker's perspective. When the access to the intended lines is done, the victim finally sends an interrupt to attacker signaling the end of the cache access.

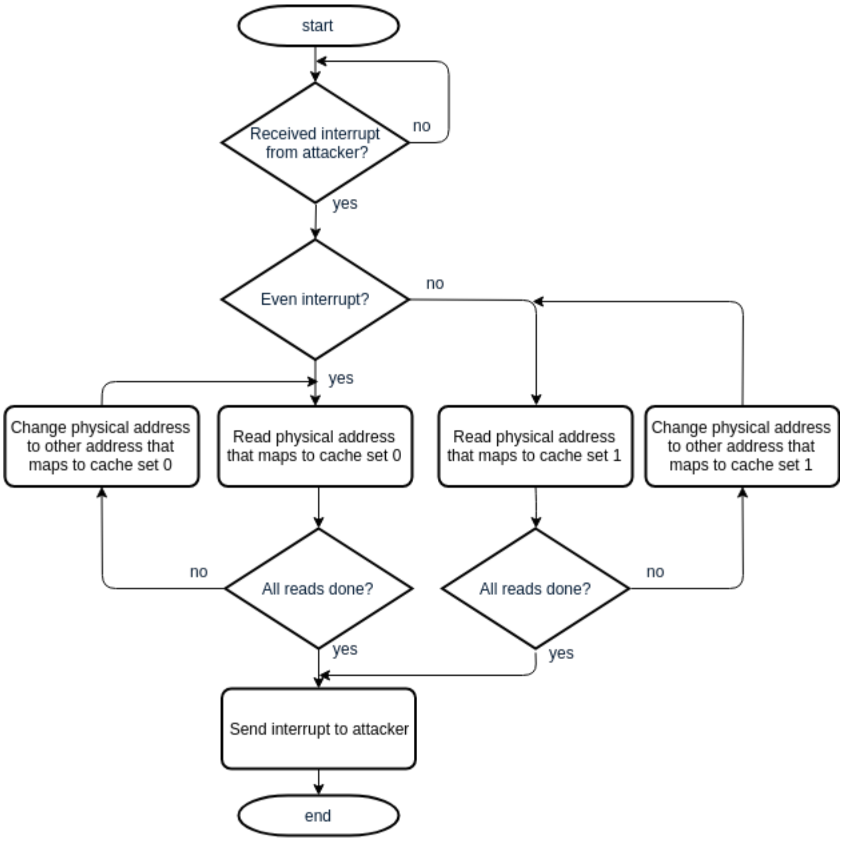

start
Received interrupt from attacker?
no
yes
Even interrupt?
no
yes
Change physical address to other address that maps to cache set 0
Read physical address that maps to cache set 0
Read physical address that maps to cache set 1
Change physical address to other address that maps to cache set 1
no
All reads done?
All reads done?
no
yes
yes
Send interrupt to attacker
end

Figure 4.10: Victim's flow chart during the attack that allows to observe the Bits Transmission Channel.

5. Implementation

This chapter describes the implementation of the proposed channels whose design was presented in the last chapter. The implementation encompasses the system configurations and the previously designed algorithms. Furthermore, the code that allows overcoming the challenges (see Section 3.2) is highlighted, and the implementation of the countermeasure is presented too.

5.1 System Configuration

Before implementing the attacks, the system must be configured statically. The configuration occurs, in Jailhouse's case, via Jailhouse's cell configuration source files (see Section 2.3.2.1) and Linux's boot parameters. In Bao Hu's case, the configuration is done through a single configuration file that describes the assigned resources for all guests.

5.1.1 Jailhouse

This section describes Jailhouse cells' configurations without the coloring feature.

5.1.1.1 Cells Configuration

1. **Root Cell -** Firstly, Jailhouse needs the Linux kernel boot parameters *mem=* to be set in order to reserve memory for other cells. As the targeted board *ZCU104 Evaluation Kit* provides 2GB of RAM accessible in a 32-bit address map (depends on the address width of the interface master [50]), the chosen reserved memory for hypervisor and non-root cell is 0,75GB, while Linux is configured to use 1,25GB.

   In this case, the attribution is done through U-Boot, recurring to the following command:

Listing 5.1: Memory reservation code. The *mem=* kernel boot parameter sets the available physical memory and reserves the rest of the memory.

```
    setenv bootargs "mem=1280M"
```

After reserving the memory for hypervisor and non-root cell, it follows the configuration of the root cell. Due to the large size of the configuration code, only the most relevant sections of code are addressed.

It is in the root cell configuration where the hypervisor memory and other root cell's memory regions locations are defined. The hypervisor memory must be within the reserved memory in U-Boot, and it is defined by members *.phys_start* and *.size*, which represent the beginning (i.e., host physical address) and the length of the memory region, respectively.

Regarding the CPUs assigned to each cell, since the root cell owns all CPUs at start and then loses them according to non-root needs, it is attributed to the root cell all CPUs. This is done by attributing the bitmap value *0xf*, that converted to binary representation means *0b1111*. Since there are 4 cores in the targeted processor (i.e., *Cortex A53 MPCore*), 4 bits are enough to encompass all CPUs.

The first declared memory region allows the access to PS I/O peripherals registers (e.g., UART, I2C, CAN and other peripherals), while the second one allows the access to RAM with the permissions to write, read and execute data within the established boundaries defined by *.phys_start* and *.virt_start* members (*.phys_start* means the host's physical address, and *.virt_start* means the guest's physical address). The last memory region grants to root cell the access to the previously mentioned shared memory region (i.e., ivshmem) with the non-root cell.

Concerning the PCI devices, it is registered one virtual PCI device as a shared memory device in *.type* member. This virtual PCI device allows cells to discover shared memory and send each other interrupts.

Listing 5.2: Root cell configuration code.

```
.....
.hypervisor_memory = {
/* Must be within reserved memory (0x50000000-0x80000000)*/
    .phys_start = 0x7c000000,
    .size =       0x00400000,
    },
.....
/* CPUs which are assigned to a cell */
.cpus = {
    0xf, /* Here are assigned all CPUs */
},
.....
/*Memory regions this cell has access and with which rights (flags).*/
.mem_regions = {
    /* MMIO (permissive) */ {
    .phys_start = 0xfd000000,
    .virt_start = 0xfd000000,
    .size =          0x03000000,
    .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
            JAILHOUSE_MEM_IO,
    },
    /* RAM */ {
    .phys_start = 0x0,
    .virt_start = 0x0,
    .size = 0x7c000000,
    .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
        JAILHOUSE_MEM_EXECUTE,
    },
    /* IVSHMEM shared memory region for 00:00.0 */ {
    .phys_start = 0x7bf00000,
    .virt_start = 0x7bf00000,
    .size = 0x100000,
    .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE,
    },
},
```

2. **Non-root Cell -** To the non-root cell are assigned the cores 2 and 3, given by the bitmap *0b1100*. The assigned memory regions range from UART, needed to interact with the inmate, to the shared memory region with root cell. A memory region that enables the communication between hypervisor and non-root cell is needed, in order to issue hypercalls that can enable, load and start the cell. Finally, it is given a 512MB-sized RAM dedicated to the inmate, represented by *erika_inmate.bin* file.

Listing 5.3: Non-root cell configuration code.

```
.....
.cpus = {
    0xc, /* Here are assigned CPUs 2 and 3*/
},
.....
.mem_regions = {
/* UART */ {
    .phys_start = 0xff010000,
    .virt_start = 0xff010000,
    .size = 0x1000,
    .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
        JAILHOUSE_MEM_IO | JAILHOUSE_MEM_ROOTSHARED,
    },
    /* RAM for loader*/ {
        .phys_start = 0x7bef0000,
        .virt_start = 0, //needs to start at 0 for loader
        .size = 0x10000,
        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
            JAILHOUSE_MEM_EXECUTE | JAILHOUSE_MEM_LOADABLE,
    },
    /* IVSHMEM shared memory region */ {
        .phys_start = 0x7bf00000,
        .virt_start = 0x7bf00000,
        .size = 0x100000,
        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
            JAILHOUSE_MEM_ROOTSHARED,
    },
    /* communication region */ {
        .virt_start = 0x80000000,
        .size = 0x00001000,
        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
            JAILHOUSE_MEM_COMM_REGION,
    },
    /* RAM for erika_inmate.bin*/{
        .phys_start = 0x50000000,
        .virt_start = 0x50000000,
        .size = 0x20000000, //must be page size aligned
        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
            JAILHOUSE_MEM_EXECUTE | JAILHOUSE_MEM_DMA |
            JAILHOUSE_MEM_LOADABLE,
    },
},
```

5.1.1.2 Memory Layout

When the configuration is done, the system's memory (only the root cell's address space is illustrated) has the layout depicted in Figure 5.1. Within the RAM's address space, there is an overt division between the reserved and non-reserved memory, established previously in U-Boot, through kernel boot parameters. The non-reserved memory is fully taken by the root cell, which represents the first 1280MB of the RAM address space, while the rest of the memory is divided between the non-root cell and the hypervisor memory. The hypervisor memory only occupies 4MB, while the non-root cell occupies the rest of the reserved memory.

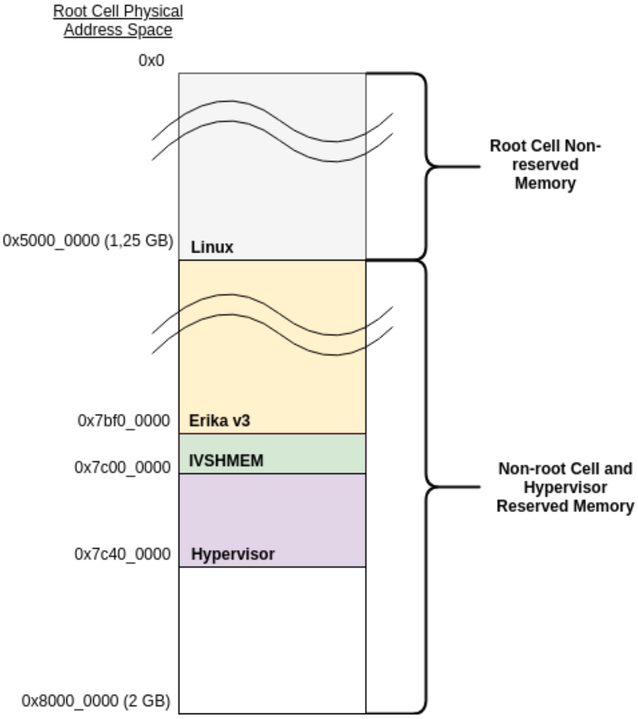

Root Cell Physical Address Space
0x0
0x5000_0000 (1,25 GB)
Linux
0x7bf0_0000
Erika v3
0x7c00_0000
IVSHMEM
0x7c40_0000
Hypervisor
0x8000_0000 (2 GB)
Root Cell Non-reserved Memory
Non-root Cell and Hypervisor Reserved Memory

Figure 5.1: RAM's layout after configuring the system.

5.1.2 Jailhouse with cache coloring

This section describes Jailhouse cells' configurations with the coloring feature implemented.

5.1.2.1 Cells Configuration

1. **Root Cell -** Using colored memory has its drawbacks. One of them is the amount of page entries that is needed by the strided memory mapping, which could easily fill up all the memory reserved for Jailhouse. As a consequence, the hypervisor memory needs to be increased.

Listing 5.4: Root cell configuration code.

```
.....
.hypervisor_memory = {
    .phys_start = 0x7d000000,
    .size =       0x01000000,
},
.....
.cpus = {
    0xf,
},
.....
.mem_regions = {
    /* MMIO (permissive) */ {
        .phys_start = 0xfd000000,
        .virt_start = 0xfd000000,
        .size =        0x03000000,
        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
            JAILHOUSE_MEM_IO,
    },
    /* Colored RAM for inmates*/ {
        .phys_start = 0x0,
        .virt_start = 0x0,
        .size = 0x7d000000,
        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
            JAILHOUSE_MEM_EXECUTE,
    },
    /* IVSHMEM shared memory region for 00:01.0 (network) */ {
        .phys_start = 0x7c000000,
        .virt_start = 0x7c000000,
        .size = 0x100000,
        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE,
    },
},
```

2. **Linux Non-root Cell -** The Linux non-root cell, since it is going to be used as the cell of the attacker, needs to have a single region that can hold both kernel image, kernel decompression space, dtb and initramfs together. To that region is given 8 colors out of 16, represented by the member *.colors*, which has the value *0xff00*.

   The loader needs another memory region, which needs to begin mandatorily on the guest's physical address *0*, due to *jailhouse-cell-linux* command implementation. Besides it, to communicate with the victim's cell (i.e., Erika non-root cell) and the root cell, where the commands are issued, it has two shared memory regions and a communication region to communicate with the hypervisor.

Listing 5.5: Non-root cell configuration code.

```
.....
.cpus = {
    0x8, /* Here is assigned CPU 3*/
},
.....
.mem_regions = {
    /* UART */ {
        .phys_start = 0xff010000,
        .virt_start = 0xff010000,
        .size = 0x1000,
        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
            JAILHOUSE_MEM_IO | JAILHOUSE_MEM_ROOTSHARED,
    },
    /* Colored RAM for kernel image and initramfs*/ {
        .phys_start = 0x40000000,
        .virt_start = 0x40000000,
        .size = 0x30000000,
        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
            JAILHOUSE_MEM_EXECUTE | JAILHOUSE_MEM_LOADABLE |
            JAILHOUSE_MEM_DMA| JAILHOUSE_MEM_COLORED_CELL,
            .colors = 0xff00,
    },
    /* RAM for loader*/ {
        .phys_start = 0x7bef0000,
        .virt_start = 0,
        .size = 0x10000,
        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
            JAILHOUSE_MEM_EXECUTE | JAILHOUSE_MEM_LOADABLE,
    },
    /* RAM */ {
        .phys_start = 0x74000000,
        .virt_start = 0x74000000,
        .size = 0x7ef0000,
        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
            JAILHOUSE_MEM_EXECUTE | JAILHOUSE_MEM_DMA |
            JAILHOUSE_MEM_LOADABLE,
    },
    /* IVSHMEM shared memory region */ {
        .phys_start = 0x7bf00000,
        .virt_start = 0x7bf00000,
        .size = 0x100000,
        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
            JAILHOUSE_MEM_ROOTSHARED,
    },
    /* communication region */ {
        .virt_start = 0x80000000,
        .size = 0x00001000,
        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
            JAILHOUSE_MEM_COMM_REGION,
    },
    /* IVSHMEM shared memory region (network) */ {
        .phys_start = 0x7c000000,
        .virt_start = 0x7c000000,
        .size = 0x100000,
        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
            JAILHOUSE_MEM_ROOTSHARED,
    },
},
.....
.pci_devices = {
    /* 00:00.0 */ {
    .type = JAILHOUSE_PCI_TYPE_IVSHMEM,
    .bdf = 0 << 3, // 00:00.0
    .iommu = 1,
    .bar_mask = {
        0xffffff00, 0xffffffff, 0x00000000,
        0x00000000, 0x00000000, 0x00000000,
    },
    .shmem_region = 4,
    .shmem_protocol = JAILHOUSE_SHMEM_PROTO_UNDEFINED, //Undefined type
    },
    /* 00:01.0 */ {
    .type = JAILHOUSE_PCI_TYPE_IVSHMEM,
    .bdf = 1 << 3, // 00:01.0
    .shmem_region = 6,
    .shmem_protocol = JAILHOUSE_SHMEM_PROTO_VETH, //Virtual peer-to-peer Ethernet
},
},
```

3. **Erika Non-root Cell -** The Erika non-root cell has a memory region where the binary *erika_inmate.bin* is located, and it is given to that region the other 8 colors that weren't set.

   This cell is also assigned with a virtual PCI device which allows to receive interrupts and share memory with Linux non-root cell.

Listing 5.6: Non-root cell configuration code.

```
.....
.cpus = {
    0x4, /* Here is assigned CPU 2*/
},
.....
.mem_regions = {
    /* UART */ {
    .phys_start = 0xff010000,
    .virt_start = 0xff010000,
    .size = 0x1000,
    .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
        JAILHOUSE_MEM_IO | JAILHOUSE_MEM_ROOTSHARED,
    },
    /* IVSHMEM shared memory region */ {
    .phys_start = 0x7bf00000,
    .virt_start = 0x7bf00000,
    .size = 0x100000,
    .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
        JAILHOUSE_MEM_ROOTSHARED,
    },
    /* communication region */ {
    .virt_start = 0x80000000,
    .size = 0x00001000,
    .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
        JAILHOUSE_MEM_COMM_REGION,
    },
    /* Colored RAM for erika_inmate.bin*/{
    .phys_start = 0x70000000,
    .virt_start = 0x0,
    .size = 0x4000000,
    .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
        JAILHOUSE_MEM_LOADABLE | JAILHOUSE_MEM_DMA |
        JAILHOUSE_MEM_COLORED_CELL,
    .colors = 0x00ff,
    },
},
```

5.1.3 Bao Hu

The Bao Hu's configuration system is much simpler than Jailhouse's because it allows to configure all the guests in the same file and it doesn't support yet the majority of the functionalities as Jailhouse does, which implies a naturally simpler approach to the assignment of memory regions and devices.

The guest images in Bao Hu are statically assigned in the configuration file, in contrast to Jailhouse where they are assigned in execution time through userspace commands.

Listing 5.7: Bao Hu guest images assignment.

```
VM_IMAGE(bm1, /home/joao/nloader/linux.bin);
VM_IMAGE(bm2, /home/joao/bao_testguest/startup_AEMv8-FVP_AArch64_GCC.bin);
```

To share memory between guests, two shared memory regions are declared: (i) the first one is used to synchronize both guests, and (ii) the second one is used as a contiguous memory region that starts at an address that maps to cache set 0, being this way a reference to the guest's access to targeted addresses.

Listing 5.8: Bao Hu shared memory declarations.

```
shmem_t bm_shmem = {
    .size = 0x1000,
    .colors = 0
};

shmem_t bm_shmem_contend_guest = {
    .size = 0x100000, // 1MB = L2 size
    .colors = 0,
};
```

Then, each guest image is assigned to a memory region, where it is given the guest's starting physical address (*.base* member) and a size for each memory region (*.size* member). It is also possible to assign a number of CPUs to each guest (*.cpu_num* member) and specify them (*.cpu_affinity* member) using the bitmap notation as it has been used in Jailhouse.

Listing 5.9: Bao Hu memory region assignment for Linux guest.

```
....
            .image = {
                .base_addr = 0x00200000,
                .load_addr = VM_IMAGE_OFFSET(bm1),
                .size = VM_IMAGE_SIZE(bm1)
            },

            .entry = 0x00200000,
            .cpu_affinity = 0x7,
            .platform = {
                .cpu_num = 1,
                .region_num = 2,
                .regions =  (struct mem_region[]) {
                    {
                        .base = 0x0000000,
                        .size = 0x20000000 //512 MB
                    },
                    {
                        .base = 0x21000000,
                        .size = 0x1000,
                        .shared = &bm_shmem
                    }
                },
                }
...
```

5.1.4 Bao Hu with cache coloring

Regarding the cache coloring implementation in Bao Hu, it is possible to assign colors to each guest, which means that the memory regions that the guest owns are only allocated to regions that map to a certain set not accessible by other guests (through *.color* member), including the shared memory regions which can be colored separately.

The distribution of colors can be done in any way, but in order to have mutual cache protection between guests, memory regions with different colors must be assigned to different guests.

To the contiguous shared memory that is used to target certain cache sets by the victim's side, it is given a color different from the attacker guest, since it is imperative for them to not contend for the same cache sets, otherwise the victim will evict the previously primed addresses and then the attacker will recognize them as evicted.

Listing 5.10: Bao Hu colored shared memory declarations.

```
shmem_t bm_shmem = {
    .size = 0x1000,
    .colors = 0
};

shmem_t bm_shmem_contend_guest = {
    .size = 0x100000, // 1MB = L2 size
    .colors = 0x1
};
```

Another point to remember is to keep colors as contiguous as possible, allowing caches to exploit the higher performance to central memory controller [32], so it is given half of the available colors to the attacker guest (given by value *0xf0*) while the rest of the colors are given to the victim guest (given by value *0x0e*).

Listing 5.11: Bao Hu memory region assignment for colored Linux guest.

```
            .image = {
                .base_addr = 0x00200000,
                .load_addr = VM_IMAGE_OFFSET(bm1),
                .size = VM_IMAGE_SIZE(bm1)
            },

            .entry = 0x00200000,
            .cpu_affinity = 0x7,
            .colors = 0xf0,
            .platform = {
                .cpu_num = 1,
                .region_num = 2,
                .regions =  (struct mem_region[]) {
                    {
                        .base = 0x0000000,
                        .size = 0x20000000 //512 MB
                    },
                    {
                        .base = 0x21000000,
                        .size = 0x1000,
                        .shared = &bm_shmem
                    }
                },
                }
```

5.2 Eviction Strategy

Before implementing the channels, it is also needed to test which eviction strategy is more effective while evicting a cache. The parameters (i.e., N-A-D parameters) from Table 3.1 that showed the best eviction rate, and the parameters used by Irazoqui et al. [47] to test AutoLock, were chosen to be tested.

This test occurs in a context where there is no victim, but only an attacker that simulates an access to a cache line between prime and probe steps, meaning that the attack doesn't occur between cores, opposed to the attacks done in the proposed channels. The attacks are repeated over 100000 times, 50000 for each hit and miss cases, and then recorded to establish a histogram.

To simulate a cache hit, the attacker doesn't access a cache line between the prime and probe steps, minimizing the cache set activity. The function that implements a cache hit and records its probing value is illustrated in Listing 5.12.

Listing 5.12: Hit simulation function.

```c
static void prime_probe_hit(libflush_session_t* libflush_session, void* address, size_t runs, size_t* histogram, size_t histogram_size, size_t histogram_scale)
{
  size_t set_index = libflush_get_set_index(libflush_session, address);

  for (unsigned int i = 0; i < runs; i++) {
    libflush_prime(libflush_session, set_index);
    size_t time = libflush_probe(libflush_session, set_index);
    histogram[MIN(histogram_size - 1, time / histogram_scale)]++;
  }
}
```

To provoke a cache miss, the attacker accesses one cache line that maps to the targeted set between the prime and probe steps, as it is illustrated in Listing 5.13.

Listing 5.13: Miss simulation function.

```c
static void prime_probe_miss(libflush_session_t* libflush_session, void* address, size_t runs, size_t* histogram, size_t histogram_size, size_t histogram_scale)
{
  size_t set_index = libflush_get_set_index(libflush_session, address);

  for (unsigned int i = 0; i < runs; i++) {
    libflush_prime(libflush_session, set_index);

    /* Access targeted address */
    libflush_access_memory(address);
    size_t time = libflush_probe(libflush_session, set_index);
    histogram[MIN(histogram_size - 1, time / histogram_scale)]++;
  }
}
```

5.3 Attack Challenges

In this section, it is described the implementation of the challenges that were addressed previously (see Section 3.2).

5.3.1 Prime Step

During the prime step, as the name suggests, the targeted cache set needs to be primed the way the attacker wants. This means evicting the cache set and exchanging the content of it with the expected data. To do this efficiently, the attacker needs primarily to know how to access the addresses for them to evict the previous content from cache set, and then reside there.

1. **Understand hidden mappings -** As the accessed addresses in the attacker's case (i.e., via Linux application) are virtual, the first challenge consists in knowing their translation to physical memory, or to use huge pages. The chosen method is to understand the virtual-to-physical translation, which requires privileged access to */proc/self/pagemap* system service.

   The */proc/self/pagemap* file is read to *memory.pagemap* variable before starting the attack as it is illustrated in Listing 5.14.

   Listing 5.14: Code to get physical address recurring to */proc/self/pagemap*.

```
bool libflush_init(libflush_session_t** session, libflush_session_args_t* args)
{
  #if HAVE_PAGEMAP_ACCESS == 1
   (*session)->memory.pagemap = open("/proc/self/pagemap", O_RDONLY);
   if ((*session)->memory.pagemap == -1) {
    free(*session);
    return false;
   }
  #endif
 return true;
}
```

   Then, when the attack starts, the attacker can get the physical address of a virtual address by reading the *memory.pagemap* variable and retrieving the page frame from which the virtual page is mapped to. With the page frame number and the virtual address known, the attacker can then calculate the physical address by appending the first 12 bits of the virtual address to the page frame number.

Listing 5.15: Code to get physical address recurring to */proc/self/pagemap*.

```c
uintptr_t libflush_get_physical_address(libflush_session_t* session, uintptr_t virtual_address)
{
  (void) session;
  (void) virtual_address;

#if HAVE_PAGEMAP_ACCESS == 1
  // Access memory
  libflush_access_memory((void *) virtual_address);

  uint64_t value;
  off_t offset = (virtual_address / 4096) * sizeof(value);
  int got = pread(session->memory.pagemap, &value, sizeof(value), offset);
  assert(got == 8);

  // Check the "page present" flag.
  assert(value & (1ULL << 63));

  uint64_t frame_num = get_frame_number_from_pagemap(value);
  return (frame_num * 4096) | (virtual_address & (4095));
#else
  return 0;
#endif
}
```

Once known the physical value of the addresses, the attacker can then check which ones share the same set index (i.e., congruent addresses). The function that searches for congruent addresses is illustrated in Listing 5.16.

Listing 5.16: Find congruent addresses used during Prime step code.

```c
void find_congruent_addresses(libflush_session_t* session, libflush_eviction_t*
    eviction, size_t index, uintptr_t physical_address)
{
  unsigned int found = 0;
  for (unsigned int i = 0; i < eviction->memory.mapping_size; i += LINE_LENGTH)
  {
    uint8_t* virtual_address_2 = (uint8_t*) eviction->memory.mapping + i;
  uintptr_t physical_address_2 = libflush_get_physical_address(session, (uintptr_t) virtual_address_2);
  uint64_t index_2 = (physical_address_2 >> LINE_LENGTH_LOG2) % NUMBER_OF_SETS;

  if ((index == index_2) && (physical_address != physical_address_2))
  {
   eviction->congruent_address_cache[index].congruent_virtual_addresses[found++] =
    virtual_address_2;
  }
 }
}
```

2. **Defeat cache replacement policy -** When the virtual addresses that map to the same cache set have finally been found, they are used to evict the cache lines from the targeted cache set. This isn't as straightforward as it seems due to the pseudo-random replacement policy implemented in the LLC. So, to overcome the self-eviction effect provoked by the replacement policy, an eviction strategy is implemented (see Section 3.2.1.2). The function that implements the eviction strategy is illustrated in Listing 5.17.

Listing 5.17: Eviction strategy used during Prime step code.

```
void evict(congruent_address_cache_entry_t* congruent_address_cache_entry)
{
/* For each address accessed */
 for (unsigned int i = 0; i < ES_EVICTION_COUNTER; i += 1) {
 /* For each time the same address is accessed repeatedly */
  for (unsigned int j = 0; j < ES_NUMBER_OF_ACCESSES_IN_LOOP; j++) {
  /* For each different address accessed in a loop */
   for (unsigned int k = 0; k < ES_DIFFERENT_ADDRESSES_IN_LOOP; k++) {
    libflush_access_memory(congruent_address_cache_entry->congruent_virtual_addresses[i+k]);
   }
  }
 }
}
```

5.3.2 Probe Step

During the probe step, the attacker only needs to check which previously accessed addresses still remain in the cache, that is, how many cache hits occur during probe step.

1. **Defeat cache replacement policy -** To avoid *cache thrashing*, the eviction set size is reduced and the access is done backwards. As the attack is run in a loop, exactly 1 way in the L2 cache will not be occupied after a few attack rounds, which has the disadvantage of missing a victim access in 1/16 of the cases. If the victim replaces one of the 15 ways occupied by the attacker, there is still one empty way to reload the address that was evicted. This reduces the chance of *cache thrashing* significantly and allows to successfully perform the attack on caches with a random replacement policy.

Listing 5.18: Backwards access strategy used during Probe step code.

```
void libflush_eviction_probe(libflush_session_t* session, size_t set_index)
{
  libflush_eviction_t* eviction = (libflush_eviction_t*) session->data;

  congruent_address_cache_entry_t congruent_address_cache_entry =
    eviction->congruent_address_cache[set_index];

  if (congruent_address_cache_entry.used == false) {
    find_congruent_addresses(session, eviction, set_index, (uintptr_t) NULL);
  }

  for (int i = ADDRESS_COUNT - 1; i >= 0; i -= 1) {
    libflush_access_memory(congruent_address_cache_entry.congruent_virtual_addresses
        [i]);
  }
}
```

2. **Accurate timing -** To time accurately the probing time of cache, it was chosen the *perf* Linux profiler, as it offers a highly precise timing and doesn't need privileged access.

   In the attack's case, *perf* uses the CPU cycles as the source to measure the time to access the cache (i.e., given by PERF_COUNT_HW_CPU_CYCLES macro). The initialization of *perf* Linux profiler is illustrated in Listing 5.19.

Listing 5.19: perf syscall used during Probe step code.

```

inline bool perf_init(libflush_session_t* session, libflush_session_args_t* args)
{
  (void) session;
  (void) args;

  static struct perf_event_attr attr;
  attr.type = PERF_TYPE_HARDWARE;
  attr.config = PERF_COUNT_HW_CPU_CYCLES;
  attr.size = sizeof(attr);
  attr.exclude_kernel = 1;
  attr.exclude_hv = 1;
  attr.exclude_callchain_kernel = 1;

  session->perf.fd = syscall(__NR_perf_event_open, &attr, 0, -1, -1, 0);
  return true;
}
```

Whenever the attacker wants to measure the time to access the cache, it only needs to get a time reference before accessing the cache, and after accessing the cache, obtain again a time reference and calculate the difference between both time references. The function that allows to obtain the time reference via *perf* is illustrated in Listing 5.20.

Listing 5.20: perf syscall used during Probe step code.

```
inline uint64_t perf_get_timing(libflush_session_t* session)
{
  long long result = 0;
  if (read(session->perf.fd, &result, sizeof(result)) < (ssize_t) sizeof(result)) {
    return 0;
  }
  return result;
}
```

5.4 Proposed Channels

To prove the existence of cache timing channels between VMs, some libraries and hypervisor features were used to deploy the attacks:

1. **libflush -** The library, named *libflush*, was used to implement an attack based on *Prime+Probe* exploitation technique. It was developed by Gruss et al. [19] to target Android mobile devices by monitoring tap and swipe events as well as keystrokes, even deriving the words entered on the touchscreen.

   The library's *Prime+Probe* implementation was taken advantage of to perform the attacks, as such implementation had the functionality of performing the necessary platform's architecture instructions.

2. **Inter-cell communication -** To enable signaling between isolated cells, Jailhouse provides a communication channel through a shared memory region, called *ivshmem*. Device drivers get the relevant information by accessing custom PCI configuration space registers, being used too to state synchronization between cells. The aforementioned registers are the following:

   (a) IntrMask - Identifies the interrupt.

   (b) IntrStatus - Carries the message of the other guest's doorbell.

   (c) IVPosition - Reports the guest's ID number.

   (d) Doorbell - Carries the message and triggers an interrupt.

   To make use of the *ivshmem*, a driver is needed to map to userspace PCI device registers and memory regions. This driver, named *UIO_PC* [51], is a relatively recent driver model that seeks to move as much functionality into userspace as possible.

5.4.1 Simple Channel

1. **Attacker -** The first thing the attacker needs to do is to use UIO driver to map *ivshmem* device registers to userspace. This is needed to enable from the userspace the cell to use one of *ivshmem*'s registers (i.e., Doorbell register) to send an interrupt to the other cell.

Listing 5.21: First channel root cell code example.

```
/* Open uio_ivshmem device file */
fd = open("/dev/uio1", O_RDWR);
if (fd < 0) {
    exit(EXIT_FAILURE);
}
```

```

/* Map ivshmem registers --> /dev/uio1 maps[0] */
if ((virt_ivshmem_reg_address = mmap(NULL, IVSHMEM_REG_LENGTH,
PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)) == (void *) -1){
      close (fd);
      exit(EXIT_FAILURE);
}
```

The next step is to get the virtual address of the *ivshmem* memory region to send the agreed physical address for the other cell to access.

Listing 5.22: First channel root cell code example.

```
/* Open memory virtual file */
fd2 = open("/dev/mem", O_RDWR);
if (fd2 < 0) {
    exit(EXIT_FAILURE);
}

/* Map ivshmem adress */
void* virt_ivshmem_address = mmap(NULL, IVSHMEM_SIZE,
PROT_READ | PROT_WRITE, MAP_SHARED, fd2, IVSHMEM_ADDR);
if(virt_ivshmem_address==MAP_FAILED) {
  exit(EXIT_FAILURE);
}
```

Then, the agreed physical address is sent, but before that, the set index of the physical address is calculated, given by the formula: *set index = (physical address » log2 of line length) % number of sets*.

Listing 5.23: First channel code example.

```
/* Choose physical address */
uintptr_t phys_devmem_address = 0x70000000;
/* Get set index */
size_t set_index = (phys_devmem_address >> LINE_LENGTH_LOG2)
% NUMBER_OF_SETS;
/* Send physical address to ivshmem memory space */
*(volatile uintptr_t *)virt_ivshmem_address = phys_devmem_address;
```

After preparing the attack, the attacker starts to prime the cache set *set_index* using *libflush*'s *libflush_prime()*. After priming, it sends the interrupt by writing to one of *ivshmem*'s registers (i.e. Doorbell register).

The function of awaiting an interrupt from non-root cell is done using a blocking read function linked to *UIO_PCI*'s device file. When the interrupt is received, the attacker probes the cache using *libflush*'s *libflush_probe()*.

The cycle is repeated *NUMBER_OF_RUNS* times, and finally, the mean value of the probing time is calculated.

Listing 5.24: First channel root cell code example.

```
for(unsigned int run = 0; run < NUMBER_OF_RUNS; run++)
  {
      /* Prime cache */
      libflush_prime(libflush_session, set_index);
        /* Send interrupt to non-root cell */
      virt_ivshmem_reg_array[Doorbell/sizeof(int)] = msg;
      /* Wait for interrupt from non-root cell */
      rv = read(fd, &buf, sizeof(buf));
      if (rv == (ssize_t)sizeof(buf))
      {
            /* Probe cache */
            time = libflush_probe(libflush_session, set_index);
      }
      time_avg += time;
      usleep(100000);
  }
  /* Calculate the mean value of probing time */
  time_avg = time_avg / NUMBER_OF_RUNS;
```

2. **Victim -** The first thing the victim has to do, is to retrieve in the first interrupt occurrence the address that the attacker has shared through *ivshmem* to know which address to access.

   In the victim counterpart, the cache access occurs in a simple way. Using the inmate's library (provided in header file *inmate.h*) function *mmio_read32()*, it is possible to access directly a physical address as Erika v3 OS doesn't support MMU [52].

   Although the access is pretty straightforward, the victim has to have some cautions. In the case of accessing more than one line, the victim has to change the accessed address maintaining the cache set index (which corresponds to bits [15:5]). This is achieved by jumping to the next cache way. Since the way size of the L2 cache is 64KB, the jump is done by adding 64KB to the address.

Finally, after accessing the 16^{th} 64KB-spaced physical addresses (i.e. evict all cache lines from same cache set), the interrupt is sent by, once again, writing to the Doorbell register through *send_irq()* function.

Listing 5.25: First channel non-root cell code example.

```
/* Retrieve targeted address from ivshmem */
if(irq_occurence == 1) address = mmio_read32((void*)IVSHMEM_MEM_ADDR);
/* For 16 lines to access */
for (unsigned long v = 0; v < (16*LENGTH_64KB); v += LENGTH_64KB)
 {
     /* Access address */
     mmio_read32((void*)(unsigned long)(address + v));
 }
/* Send interrupt to root cell */
send_irq(d);
```

5.4.2 Bits Transmission Cannel

1. **Attacker -** Regarding the second attack, the same preparation steps (e..g, mapping and choosing target addresses) are done, with the exception that instead of one cache set, the attack must monitor two cache sets. So, two physical addresses are chosen based on their set index difference and sent to the non-root cell.

   The implementation of the attack is also similar to the one that was done on the first channel. The two chosen sets *set_index* and *set_index_2* are primed and probed one after another, and there is no calculation of the mean probing time for each set, because as the victim access is scheduled to emphasize the probing time difference between accessed and non-accessed set, there is no need to calculate a mean value.

Listing 5.26: First channel code example.

```
for(unsigned int run = 0; run < NUMBER_OF_RUNS; run++)
  {
      /* Prime cache */
      libflush_prime(libflush_session, set_index);
```

```
    libflush_prime(libflush_session, set_index_2);
    /* Send interrupt to non-root cell */
    virt_ivshmem_reg_array[Doorbell/sizeof(int)] = msg;
    /* Wait for interrupt from non-root cell */
    rv = read(fd, &buf, sizeof(buf));
    if (rv == (ssize_t)sizeof(buf))
    {
          /* Probe cache */
          time = libflush_probe(libflush_session, set_index);
          time_2 = libflush_probe(libflush_session, set_index_2);
    }
    usleep(100000);
}
/* Calculate the mean value of probing time */
time_avg = time_avg / NUMBER_OF_RUNS;
```

2. **Victim -** In the second channel's case, the access to different cache sets occurs in an alternate way, so for each interrupt, the victim accesses a different cache set. After retrieving both physical addresses that map to the targeted cache sets, the victim accesses the first cache set in case of an even interrupt occurrence, and the other cache set is accessed in case of an odd interrupt occurrence.

Listing 5.27: Second channel non-root cell code example.

```
/* Retrieve targeted address from ivshmem */
if(irq_occurence == 1) address = mmio_read32((void*)IVSHMEM_MEM_ADDR);
/* Access cache set 0 in case of even interrupt */
if(irq_occurence % 2 == 0)
{
    for (unsigned long v = 0; v < (16*LENGTH_64KB); v += LENGTH_64KB)
    {
        mmio_read32((void*)(unsigned long)(address + v));
    }
}
/* Access cache set 1 in case of odd interrupt */
else {
    for (unsigned long b = 0; b < (16*LENGTH_64KB); b += LENGTH_64KB)
    {
        mmio_read32((void*)(unsigned long)(address_2 + b));
    }
}
/* Send interrupt to root cell */
send_irq(d);
```

5.5 Countermeasures

The chosen strategy to mitigate the occurrence of side-channel attacks is the cache coloring approach, which assigns different colors to the available cores partitioning the LLC.

Colors Selection

Before implementing the cache coloring, the number of available colors needs to be known. To calculate the number of colors, the L2 cache architecture needs to be taken into account. The targeted board's L2 cache is 1MB-sized and its associativity is 16-way set-associative, which corresponds to a way size of 64KB. As such, the number of colors available is the number of pages that can fit in a way, which corresponds to 16 colors.

There is a precaution to not divide the L1 cache into more colored partitions. If the L1 is colored (i.e, the highest index bit is colored), the core will only have access to half of the available sets (there is no need to partition memory within a core since the attack is cross-core). Hence, there will be fewer lines to be used by the core, which will increase the contention between them on future accesses. This contention leads to an increase in the miss rate.

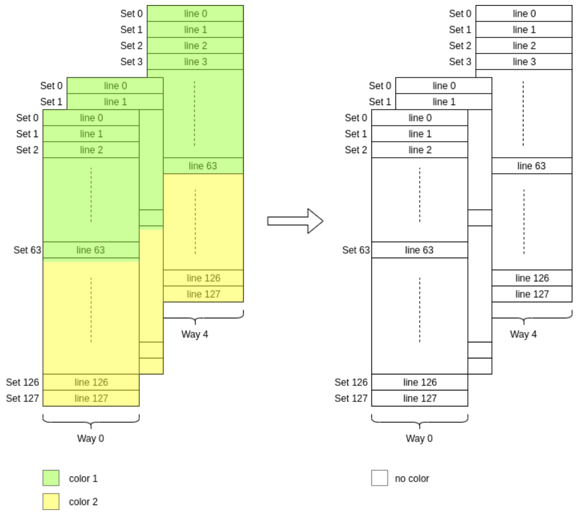


Figure 5.2: L1-I cache with the highest index bit colored (left picture) and without coloring (right picture).

To avoid the performance penalty of coloring, the colors are configured to only encompass the L2 cache. Thus, instead of the available 16 colors, there will be 8 colors that can only partition the L2 cache, as the Figure 5.3 illustrates.

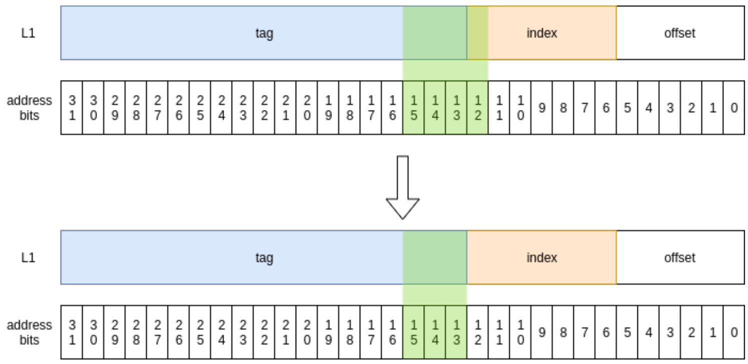


Figure 5.3: Coloring assignment with L1 cache coloring (above picture) and without L1 cache coloring (below picture).

Colored Pages Allocation

Once the number of available colors is known, the hypervisor proceeds to allocate the colored pages according to the VMs' configuration files. In Bao, since the VM and hypervisor images are loaded to certain contiguous physical memory spaces at the boot time (when U-Boot is running), they need to be re-colored [1] by the hypervisor.

As aforementioned, the color configuration is given by a bitmap at VM instantiation time. The hypervisor uses the color bitmap to search through the VM's memory region page pool, which keeps track of the colored pages. The search is done always by looking for the next page that fits in the colored mask and isn't colored. Once a color-compliant page has been found, the hypervisor maps the page.

Listing 5.28: Function that searches for the next available page.

```c
static inline uint64_t pp_next_clr(uint64_t base, int from, uint64_t colors){

    uint64_t clr_offset = (base/PAGE_SIZE) % (COLOR_NUM * COLOR_SIZE);
    uint64_t index = from;

    while(!((colors >> ((index + clr_offset) / COLOR_SIZE % COLOR_NUM)) & 1))
        index++;

    return index;
}
```

The function that is used to compute the next color-compliant page is described in Listing 5.28. It first needs to calculate the offset at which the page to be colored is, relatively to the beginning color mask. Then, it checks if the concerning page, represented by *index*, fits into the color mask. Since the allocation

[1] Re-coloring: process of moving uncolored memory spaces to colored memory spaces.

of color pages occurs by looking for the next color-compliant page, it is assumed that the *index* represents the last allocated page, and there are no allocated color pages ahead of it.

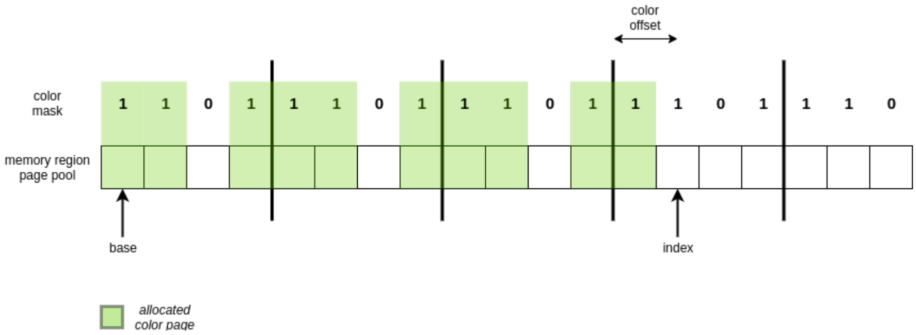


Figure 5.4: Hypervisor's search mechanism for the next colored page. The *base* value indicates the beginning of the pool page. The *index* value indicates the current page being checked on if it is color-compliant. The *color offset* value represents the offset of the *index* in relation to the beginning of the color mask. The *color mask* repeats itself every four pages. The *page pool* has 19 pages, of which 10 of them are already allocated.

6. Evaluation

In this chapter, the experimental results of the attacks are presented. The results have the purpose to confirm if there exists a channel between statically partitioned cores, or if there is no dependency between them. To evaluate the channels, different kinds of graphical representations have been used, namely the channel matrix and the graph line.

The first section of this chapter is dedicated to the description of the experimental setup used in the deployment of the attacks. Then, the results obtained for two tested eviction strategies are discussed. Using the most suitable eviction strategy, follows the deployment of the proposed attacks and the comparison to their counterparts with a mitigation strategy implementation.

6.1 Experimental Setup

The environment where the tests took place is characterized, essentially, by two main components: (i) the hardware where the attacks were deployed, and (ii) the software tools used to build the different VMs and libraries.

Test Device

The tests were done on the *ZCU104 Evaluation Kit* that features a *Zynq UltraScale+ MPSoC* device with support to many common peripherals and interfaces for embedded vision use case. The included *ZU7EV* device is equipped, among other processors, with a quad-core *Cortex-A53* applications processor and 16nm programmable logic. The Table 6.1 resumes all main characteristics of the *Cortex-A53 MPCore*.

Software Tools

To build the Linux kernel image it was used the PetaLinux Software Development Kit (SDK) (i.e., Petalinux tools 2018.2 release) that contains everything necessary to build, develop, test and deploy embedded Linux systems on ARM platforms. For Jailhouse v0.10, the image is built using SD Card as the root file system, while in Bao's case, the image is built using initramfs as the root file system. For the

Processor (cores)	Characteristics
ARM Cortex-A53 (4)	Architecture: Armv8-A Clock speed: 1536 MHz RAM: 2 GB L1 cache: 4 x 32 KB, 4-way, 128 sets L2 cache: 1 MB, 16-way, 1024 sets

Table 6.1: Hardware characteristics of *Cortex-A53* processor.

Linux image to be built against *ZCU104 Evaluation Kit* platform, it is complemented with *ZCU102 BSP* that contains device tree information, hardware description files, and other system setup files.

In regards to the ERIKA v3 image, it was used the Eclipse framework with RT-Druid as a plugin to build an ERIKA v3 image that includes the main application running on it.

To deploy the attacks, as it has been mentioned, it was used the library *libflush*. The library is compiled using Linaro's *aarch64-linux-gnu* v5.4.0 toolchain. Regarding the configuration of the library, it is set in *config.mk* file to use *zeroflte* device configuration and *perf* as time source.

In Bao, the device tree for Linux image is customized manually while Jailhouse's device tree is already built by Petalinux. The Device-tree Compiler (DTC) used to compile the customized Linux's device tree for Bao is *dtc* v1.4.5 while the toolchain used to build Bao is ARM's *aarch64-elf* v8.2.1 toolchain.

The heatmap results were obtained by using *Plotly Chart Studio* software, while the rest of the line graphs were obtained by using *LibreOffice Calc* and *Microsoft Excel*.

6.2 Eviction Strategy

To evaluate the eviction strategies, a line graph was used to display the distribution of the cache hit and cache miss results over the 100000 iterations done in the platform. The following results reveal the efficiency of the eviction strategies to evict cache sets and their implication to the success of the attack.

6.2.1 AutoLock's Eviction Strategy

The first eviction strategy, used by Irazoqui et al. [47], is illustrated in Figure 6.1. The graph reveals a considerable distribution of the results, which can be observed by the y-axis that shows at best 12154 (in 50000) occurrences for a certain probing time interval. While it is possible to distinguish between hit and miss results the majority of the time, some of them merge making it difficult to distinguish at times which one is which. This uncertainty when probing the cache isn't useful to the deployment of the attack as it doesn't establish a clear time border between cache misses and cache hits.

The success of the attack depends mostly on the capacity of the eviction strategy to evict a whole cache set. So, an eviction strategy that cannot distinguish with certainty a cache miss from a cache hit cannot be used to deploy the attack as it might lead to incorrect observations.

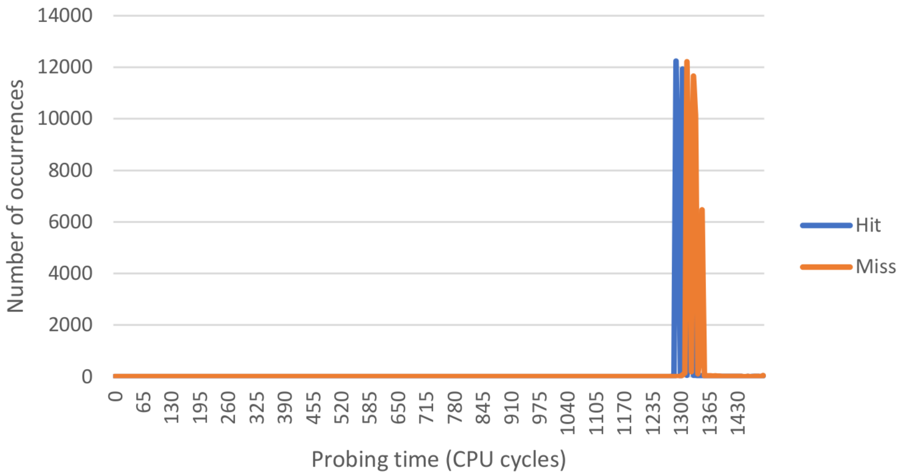


Figure 6.1: Line graph for cache timing results on *ZCU104 Evaluation Kit* using N-A-D 23-2-5. The x-axis represents the probing time that each cache miss or hit took and the y-axis represents the number of times that a cache miss or hit occurred.

6.2.2 ARMageddon's Eviction Strategy

The second eviction strategy, used by Lipp et al. [28], shows a much more concentrated distribution of the results as it can be seen in Figure 6.2. As the eviction strategy evicts more effectively the cache set, the cache hit and cache miss probing times appear to be more distant and concentrated, allowing a better realization to the attacker of the occurrence of cache hits and misses.

With this display of results, it is possible to trace a threshold value that can discern an observation of a cache miss or a cache hit. In this case, the value of 1130 CPU cycles could be safely chosen to infer, based on the probing time acquired value, the occurrence of a cache miss or cache hit.

Comparing to the first eviction strategy, it is possible to observe that the second eviction strategy is more suitable to be used in the attack deployment, as it reveals a better eviction efficiency.

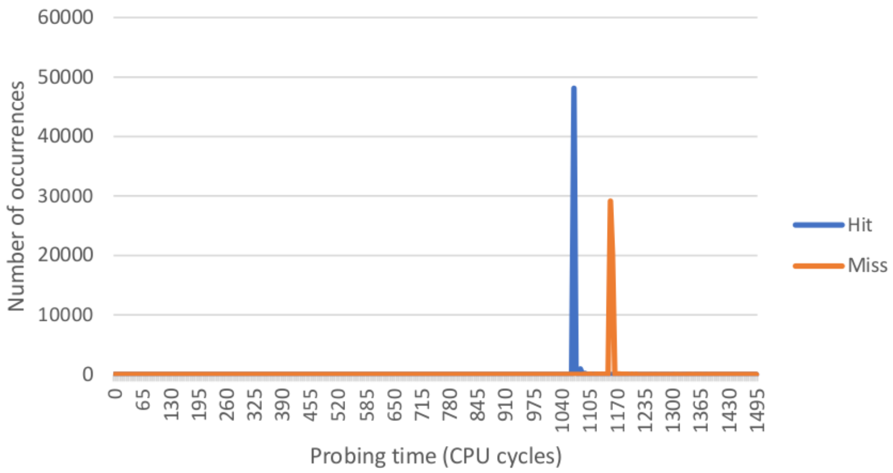


Figure 6.2: Line graph for cache timing results on *ZCU104 Evaluation Kit* using N-A-D 25-2-6. The x-axis represents the probing time that each cache miss or hit took and the y-axis represents the number of times that a cache miss or hit occurred.

6.3 Proposed Channels

6.3.1 Simple Channel

To evaluate the Simple Channel, it was used a heat map which specifies the dependency of an observed probing time given a certain number of accessed cache lines, where a brighter color represents a higher probability. To avoid the occurrence of false positives (i.e., observation of a probing time of fewer cache lines than expected) or false negatives (i.e., observation of a higher probing time than expected), 500 consecutive attacks were run. To a certain number of accessed cache lines from a certain cache set, 500 probing values are recorded and then it is calculated the number of occurrences of each value. Given the number of occurrences, follows the calculation of their probability of occurring within a 10 CPU cycles interval. This probability is represented in the heat map using a sequence of darker to brighter colors (i.e., represented in z-axis).

Bao

The heat map of the observed channel on Bao is illustrated in Figure 6.3. The observation of the heat map shows the existence of a well-defined channel that establishes the correlation between cache lines accessed and probing time (i.e., a directly proportional relation). The more the number of cache lines the victim accesses, the more is the probability of the attacker experiencing a greater probing time.

This probability is associated with the brighter color that demonstrates horizontal variation across the heat map.

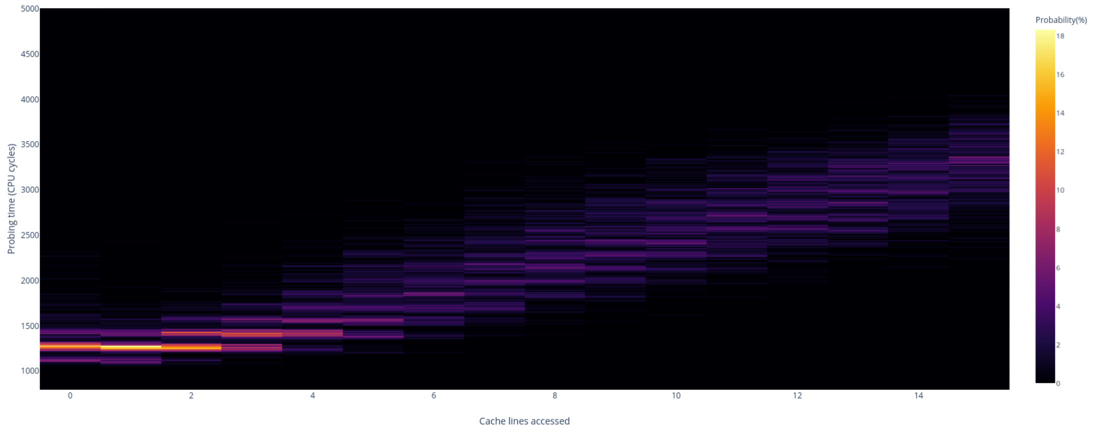


Figure 6.3: Channel matrix for the unmitigated LLC channel on *ZCU104 Evaluation Kit* using a bare metal guest as the victim. The x-axis represents the number of accessed cache lines, the y-axis represents the probing time of each attack run and the z-axis represents the probability of the probing time value happen given a certain number of cache lines accessed by the victim.

Since the attacker previously prepares the shared cache set with known cache lines (i.e., priming the cache), it will wait for the victim to access those lines. As the victim accesses more cache lines from the shared cache set, the number of evicted lines from that cache set will increase. Thus the attacker, when trying to access again the cache set (i.e., probing the cache), will experience more cache misses. The increase in the number of cache misses means naturally a greater probing time. This dependency between the attacker's probing time and the victim's number of cache lines might lead to a leak of critical information of co-located VMs that only share a cache level.

An observation that can be made is the distribution of probing time values for a certain number of cache lines accessed. To use as an example, for 8 cache lines accessed by the victim, the attacker might obtain probing time values that range from 1500 CPU cycles to 3000 CPU cycles. This distribution might be caused by the pseudo-random policy employed by the L2 cache. The more the number of cache lines accessed, the higher the probability of newly-allocated cache lines evict the other victim accessed cache lines. The *Prime+Probe* runs that measured 3000 CPU cycles could have occurred in a case where the victim could allocate successfully the 8 cache lines in the cache set, while the runs that measured 1500 CPU cycles could have occurred in cases where most of the 8 cache lines accessed by the victim would be evicted by the newly-allocated victim cache lines. This phenomenon could explain too the concentration of results when fewer cache lines are accessed by the victim. The lesser the number of cache lines accessed by the victim, the lesser is the chance of occurring self-eviction. Thus, the probability of obtaining similar values is higher. As an example, for 1 cache line accessed by the victim, the probability of the attacker obtaining probing times between 1350 and 1360 CPU cycles is 18%. For a bigger number of cache lines accessed, there is no other interval with such probability, while for less number of cache lines accessed, there are more intervals with probability close to 18%.

It can also be observed from the heat map that the probing time of the cache sets is much higher than the expected probing time given the cache latency values (i.e., 16 lines accessed from L2 are reported to have 16x15 cycles of latency). This increase in probe time is caused by the use of the *perf* tool instead of directly accessing the PMU counter values. The *perf* tool uses system calls (e.g. read syscall) to read the PMU-based hardware counters, which introduces some overhead to the measurements [49].

6.3.2 Bits Transmission Channel

To evaluate the Bits Transmission Channel, it was used a line chart that overlaps two different measures done to two different cache sets consecutively. The chart compares the probing time of different cache sets in each attack run, revealing which set was accessed and what not during each attack iteration.

Jailhouse

The line graph, illustrated in Figure 6.4, describes two lines that represent the probing time of two sets throughout 50 *Prime+Probe* iterations. During each iteration, one cache set has a higher probing time than the other. The cache set with the higher probing time has the most evicted lines by the victim. This way, the Bits Transmission Channel serves as a channel that leverages the directly proportional relation verified by the Simple Channel to infer, between 2 cache sets, which one is the victim accessed cache set.

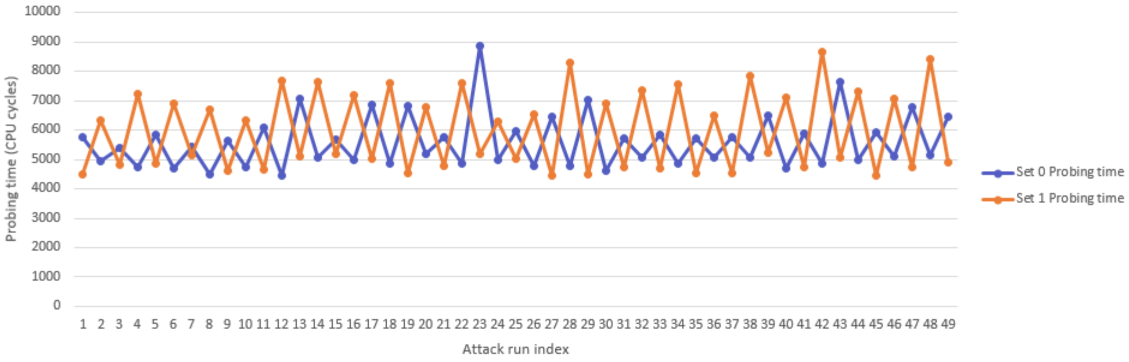


Figure 6.4: Sample sequence of attacker's access time on *ZCU104 Evaluation Kit*. The blue line represents the probing time of set *0*, while the yellow line represents the probing time of set *1*. The x-axis represents the index of the attack run and the y-axis represents the probing time of each attack run.

This channel simulates the transmission of bits between victim and attacker by checking the probe time of different sets. The victim was designed to alternate the access to two agreed sets, leaking this way information to the attacker of which set the victim accessed. To emphasize the probing time difference (i.e., make the leak information more noticeable) between accessed and not accessed cache set, the victim accesses 16 cache lines of the intended cache set and the other cache set is not accessed.

In this particular case, the chart reveals that the peaks of cache set *0* occur during the troughs of cache set *1* and vice-versa. The chart reveals too, that the first accessed cache set by the victim is cache set *0*, which then alternates the access with the other set, transmitting the sequence "*01010...*".

An observation that can be made is the relative non-linearity between iterations. To use as an example, the 25^{th} iteration demonstrates a much higher probing time difference between cache sets than the 3^{th} iteration. While this specific example could infer correctly to the attacker the cache set accessed by the victim, it might lead to wrong assumptions about the victim's activity. This phenomenon, transverse to all the cases, could be caused by multiple factors: (i) the first supposition, derives from the problem mentioned in Section 3.2 that, in some cases, higher-level caches might already have cache lines that map to the targeted cache set which leads to a lower probing time, and (ii) the second supposition, that explains the increase in the measured values, relies on the eviction strategy's inefficacy to evict a whole cache set that might lead to the mistaking of cache hits with cache misses when probing. Previously (see Section 3.2.1.2), it has been discussed that some eviction strategies might not have an eviction rate of 100%, which can cause the non-eviction of some cache lines by the attacker. Then, when the probe step occurs, the non-evicted cache lines can be perceived as cache misses by the attacker when, in fact, were cache lines that weren't evicted by the attacker. This imperceptible cache miss might lead to a misunderstood accessed line by the victim, which will cause an increase in the probe time that wasn't caused by the victim's activity but by the eviction's strategy inefficacy to evict a whole cache set.

6.4 Countermeasures

6.4.1 First Channel

Bao

The heat map of the first channel with the application of the cache coloring countermeasure is illustrated in Figure 6.5. It is noticeable the lack of horizontal variation when different cache lines of the same set are accessed, which means that the outputs are independent of inputs. So, it can be concluded that with the implemented mitigation strategy the channel stops existing.

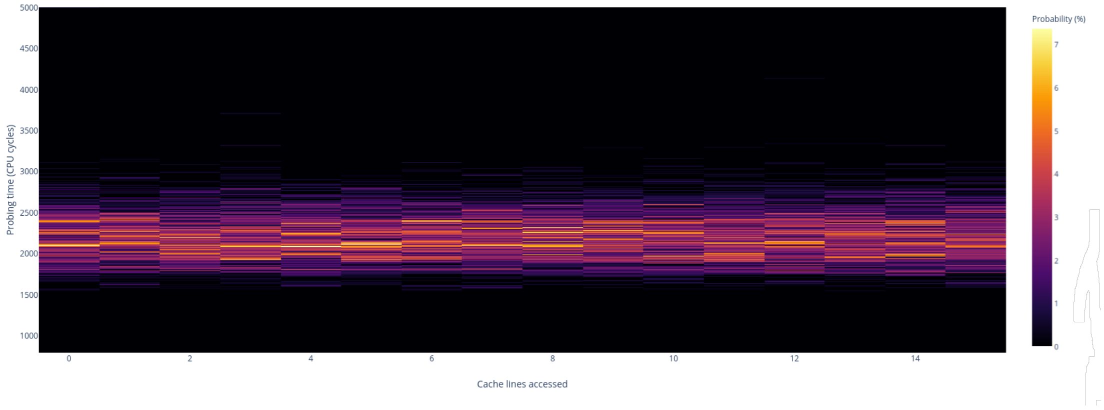


Figure 6.5: Channel matrix for the mitigated LLC channel on *ZCU104 Evaluation Kit* using a bare metal guest as the victim. The x-axis represents the number of accessed cache lines, the y-axis represents the probing time of each attack run and the z-axis represents the probability of the probing time value happen given a certain number of cache lines accessed by the victim.

The coloring partitioning scheme has the particularity of isolating co-located VMs in detriment of the cache set accessible space's reduction. The hardware isolation between partitions is achieved by giving different colors to the cache-contending partitions. When the attacker and the victim have different colors, they don't contend for the same cache lines. So, the victim cannot evict cache lines from the attacker and vice-versa. This means that, when the attacker primes the shared cache set, the victim won't be able to access the cache lines that were previously accessed by the attacker. Hence, when the attacker probes the cache set, it will find no cache misses because the cache lines weren't evicted by the victim. This independence between the attacker and the victim's cache activity is represented by the invariance of the attacker's probing time given the number of the victim's accessed cache lines. Since the probability of the attacker obtaining a probe time between 1750 and 2500 CPU cycles is independent of the caches lines accessed by the victim, it can be concluded that the attacker cannot infer any cache activity by the victim. This means that the coloring strategy guarantees determinism at the cache level.

Although the cache coloring strategy guarantees determinism, it comes with a cost. As the cache set accessible space is reduced by half to each partition (i.e., the attacker owns half of the colors, while the victim owns the other half), more likely will be for the partitions to access cache lines that map to the same cache set. Thus, the probability of newly-allocated cache lines evicting other allocated cache lines increases. This leads to the deterioration of the victim's performance, but there is another factor that needs to be taken into account when considering the effects of the color partitioning in the system's performance. Even though the cache's space is reduced, there won't be more victim's evictions forced by the attacker's cache activity. Hence, there is a decrease in the miss rate due to the lack of contention on the cache sets. It can be concluded that coloring efficacy is a delicate balance between the degradation caused by cache size reduction and the isolation enhancement introduced by cache partitioning.

To evaluate the implemented system's performance on *ZCU104 Evaluation Kit*, the LMBench's *lat_mem _rd* benchmark was used. A partition measures the latency of a memory read, using *lat_mem _rd* benchmark, while the other partition accesses the cache. The *lat_mem _rd* benchmark is employed to read a 20MB memory area with a 64B stride distance. The behavior of the system with different configurations is illustrated in Figure 6.6.

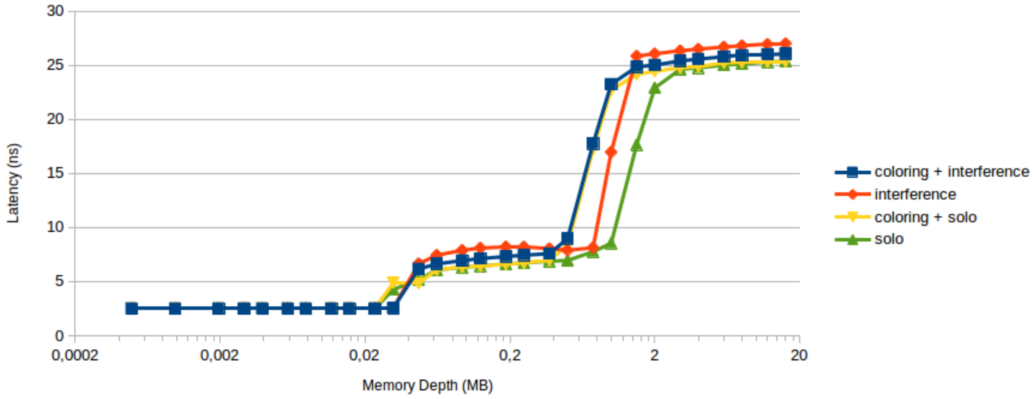


Figure 6.6: Memory latency of a partition on *ZCU104 Evaluation Kit*. The blue line represents a system with coloring and interference of the other partition. The red line represents the system with interference and without coloring. The yellow line represents a system with coloring and without interference. The green line represents a system without coloring and interference. The x-axis represents the depth of the memory accessed by the partition, and the y-axis represents the memory read latency for a certain memory depth.

The logarithmic graph describes a similar memory read latency when there is a coloring scheme implemented, whether there is interference or not. When there is a co-located VM polluting the cache (i.e., blue line), the other partition experiences the same memory read latency as if it was running solo (i.e., red line). This behavior resemblance is represented in the line graph by the overlapping of blue and red lines, which means that a critical partition (i.e., victim) runs in a deterministic fashion whether there is another partition (i.e., attacker) using the cache or not. Thus, this enforces the idea of determinism given by the color partitioning scheme.

The graph describes also an increase in the memory read latency when the coloring strategy is implemented. The partitions that are isolated between each other (i.e., blue and yellow lines) start to have an increase in the memory read latency at a lower memory depth than the non-isolated ones (i.e., red and green line). While the red and green lines start to increase their memory read latency when the L2 cache size is achieved (i.e., 1MB), the blue and yellow lines start to increase their latency when half of the L2 cache size is achieved (i.e., 512KB). This happens due to the reduction of the accessible cache set size given by the color partitioning.

7. Conclusion

This dissertation presents microarchitectural attacks that allow making a side-channel analysis based on the timing behavior of the cache. The presented attacks have been done under certain conditions that resemble a real-world application in the context of the automotive industry. The attacks occur in systems where VMs with different critical levels (i.e., mixed-criticality systems) are assigned to different cores within the same processor. The VMs are also isolated between each other by software, and this implies: (i) no memory sharing, (ii) full concurrency, and (iii) cache sharing between cores, as it has been described in the Analysis chapter (see Section 3.1).

Regarding the proposed goals, they have been achieved with success as they proved the lack of determinism and insecurities of mixed-criticality systems supported by so-called isolation enforcer hypervisors, using the memory hierarchy as a communication medium between guests. The cache coloring countermeasure proved to be an effective but performance-wise costly solution to these attacks, as it increased the memory latency of both partitions.

7.1 Future Work

There are many suggestions to hamper the occurrence of attacks and many ways to improve the attack's speed and resolution. The presented attacks in this dissertation relied on a library that used an eviction strategy to prime a cache set, but it is known that this isn't the fastest neither most effective way to evict cache lines from a set in the ARMv8-A architecture. To improve the attack's efficiency, the author proposes to implement the same attacks using the provided ARMv8-A cache flush instructions instead of an eviction strategy.

Two other improvements concerning the *libflush* library are suggested. The first one is related to the use of huge pages to directly target PIPT and VIVT caches without the need of knowing the address translations. The *libflush* library leverages a given privileged access to *proc/self/maps* file to see the translations that allow the user to access a specific cache set, while the use of huge pages has the advantage of not relying on any privilege mode to access a specific cache set. The other suggestion is related to the way that the probe time is measured. To avoid the noise introduced by the interaction with higher-level caches, the probing time can be acquired by measuring the time of every load from the eviction set, instead of measuring the total probe time. This way, the attacker knows from which memory level each line has been accessed (i.e., L1, L2 or main memory).

To give use to these attacks, the author has already implemented a rudimentary transmission channel that simulates the leak of information through cache observation (i.e., Bits Transmission Channel). So, the author proposes to extrapolate even more the application of these attacks and decode co-located VM information exploiting speculative execution [53] or out-of-order execution [8]. To mitigate these attacks, the author recommends the future Linux releases to not give userspace access to the PMU values through system calls as they expose a lot of the hardware's activity. Although there are other userspace ways to time cache accesses documented in this dissertation, there is no other known userspace tool that can be as precise as *perf*.

Bibliography

[1] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware,” *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.

[2] G. Heiser, “The role of virtualization in embedded systems,” in *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pp. 11–16, ACM, 2008.

[3] M. Strobl, M. Kucera, A. Foeldi, T. Waas, N. Balbierer, and C. Hilbert, “Towards automotive virtualization,” in *Applied Electronics (AE), 2013 International Conference on*, pp. 1–6, IEEE, 2013.

[4] S. Pinto and N. Santos, “Demystifying arm trustzone: A comprehensive survey,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, p. 130, 2019.

[5] A. Biondi, M. Marinoni, G. Buttazzo, C. Scordino, and P. Gai, “Challenges in virtualizing safety-critical cyberphysical systems,” in *Proceedings of Embedded World Conference*, pp. 1–5, 2018.

[6] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, “Bao: a lightweight static partitioning hypervisor for modern multi-core embedded systems,” in *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, (Bologna, Italy), to appear 2020.

[7] J. Martins, J. Alves, J. Cabral, A. Tavares, and S. Pinto, “μrtzvisor: A secure and safe real-time hypervisor,” *Electronics*, vol. 6, no. 4, p. 93, 2017.

[8] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown,” *arXiv preprint arXiv:1801.01207*, 2018.

[9] Q. Ge, Y. Yarom, F. Li, and G. Heiser, “Your processor leaks information-and there’s nothing you can do about it,” *arXiv preprint arXiv:1612.04474*, 2016.

[10] Q. Ge, Y. Yarom, and G. Heiser, “No security without time protection: We need a new hardware-software contract,” in *Proceedings of the 9th Asia-Pacific Workshop on Systems*, p. 1, ACM, 2018.

[11] J. Van Bulck, F. Piessens, and R. Strackx, “Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 178–195, ACM, 2018.

[12] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, “SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems,” in *IEEE Symposium on Security and Privacy (S&P)*, (Los Alamitos, CA, USA), to appear 2020.

[13] P. C. Kocher, “Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems,” in *Annual International Cryptology Conference*, pp. 104–113, Springer, 1996.

[14] P. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” in *Annual International Cryptology Conference*, pp. 388–397, Springer, 1999.

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

[16] Y. Yarom and K. Falkner, “Flush+ reload: A high resolution, low noise, l3 cache side-channel attack.,” in *USENIX Security Symposium*, vol. 1, pp. 22–25, 2014.

[17] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-level cache side-channel attacks are practical,” in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 605–622, IEEE, 2015.

[18] G. Irazoqui, T. Eisenbarth, and B. Sunar, “Cross processor cache attacks,” in *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pp. 353–364, ACM, 2016.

[19] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, “Armageddon: Cache attacks on mobile devices.,” in *USENIX Security Symposium*, pp. 549–564, 2016.

[20] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, “Virtualization on trustzone-enabled microcontrollers? voilà!,” in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 293–304, IEEE, 2019.

[21] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares, “Towards a trustzone-assisted hypervisor for real-time embedded systems,” *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 158–161, 2016.

[22] S. Pinto, A. Tavares, and S. Montenegro, “Space and time partitioning with hardware support for space applications,” *Data Systems in Aerospace, European Space Agency, ESA SP*, vol. 736, 2016.

[23] S. Pinto, A. Oliveira, J. Pereira, J. Cabral, J. Monteiro, and A. Tavares, “Lightweight multicore

virtualization architecture exploiting arm trustzone," in *IECON 2017-43rd Annual Conference of the IEEE Industrial Electronics Society*, pp. 3562–3567, IEEE, 2017.

[24] M. Mounika and C. Chinnaswamy, "A comprehensive review on embedded hypervisors," vol. 5, no. 5, 2016.

[25] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, 2013.

[26] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral, "Ltzvisor: Trustzone is the key," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[27] U. Drepper, "The cost of virtualization.," *Acm Queue*, vol. 6, no. 1, pp. 28–35, 2008.

[28] M. Lipp, *Cache attacks on arm*. PhD thesis, Master thesis, Graz, University Of Technology, 2016.

[29] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look mum, no vm exits!(almost)," *arXiv preprint arXiv:1705.06932*, 2017.

[30] M. Baryshnikov, "Jailhouse hypervisor, bachelor project," 2016.

[31] J. Kiszka, "Hard partitioning for linux: The jailhouse hypervisor," *http://events.linuxfoundation.org/sites/events/files/slides/LinuxConNA-2015-Jailhouse.pdf*, 2015.

[32] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic memory hierarchy and virtualization for modern multi-core embedded systems," in *25th IEEE real-time and embedded technology and applications symposium, RTAS*, 2019.

[33] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 249–266, 2019.

[34] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling (2019)," *arXiv preprint arXiv:1905.05726*, 2019.

[35] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 991–1008, 2018.

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

[37] S. E. J. W. R. Trimble, W. Oblitey, *Covert Storage Channels: A Brief Overview.* 2013.

[38] “Arm cortex-a53,” *https://www.7-cpu.com/cpu/Cortex-A53.html*, 2018.

[39] D. H. Woo and H. Lee, “Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors,” in *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[40] D. J. Bernstein and P. Schwabe, “A word of warning,” in *Workshop on Cryptographic Hardware and Embedded Systems*, vol. 13, 2013.

[41] A. Langley *https://github.com/agl/ctgrind*, 2010.

[42] R. Zhang, X. Su, J. Wang, C. Wang, W. Liu, and R. W. Lau, “On mitigating the risk of cross-vm covert channels in a public cloud,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 8, pp. 2327–2339, 2014.

[43] D. Cock, Q. Ge, T. Murray, and G. 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*, pp. 570–581, ACM, 2014.

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

[45] V. Varadarajan, T. Ristenpart, and M. Swift, “Scheduler-based defenses against cross-vm side-channels,” in *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 687–702, 2014.

[46] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, “Prefetch side-channel attacks: Bypassing smap and kernel aslr,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 368–379, ACM, 2016.

[47] M. Green, L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth, “AutoLock: Why Cache Attacks on ARM Are Harder Than You Think,” in *26th USENIX Security Symposium (USENIX Security 17)*, pp. 1075–1091, 2017.

[48] D. Gruss, C. Maurice, and S. Mangard, “Rowhammer. js: A remote software-induced fault attack

in javascript," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 300–321, Springer, 2016.

[49] V. M. Weaver, "Linux perf_event features and overhead," in *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, vol. 13, 2013.

[50] *Zynq UltraScale+ Device Technical Reference Manual*. 2018.

[51] H. Schild *https://github.com/henning-schild-work/ivshmem-guest-code/*, 2018.

[52] P. Gai, "Mmu support in erika," *http://www.erika-enterprise.com/forum/viewtopic.php?t=1230*, 2019.

[53] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.