Principled Elimination of Microarchitectural Timing Channels through Operating-System Enforced Time Protection

Qian Ge

Submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy

School of Computer Science and Engineering

University of New South Wales

Sydney, Australia

October 2019

Thesis/Dissertation Sheet

Surname/Family Name	:	**Ge**
Given Name/s	:	**Qian**
Abbreviation for degree as give in the University calendar	:	**PhD**
Faculty	:	**Engineering**
School	:	**Computer Science and Engineering**
Thesis Title	:	**Principled Elimination of Microarchitectural Timing Channels through Operating-System Enforced Time Protection**

Abstract 350 words maximum: (PLEASE TYPE)

Microarchitectural timing channels exploit resource contentions on a shared hardware platform to cause information leakage through timing variance. These channels threaten system security by providing unauthorised information flow in violation of the system's security policy. Present operating systems lack the means for systematic prevention of such channels. To address this problem, we propose time protection as an operating system (OS) abstraction, which provides mandatory temporal isolation analogous to the spatial isolation provided by the established memory protection abstraction.

In order to fully understand microarchitectural timing channels, we first study all published microarchitectural timing attacks, their countermeasures and analyse the underlying causes. Then we define two application scenarios, a confinement scenario and a cloud scenario, which between them represent a large class of security-critical use cases, and aim to develop a solution that supports both.

Our study identifies competition for limited hardware resources as the underlying cause for microarchitectural timing channels. From this we derive the requirement that proper isolation requires that all shared resources must be partitioned, either spatially or temporally (time-shared). We then analyse a number of recent processors across two instruction-set architectures (ISAs), x86 and Arm, for their support for such partitioning. We discover that all examined processors exhibit hardware state that cannot be partitioned by architected means, meaning that they all have uncloseable channels. We define the requirements hardware must satisfy for timing-channel prevention, and propose an augmented ISA as a new, security-oriented hardware-software contract.

Assuming conforming hardware, we then define the requirements that OS-provided time protection must satisfy. We propose a concrete design of time protection, consisting of a set of policy-free mechanisms, and present an implementation in the seL4 microkernel. We evaluate the efficacy and efficiency of the implementation, and show that it is highly effective at closing timing channels, to the degree supported by the underlying hardware. We also find that the performance overheads are small to negligible. We can conclude that principled prevention of timing channels is possible though mandatory, black-box enforcement by the OS, subject to hardware manufacturers providing mechanisms for scrubbing all shared microarchitectural state.

The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years must be made in writing. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

FOR OFFICE USE ONLY	Date of completion of requirements for Award:

ORIGINALITY STATEMENT

INCLUSION OF PUBLICATIONS STATEMENT

UNSW is supportive of candidates publishing their research results during their candidature as detailed in the UNSW Thesis Examination Procedure.

Publications can be used in their thesis in lieu of a Chapter if:

- The student contributed greater than 50% of the content in the publication and is the "primary author", ie. the student was responsible primarily for the planning, execution and preparation of the work for publication
- The student has approval to include the publication in their thesis in lieu of a Chapter from their supervisor and Postgraduate Coordinator.
- The publication is not subject to any obligations or contractual agreements with a third party that would constrain its inclusion in the thesis

Please indicate whether this thesis contains published material or not.

☐ *This thesis contains no publications, either published or submitted for publication (if this box is checked, you may delete all the material on page 2)*

☒ *Some of the work described in this thesis has been published and it has been documented in the relevant Chapters with acknowledgement (if this box is checked, you may delete all the material on page 2)*

☐ *This thesis has publications (either published or submitted for publication) incorporated into it in lieu of a chapter and the details are presented below*

CANDIDATE'S DECLARATION

I declare that:

- I have complied with the Thesis Examination Procedure
- where I have used a publication in lieu of a Chapter, the listed publication(s) below meet(s) the requirements to be included in the thesis.

Name **QIAN GE**	**Signature**	**Date (dd/mm/yy)**

Postgraduate Coordinator's Declaration (to be filled in where publications are used in lieu of Chapters)

I declare that:

- the information below is accurate
- where listed publication(s) have been used in lieu of Chapter(s), their use complies with the Thesis Examination Procedure
- the minimum requirements for the format of the thesis have been met.

PGC's Name	**PGC's Signature**	**Date (dd/mm/yy)**

Abstract

Microarchitectural timing channels exploit resource contentions on a shared hardware platform to cause information leakage through timing variance. These channels threaten system security by providing unauthorised information flow in violation of the system's security policy. Present operating systems lack the means for systematic prevention of such channels. To address this problem, we propose *time protection* as an operating system (OS) abstraction, which provides mandatory temporal isolation analogous to the spatial isolation provided by the established memory protection abstraction.

In order to fully understand microarchitectural timing channels, we first study all published microarchitectural timing attacks, their countermeasures and analyse the underlying causes. Then we define two application scenarios, a confinement scenario and a cloud scenario, which between them represent a large class of security-critical use cases, and aim to develop a solution that supports both.

Our study identifies competition for limited hardware resources as the underlying cause for microarchitectural timing channels. From this we derive the requirement that proper isolation requires that all shared resources must be partitioned, either spatially or temporally (time-shared). We then analyse a number of recent processors across two instruction-set architectures (ISAs), x86 and Arm, for their support for such partitioning. We discover that all examined processors exhibit hardware state that cannot be partitioned by architected means, meaning that they all have uncloseable channels. We define the requirements hardware must satisfy for timing-channel prevention, and propose an *augmented ISA* as a new, security-oriented hardware-software contract.

Assuming conforming hardware, we then define the requirements that OS-provided time protection must satisfy. We propose a concrete design of time protection, consisting of a set of policy-free mechanisms, and present an implementation in the seL4 microkernel. We evaluate the efficacy and efficiency of the implementation, and show that it is highly effective at closing timing channels, to the degree supported by the underlying hardware. We also find that the performance overheads are small to negligible. We can conclude that principled prevention of timing channels is possible though mandatory, black-box enforcement by the OS, subject to hardware manufacturers providing mechanisms for scrubbing all shared microarchitectural state.

Contents

Acknowledgements

I would like to thank my supervisor, Gernot Heiser, for his support, guidance, and patience during my PhD. My PhD topic, principled elimination of microarchitectural timing channels through operating-system enforced time protection, is extremely challenging, and this work would not be possible without his extremely high standard and wise insight. Being his student was very challenging, but also rewarding.

Thanks to my partner, Adrian Danis, who gave me tremendous encouragement during those very dark days while I was struggling with both work and life. Thanks also to my families, Lihua Zhang, Xingjie Ge, Jiale Li, Lijuan Zhang, and Lili Zhang, for their great support and understanding.

Thanks to past and current students, engineers and researchers in the Trustworthy Systems team, who have helped me: Peter Chubb, Anna Lyons, Luke Mondy, and Yuval Yarom.

Special thanks to my friends: Cai Li, and Lei Zeng, for being there at the moment I needed it the most.

Particularly, I would like to thank those who helped with proof reading my thesis draft: Peter Chubb, Anna Lyons, Branden Robinson, Victor Phan, Sylvain Gauthier, Oliver Scott, Damon Lee, Amos Robinson, Santiago Bautista, Joel Beeren, Alex Legg and my supervisor, Gernot Heiser.

Bow to all of you, as the most formal way to show my respect.

Publications

This thesis is partially based on work described in the following publications:[1]

- Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: the missing OS abstraction. In *EuroSys Conference*, Dresden, Germany, March 2019. ACM. **Best Paper Award**.
  Citation count: 9.
- Qian Ge, Yuval Yarom, and Gernot Heiser. No security without time protection: We need a new hardware-software contract. In *Asia-Pacific Workshop on Systems (APSys)*, Korea, August 2018b. ACM SIGOPS. **Best Paper Award**.
  Citation count: 9.
- Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8:1–27, April 2018a.
  Citation count: 135.

Other papers published during my PhD:

- Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE Symposium on High-Performance Computer Architecture*, pages 406–418, Barcelona, Spain, March 2016. IEEE.
  Citation count: 145.
- Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, San Jose, CA, US, May 2015. IEEE.
  Citation count: 455.
- David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of some timing channels on seL4. In *ACM Conference on Computer and Communications Security*, pages 570–581, Scottsdale, AZ, USA, November 2014. ACM.
  Citation count: 65.

[1]The citation counts are based on statistics offered by Google scholar on September 2019.

Acronyms

AES avanced encryption standard. 30, 40–45, 51

aISA augmented ISA. 83–85, 103, 104

ALU arithmetic logic unit. 18

APIC advanced programmable interrupt controller. 120

ASID address space ID. 8, 9, 13, 24, 25, 110, 112–114, 123, 124

ASLR address space layout randomization. 31, 39–42

BHB branch history buffer. 13, 17, 20, 37, 39, 43, 56, 59, 64, 65, 68, 69, 78, 81–83, 104, 117, 125, 128, 133

BPU branch prediction unit. 48, 117

BTB branch target buffer. 13, 17, 20, 29, 37, 40, 43, 56, 59, 64, 67–69, 76, 78, 80, 82, 83, 104, 117, 125, 128, 132

CAT cache allocation technology. 50, 161, 162

CPU central processing unit. 1, 2, 10, 13–18, 21–23, 30, 37–40, 43, 46–48, 50, 65, 69, 72, 81, 120, 150, 155, 159, 162

DES data encryption standard. 51

DoS denial-of-service. 31, 34, 56

DRAM dynamic random-access memory. 10, 17, 18, 29, 32–34, 56

ECC error-correcting code. 18, 32, 34

ECDSA elliptic curve digital signature algorithm. 29, 42

FIFO first in, first out. 117

FPU floating point unit. 17, 37, 43

GPU graphics processing unit. 10, 18, 33

HTTP HyperText Transfer Protocol. 54

IaaS infrastructure-as-a-service. 1

IBC indirect branch control. 5, 49, 71, 78–83, 117, 161

IBS instruction-based scheduling. 46, 47

IPC inter-process communication. 22–25, 27, 108, 110, 149, 150

IPI inter-processor interrupt. 41, 123

IRQ interrupt request. 25, 107, 112, 114, 148

ISA instruction-set architecture. 4, 5, 8–10, 63, 64, 83, 85, 103, 157, 160

L1 first-level. 11–14, 17, 18, 20, 43, 49–51, 53, 56, 58, 59, 72, 85, 86, 103, 104, 117, 125–127, 139, 150, 153

L1-D first-level data. 10, 11, 15, 16, 29, 36–38, 40, 49, 51, 64–67, 69, 72, 78–80, 82, 85, 86, 105, 117, 127–129, 134, 150, 151

L1-I first-level instruction. 10, 11, 29, 37, 41, 51, 64, 66, 67, 69, 72–74, 79, 80, 82, 83, 85, 117, 127, 128, 131, 133, 150, 151

L2 second-level. 10, 11, 13, 14, 17, 18, 43, 46, 60, 72, 104, 126, 128, 129, 134, 138, 145, 149–155, 157, 161

L3 third-level. 10, 11, 13, 17, 21, 36, 126, 150, 151, 153

LLC last-level cache. 10, 16, 20, 21, 31, 34–36, 41–43, 50, 51, 53, 56, 59, 60, 83–86, 104, 126, 127, 142, 148, 161

LRU least-recently used. 78, 117

MBA memory bandwidth allocation. 32

MI mutual information. 63, 74–82, 128, 130–132, 136–139, 144–146, 148

MLS multi-level secure. 61

MMU memory management unit. 8, 13, 31, 38

MSR machine state register. 71

NUMA non-uniform memory access. 106

OS operating system. 1–4, 8, 9, 19–21, 30, 37–39, 42, 48, 49, 53–61, 63, 65, 77, 78, 83–85, 103, 104, 120, 153, 158–162

PaaS platform-as-a-service. 42

PARA probabilistic adjacent row activation. 33, 34

PIPT physically-indexed, physically-tagged. 12, 13, 33

PIVT physically-indexed, virtually-tagged. 12

PMU performance monitoring unit. 47

POSIX portable operating system interface. 153

QoS quality of service. 34, 50, 56

RIDL rogue in-flight data load. 39

RPcache random permutation cache. 45, 46

RSA Rivest-Shamir-Adleman. 31, 32, 34, 40–42

RSB return stack buffer. 40, 43

SGX software guard extensions. 38, 39

SMM system management mode. 38

SMT simultaneous multithreading. 16, 17, 56

SoC system on a chip. 33

SP stream processor. 18

SVG scalable vector graphics. 30

TCB thread control block. 24, 25, 105, 107, 108, 110, 112, 116, 142

TIFF tagged image file format. 154

TLB translation look-aside buffer. 2, 13, 17, 20, 21, 29, 30, 37, 40, 42, 43, 48, 51, 53, 56, 59, 64, 67, 69, 71, 72, 76, 80, 82, 85, 103, 104, 117, 123, 128, 131, 132, 150, 155, 157

TP texture processor. 18

TPR task priority register. 120

TSX transactional synchronization extension. 31, 36, 42, 43

VIPT virtually-indexed, physically-tagged. 12

VIVT virtually-indexed, virtually-tagged. 12, 13

VM virtual machine. 1, 38, 39, 41–43, 45, 46, 49, 50, 52, 55, 57, 58, 86, 104, 148, 159

VMM virtual machine monitor. 1, 38, 85

List of Figures

List of Tables

Listings

1 Introduction

To achieve contiguous growth of compute power, computer architectures have become increasingly complicated as a result of consolidating more and more functional features on hardware. Currently, central processing units (CPUs) are equipped with various functional units, such as multi-stage pipelines (e.g., commonly more than 10 stages on Intel processors [Intel, c]), sophisticated speculative execution engines, on-core caches, or internal buses [Intel, 2018b].

Meanwhile, both the capacity and complexity of hardware caches have continued to increase. For example, Intel's Conroe architecture, which was released in 2006, contains two levels of caches with a maximum capacity of 4 MiB [Int, 2018a], whereas Intel's Skylake architecture which was released in 2015 has three levels of caches offering more than 30 MiB of capacity [Int, 2018c]. Additionally, this trend occurs on all the other hardware components that cache the execution history, including units that store destinations of recently taken branch instructions, or buffers that cache the result of recently solved virtual-to-physical address translation. For the rest of this thesis, we call these components *cache-like* hardware components.

Due to the increased processing power of computer hardware, industry has consolidated more and more software services on a single hardware platform, which are shared by mutually distrusted programs or virtual machines (VMs). Systems with such sharing range from cloud computing services to embedded platforms. For instance, infrastructure-as-a-service (IaaS) [Microsoft, 2018], a cloud computing service, hosts multiple mutually distrusting VMs on a hardware platform, through the management service provided by the bottom layer virtual machine monitor (VMM) or hypervisor. Similarly, the embedded operating system (OS) schedules applications authored by different companies together on a smartphone platform.

Sharing a hardware platform among mutually distrusting software entities indeed benefits the economy by offering virtualised machines as a service. However, sharing requires the system to provide much stronger isolation services to mutually distrusting VMs or software components, to achieve the same level of protection as if they are executing on dedicated hardware. Therefore, the software platform has to enforce a *security policy* which controls or limits information flow among different software entities (i.e., security domains), such as VMs, mobile applications, or web pages.

1.1 Microarchitectural Timing Channels

Microarchitectural timing channels threaten security on shared systems, by exploiting timing variances resulting from the shared use of hardware components, such as caches. These channels transfer secret information through timing which is affected by the shared microarchitecture state, such as cached contents of memory or the history of recently executed branch instructions. To transmit information, a sender program deliberately or incidentally manipulates hardware state during its execution, which affects the measurable efficiency of a receiver program.

The *cache-based timing channel* is a classic microarchitectural timing attack [Acıiçmez, 2007; Hu, 1992; Osvik et al., 2006; Tromer et al., 2010], which transmits information between a sender and a receiver through competing shared cache lines. For sending a signal, the sender modulates its cache footprint during its execution, which then can be detected by the receiver through measuring its latency by systematically touching those cache lines. Low latency implies that a line is still cached from a previous access, whereas a high latency implies that the sender has evicted the line during its execution.

Timing channel attacks are applicable not only to caches, but also to other hardware components that cache execution histories, such as the branch predictor [Evtyushkin et al., 2016a], or the translation look-aside buffer (TLB) [Hund et al., 2013]. Recently, researchers have demonstrated a series of sophisticated microarchitectural timing channel attacks that exploit side effects of CPU features designed for performance improvement, including speculative execution [Kocher et al., 2019], and microinstruction scheduling on an out-of-order execution pipeline [Lipp et al., 2018; Van Bulck et al., 2018].

The cause of microarchitectural timing channels is resource contention on shared hardware components that have limitations on either capacity or bandwidth, depending on the type of resource. The above mentioned cache-based timing channel exploits the capacity limitation on a shared cache. As they hold microarchitectural state, caches and cache-like components are called *stateful resources*. In contrast, bandwidth-limited resources, such as interconnects, are *stateless* and require truly concurrent access to generate a timing channel. For example, a sender can transfer information through its bandwidth consumption on a shared bus, which can be detected by a concurrently executing receiver though sensing its share of the available bandwidth.

1.2 Research Aims

This research focuses on the open problem of preventing microarchitectural timing channels through the security enforcement by the OS. Our approach is called *time protection*, in analogy to the established memory protection. We focus on providing a system solution for preventing timing channels on stateful resources, as preventing stateless channels requires bandwidth partitioning which does not exist on contemporary mainstream hardware.

Understanding the landscape To achieve our research goal, we examine all published microarchitectural timing attacks and related countermeasures. Through our study, we discover that all types of stateful resources, including cache and cache-like components, have been exploited as timing channels in intra-core as well as inter-core attacks. As these attacks result from sharing, a principled defence must prevent such sharing by partitioning all resources, either spatially or temporally. Such partitioning should be provided by the OS, in line with its general responsibility for enforcing security. The former requires resetting all history-dependent microarchitectural states between time slices, whereas the latter requires that the OS has control on resource allocation.

Virtualising all time sources can also mitigate timing channels as it removes the shared time source between the sender and the receiver. However, this mechanism is not only a very expensive solution [Li et al., 2013] but also impossible to apply to domains that interact with the physical world, including direct access to networks. Therefore, we focus on the above mentioned partitioning mechanisms.

Time protection—a core OS duty We propose to provide *time protection* in the OS, treating temporal isolation analogously to memory protection used for preventing spatial interference. This requires that a hardware platform provide options to either partition or reset related microarchitectural states. Therefore, the OS can provide mechanisms to create domains as a security container for user-level threads, with the aim of preventing microarchitectural timing channels between domains through either resetting or partitioning shared hardware states (Figure 1.1).

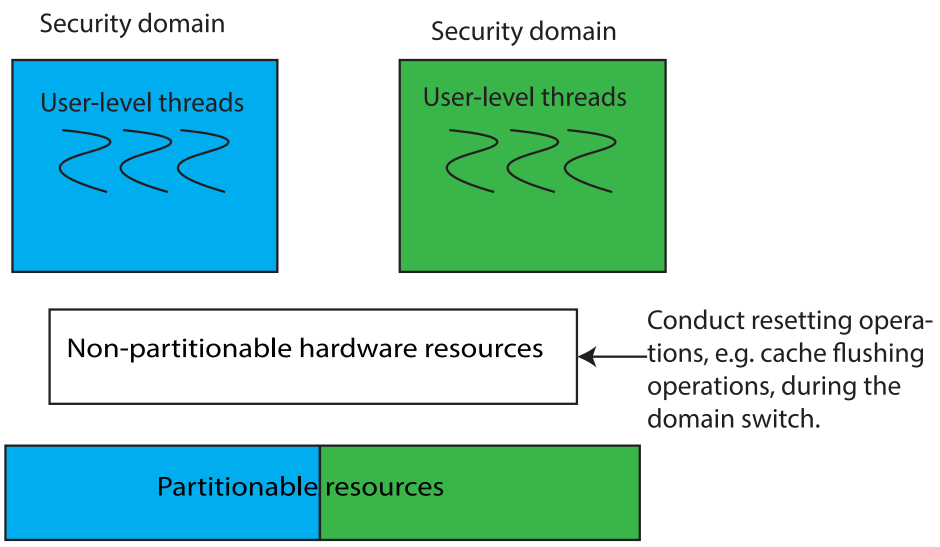

Security domain
Security domain
User-level threads
User-level threads
Non-partitionable hardware resources
Conduct resetting operations, e.g. cache flushing operations, during the domain switch.
Partitionable resources

Figure 1.1: An overview of the system solution for mitigating microarchitecture timing channels.

Investigating hardware resetting operations On mainstream hardware, an OS or a hypervisor can partition caches that are indexed by physical address with a software technique called cache colouring [Bershad et al., 1994; Kessler and Hill, 1992; Liedtke et al., 1997]. However, the cache colouring technique is not suitable for core-private caches or cache-like components that are indexed by virtual address, as the layout of the virtual address space is out of the OS's control. Hence, temporal partitioning (time multiplexing) is the only possible approach. This requires resetting the hardware state with flushing operations becomes the only option for mitigating timing attacks on virtually-indexed hardware components.

In order to understand the feasibility of this approach, we investigate the effectiveness of hardware resetting operations for mitigating those channels. We implement all possible timing attacks that are based on contending core-private caches or cache-like components. Then we analyse both original and mitigated channels on multiple generations of x86 and Arm processors. From this, we discover that selected hardware platforms are effective in mitigating many of the studied channels. However, there are residual channels on hardware state used for predicting branch instructions on studied x86 processors, and on states used for executing branch instructions on the Arm Cortex-A53 processor. The primary cause of these residual channels is that the current instruction-set architecture (ISA) is not a sufficient contract for ensuring temporal isolation, as it is completely hides the microarchitectural state and consequently does not provide the mechanisms required for temporally partitioning. As such, the ISA is insufficient for letting the OS enforce security. Therefore, we propose an *augmented ISA* as a new hardware-software contract to provide sufficient support on enforcing temporal isolation.

Achieving the proposed system solution We propose OS mechanisms for providing time protection, through a combination of partitioning and resetting techniques, with the assumption that the hardware can provide sufficient support for complete temporal isolation.

We implement our prototype on the seL4 microkernel, not only because of its integrity and confidentiality guarantees [Klein et al., 2014], but also for benefiting from its existing mechanisms for memory allocation. Additionally, previous work demonstrated that the kernel does not contain any storage channels [Murray et al., 2013], which gives us strong confidence in investigating the only possible remaining class of covert channels— timing channels.

Our solution allows not only partitioning user-level memory (including memory managed by the kernel), but also the kernel image itself with a new kernel mechanism called *kernel cloning*. Kernel cloning allows creating an almost complete copy of the kernel image that can be assigned to corresponding security domains, with an option of also partitioning hardware interrupts. With existing kernel mechanisms, we implement a system prototype and demonstrate that time protection effectively mitigates studied channels with a low impact on system performance.

1.3 Research Contributions and Thesis Outline

1.3.1 Research contributions

Our research produces the following contributions:

- We conduct a systematic study on all published microarchitectural timing attacks and related countermeasures (Chapter 3);
- we study the effectiveness of hardware-manufacturer provided operations on mitigating intra-core timing channels, for multiple generations of x86 and Arm processors (Chapter 5);
- we discover that the indirect branch control (IBC) mechanism [Intel, 2018c] introduced by Intel for mitigating the Spectre attack [Kocher et al., 2019] is partially effective at mitigating intra-core timing channels but leaves residual channels (Section 5.4);
- we propose an augmented ISA, a new software-hardware contract, as a solution for preventing any hidden microarchitectural state being exploited as timing channels. (Section 5.5);
- we establish the system requirements for enforcing confinement in the presence of microarchitectural timing channels (Section 6.1);
- we propose the *kernel cloning* mechanism for constructing dedicated kernel images that can be assigned to corresponding security domains (Section 6.2);
- we extend the cloning mechanism with an option for partitioning hardware interrupt sources (Section 6.2);
- we demonstrate the implementation on seL4, supporting both x86 and Arm multicore platforms (Section 6.4);
- we evaluate our system prototype for mitigating microarchitectural timing channels, showing that it is effective at closing studied microarchitectural timing channels, within limitations of present hardware (Section 6.5.2);
- we show that our mechanism introduces very low overhead on system performance (Section 6.5.3).

1.3.2 Thesis outline

This thesis is structured as follows. In Chapter 2, we provide the necessary background on timing channels, related microarchitecture, cache colouring, microkernels, and the seL4 microkernel. In Chapter 3, we define our research scope and examine the existing work on microarchitectural timing attacks and corresponding countermeasures. In Chapter 4, we define threat scenarios of this work, as well as hardware requirements for mitigating studied channels, and propose a system solution for providing time protection. In Chapter 5, we show our study on the effectiveness of hardware provided operations on providing temporal isolation for mitigating intra-core timing channels, and our proposed augmented ISA as a solution for the lack of definition on time protection in the existing ISA. In Chapter 6, we demonstrate the kernel cloning mechanism which provides the service for assigning dedicated kernel images and hardware interrupts to security domains. We also evaluate the effectiveness of our time protection mechanisms and their performance overhead. Finally, in Chapter 7, we conclude our research with a summary of our work, including a discussion of the future work.

2 Background

2.1 Timing Channels

Interest in information leakage and secure data processing historically was centred on sensitive, and particularly cryptographic, military and government systems [Denning, 1976]. However, many–such as Lampson [1973]– recognised the problems faced by tenants of an untrusted commercial computing platform of the sort that is now commonplace. The US government's "Orange Book" standard [DoD], for example, collected requirements for systems operating at various security classification levels. The Orange Book introduced standards for information leakage in the form of limits on channel bandwidth. While these were seldom if ever actually achieved, this approach strongly influenced the direction of later work.

Threat models for information leakage are often classified as either *covert channels* or *side channels*. Covert channels allow colluding programs to transfer information by means that bypass system security policy. A typical attack scenario includes a *Trojan* (sender), who has access to sensitive information, and a *spy* (receiver), who does not have access to that sensitive information but is able to communicate easily with the external world. A side channel is similar to a covert channel, except that instead of relying on a Trojan to transfer information that is deliberately leaked, an attacker exploits the side channel to recover sensitive information owned by a *victim* through existing system services. Side channel attacks are much harder to implement than covert channel attacks as they steal information from non-colluding victims. As a result, the implementations of side channel attacks are normally more complicated and more interesting than covert channels which leak information through similar attacking mechanisms.

Covert channels are categorized as either *storage* or *timing* channels. The common distinction between the storage and timing channel is whether a shared time source is required [Schaefer et al., 1977; Wray, 1991]. Storage channels communicate through any system state affected by the sender that can be directly read from the receiver, such as a register value, the return value of a system call, or even the existence of a system object (e.g., a user-level thread). A storage channel exploits something that is directly visible in software, which can be removed completely as shown by past work [Murray et al., 2013].

By contrast, timing channels, which exploit timing variance for communication, are much harder to reason about. A classic example of a timing channel is a secret-dependent code path or data access pattern that directly influences execution time. The usual defence is to code for deterministic execution by applying constant-time techniques, which is very hard to achieve [Bernstein, 2005]. Furthermore, the absence of timing channels is rarely formally specified, due to the difficulty of establishing a reliable model of the timing behaviour of hardware platforms [Klein et al., 2011].

In this work, we are concerned with *microarchitectural timing channels*, where information leakage is through shared microarchitectural components. The microarchitectural components implement functionalities of a processor which are abstracted away by the programming interface, the ISA, of that processor. In other words, the microarchitectural components contain details of hardware implementation, and are normally not exposed to software by manufacturers. For example, an attacker can observe the aggregate number of hardware cache conflicts created by a victim program by measuring the total execution time of accessing a cache-sized buffer [Osvik et al., 2006; Percival, 2005].

2.2 Virtual-to-physical Address Translation

The OS provides an abstraction of the memory available on a system through a combination of hardware and software techniques: providing virtual memory to user-level programs which is translated to physical memory by hardware during execution. From user-level programs' point of view, the memory appears as a contiguous address space, even though the content is sparsely stored in the physical memory. The address used by user-level programs is called the *virtual address*, whereas the address used by the hardware for fetching and storing memory content is called the *physical address*.

Figure 2.1 presents an overview of virtual-to-physical address translation. To conduct the translation, the hardware system divides the virtual address space into equal-sized regions called virtual *pages*. As a result, a virtual address comprises a virtual *page number* and a *page offset*. Similarly, the hardware system divides memory into *frames* that are the same size as virtual pages, and a physical address is formed by a frame number and a frame offset. The hardware only needs to translate the page number into a frame number as the offset does not change. While a program executes, the hardware translates the virtual page number into physical frame number, using the mapping scheme enforced by a hardware component called the *memory management unit (MMU)*. The hardware also identifies the address mapping translated from different address spaces (i.e., processes) with an identifier, the address space ID (ASID).

Contemporary processors use a multi-level page table structure to record the mapping from virtual to physical address, which is a radix tree indexed by virtual page numbers. Each valid page table entry contains either a pointer to the next level page table or the mapped frame number. To conduct a virtual address translation, the MMU first breaks

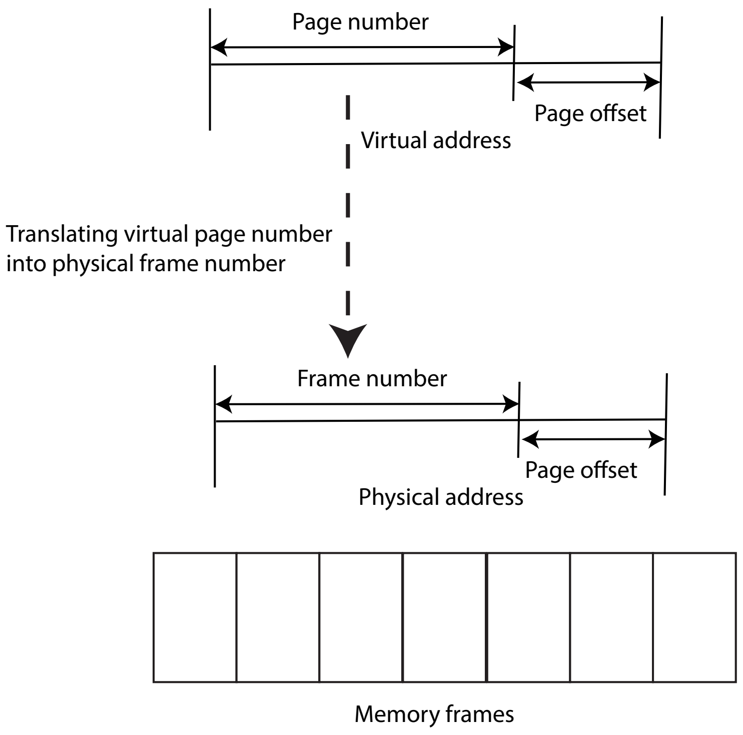


Figure 2.1: The virtual-to-physical address translation.

the virtual address into a page number and a page offset. Then, the MMU uses this page number as an index to walk the multi-level page table radix tree until it finds a leaf node. A leaf node is either a valid page table entry, pointing to a mapped physical frame, or an invalid entry. If a valid translation exists, the translated physical address is the is the sum of the frame number and page offset. It is important to note that traversing the multi-level page table requires at least one memory access per traversal if all page tables are stored in memory.

Although the translation process is enforced by the hardware, it is the OS's responsibility to provide the mapping, including assigning an ASID for the translated virtual address space. In other words, the OS manages the memory as a resource by assigning physical frames to virtual pages in an user-level program's address space. Because of the virtual-to-physical address translation, user-level programs control the layout of their virtual address space (i.e., virtual addresses), whereas the OS manages physical frames (i.e., physical addresses) used by pages in those virtual address spaces.

2.3 Relevant Microarchitectural Components

A processor provides a programming interface through its ISA, which abstracts over functionally irrelevant *microarchitectural* implementation details, such as pipelines, caches, or memory buses. These microarchitectural components contain the implementation details of the hardware, which are typically not exposed to software for portability reasons. For example, a program written for an earlier implementation of the x86 architecture, such as Intel's Conroe processor [Int, 2018a], can also execute on a later implementation of the x86 architecture, such as Intel's Skylake processor [Int, 2018c], because those processors modify only the microarchitectural implementation but not the ISA. In other words, the ISA works as the hardware-software contract for governing the behaviour of a given architecture.

Although not exposed through the ISA, those microarchitectural components contain *hidden state* that can be observed in the timing of program execution as a result of resource contention. Competition between processes (external contention) or within a process (self-contention) can both reveal hidden state, which can be exploited as timing channels. An example of a timing channel due to external contention is where an attacking program manipulates its footprint on a shared cache, causing slowdown of a victim program due to cache conflicts generated by the attacking program. Resource competition can also result from conflicts between different stages of program execution, such as between the start and the end of a program. These internal competitions create a timing channel if those conflicts are related to secret information held by the program. We will explain timing channels due to self-contention in more detail in Section 3.1.2.

This section outlines the background of the most important microarchitectural components that have been used to explore microarchitecture-based timing channels due to contention. In this section, we explain caches (Section 2.3.1), prefetching (Section 2.3.2), pipelining (Section 2.3.3), in-flight data (Section 2.3.4), buses and interconnects (Section 2.3.5), hardware multithreading and multicore (Section 2.3.6), dynamic random-access memory (DRAM) (Section 2.3.7), and the graphics processing unit (GPU) (Section 2.3.8). In Chapter 3, we will explain all related attacks as well as the scope for this work.

2.3.1 Caches

The hardware cache is a small quantity of memory that is located closer to processors than main memory. Because accessing the cache is much faster than accessing main memory, the CPU always searches relevant caches first before triggering any main memory request. A cache *hit* is a state in which a data fetching request from the CPU is contained in the cache. Conversely, a cache *miss* is a state in which the cache does not contain the data requested by the CPU, so main memory must be accessed. In other words, a cache hit is much cheaper than a cache miss. The effectiveness of cache relies on the hit rate, the fraction of requests satisfied from the cache.

To balance the design trade-off between complexity and efficiency, there are normally multiple levels of caches on contemporary hardware—each lower in the hierarchy being larger and slower than the one above. The size of each level is designed to balance cost and response time, by maintaining a high hit rate. We use the cache hierarchy on a modern Intel multi-core processor as an example (Figure 2.2). There are three levels of caches on this example Intel processor: the first-level data (L1-D) and first-level instruction (L1-I)

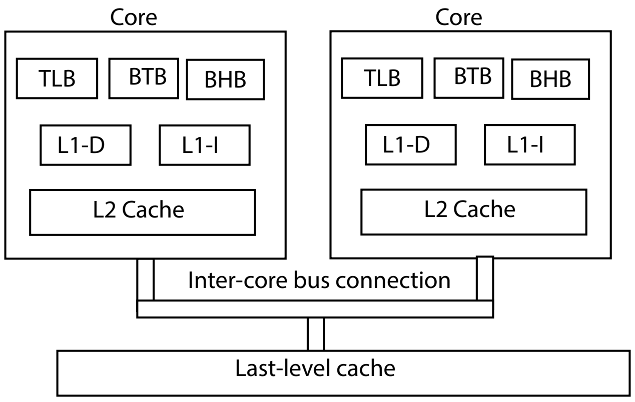


Figure 2.2: The hierarchical cache structure on a multicore Intel processor.

caches, a unified second-level (L2) cache that is core-private, and a last-level cache (LLC), the third-level (L3) cache, that is shared by all cores on a multicore processor (package).

The structure of cache hierarchy is microarchitecture-dependent, and the size of each cache is processor-dependent. For example, the Arm Cortex-A9 processor [ARM, 2010] on the Sabre platform [NXP, 2019] has a 32 KiB L1-D cache and a 32 KiB L1-I cache that are core-private, and a 1024 KiB L2 cache that is shared by all cores in the processor. In contrast, the Arm Cortex-A53 processor [ARM, 2016] has the same capacity on first-level (L1) caches, but a smaller L2 cache (512 KiB). The situation is different on Intel's i7-4770 processor, which is equipped with core-private L1-D (32 KiB), L1-I (32 KiB), and L2 (512 KiB) caches, and a core-shared 8096 KiB L3 cache [Intel, 2019a].

Figure 2.3 shows the internal structure of a hardware cache. The cache consists of *lines*, each of which holds one aligned power-of-two-sized block of adjacent bytes loaded from memory. Each cache line also stores information about the corresponding memory address, called cache *tags*. Cache lines are normally grouped into a number of *sets*, working as a *set-associative* cache. The number of cache lines in a set is called the cache's *associativity* (ways).

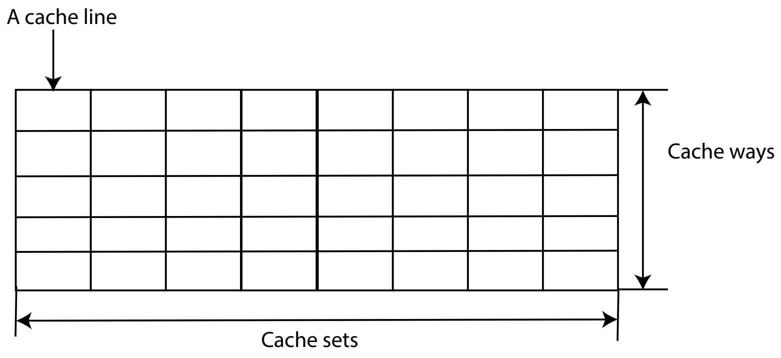


Figure 2.3: The internal structure of a hardware cache.

Caches are indexed by either virtual or physical addresses. As shown in Figure 2.4, address bits are divided into offset bits, indexing bits, and tagging bits. To locate data for a given address, the hardware first maps the address into a cache set with a indexing function that takes the value contained in the indexing bits as the input. Then, the hardware matches the value contained in the tagging bits against all tags within that set in parallel, an associative lookup, to locate the cache line within the set. Lastly, the hardware locates the data stored in the line by using offset bits.

When a cache miss occurs, the cache loads the corresponding line from memory, and replaces an existing line from the mapped set if that set reaches its capacity. A 1-way associative cache is also called a direct-mapped cache, where there is only one possible way to store any cache line. By contrast, a fully-associative cache, a single-set cache, can hold a cache line in any location.

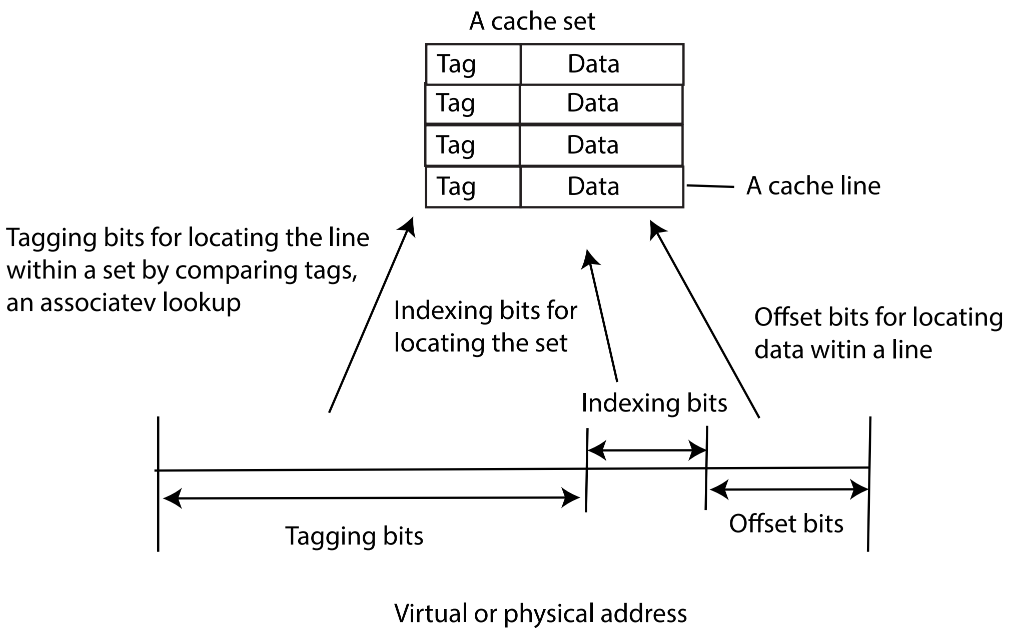

A cache set
Tag
Data
Tag
Data
Tag
Data
Tag
Data
A cache line
Tagging bits for locating the line within a set by comparing tags, an associatev lookup
Indexing bits for locating the set
Offset bits for locating data witin a line
Indexing bits
Tagging bits
Offset bits
Virtual or physical address

Figure 2.4: The indexing scheme of a hardware cache.

Caches can be classified into four types, based on the type of address used for indexing or tagging. A virtually-indexed, virtually-tagged (VIVT) cache uses the virtual address for both indexing and tagging, whereas a virtually-indexed, physically-tagged (VIPT) cache uses the virtual address for indexing but physical address for tagging. A physically-indexed, physically-tagged (PIPT) cache indexes and tags cache lines with the physical address, while a physically-indexed, virtually-tagged (PIVT) cache indexes cache lines with the physical address but tags them with the virtual address. While indexing bits are within the page offset, the cache can be regarded as either physically-indexed or virtually-indexed, because the virtual-to-physical translation does not translate those page offset bits.

The hardware architecture selects the type of address used for indexing and tagging, depending on the internal structure of a cache and the latency of accessing the cache. Normally, L1 caches, which have low access latency, are virtually-indexed, avoiding the cost of waiting for the virtual-to-physical translation. L1 caches are also commonly described as physically-indexed, if the size of a cache way is no bigger than the size of a page in L1 caches. Hence, indexing bits are not translated during the virtual-to-physical translation process. By contrast, L2 or L3 caches are normally physically-indexed, as their higher latency makes waiting on virtual-to-physical translation more affordable.

The function used for indexing is also a trade-off between cache latency and hardware complexity. The higher-level (i.e., L1 or L2) caches normally use a one-to-one mapping function: the value contained in the indexing bits represents the set number for a given address. As the lower-level caches are slower to access, the hardware can afford more complicated mapping schemes on physical addresses to reduce conflict misses. For example, recent Intel architectures have a two-level hashing function for distributing cache traffic [Hund et al., 2013; Maurice et al., 2015; Yarom et al., 2015].

There are more than just data or instruction caches available on a processor. The TLB caches the most recently resolved virtual-to-physical address translations (Section 2.2), in order to reduce address translation latency. The TLB is a virtually-indexed cache, which is normally tagged with an ASID to identify the address space which owns a cached translation. Moreover, hardware manufacturers have developed low-latency caches for caching page table entries in order to reduce the number of required memory accesses during a page table traversal (Section 2.2). There are two different designs for this type of cache: the *page table cache*, and the *translation cache*. The page table cache stores page table entries, working as a conventional PIPT data cache [Barr et al., 2010]. An example implementation of the page table cache is AMD's page walk cache, a high-speed read-only cache that stores recently accessed page table entries [Bhargava et al., 2008]. In contrast, the translation cache works as a virtually-indexed cache, tagged by the page table indices used for traversing page tables [Barr et al., 2010]. The translation cache allows the MMU to search cached page table entries from different levels in parallel. An example implementation of the translation cache is Intel's paging-structure cache [Intel, a], which works as a split translation cache—storing page table entries from different levels separately.

The hardware also contains caches that are used for accelerating the execution of branch instructions, including the *branch target buffer (BTB)* and *branch history buffer (BHB)*. The BTB stores the destination of recently executed branch instructions. Similarly, the BHB caches the outcome of recent conditional branch executions, assisting the CPU in predicting the destinations of future conditional branches. The BTB and BHB commonly work as VIVT caches, located inside a core, as the physical address is not available inside the core and as their functionality (assisting the CPU pipeline) cannot tolerate the latency of virtual-to-physical translation that is conducted outside the core.

Additionally, the *decoded instruction cache*, which works as a VIVT cache, stores recently fetched and decoded instructions in the form of micro-ops which can be processed and then executed by execution units in the core. We will explain the models used for instruction decoding and execution in Section 2.3.2.

2.3.2 Prefetching

Prefetching is the process by which the CPU fetches instructions or data before they are requested, loading them from their original storage into local storage, such as L1 caches. Speculatively prefetching instructions and data into local cache can prevent the CPU from waiting on cache misses, thus increasing the throughput of the CPU.

The prefetcher triggers fetching requests based on the most recently-executed instruction or -accessed data streams. If data stored in address v has been visited recently, the data stored in adjacent addresses is highly likely be used in the near future due to spatial locality. Therefore, the prefetcher increases cache hit rates by loading such data. Moreover, the prefetcher observes strides on data accesses, and prefetches the data that is stored a stride away from the previously accessed address. For example, the Arm Cortex-A9 processor implements a prefetcher that monitors past caches misses, automatically prefetching data from two independent data streams [ARM, 2010]. Hence the prefetcher also hold state which is affected by past history, such as the history generated by recently-executed programs.

The hardware can implement prefetching on multiple cache levels. For example, the Arm Cortex-A9 processor not only prefetches data and instruction to L1 caches, but also generates prefetching hints to the external L2 cache that is managed by a separate controller [ARM, 2010].

2.3.3 Pipelining

Instruction pipelining divides an instruction into a series of sequential steps that are performed by different execution units. To speed up CPU throughput, steps that belong to different instructions are processed in parallel, as long as there is no bottleneck, such as instruction dependencies, on any execution units [Healy et al., 1999].

Superscalar execution To increase the throughput of a CPU, the *superscalar execution* model manages multiple instruction pipelines in parallel by operating on a number of execution units concurrently. Here we use the modern Intel architecture as an example. Intel architecture divides the CPU into three parts [Lipp et al., 2018; van Schaik et al., 2019]: a *front-end* engine that preprocesses and decodes instructions into a stream of *micro-ops*, an *execution engine* that schedules micro-ops among multiple execution ports (i.e., dispatches micro-ops to execution units), and a memory subsystem that conducts memory load and store operations. In the execution engine, there are multiple channels, called *execution ports*, which connect the micro-op scheduler to the execution units where the micro-ops are executed. The design of the execution port is microarchitectural dependent. For instance, Intel's Skylake microarchitecture implemented eight ports: ports zero, one, five and six are used for executing arithmetic micro-ops, whereas ports two, three, four and seven are dedicated to memory-based micro-ops (e.g., loads and stores) [Aldaya et al., 2018]. To compare, Intel's Nehalem microarchitecture implemented six ports: ports zero, one, five for conducting arithmetic operations, whereas ports two, three, four for executing memory-based operations [Intel, c].

Speculative execution Contemporary CPU design uses *speculative execution* to maximise the performance of a CPU by predicting the outcome of branch instructions at the front-end. In other words, the stream of micro-ops generated by the front-end is not necessarily equivalent to the instructions written in a program due to branch prediction. Whilst calculating the outcome of a branch instruction is time-consuming, as might happen if the destination of the branch needs to be loaded from memory, the CPU attempts to execute instructions after the branch based on a guessed outcome (e.g., the destination address). Moreover, the CPU conducts those guesses based on the most-recently-executed branch instructions. That is, the CPU predicts the outcome of currently executing instructions based on the recent execution history. After the correct result becomes available, the CPU chooses to either commit or abort the speculatively-executed computation.

Out-of-order execution The *out-of-order execution* model reorders the instruction stream in the execution engine, allowing an instruction to be executed before or in parallel with preceding instructions [Fog, 2018]. To achieve that, a modern CPU divides an instruction into several micro-ops and schedules them on the functional units in the pipeline. An instruction can only be regarded as completed once all of its micro-ops and preceding instructions are finished.

The above mentioned superscalar execution model typically implements an out-of-order execution model that executes micro-ops decoded from different instructions in parallel. Furthermore, the CPU can execute micro-ops speculatively while conditions for making a correct decision are not immediately available. To fix a faulty speculation, the CPU reissues the incorrectly executed micro-op. The CPU commits an instruction once all of its micro-ops and preceding instructions are finished. In contrast, the CPU aborts an instructions if any of its micro-ops is regarded as faulty (e.g., crossing the privilege boundary).

2.3.4 In-flight data

Modern CPUs have many potential sources of *in-flight data* which carry information on currently executing load and store operations. To execute a load or store, the CPU first translates the source or destination from its virtual address to a mapped physical address, then acquires permission to access the corresponding location for conducting the actual read or write. All these steps are potentially time consuming and may cause pipeline stalls. To optimise, the CPU records ongoing read or write requests in internal buffers, then continues processing other instructions. In this work, we focus on three internal microarchitectural components that store in-flight data: *load* and *store buffers*, and *line fill buffers*.

Load buffers Load-buffer entries store information about dispatched load operations, such as virtual addresses, ordering constraints due to adjacent loads or stores, and the current status. The load buffer is normally located between the execution engine (Section 2.3.3) and the L1-D cache, serving the ongoing loads issued by the *load port*, a type of port for load operations, in the execution engine (Section 2.3.3).

Store buffers The store buffer tracks pending store requests sent from the execution engine (Section 2.3.3), recording information about both address and data. Moreover, the store buffer can also be involved in pipelining optimizations, such as the *store-to-load forwarding* [Abramson et al., 1995; Mekkat et al., 2015] which forwards the recent store to the load buffer if a pending load is requested from the same physical address. In other words, the store-to-load forwarding searches for any pending loads that fetch from same physical addresses as pending stores, and forwards the store values if any matching pairs are found.

Both store and load requests use virtual addresses which need to be translated into physical address (Section 2.2), a potentially time consuming process (Section 2.3.1). To optimize, the store-to-load forwarding can be speculatively executed without waiting on the virtual-to-physical address translation. For instance, recently released Intel processors predict whether the physical addresses of a given pair of load and store are identical by comparing only partial virtual addresses [Minkin et al., 2019; Yoaz et al., 1999]. Thus the processor can execute the store-to-load forwarding speculatively before translated physical addresses are available. Later, the processor decides either to commit or revise the load once the physical addresses are available.

Line fill buffers The line fill buffer records outstanding load or store requests that suffer from cache latencies, such as L1-D cache misses. The line fill buffer works as a memory interface for the pipeline, connecting the pipeline to the cache hierarchy and main memory. Moreover, the line fill buffer can reduce cache stalls while resolving cache misses or participate in optimisations performed on load or store buffers. For instance, the line fill buffer in Intel processors allows non-blocking reads from the L1-D cache while the cache hierarchy is trying to resolve L1-D misses [Dundas, 2002; Intel, b]. Moreover, the line fill buffer in Intel processors assists speculatively executed store-to-load forwarding if stores are pushed through the line fill buffer to either the L1-D cache or main memory [Abramson et al., 1997, 1999; Bodas et al., 1998].

2.3.5 Buses and interconnects

Buses host traffic generated from the CPU to external memory, including external caches (e.g., the LLC). Interconnects connect all hardware components on a multicore system, working as a network router for transferring messages among components. The interconnect on contemporary hardware works as a sophisticated network with the help from multiple components, including packet buffers, channels, ports, and switches. Due to being shared by cores on a system, buses and interconnects are vulnerable to saturation [Wassel et al., 2013], as they are under-provisioned.

2.3.6 Hardware multithreading and multicore

The simultaneous multithreading (SMT) allows multiple execution contexts (i.e., hardware threads) to execute concurrently, each with its own hardware state. SMT increases CPU throughput by scheduling hardware threads among the functional units. Intel's hyperthreading technique is an example of an SMT implementation where multiple hardware threads are simultaneously scheduled on the CPU pipeline. In SMT, threads compete to access functional units (e.g., the floating point unit (FPU)), speculation resources (e.g., the BTB or the BHB), and L1 caches. In this work, we also regard the sources of in-flight data (Section 2.3.4) as part of the execution engine (Section 2.3.3) shared by hardware threads.

In the superscalar execution model (Section 2.3.3), micro-ops that belong to different hardware threads (threads in the SMT model) are scheduled together thereby competing for ports. As ports are designed for different kinds of operations (Section 2.3.3), a hardware thread can easily create contention on a port by executing operations that can only be executed on that port. The fine-grained sharing between hardware threads introduces tremendous threats to system security, in particular by exploiting port contention in real time.

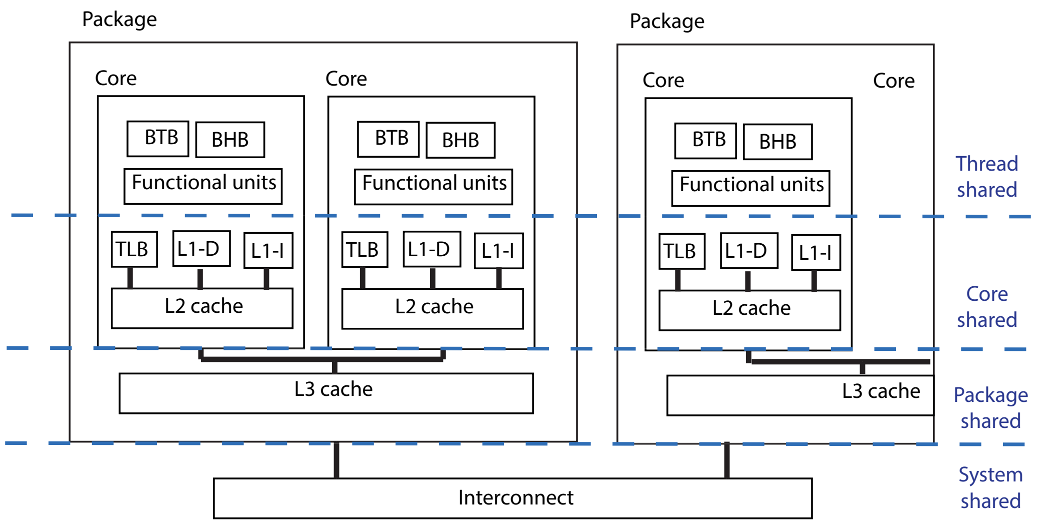


Figure 2.5: The shared resources in a multi-core Intel system.

A modern multicore system has complicated resource-sharing hierarchies. Figure 2.5 shows an example of such sharing hierarchy on an Intel multi-core system. The hierarchy contains hardware threads, cores, and packages. Hardware threads share all the core-private resources, including speculation resources (e.g., BTB and BHB), TLB, functional units, and L1 caches. The L2 cache can also be a core-private resource, working as an intermediate cache between the L1 and L3 caches. All cores share the L3 on the package, whereas everything running in the system shares the interconnect and buses.

2.3.7 DRAM

The DRAM typically operates as the main memory in a system. A DRAM consists of multiple DRAM *chips* that each contains multiple *banks*. In each bank, there is a two-dimensional array of DRAM *cells*, which are the smallest unit of storage in the DRAM [JEDEC, 2012] . The data stored in each cell is decided by the amount of charge stored in its *capacitor*: the charged state represents bit one, and the discharged state represents bit zero. Moreover, every cell is connected by a horizontal *wordline* and a vertical *bitline*. All cells connected by the same wordline form a *row*, whereas all cells connected by the same bitline form a *column* [Kim et al., 2014]. Each bank also has a *row buffer* that caches the value of the most recently accessed row.

To access a word, the DRAM controller first opens the corresponding row from the same bank in all chips, then transfers all data stored on these rows to row buffers in each bank. Finally, the DRAM controller accesses data stored in row buffers for requested reads or writes. Before opening another row, the controller must close the currently open row.

Data stored in DRAM cells can be corrupted by losing charge or external events. To address that, the DRAM controller periodically recharges cells, refreshing the charge stored in capacitors. This process is called *refreshing*. Additionally, external events, such as the exposure to cosmic rays [Hwang et al., 2012], can also corrupt data stored in cells.

To protect data, the DRAM design can choose to contain error-correcting code (ECC) memory that stores extra parity bits for error correction and detection. An ECC implementation can correct up to n bits of errors and detect maximumly m bits of errors, where n and m are determined by the error correction algorithm as well as the number of parity bits in the ECC memory. Supporting ECC memory requires the system to enlarge the memory bus for the extra parity bits. In commodity systems, the memory controller is the only unit that conducts ECC correction. Once an error is detected, the memory controller corrects the error and writes back the corrected values.

2.3.8 GPU

The GPU is a type of co-processor that is specialised in image rendering. Comparing to the CPU, the GPU devotes more computation engines, i.e., arithmetic logic units (ALUs), to data processing rather than general purpose processing. The fundamental processing unit in GPU is called *stream processor (SP)* that each contains multiple ALUs. Furthermore, the SP executes *shaders* which are specialized programs designed for conducting graphics rendering in parallel. Additionally, the GPU has multiple texture processors (TPs) that load input textures requested by SPs during computation [Frigo et al., 2018].

The GPU also contains private caches that store recently accessed vertices and textures. For instance, there are two levels of caches on the Adreno 330, an integrated GPU used by embedded Android devices: small but efficient L1 caches (1 KiB for each TP) for textures, and a unified L2 cache (32 KiB) for both vertices and textures [Frigo et al., 2018].

The GPU can either work as a dedicated GPU with its own memory or an integrated GPU. When working as an integrated co-processor, the GPU shares the system memory with the processor.

2.4 Cache Colouring

Cache colouring is a software approach to partition physically-indexed set-associative caches, originally proposed for improving system performance [Bershad et al., 1994; Kessler and Hill, 1992] or to guarantee performance of real-time tasks [Liedtke et al., 1997]. There are also other methods for partitioning caches, which will be introduced in Section 3.3.5. In this section, we focus on the cache colouring technique due to its importance for this work–conducting spatial partitioning on physically-indexed caches using the memory management model of the seL4 microkernel. We will explain the method for implementing cache colouring in seL4 in Section 2.6.3. Additionally, we will describe related work that applied cache colouring as a method for mitigating cache-based timing channels in Section 3.3.5.

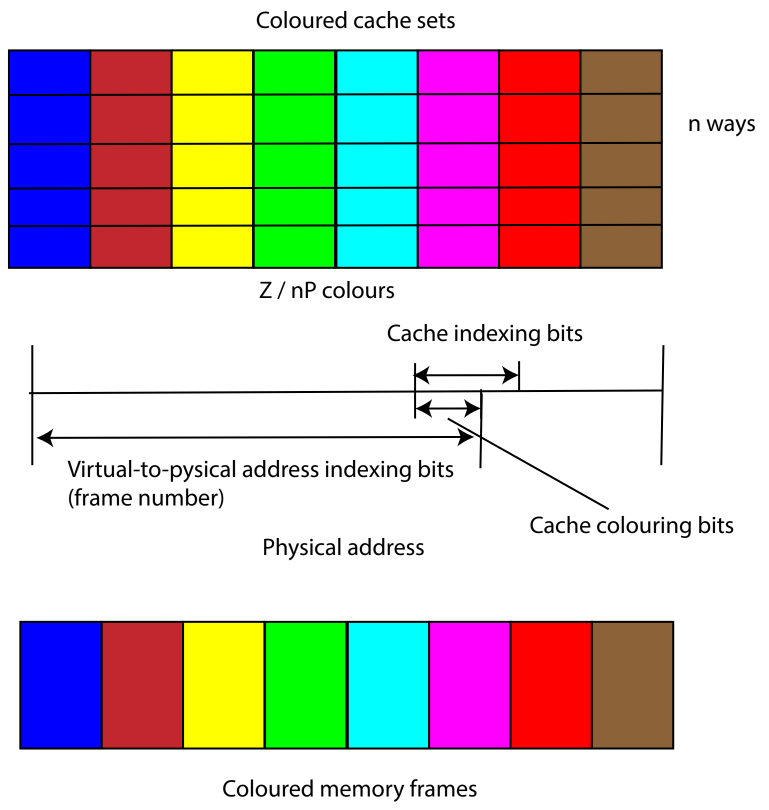


Figure 2.6: The cache colouring mechanism.

Cache colouring, also called page colouring, takes advantage of overlapping bits in two mapping schemes: cache-indexing (Section 2.3.1) and virtual-to-physical address translation (Section 2.2). Figure 2.6 presents an overview of the cache colouring mechanism. Overlapping bits shown in Figure 2.6 are cache colouring bits, which assign cache sets to memory frames, regarded as the *colour* of the frame.

Cache colouring is an OS mechanism, which requires that the OS has complete control on allocating colours to security domains. Because the virtual address is under the control of the user-level application, cache colouring is conducted through physical addresses, which assigns physically-indexed cache sets to memory frames. Hence the cache colouring has two prerequisites: a physically-indexed cache, and that the indexing bits used by cache mappings contribute to the frame number. That is, cache colouring requires that frames with different values in their colouring bits do not map to the same cache set. For an n-way set-associative cache of size Z bytes, there are Z/nL cache sets if the cache line size is L. With a frame size of P bytes, there are Z/nP cache colours.

The cache colouring technique cannot be used on virtually-indexed caches, such as the L1 caches, as virtual address allocation is not under the OS's control. Similarly, the OS cannot colour BTBs, BHBs, nor TLBs, as these are all virtually-indexed caches (Section 2.3.1).

Cache colouring can also be used to protect against cache-based timing channels, by assigning security domains that contain a number of software components and processes with disjoint frame colours. Our work regards a security domain as a single unit in a system's security policy for enforcing time protection. Necessarily, because frames with different colours will be mapped to different cache sets, cache lines from those coloured frames cannot reside in the same cache set. Hence, frames with different colours cannot interfere in the cache. Figure 2.7 demonstrates a simple system scenario where security domains occupy different coloured cache sets by created with only coloured frames. Due to the absence of sharing, threads hosted by different security domains cannot transmit information via cache-based timing channels.

Similarly to other hardware-partitioning techniques, as will be introduced in Section 3.3.5, cache colouring can mitigate cache-based timing attacks relying on either consecutive (Section 3.2.3.1) or concurrent cache accesses (Section 3.2.3.2).

On more recent Intel architectures, the LLC contains multiple cache slices that are connected by a ring bus [Yarom et al., 2015]. Thus the cache mapping process has two stages: locating a cache slice, and locating a cache line within a slice. According to previous work, Sandy Bridge and newer Intel microarchitectures hash the physical address to locate a cache slice [Hund et al., 2013; İnci et al., 2016; Irazoqui et al., 2015b; Maurice et al., 2015; Yarom et al., 2015]. Within a cache slice, the hardware uses a simple one-to-one function for mapping cache sets [Liu et al., 2015].

Knowledge of the hash function used for mapping cache slices can increase the number of cache colours if the hash function for slice index uses a different input than that used for mapping cache sets within a cache slice. Consequently, the result of calculating the

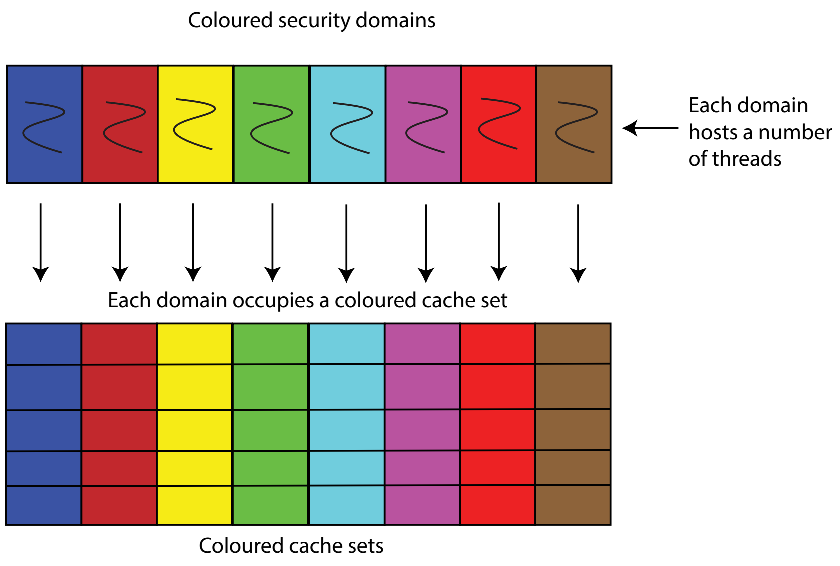

Coloured security domains
Each domain hosts a number of threads
Each domain occupies a coloured cache set
Coloured cache sets

Figure 2.7: Assigning security domains with different cache colours.

slice index does not affect the result of calculating the cache sets. For example, a 4-slice L3 cache has 128 colours if each of the slices has 32 colours. If the hashing function is known, the OS can assign frames to cache slices as well as cache sets within a slice based on physical addresses (frame numbers).

However, reverse engineering the details of hashing functions used by the LLC may not be possible for future CPUs as the complexity of the hardware design increases. Still, cache colouring is possible without the knowledge of the hashing function by only colouring cache sets within a slice [Yarom et al., 2015], paying the expense of using fewer colours than actually offered by slicing. In this work, we conduct the cache colouring on experimental Intel processors using colours within a slice, as we do not make any assumptions on the undocumented hashing function.

One shortcoming of cache colouring is its inability to support huge pages. Huge pages are large frames designed to alleviate TLB pressure, such as 2 MiB-sized pages on x86. When large pages are enabled, systems have fewer available colours (often one) as there is reduced overlapping (frequently none) between memory frame numbers and cache-mapping bits. Additionally, coloured bits are only a subset of the memory frame numbers; hence the cache colouring breaks the memory into chunks. Each chunk contains frames with disjointed colours. As a result, coloured memory is generally not contiguous, as memory with same colours are located in separate chunks.

The other drawback of cache colouring is that reallocating colours requires copying all the preempted frames to one of the remaining colors, which can be an expensive task for taking a color away from a partition. However, a system can regard this task as affordable if the reallocation is not triggered frequently.

2.5 The L4 Microkernel Family

A microkernel offers the near-minimum amount of system services that are required to implement an OS. Compared with the monolithic kernel, a microkernel does not contain many functions associated with a traditional OS, such as device drivers, process management, memory management, or file systems. These OS functions are all implemented by user-level servers (i.e., applications), using general-purpose system mechanisms provided by the kernel [Heiser and Elphinstone, 2016; Levin et al., 1975; Liedtke, 1995]. Normally, the microkernel executes at the most privileged level offered by CPU, which is necessary to conduct context switching, enforce virtual memory configurations, and handle hardware interrupts. In contrast, everything else, such as user-level device drivers or memory allocators, are running at lower privilege.

On a microkernel, user-level threads communicate through inter-process communication (IPC) messages, a message-passing mechanism offered by kernel. The performance of IPC is critical for microkernel systems, because all system services offered by both user-level applications and the kernel are obtained through IPC messages. Previously, Liedtke [1993] demonstrated an efficient IPC message-passing design on his L4 kernel, which is a factor of 10–20 performance improvement over previous microkernel implementations [Accetta et al., 1986]. The success of the L4 kernel triggered a revolution in microkernel design, driven by principles including minimality, generality, efficient IPC, and user-level device drivers [Härtig et al., 1997; Heiser and Elphinstone, 2016; Liedtke, 1995].

Minimality and generality Ideally, a microkernel should only contain the minimum set of mechanisms for implementing user-level services, and there should be no policy in the kernel. In other words, a service can only be tolerated inside the microkernel if relocating it to user-level would prevent fulfilling other system required functionalities [Heiser and Elphinstone, 2016; Liedtke, 1995]. Furthermore, the kernel mechanisms are designed for general-purpose usage, allowing a broad spectrum of systems to be built, from embedded systems to cloud platforms.

IPC As mentioned before, an efficient IPC implementation is important for microkernels as IPC is on the critical path for invoking system services. Synchronous IPC copies message contents from the sender directly to the receiver, avoiding the cost of buffering the message in the kernel. For short messages that can be passed in CPU registers, the kernel can further optimise IPC performance by only conducting a context switch, leaving those message registers untouched.

User-level device drivers The minimality principle, only containing a minimum set of mechanisms but not any policies inside a microkernel, motivates relocating device drivers to user-level. In such a design, hardware interrupts are modeled as IPC messages sent by the kernel to user-level drivers. An outstanding benefit of moving device drivers to user level is to largely decrease the size of the kernel, resulting in a smaller trusted computing base [Rushby, 1984] that is furthermore verifiable [Klein et al., 2009]. The kernel still contains a small set of drivers essential for system services, including a timer driver for enforcing system ticks and an interrupt controller driver for distributing interrupts to user-level applications (e.g., device drivers).

2.6 The seL4 Microkernel

2.6.1 Overview

seL4 [seL4] is a high-assurance microkernel designed for security- and safety-critical systems. The implementation of seL4 is formally proved to be correct against its functional model [Klein et al., 2009], even to the level of the executable binary [Sewell et al., 2013]. Also, the formal model of the kernel provides integrity and confidentiality guarantees [Klein et al., 2014], demonstrating the kernel is free from any covert storage channels [Murray et al., 2013].

In common with other security-focused systems [Bomberger et al., 1992; Shapiro et al., 1999], seL4 is a capability-oriented microkernel: capabilities [Dennis and Van Horn, 1966; Karger, 1988] grant authorisation to access objects. A capability represents an object and associated rights to operate on that object. To perform an operation, such as acknowledging the receipt of a hardware interrupt, a user-level thread must present its corresponding capability with sufficient access rights for the invoked kernel service [Data61, 2017b]. Accordingly, all seL4 system calls operate on capabilities that are managed by the kernel but created upon request from user-level applications.

With the capability-based access control model, the system can selectively grant rights to user-level applications in order to isolate those applications from each other. For example, the system can prohibit communication between applications by not sharing any memory frames, represented by Frame capabilities, nor endpoints that are used for send and receive IPC messages, represented by Endpoint capabilities. This system model enables a high degree of assurance in the isolation provided by seL4, as the kernel only permits operations that are explicitly approved by corresponding capabilities. We will examine types of kernel objects that are referenced by capabilities in Section 2.6.2.

There is no concept of a processes on seL4; rather, the kernel uses a lower-level abstraction, the thread, to represents an executable context on a CPU. User-level processes are regarded as threads on seL4, as the kernel does not provide any process or thread management. A user-level application can contain multiple threads which share a virtual address space. seL4 regards process management as system policy, thus it should be implemented at user level with kernel-provided mechanisms. Similarly, the kernel does not manage user-level address spaces, but provides system mechanisms for constructing the layout of user-level address spaces.

The multicore implementation of seL4 uses a big lock to maintain the coherency of the kernel data [Peters et al., 2015]. The design was chosen for its performance and verifiability on a small sized microkernel. The kernel acquires the lock before serving a system call and releases the lock before resuming the next running thread. On a multicore system, threads can only be scheduled on their core, marking as the *affinity* of a thread. The kernel maintains a scheduling queue for each core, collecting the ready-to-run threads on that core.

2.6.2 The memory management model in seL4

Based on the design goal of separating policy from kernel mechanism [Levin et al., 1975] (Section 2.5), seL4 does not impose any memory-allocation nor thread-management policy; rather, the memory allocation task is delegated to user level. The kernel has no heap and uses only a bounded stack. Beyond that, the kernel provides a model for managing kernel memory at user-level, which is motivated by enabling reasoning about resource usage and isolation [Elphinstone and Heiser, 2013].

Figure 2.8 gives an overview of the memory model in seL4. After booting, the kernel hands over all free memory as Untyped capabilities to the initial thread. Those Untyped capabilities can be further re-typed into other objects, such as memory frames for user-level address mappings or thread control blocks (TCBs). In the same vein, all the other types of kernel memory, including page tables used to create user-level address spaces, capability management spaces, or inter-process communication endpoints, are created by user-level processes.

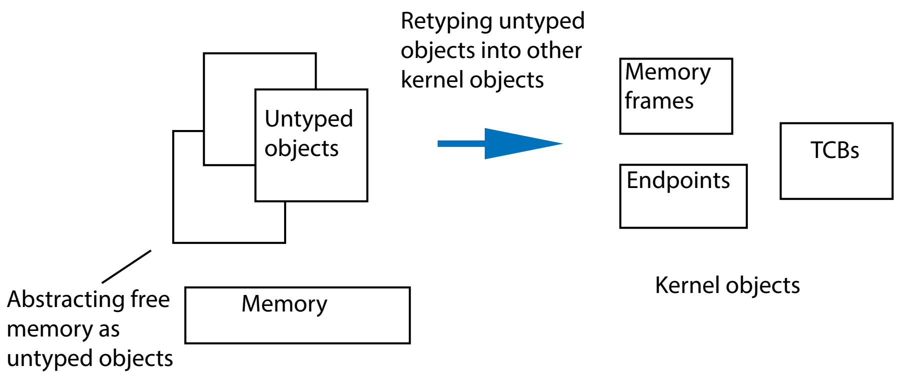

Retyping untyped objects into other kernel objects
Untyped objects
Memory frames
TCBs
Endpoints
Kernel objects
Abstracting free memory as untyped objects
Memory

Figure 2.8: The memory model in seL4.

We now explain the memory management model of seL4 using kernel objects used on 32-bit Arm processors, as listed in Table 2.1.

Untyped Kernel metadata that is designed explicitly as kernel objects and subjected to capability-based access control. Instead of being dynamically created by the kernel, all kernel objects must be explicitly created from Untyped memory, application-controlled memory regions that are created by the kernel during system start-up.

Kernel object	**Description**
Untyped	Memory from which kernel objects can be created
TCB	Thread control block
CNode	Capability storage; can be used as the CSpace of a thread
PageDirectory	Top-level page directory used for paging structure; the VSpace
PageTable	Second-level page directory used for paging structure
Page	A frame of physical memory
ASIDControl	Top-level ASID pool used for generating second-level ASID pools
ASIDPool	Second-level ASID pool
IRQHandler	An interrupt source
IRQControl	Master capability for creating other IRQHandler capabilities
Endpoint	Port-like object for synchronous IPC messaging
Notification	A signalling mechanism

Table 2.1: Kernel objects in seL4 for 32-bit Arm processors.

TCB seL4 offers services to user-level applications through capabilities. A TCB capability represents a TCB representing a thread, which has an associated capability space (CNode) and virtual address space (VSpace).

CNode A CNode has a fixed number of slots. A slot in a CNode can be empty or store a capability. A CSpace represents the namespace used by an application for storing its capabilities, and any capability invoked by this application must be valid in its namespace. A CNode can be configured as the CSpace of a thread by associating that CNode with its TCB (the TCB object). Performing deletion on capabilities removes those capabilities from corresponding CSpace. Moreover, deletion resets memory used by kernel objects pointed by those capabilities if they are the only existing capabilities for those objects.

PageDirectory A PageDirectory capability represents the top-level page directory used for virtual-to-physical address translation (Section 2.2). In seL4, each virtual address space is managed as a VSpace. Depending on the requirements of the underlying hardware, a VSpace can be composed of various objects for managing virtual memory (Section 2.2), including a top-level page directory (a PageDirectory capability), page tables (PageTable capabilities), memory frames (Page capabilities), and an ASID that is assigned with the ASIDPool capability.

ASIDControl The seL4 microkernel uses a two-level ASID mapping: the kernel only creates the first-level ASIDPool at boot, and the second-level ASIDPool is created and installed by the initial thread using the ASIDControl capability.

IRQHandler To delegate hardware interrupts an authorised thread can invoke the `seL4_-IRQControl` system call on the IRQControl capability, the master capability for managing interrupts. This allows threads to create IRQHandler capabilities on given hardware interrupt request (IRQ) sources. Furthermore, a user-level thread can subscribe interrupt notifications from an IRQ by associating a Notification capability which provides a signalling mechanism with the corresponding IRQHandler. Once the interrupt is received, the thread acknowledges the receipt of the interrupt by using the `seL4_IRQHandler_Ack` system calls.

Endpoint seL4 provides the message-passing mechanism between user-level threads through Endpoint capabilities. To send an IPC message, a user-level thread invokes the `seL4_Send` system call on an Endpoint capability, and the recipient thread receives the message via the `seL4_Wait` (i.e., listening) on a copy of that Endpoint capability.

2.6.3 Implementing cache colouring in seL4

The following properties of seL4's memory management model contribute to resource isolation:

- The kernel memory is explicitly managed as kernel objects;
- the capability-based access control model simplifies calculating the resource usage of a given thread;
- kernel objects are created and managed at user-level; and
- the kernel does not contain any memory-management policy.

A user-level management thread can easily deploy memory management policies at user-level using the kernel mechanisms provided by seL4.

Figure 2.9 demonstrates the application of cache colouring to create security domains: every kernel object used by a coloured domain has the same cache colour. To achieve that, the management thread first partitions memory into coloured pools, by splitting Untyped objects into smaller sized Untyped objects according to the colour of their memory frame (Section 2.4). Then, the management thread creates coloured system objects using the Untyped memory in each pool, which are later used to create coloured security domains.

By applying cache colouring in this way, the coloured threads belonging to different domains do not share lines in shared caches, for any uses by user-level address space (VSpace) or kernel metadata (kernel objects). Critically, those isolated domains are not able to communicate if there is no Endpoint capability available for IPC. To create a static system, the management thread can commit suicide after system initialisation, destroying the possibility of revoking those coloured objects. Thus the system remains strictly partitioned forever.

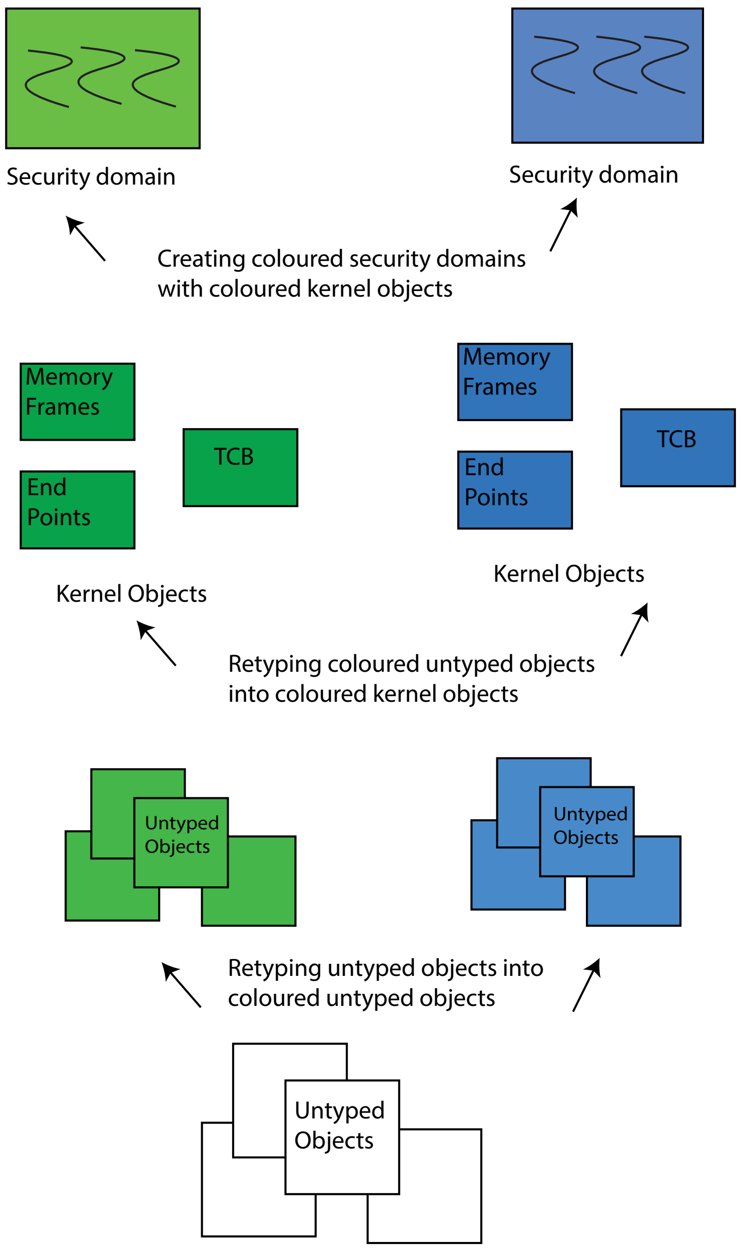

Security domain
Security domain
Creating coloured security domains with coloured kernel objects
Memory Frames
End Points
TCB
Kernel Objects
Memory Frames
End Points
TCB
Kernel Objects
Retyping coloured untyped objects into coloured kernel objects
Untyped Objects
Untyped Objects
Retyping untyped objects into coloured untyped objects
Untyped Objects

Figure 2.9: Creating coloured security domains using the memory model in seL4.

3 | Related Work

This chapter is the subject of the following paper of which I was the primary author: Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8:1–27, April 2018a. The analysis and classification of all published timing channel attacks and corresponding countermeasures was performed primarily by myself, with assistance from Yuval Yarom and David Cock, under the supervision of Gernot Heiser.

3.1 Scope

This work focuses on mitigating microarchitectural timing channels based on resource contention on both single-core and multicore systems. Hence, we ignore external channels, such as the DRAM open rows channel [Pessl et al., 2016], in this work. More specifically, we do not cover timing channels exploiting resource contention in the following methods.

3.1.1 Hardware multithreading

We do not cover timing attacks between hardware threads (Section 2.3.6) that are concurrently executing on a core. Hardware threading (i.e., Hyperthreading on Intel platforms) introduces a tremendous threat to system security [Percival, 2005], due to the concurrent sharing between hardware threads. For example, the PortSmash attack exploited the contention on ports in the execution engine (Section 2.3.3) on Intel's Skylake processors [Aldaya et al., 2018]. As mentioned in Section 2.3.6, a hardware thread can easily create contention on a port by continuously executing the corresponding instruction, thereby slowing down other hardware threads. The PortSmash attack utilised this feature by repeatedly executing instructions on both double and add ports, which eventually lead to breaking the elliptic curve digital signature algorithm (ECDSA) algorithm [Aldaya et al., 2018].

Moreover, previous work demonstrated timing channels by exploiting contentions not only on functional units on pipeline (contending ports) [Acıiçmez and Seifert, 2007; Aldaya et al., 2018; Wang and Lee, 2006], but also on various caches or cache-like components (contending cache lines), including the line fill buffer [Schwarz et al., 2019], the TLB [Gras et al., 2018], the BTB [Acıiçmez et al., 2007a,b], the L1-I cache [Acıiçmez, 2007; Acıiçmez et al., 2010], and the L1-D cache [Brumley and Hakala, 2009; Osvik et al., 2006; Percival, 2005; Tromer et al., 2010]. Attacks that create contention on caches use different techniques than those that contend ports. We will explain attacking techniques used for generating cache contention in Section 3.2.1.

Timing channels between hardware threads are inevitable due to the large amount of concurrently shared microarchitectural states. For preventing those timing channels, the OS has to prohibit the concurrent sharing by either disabling hardware threading or allocating all hardware threads of a core to the same security domain. Disabling hyperthreading is the most effective method for mitigating timing channels between hardware threads, which is becoming a common practice on public clouds [Marshall et al., 2010]. Furthermore, OpenBSD has already disabled hardware threads by default on Intel processors [OpenBSD, 2018] in order to mitigate the TLBleed attack [Gras et al., 2018], a side channel attack that successfully broke the elliptic curve function used for cryptographic multiplication in `libgcrypt` [Bernstein, 2006] by causing contention on shared TLBs between hardware threads.

3.1.2 Timing channels due to self-contention

Moreover, we do not cover timing attacks due to self-contention, where the total execution time of a victim program is related with the secret.

Such kind of attacks detect the accumulated number of cache hits and misses while the victim application is processing different input values. This can then be measured as the total execution time of the victim application. For example, Bernstein [2005] demonstrated a practical attack against the avanced encryption standard (AES) by measuring the latency of a remote AES service, representing self contention as an exploitable side channel. Similarly, Andrysco et al. [2015] discovered that the latency of floating-point operations is related to types of operands on various floating-point instructions. Those measurable timing variances can directly lead to a side channel attack on a scalable vector graphics (SVG) filter, revealing arbitrary pixels from attacked web pages browsing on Firefox.

A common approach to mitigating timing channels due to self-contention is applying the constant-time approach (Section 3.3.1): eliminating any secret-dependent behaviours on consuming hardware resources which can directly cause timing difference [Ge et al., 2018a]. In other words, the sequence of accessing hardware caches or taking branch instructions does not depend on the sensitive information, such as the encryption key or the plaintext [Bernstein, 2005; Brickell, 2011]. The constant-time implementation of a given program (e.g., an AES implementation), is always hardware platform dependent, as the implementation on one hardware platform may behave differently on another hardware platform [Cock et al., 2014]. Investigating the constant-time technique is out of the scope of this project, as we are focusing on OS-enforced isolation that does not depend on assumptions on applications.

Software timing channels also belong to this category, as they transmit information by deliberately altering execution length. A simple example of such channels is a Trojan program deliberately yielding the CPU for transmitting information, while time-multiplexing a core with a spy program. Software timing can be mitigated by padding the execution time of the Trojan [Askarov et al., 2010; Braun et al., 2015; Cock et al., 2014].

We also regard the attack that exploited the timing of transactional aborts in Intel's transactional synchronization extension (TSX) [Intel, a] as a timing channel due to self-contention. TSX is Intel's implementation of the hardware transaction memory [Wei et al., 2015], supporting user-level transactions that can result in either commits or aborts. During a transaction, TSX triggers the abort if the user-level program tries to access kernel addresses. Jang et al. [2016] discovered that the timing of raising the abort is different for differently configured kernel pages. On tested Haswell and Skylake processors, the abort caused by reading from a mapped kernel page is triggered faster than the one caused by reading from an unmapped kernel page. Similarly, trying to execute an executable kernel page caused a faster abort than non-executable ones. The main reason for this timing channel is that the MMU takes more time to confirm an unmapped or non-executable page before raising an abort. Using this timing channel, Jang et al. derandomised the address space layout randomization (ASLR) [Bhatkar et al., 2003] which is designed to prevent code-injection attacks in kernel. The standard defence against this attack is kernel address space isolation, which will be introduced in Section 3.3.4.

3.1.3 Buses and interconnects

In addition, we do not cover timing channels based on shared bus or interconnect (Section 2.3.5) [Hu, 1991; Wu et al., 2012], as attacks can easily exploit bus-based timing attacks by maliciously consuming bus traffic [Woo and Lee, 2007; Zhang et al., 2016]. As mentioned in Section 1.1, the bus and interconnect are bandwidth-limited resources which require concurrent access for creating any contention. For example, a thread running on a core can generate a large number of data-fetching requests, saturating the shared bus. Under such circumstances, threads running on other cores can suffer from a denial-of-service (DoS) attack if their progress is dependent on the bus bandwidth. Similarly, the interconnect network is vulnerable to DoS attacks caused by aggressive network traffic generated from malicious programs [Wassel et al., 2013].

Hu [1991] demonstrated that the system bus can be used as a covert channel, by modulating the traffic on the shared bus of a multicore system. For example, a thread (the Trojan) running on a core can generate a huge amount of bus traffic by sweeping a buffer that is larger than the LLC (hence causing bus traffic), which can be detected by a spy thread running on another core by monitoring its progress on accessing a similar sized buffer. Previous work demonstrated that this type of channel is exploitable as a covert channel on native machines as well as virtualised environments (e.g., cloud platforms) [Hu, 1991; Wu et al., 2012].

Similarly, threads running on separate cores suffer performance interference from competition on the interconnect network on multi-core systems. A program can hijack network traffic by generating memory requests, hence dominating the network bandwidth. Previous work simulated both covert and side channels that are based on network interference [Wang and Suh, 2012]. To demonstrate a covert channel, a Trojan program manipulates network traffic load, encoding bit "1" through high traffic load and bit "0" though low traffic load. Meanwhile, a spy receives the signal by measuring the throughput of its own memory fetches. Wang and Suh simulated a side channel attack on Rivest-Shamir-Adleman (RSA) encryption, with the assumption that every exponentiation execution generates memory requests due to cache misses. By monitoring the network traffic, a spy program can conduct a cryptographic attack, building correlation between the network traffic and exponentiation executions in RSA that represents the number of bits "1" in the secret key. Wang and Suh assume an unrealistically small cache and demonstrated the side channel in simulations only, hence there is no reason to assume this could be reproduced on realistic hardware.

Although buses and interconnects have been used for covert channels, there have been no side-channel attacks being demonstrated in a realistic setting. Mitigating covert channels on those resources requires either time-multiplexing all cores by assigning all cores (hence bus and network traffic) to programs from the same security domain, or partitioning the shared bandwidth with hardware mechanisms. However, contemporary mainstream hardware does not support bandwidth partitioning. Intel recently introduced the memory bandwidth allocation (MBA) mechanism, which imposes an approximate limit on bandwidth consumption introduced by a core [Intel, e]. The technique is insufficient for preventing covert timing channels, due to the approximate enforcement.

This means that covert channels through interconnects are unavoidable on contemporary hardware, and we have to restrict ourselves to scenarios where interconnects cannot be used as channels. We therefore assume that the system either runs on a single-core platform or time-multiplexing cores with *co-scheduling* security domains [Ousterhout, 1982], at least while the system is processing sensitive information. This assumption is clearly restrictive, but it is the best we can do on present hardware. Co-scheduling schedules all user-level processes that belong to the same security domain across all available cores, thus there is only one domain consuming the cross-core bandwidth.

3.1.4 DRAM attacks

Furthermore, we do not cover attacks based on DRAM contention, including attacks on row buffers, attacks causing bit-flips (i.e., Rowhammer attacks), or attacks on the ECC correction time.

Row buffers As introduced in Section 2.3.7, modern DRAM comprises a hierarchical structure of chips, banks, and cells. Cells are the smallest unit of storage in DRAM. Cells connected by the same wordline form a row. In each bank, there is a row buffer that caches the most recently accessed row. To access data, the DRAM first opens a row then fetches data stored in that row to the row buffer in that bank. If the row is already opened, the DRAM directly accesses the data cached in the row buffer (a row hit). Otherwise, the DRAM closes the opened row before opens the selected row (a row miss).

Pessl et al. [2016] demonstrated that the timing difference between a row miss and a row hit can be easily identified on x86 and Arm platforms, due to the fact that the row miss generates a higher memory latency than the row hit. Moreover, Pessl et al. reverse-engineered the DRAM addressing scheme using the timing channel on row buffer conflicts. Additionally, they implemented a covert channel based on row buffer conflicts, achieved a transfer rate of up to 2.1 MiB/s on the Haswell desktop platform (i7-4760) and 1.6 MiB/s on the Haswell-EP server platform (2x Xeon E5-2630 v3). Lastly, Pessl et al. conducted a side channel attack which learned keystrokes in the Firefox address bar: a spy process learnt when a specific memory location was accessed by a victim process running on the other core through row conflicts.

One possible mitigation of the row buffer attack is to partition the DRAM banks, prohibiting any contention on the row buffer in the bank. Similarly with PIPT caches (Section 2.3.1), the DRAM banks are physically indexed. [Kloda et al., 2019] demonstrated a method to extend the cache colouring technique (partitioning cache sets using physical frame numbers, Section 2.4) to partition DRAM banks by applying the knowledge of the DRAM addressing scheme [Pessl et al., 2016].

Rowhammer The continued scaling of the DRAM manufacturing technology increases the density of cells, resulting in smaller cells and less distance between them. These high-density DRAMs have a high risk of *disturbance error*, a phenomenon in which the interference between cells causes errors [Mandelman et al., 2002].

Although manufacturers have employed mitigations, Kim et al. [2014] discovered that 110 among 129 tested DRAM modules are vulnerable to disturbance errors. Their attack, called Rowhammer, induced disturbance errors by accessing a memory address repeatedly. The Rowhammer attack repeatedly activates a row in order to interfere with operations in adjacent rows through voltage fluctuations on the wordline of the opened row. Eventually, the disturbance causes cells in adjacent rows to lose charge before being restored (Section 2.3.7), inducing bit-flips. Most importantly, the Rowhammer attack allows an attacker to generate bit-flips in memory frames that belong to a victim process, bypassing memory protection as these frames are not mapped in the attackers' address space (Section 2.2). Later, Seaborn and Dullien [2015] demonstrated a probabilistic corruption on system page table entries using the Rowhammer attack, which caused the attacker to gain kernel privileges on x86 platforms. Moreover, Bosman et al. [2016] exploited a

JavaScript-based Rowhammer attack, a reliable Rowhammer attack on the Microsoft Edge browser.

The Rowhammer attack can also be launched from the integrated GPU (Section 2.3.8) on a system on a chip (SoC), as the main memory is shared between the GPU and the processor. Frigo et al. [2018] demonstrated the Glitch attack on the integrated GPU (Adreno 330), which is a remote end-to-end Rowhammer attack that allows an attacker to fully compromise the Firefox browser on Android platforms. To build the attack, Frigo et al. first reverse-engineered the cache architecture of the GPU, as well as its replacement policy. Then, they created an eviction set that allowed the attacking program, a vertex shader, to generate aggressive memory accesses that eventually caused the Rowhammer attack—generating bit-flips to overwrite read or write primitives enforced by the JavaScript sandbox in Firefox.

To mitigate the Rowhammer attack, Kim et al. [2014] proposed the probabilistic adjacent row activation (PARA) mechanism which ensures that the activation on a row also opens its adjacent rows with some low probability. Kim et al. evaluated the PARA on a DRAM simulator, and demonstrated that the mitigation is effective with negligible performance impact.

ECC The ECC (Section 2.3.7) stores extra parity bits for error correction and detection on DRAM chips. Cojocar et al. [2019] discovered that the error correction process contains measurable latency, which can be used as a timing channel. Cojocar et al. demonstrated ECCploit, a Rowhammer attack on ECC enabled DRAM chips. To prepare the attack, they reverse-engineered the ECC algorithm used on the target platform, with help from the ECC timing channel. Then, they built a memory model for locations of correctable bit flips, and constructed error patterns that cannot be corrected by the ECC. Lastly, the attack launched the Rowhammer attack using these uncorrectable patterns, which successfully corrupted page table entries [Seaborn and Dullien, 2015], RSA public keys [Razavi et al., 2016], and binary code (instructions stored in memory) [Gruss et al., 2018].

To mitigate the timing channel on the ECC, the system needs to remove any timing variances on ECC related operations conducted on memory controller. One potential solution is the in-DRAM ECC which performs ECC operations on die rather than on the memory controller [Cha et al., 2017].

3.1.5 DoS attacks

Lastly, our work does not involve any DoS attacks—system performance degradation due to aggressive resource consumption in real time [Woo and Lee, 2007]. The most common DoS attacks include polluting shared caches, such as the LLC, therefore dramatically slowing down all the other cores [Allan et al., 2016; Cardenas and Boppana, 2012].

Although the DoS attack is an active concern for ensuring the quality of service (QoS) of co-tenanted systems, our work focuses on mitigating timing channels due to resource contention. We exclude the DoS attacks from our research scope as QoS and system security are different. Still, the mechanisms introduced in our work can be applied to eliminate DoS attacks due to sharing.

3.2 Timing Attacks

3.2.1 Attacking techniques

To understand how microarchitectural attacks work, we study attacking techniques that have been used on modern hardware. Because covert channels and side channels use the same technique (Section 2.1), we assume a malicious attacker thread uses cache contention to target a secret held by a victim thread to simplify the description. As discussed in Section 2.3.1, hardware caches have a limited capacity, and loading a line from a cache is much cheaper than loading from the main memory. The critical factor of a cache-based timing attack is exploring the timing differences between cache hits and misses on a shared hardware cache through resource contention.

PRIME+PROBE This technique times repeated accesses to the attacker's working set, to detect an eviction caused by the victim. Firstly, the attacker *primes* cache sets by filling one or more sets with its own lines. Then, the attacker waits on the victim to finish an execution. Lastly, the attacker *probes* the victim's cache usage by timing access to previously loaded cache lines. If a cache line was evicted, the time to access the line is high, implying the victim visited the address that maps to the same cache sets. The XLATE+PROBE attack is a variant of PRIME+PROBE, which uses cached page table entries in the LLC as the probing set.

Covert channel using PRIME+PROBE A cache-based covert channel involves creating cache contention between the Trojan and spy. Figure 3.1 demonstrates a covert channel attack using the PRIME+PROBE technique: the Trojan encodes information in the number of cache sets accessed, and the spy decodes the message by probing on his colliding cache sets. The Trojan and spy are scheduled together on the same core, sharing all the on-core hardware resources, including caches. When scheduled, the Trojan decides on accessing a buffer (for sending a bit "1") or being idle (for sending a bit "0"), whereas the spy detects Trojan's activity by measuring the latency on accessing his cache-sized buffer. A longer latency represents that the Trojan accessed the buffer, as the spy senses cache misses; a shorter latency represents that the Trojan was idling, as the spy senses no cache miss. The Trojan's buffer does not need to be cache-sized: having a partial coverage on the shared cache is sufficient to create the attack. Moreover, more information can be sent than just a bit. For example, the Trojan and spy can communicate based on a sophisticated encoding scheme, hence enlarging the throughput of the channel.

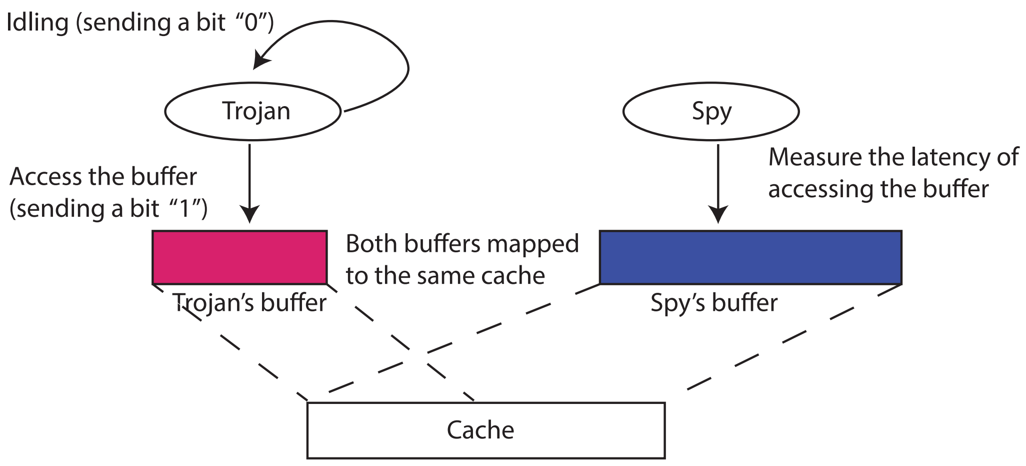

Idling (sending a bit "0")
Trojan
Spy
Access the buffer (sending a bit "1")
Measure the latency of accessing the buffer
Both buffers mapped to the same cache
Trojan's buffer
Spy's buffer
Cache

Figure 3.1: A covert cache-based timing channel using the PRIME+PROBE technique.

PRIME+ABORT This technique is similar to the PRIME+PROBE attack as it is also designed to detect cache contention on primed cache sets. Rather than timing the cost of a probe (i.e., revisiting the primed set), PRIME+ABORT utilises transactional aborts in Intel's TSX [Intel, a], Intel's implementation of hardware transactional memory [Wei et al., 2015], to detect any cache contention caused by other programs that share the primed cache sets. Intel's TSX allows multiple programs to access the shared memory in parallel [Herlihy and Moss, 1993], and triggers an abort if any suspected conflicts occur during a transaction. All changes made during a transaction are guaranteed to be executed atomically if no conflict is detected. To use TSX, a program declares the start and the end of a transaction section with `XBEGIN` and `XEND` instructions. Within a TSX transaction, the hardware tracks the read and write sets of the current transaction, and raises an abort if either any cache lines in the write set are evicted from the L1-D cache, or any cache lines in the read set are evicted from the L3 cache (i.e., the LLC In Intel's processors). There are also other reasons for a transactional abort but they are not related to the PRIME+ABORT attack.

There are two versions of PRIME+ABORT: PRIME+ABORT-L1 and PRIME+ABORT-L3. To conduct a PRIME+ABORT-L1 attack, the attacker first starts a TSX transaction, and primes single L1-D cache set using write operations (holding the cache set in his write set). Then, the attacker waits for an abort before detecting any conflicts caused by other processes on the primed cache set. PRIME+ABORT-L1 can only attack processes running on the same core, as it relies on the write set being evicted from the core-private L1-D cache. To address that restriction, the PRIME+ABORT-L3 holds the attacker's read set in L3 cache sets. Thus a TSX abort can detect cache contentions generated by processes running on other cores that share the same L3 cache. In Intel processors, the L3 cache is the LLC that is shared by all cores in the processor. The XLATE+ABORT attack is a variance of PRIME+ABORT that constructs probing set with cache page table entries on the LLC.

Evict+Time This approach measures the execution time of a victim, and evaluates the time difference before and after the attacker *evicts* targeted cache lines. A growing execution time indicates that the victim accessed those lines of interest, which is a reflection of cache misses.

Flush+Reload The FLUSH+RELOAD technique requires not only shared virtual memory between the attacker and the victim (e.g., shared libraries or page deduplication [Bugnion et al., 1997; Miłoś et al., 2009]) but also the ability to flush cache lines by virtual address. To implement the attack, the attacker first *flushes* a cache line of interest, then *reloads* the cache line after the victim has executed. A fast reload means that the victim accessed the cache line, whereas a slow reload shows the opposite. The advantage of FLUSH+RELOAD over PRIME+PROBE or EVICT+TIME is that the attacker can accurately detect the usage of a specific cache line, rather than only a cache set. A variant of the FLUSH+RELOAD attack, called EVICT+RELOAD, does not need a dedicated cache flushing instruction. Similarly, FLUSH+FLUSH is another variant of FLUSH+RELOAD, which measures the timing variance of a second x86 `clflush` instruction to detect whether the targeted cache line was cached or remained flushed.

Transient execution As introduced in Section 2.3.3, the out-of-order execution model can execute micro-ops speculatively, and decide to either abort or commit an executed micro-op when all conditions for making a decision are met. Before being aborted or committed, the execution of a micro-op is called *transient execution*. The transient execution does not affect the correctness of the architectural state, however, it can affect the microarchitectural state of a system. For example, a cache line that is loaded due to speculatively executed micro-op stays in the cache even though the micro-op is reissued. Likewise, a micro-op can cause a faulty load even when the CPU later decides to abort the corresponding instruction.

Transient execution attacks exploit microarchitectural states influenced by transient executions. Typically, the attack first executes a few instructions transiently, which are later aborted by the CPU. Then the attacker probes any changes to the cache state using other cache attacking techniques, such as FLUSH+RELOAD.

3.2.2 Timing attacks on core-shared state

Previous work has demonstrated both covert- and side-timing channels on hardware components shared by threads time-multiplexing on a core. As shown in Figure 2.5, resources shared by threads time-multiplexing on a core are functional units (e.g., the FPU), the BTB, the BHB, the TLB, the L1-D cache, and the L1-I cache. We also regard execution engines involved in the superscalar execution model (Section 2.3.3) as functional units shared by threads on a core, including store buffers and line fill buffers.

FPU The FPU contains a large state that is normally expensive to save and restore during a context switch. To optimise context switching latency, the OS simply disables FPU access and lazily switches FPU content if and when a process causes an exception by executing any floating-point operation. As a result, the latency of floating-point operations indicates other processes' activity on the FPU. Hu [1992] demonstrated that a thread can use the PRIME+PROBE technique on the FPU state to create a covert channel.

Speculative execution The speculative execution feature (Section 2.3.3) has been proved to be timing-attack prone, as demonstrated by the innovative Spectre attack [Kocher et al., 2019].

The Spectre attack demonstrates that side effects of a speculatively executed instruction, such as cache line state, remains even though the mis-predicted instruction was aborted. In other words, the speculatively loaded cache line remains in the cache even though the related branch instruction was mis-predicted. The Spectre attack is a transient execution attack.

As CPU predicts branch instructions based on the recent history, the Spectre attack first trains the branch predictor with its branch instructions. Then, it executes an instruction for loading a cache line that is not allowed under system's permission, such as visiting an address that is not within its address space. Due to the speculative execution, the CPU speculatively executes that instruction under the condition that the correct decision cannot be made immediately [Kocher et al., 2019]. Later, the attacker can detect the cache line loaded by speculatively executed instructions through a FLUSH+RELOAD attack.

Out-of-order execution The pipelining execution model has been demonstrated as a source of timing channel attacks. The Meltdown attack reveals the content of the kernel's address space by leveraging the out-of-order execution engine (Section 2.3.3) on the CPU pipeline [Lipp et al., 2018]. All attacks in this section are transient execution attacks.

Lipp et al. [2018] discovered that a hardware page fault exception can only be injected once the causing instruction is retired. Hence, the content stored in the faulting address has already been loaded to the cache, even though it violates the memory protection conducted by the MMU. Based on this, they conducted a side channel attack efficiently revealing the kernel's memory content using the FLUSH+RELOAD technique to probe the state of a target cache line. The attack issues instructions that trigger memory accesses with dependencies on kernel's memory content. Even though later aborted, these instructions successfully loaded cache lines that are indexed by kernel's memory content, due to the out-of-order execution engine in the pipeline. Then, the attacking program conducts uses the FLUSH+RELOAD technique to detect the loaded cache lines, which eventually leads to reverse engineer the kernel's memory contents.

Similarly, the Foreshadow [Van Bulck et al., 2018] timing channel attack successfully breaks the security protection provided by Intel's software guard extensions (SGX) technology. Intel's SGX is designed to provide confidentiality and integrity guarantees to applications hosted in its enclaves [Anati et al., 2013; Intel, a]. However, the Foreshadow attack breaks this promise by leaking plaintext enclave secrets from the L1-D cache. Furthermore, the next generation of the Foreshadow attack, the Foreshadow-NG attack [Weisse et al., 2018], allows the attacker to break other security boundaries enforced by the hardware. The Foreshadow-NG attack breaks the boundaries for hosting VMMs, VMs, OSes, and programs running in the system management mode (SMM), which is a special operating mode dedicated to handling system-wide hardware functions such as power management [Intel, c].

Store buffers As introduced in Section 2.3.4, the store buffer records information on outstanding store requests sent from the pipeline, as well as participates in pipelining optimisations, such as speculatively executed store-to-load forwarding.

Minkin et al. [2019] discovered that speculatively executed store-to-load forwarding allows the recently released Intel CPU to incorrectly pass writes to subsequent reads. The testing CPUs only use partial address bits to match the store address (for writing) against the load address (for reading), causing the adversary to load from an unrelated write. Even though later aborted, the faulty load remains in the cache. To exploit this, they invented the Fallout timing channel attack, which is a transient execution attack. The attacker first generates a faulty load that results in an incorrect cache line load, then conducts a FLUSH+RELOAD side channel attack to reveal the loaded cache line. The Fallout attack not only leaks recently written kernel data at user-level but also derandomises the kernel's ASLR.

Line fill buffers The line fill buffer (Section 2.3.4) serves ongoing data transmissions that involve cache hierarchy and main memory. Each entry in the line fill buffer records the destination, data, and status of this transmission.

According to the research conducted by van Schaik et al. [2019], line fill buffers in recently released Intel CPUs also participate in speculatively executing store-to-load forwarding. Similar to the store buffer, the line fill buffer can forward an ongoing store to a pending load only when their partial addresses match. To exploit that feature, van Schaik et al. invented a transient execution attack, called the rogue in-flight data load (RIDL) attack [van Schaik et al., 2019]. The attack utilises speculatively executed load operations (including the load port introduced in Section 2.3.4) on Haswell, Skylake, Coffee Lake, and Kaby Lake microarchitectures: the CPU speculatively executes the attacker's load operation based on the victim's previous stored secret that is recorded in the line fill buffer. Once the faulty cache line is loaded, the attacker performs a FLUSH+RELOAD attack to extract the secret value. As a result, the RIDL attack allows an attacking program running in unprivileged mode to leak information owned by another application, the OS, another VM or even a program protected by Intel's SGX.

Similarly, the ZombieLoad timing attack [Schwarz et al., 2019] exploited the transiently executed faulty loads on the line fill buffer, stealing data owned by the kernel from a user-level program. The ZombieLoad attack is also a transient execution attack that uses FLUSH+RELOAD to detect faulty loaded cache lines. According to Schwarz et al., ZombieLoad can recover kernel memory at one byte per 10 s with 38% accuracy.

BHB The speculative execution engine predicts subsequent conditional branch instructions based on the history stored in BHB entries which are indexed by virtual addresses. As virtual addresses are under the control of user-level programs, a process can easily train the branch prediction state by polluting the BHB entries with its execution history, which the goal of manipulating branch predictions for other programs executing on the same core. This BHB attack uses the PRIME+PROBE technique.

Cock et al. [2014] discovered that the branch mis-predictions from a separate process could affect the reported cycle counter value on the Arm Cortex A8 processor (AM3358). This timing variance can be exploited as a side channel by an attacker thread through corrupting the branch prediction history.

Evtyushkin et al. [2016b] demonstrated a mechanism for exploiting the branch prediction state as a covert timing channel. The attack exploits the branch mis-prediction rate that is impacted by the Trojan's execution history, causing measurable timing variations on the spy's execution time. As a result, the Trojan and spy create a covert timing channel by having a protocol on encoding messages into the mis-prediction rate. We will demonstrate our implementation of this channel in Section 5.3.1.5 for measuring the effectiveness of manufacturer-provided resetting operations for mitigating intra-core covert timing channels.

BTB Evtyushkin et al. [2016a] implemented a side channel attack using BTB collisions, which makes ASLR ineffective on Linux. The ASLR technique [Bhatkar et al., 2003] was initially introduced to prevent code-injection attacks, by randomizing both kernel-level and user-level virtual address space layout. Evtyushkin et al. discovered that the Intel Haswell platform only uses partial virtual address bits for both indexing and tagging BTB entries. Hence, they created collisions on shared BTB entries with the PRIME+PROBE technique. The prime set was created with virtual addresses in the same address space and the same virtual address from different address spaces. Based on the revealed partial virtual address bits they eventually disclosed the virtual address space layout of the targeted system.

RSB The return stack buffer (RSB) is a small buffer that the CPU uses to predict the return addresses of function-call operations: the function calling instruction causes a stack push, whereas a return instruction causes a stack pop. In other words, the RSB mirrors the local stack.

Bulygin [2008] demonstrated a side channel attack on the RSB which can be used to break RSA [Brumley and Boneh, 2003]: the attacker can detect an end reduction in the

Montgomery modular multiplication [Montgomery, 1985], by counting the number of RSB entries that had been used by the RSA encryption. The attack conducted a form of the PRIME+PROBE attack on RSB entries. The attacking program first primes the RSB entries with its function calling instructions, then it invokes the RSA encryption process which can potentially replace the primed RSB entries. Later, the attacker measures its footprint left on the RSB entries by conducting return instructions, to detect the replacement triggered by the RSA encryption.

TLB Hund et al. [2013] showed that TLB contention allows an attacker to defeat ASLR [Bhatkar et al., 2003], with a variant of the FLUSH+RELOAD technique. They exploited the fact that the latency of segmentation faults reveal TLB mappings cached by the kernel for servicing system calls: invalid mappings are not cached in the TLB so any subsequent access triggers an expensive page table walk; valid mappings produce a much more rapid segmentation fault. They successfully defeated ASLR on both Windows 7 Enterprise and Ubuntu Desktop 11.10 running on three different Intel microarchitectures, with a worst-case accuracy of 95% on the Intel Sandy Bridge architecture.

L1-D and L1-I caches To evaluate the security of the VAX/VMM system, Hu [1992] stated that shared caches can lead to timing channels, which can be easily exploited using the PRIME+PROBE technique, which was confirmed by later work. With knowledge of AES's S-boxes, Osvik et al. [2006] and Tromer et al. [2010] attacked AES secret keys by detecting L1-D cache contention using PRIME+PROBE and EVICT+TIME techniques. Similarly, Acıiçmez [2007] presented an RSA attack using the PRIME+PROBE technique on the L1-I cache, assuming frequent preemptions on RSA processes. Later, Zhang et al. [2012b] showed similar key-recovery attacks were even practical on ElGamal [ElGamal, 1985] between virtual machines, by using inter-processor interrupts (IPIs) to preempt the victim frequently.

3.2.3 Timing attacks on package-shared state

As shown in Figure 2.5, resources shared by threads running on different cores of a package are the LLC and buses for inter-core traffic. This work does not include timing channels based on shared buses or interconnects (Section 3.1.3). We focus on the timing channels based on the LLC shared by cores within a package.

The LLC holds a footprint left by all threads running on a package, which can be used as either a covert or a side timing channel by threads sharing a core (Section 3.2.3.1) or concurrently running on multiple cores on the same package (Section 3.2.3.2).

3.2.3.1 Time slicing

Since Hu [1992] demonstrated the possibility of transmitting information through cache collisions using the PRIME+PROBE technique, many researchers have explored covert timing attacks on the shared LLC between normal processes [Percival, 2005] or between VMs on a public cloud (Amazon EC2) [Ristenpart et al., 2009] on a single core. Moreover, Ristenpart et al. [2009] detected co-residency on the Amazon EC2 platform with LLC contention.

With the FLUSH+RELOAD technique, Gullasch et al. [2011] attacked AES in OpenSSL 0.98n with help from the completely fair scheduler on Linux to pre-empt the AES thread frequently. Similarly, Irazoqui et al. [2014] used FLUSH+RELOAD to break AES in OpenSSL 1.0.1f with page sharing offered by the VMware ESXi5.5.0.

Past work also demonstrated applying the EVICT+TIME technique to create cache contention on the LLC. Hund et al. [2013] measured the effect of EVICT+TIME on system-call latencies on Linux, and was able to decode the kernel address space layout.

To conduct a PRIME+PROBE attack on the LLC, the attacker has first to solve the mapping scheme, which can be as complicated as a two-level hashing function on more recent Intel platforms [Hund et al., 2013; İnci et al., 2016; Intel, b; Irazoqui et al., 2015b; Maurice et al., 2015; Yarom et al., 2015]. Irazoqui et al. [2015a] overcame that problem by building probing buffers on huge pages, and was able to recover an AES key on both Xen 4.1 and VMware ESXI 5.5. Oren et al. [2015] demonstrated that a PRIME+PROBE attack was also possible without huge pages. Oren et al. presented schemes for collecting the cache footprint as a signature of the target's actions, achieving a covert channel with 320 kb/s throughput and a side channel that identifies mouse or network activities. Lipp et al. [2016] showed that the Arm architecture is also vulnerable to the PRIME+PROBE with an AES attack.

The processor can also use the LLC to store recently accessed page table entries (Section 2.2). Gras et al. [2017] discovered that the cached page table entries can be detected by using the EVICT+TIME attack on the LLC, resulting in an attacking program, running as a JavaScript in Firefox and Chrome, that can derandomise the ASLR implementation in these browsers. To conduct that attack, Gras et al. first detected cached page table entries that are loaded due to resolving a TLB miss (Section 2.3.1) through the EVICT+TIME attack on LLC sets. Then they broke the ASLR implementation through the connection between page table entries and the virtual-to-physical address translation configured for ASLR.

3.2.3.2 Multicore

The challenges of building a covert channel between concurrently running threads include coping with the scheduling uncertainties on the hosting system [Wu et al., 2012], reverse engineering the virtual-to-physical address mappings [Liu et al., 2015], and solving the hash table mapping function on the LLC [Hund et al., 2013; İnci et al., 2016; Irazoqui et al., 2015b; Maurice et al., 2015; Yarom et al., 2015].

Xu et al. [2011] demonstrated a covert timing channel using the PRIME+PROBE technique: the attack achieved 3.2 b/s throughput on Amazon EC2 [Xu et al., 2011], with an improved transmission protocol compared to the initial design by Ristenpart et al. [2009] on a multicore host (Section 3.2.3.1).

The FLUSH+RELOAD attack is also applicable on multicore between OS hosted processes or VMs on public clouds. Yarom and Falkner [2014] successfully attacked RSA with the FLUSH+RELOAD attack, with the assumption that the attacker has read access to the in-memory RSA implementation through either memory mapping or page de-duplication [Bugnion et al., 1997; Miłoś et al., 2009]. Later, Yarom and Benger [2014], Benger et al. [2014] and van de Pol et al. [2015] conducted an attack on the secret key used in ECDSA with a similar technique. Additionally, Zhang et al. [2014b] implemented a FLUSH+RELOAD attack between co-resident VMs on DotCloud [DotCloud], to steal fine-grained information (e.g., end-user's shopping activity) on a platform-as-a-service (PaaS) cloud platform.

Gruss et al. [2015] generalised the FLUSH+RELOAD technique with pre-computed cache template matrices, allowing attackers to efficiently compute the similarities between the matrices and online cache-hit profiles for both data and instruction accesses. The cache template matrices attack is capable of attacking both keystrokes and AES on OpenSSL. Furthermore, Gruss et al. [2015] conducted the EVICT+RELOAD attack, a variant of the FLUSH+RELOAD attack as mentioned in Section 3.2.1, which does not need a dedicated cache flushing instruction. Moreover, Gruss et al. [2016b] achieved a bandwidth of 496 Kib/s for a covert channel using FLUSH+FLUSH, another variant of FLUSH+RELOAD, demonstrating FLUSH+FLUSH is faster than FLUSH+RELOAD.

To build a PRIME+PROBE attack on the LLC, Liu et al. [2015] presented a solution for constructing a prime buffer for the physically-indexed and -tagged LLC sets. They implemented not only a high throughput covert channel (1.2 Mb/s), but also a side channel attack on the square-and-multiply exponentiation algorithm in ElGamal (GnuPG 1.4.13). İnci et al. [2016] extended the attack to the Amazon EC2 cloud, leaking both cloud co-location information and ElGamal private keys (GnuPG 1.4.18).

As introduced in Section 3.2.1, the PRIME+ABORT attack is similar to the PRIME+PROBE attack, but replaces the probing stage with a transactional abort in Intel's TSX. Disselkoen et al. [2017] demonstrated a PRIME+ABORT-L3 attack which reveals the secret key used by AES through conflicts on the LLC detected by transactional aborts.

van Schaik et al. [2018] introduced the XLATE+PROBE attack that builds the probing set used in the PRIME+PROBE attack with cached page table entries in the LLC. Their attack allows an attacking program to break the T-table implementation of the AES implementation in OpenSSL. Moreover, the attack is efficient even though the attacking program and the victim program (an AES server) are partitioned on the LLC using either set or way partitioning schemes (Section 3.3.5), due to the fact that cache sets used by page table entries are still shared. Also, van Schaik et al. extended the attack using the transactional aborts in Intel's TSX. Their extended attack uses the XLATE+ABORT attacking technique, which detects any conflicts on cached page table entries caused by other processes during a TSX transaction. The XLATE+ABORT attack breaks AES in OpenSSL 1.0.1e.

3.2.4 Summary

We observe that microarchitectural timing attacks appear on all levels of the resource hierarchy, from thread- and core-level shared resources to package-shared resources. In terms of types of explored timing channels these resources have been used for both side and covert channel attacks.

Attacks are first demonstrated on high-level shared resources, such as those on the FPU or L1 caches. Attacks on L1 caches are done through exploiting the cache contentions between processes [Acıiçmez, 2007; Acıiçmez and Schindler, 2008; Neve and Seifert, 2006; Osvik et al., 2006; Tromer et al., 2010] or even across VMs [Zhang et al., 2012b]. Furthermore, attacks on L1 caches are also applicable to the L2 cache, if there is a core-private L2 cache available. Similar attacks are also exploited on intra-core cache-like components, including the BHB [Cock et al., 2014; Evtyushkin et al., 2016b], the BTB [Evtyushkin et al., 2016a], and the TLB [Hund et al., 2013]. The attack on the RSB [Bulygin, 2008] is different to attacks on caches, because the RSB does not contain any indexing scheme but works as a rolling buffer.

Later, attacks are demonstrated on lower-level shared resources like the LLC. Timing attacks based on the LLC can be classified into two categories: relying on shared memory (e.g., page duplication) or no such requirement. Started by FLUSH+RELOAD [Gullasch et al., 2011], attacks relying on shared memory have been demonstrated between processes through the shared library code, between programs sharing the same core [Gullasch et al., 2011; Irazoqui et al., 2014], as well as between programs running on different cores [Benger et al., 2014; Gruss et al., 2015; van de Pol et al., 2015; Yarom and Benger, 2014; Yarom and Falkner, 2014]. The cross-core FLUSH+RELOAD attack has also been demonstrated between co-resident VMs [Zhang et al., 2014b], based on memory shared due to deduplication [Bugnion et al., 1997; Miłoś et al., 2009].

LLC attacks without relying on shared memory are a recent evolution: they have been demonstrated by time-multiplexing a core [Gras et al., 2017; Irazoqui et al., 2015a; Lipp et al., 2016; Oren et al., 2015], or cross-cores [Disselkoen et al., 2017; İnci et al., 2016; Liu et al., 2015; van Schaik et al., 2018].

A series of innovative attacks have demonstrated exploitation of transitive executions in CPUs. These attacks abused the transitively executed micro-ops due to the speculative execution [Kocher et al., 2019], the out-of-order execution [Lipp et al., 2018; Van Bulck et al., 2018; Weisse et al., 2018], and the store-to-load forwarding on the store buffer [Minkin et al., 2019] and the line fill buffer [Schwarz et al., 2019; van Schaik et al., 2019].

3.3 Countermeasures

3.3.1 Constant-time techniques

To protect cryptographic computation, a common technique is to ensure behaviour is never dependent on a secret (e.g., branch operations do not depend on secrets). This approach has been applied both to local contention-based channels and remote timing channel attacks. For example, the design of the NaCl library [Bernstein et al., 2012] implements many constant-time techniques for avoiding OpenSSL's vulnerabilities.

Program analysis framework Several research projects presented tools and formal frameworks for analysing the behaviour of constant-time implementations. Langley [2010] traced the flow of secret information based on a modified Valgrind [developers] program analysis tool. Similarly, Köpf et al. [2012] and Doychev et al. [2015] described methods for estimating an upper bound on information leakage through timing variations. Moreover, FlowTracker [Silva et al., 2015] conducted analysis on side-channel attacks using a modified LLVM compiler [Lattner and Adve, 2004].

Hardware support One important approach to avoid secret-dependent table lookups is to provide hardware operations for cryptographic primitives [Page, 2003]. The x86 architecture provides a set of instructions for AES implementations, assisting encryption, decryption, key expansion and all modes of operations in AES [Gueron, 2009, 2010], which is supported by both Intel [Xu, 2010] and AMD [AMD] architectures. Similar support is also available on other architectures, including Arm [ARM], and SPARC [ORACLE].

Language-based approaches Language-based approaches provides specialised language semantics with constant execution latency, together with a corresponding hardware design [Zhang et al., 2012a]. This type of solution emphasises the importance of coordination between hardware and software.

Discussion Constant-time techniques are difficult to implement due to their high degree of complexity [Bernstein, 2005]. For instance, the sequence of cache accesses needs to be secret-independent for preventing any possible cache-based timing attacks. Previously, Brickell [2011] noted having secret-dependent memory access within a cache line cannot leak secret data. However, Bernstein and Schwabe [2013] showed Intel processors can leak cache-offset bits, which was exploited recently by Yarom et al. [2016] with the CacheBleed attack on the OpenSSL implementation. Moreover, Coppens et al. [2009] listed several possible leaks even though the sequence of memory access is secret-independent, such as instruction timing, or register dependencies. To address these issues, they designed a compiler that prohibits control-flow dependencies on secret keys of cryptographic algorithms on x86.

Another shortcoming of the constant-time technique is that a constant-time program may not perform consistently across hardware platforms: Cock et al. [2014] discovered that the constant-time fix on OpenSSL 1.0.1e for the Lucky 13 remote side-channel timing attack [AlFardan and Paterson, 2013], does not remove the vulnerability on the Arm AM3358 platform.

The performance impact of the constant-time technique is application dependent.Applying the constant-time technique to a specific application is generally more affordable than applying it to all programs. The NaCl library [Bernstein et al., 2012] outperformed its peers, with the benefit of promising no secret-dependent behavior on either loads or branch instructions. In contrast, applying the constant-time technique in the LLVM compiler [Lattner and Adve, 2004] introduced up to a 2.4 times slowdown to the openSSL test [Coppens et al., 2009].

3.3.2 Injecting noise

The noise injection technique prevents a timing channel by corrupting the attacker's measurement with garbage data (i.e., noise). Previous work suggested injecting noise into both timing sequence and resource usage patterns.

Noise injection on timing Theoretically, injecting noise into all measurable timing sequences in a system can prevent timing attacks, as any of the attacker's timing measurements are useless due to the noise. *Fuzzy time* is a technique to inject noise into all visible events in a system, such as system ticks or interrupt delivery [Hu, 1991].

Vattikonda et al. [2011] virtualised the timestamp counter value read by VMs, by inserting noise into the return value of the `rdtsc` instruction, the instruction for reading the timestamp counter on x86. Similarly, Martin et al. [2012] implemented a system solution for fuzzing any possible timing variance measured through hardware (`rdtsc`) or software clocks, with the assumption of no external timing source. They conducted a statistical analysis on this solution, demonstrating the mitigation on timing channels.

Noise injection on resource consumption Brickell et al. [2006] suggested an alternative AES implementation involving compacting, randomising and preloading lookup tables, in order to introduce noise to the cache footprint left by AES executions.

Wang and Lee [2007] invented the random permutation cache (RPcache), a cache containing both randomised indices and protection attributes. With RPcache, each process owns a permutation table for memory-to-cache mapping. Furthermore, each cache line contains an owner identification which is used to locate the permutation table owned by corresponding process. Cache lines belonging to different processes cannot evict each other. To evict a cache line, RPcache randomly selects a cache set as the victim set. Later, Kong et al. [2009] proposed an implementation which supports explicitly requested RPcache functions for sensitive data processing, such as AES table lookups. Kong et al. evaluated the solution on simulated hardware, demonstrating the potential for low overhead on real hardware implementation.

To eliminate reuse-based attacks, Liu and Lee [2014] designed a cache with a random replacement policy. Their design does not allocate cache lines for serving cache misses. Instead, the mechanism directly sent requested data to the processor. For generating a randomised footprint on the cache, the mechanism allocates cache lines filled with randomised fetches within a configurable neighbourhood window of the missing memory line.

Zhang et al. [2014a] introduced a bystander VM hosted on Xen to inject noise into the cross-VM L2-cache covert channel. They created a continuous-time Markov process for modeling the PRIME+PROBE-based cache covert channel, and analysed the impact of the bystander VM on the error-rate of the modeled channel. They draw the conclusion that the bystander VM cannot significantly impact the cross-VM covert channel by only adjusting its CPU consumption. Moreover, the bystander VM must modulate its working set and memory access rates for effectively weaken the channel bandwidth.

Discussion Noise injection is inefficient for system security assurance. Completely closing the timing channel requires anti-corrected "noise" injection [Cock et al., 2014], which is impossible to produce in many system circumstances. Most importantly, noise injection reduces the signal-noise ratio but cannot eliminate the signal, and the amount of noise needed to reduce this ratio is massive hence severely degrade system performance.

The performance impact introduced by noise injection depends on the scale of the application. For example, the RPcache only introduced a small performance hit, 1%, in the SPEC2000 benchmark for injecting noise into a 32 KiB cache [Wang and Lee, 2007]. Similarly, dedicating a small RPcache for processing sensitive data introduced very low overhead [Kong et al., 2009]. However, the amount of noise required increases significantly as the system reduces the residual channel further. This dramatically penalises system performance even though the channel bandwidth can be reduced by no more than about two orders of magnitude [Cock et al., 2014].

3.3.3 Enforcing determinism

Another line of work attempted to eliminate timing channels by enforcing a deterministic system sequence with virtual time and black-box techniques.

Virtual time The virtual-time approach provides only virtualised clocks, whose progress is completely deterministic. Virtualising time implies that the progress of virtual time is independent of any secret data processing. Aviram et al. [2010a] repurposed Determinator [Aviram et al., 2010b], a framework for debugging concurrent programs with deterministic execution sequences, as a cloud computing environment with only virtual time. Ford [2012] further extended this model to allow scheduled external I/O events. Later, Wu et al. [2015] produced a hypervisor-enforced deterministic execution system for both internal and external event sequences. Wu et al. also introduced the concept of *mitigation interval*, a time interval that bounds any theoretical information leakage. Instead of providing a deterministic execution environment, StopWatch [Li et al., 2013] hosts three replicas of a system process, and virtualises the x86 time-stamp counter with the median of the times obtained from the replicas.

Compared with prohibiting access to any real time source, instruction-based scheduling (IBS) is a more limited form of deterministic execution that only prevents an attacker from using the system preemption tick as a clock source. IBS guarantees deterministic progress of a program in every system tick by generating preemption interrupts after a fixed number of instructions. Dunlap et al. [2002] proposed IBS as a debugging technique, which was later integrated into Determinator [Aviram et al., 2010a] with the help from the CPU's performance monitoring unit (PMU). Additionally, Stefan et al. [2013] demonstrated IBS as an approach to mitigate timing channels. Cock et al. [2014] discovered that the imprecise delivery of PMU's interrupts have negative impact on the effectiveness of this technique. To schedule processes after a precise number of instructions being executed, the system has to configure the PMU to trigger an interrupt earlier than the targeted count, then single-stepping the CPU until the target is reached [Dunlap, 2006]. According to Wu et al. [2015], the single-stepping technique introduced 30% overhead to CPU-bound benchmarks for a 1 ms mitigation interval, and 5% overhead for a 100 ms mitigation interval.

Black-box mitigation The Black-box approach controls the timing of externally visible events, by achieving determinism for a system as a whole.

Köpf and Dürmuth [2009] proposed a bucketing solution for system response time, to calculate a theoretic upper-bound on information leakage. Additionally, Askarov et al. [2010] enhanced Köpf and Dürmuth's work with an exponential back-off policy to calculate a hard upper bound on information leakage. Cock et al. [2014] demonstrated that such a kind of policy can be implemented efficiently on the seL4 microkernel to mitigate the Lucky 13 attack on OpenSSL TLS [AlFardan and Paterson, 2013].

Braun et al. [2015] designed a set of compiler directives for automatically generating fixed-time functions with temporal padding. In addition, the padding masks any possible timing leaks caused by limited temporal resolution with randomisation.

Discussion Virtualising all types of system time is effective against all types of timing attacks, because of the non-existence of both internally and externally measurable timing sequences. However, these systems introduce a heavy performance penalty because they are fully deterministic. For instance, StopWatch introduced 2.8 times overhead on network-intensive benchmark and 2.3 times overhead on computation-intensive benchmarks [Li et al., 2013]. Secondly, fully deterministic systems, such as the Determinator [Aviram et al., 2010b], rely on custom-written software and cannot support legacy applications that require real time. Most importantly, preventing any access to real time is often infeasible, as any internal (e.g., the progress of a program) or external (e.g., network bandwidth) events can be used as timing references.

Rather than synchronising all time sources measurable by individual processes, the black-box technique controls only the timing of externally visible events. Hence, the black-box technique is only effective against remotely exploitable attacks that exploit system response latency, as all the other internally visible timing sequences remain unchanged. Comparing with virtualising all possible timing sources, this approach is not only easier to implement but also less expensive.

Our work is different compared to the existing black-box approach by regarding a security domain as a black-box, to prevent information leakage between security domains through timing. The OS mechanisms introduced in this work conduct security enforcement for supporting confinement on microarchitectural timing channels.

3.3.4 Partitioning time

Time partitioning mitigates attacks that exploit contention on shared resources by granting exclusive access within a timeslice and carefully managing the transition between timeslices.

Kernel address space isolation To mitigate the timing attack on prefetch instructions, Gruss et al. [2016a] proposed to separate kernel and user address spaces with dedicated kernel address space mappings. The kernel has exclusive access to its address space. As a result, the kernel updates the page directory setting before starting any kernel executions, switching the virtual-to-physical address mapping from user space to kernel space. This technique also mitigates the Meltdown attack [Lipp et al., 2018] which loads cache lines that are indexed from kernel's memory content.

Execution leases Tiwari et al. [2009] designed a new hardware mechanism, execution leases, to share execution resources among threads. When a lease expires, a trusted entity gains control, and expels any untrusted operations. They prototyped an in-order, un-pipelined CPU core that contains extra hardware components for maintaining lease contexts. The system is inherently slow because its design prohibits performance optmizations, such as a TLB or branch prediction unit (BPU). Later, Tiwari et al. [2011] extended the execution lease with their CPU implementation (a Star-CPU), a microkernel, and an I/O protocol, providing top-to-bottom information flow guarantees.

Minimum timeslice To attain a fine-grained view of the cache footprint, many attacks require frequently preempting the victim process, such as the PRIME+PROBE attack implemented by Zhang et al. [2012b]. To prevent this type of attack, Varadarajan et al. [2014] introduced latency on responding to system events, extending the minimum preemptable period.

Flushing store buffers Because the store buffer is not shared between hardware threads, flushing the store buffer while switching between threads from different security domains is sufficient to mitigate the Fallout attack [Minkin et al., 2019] (Section 3.2.2), the attack that exploits transient write forwarding as a timing channel. Minkin et al. discovered that the `mfence` instruction provided by Intel [Intel, e] can mitigate the Fallout attack. The `mfence` instruction not only drains the store buffer but also serialises previously issued load and store operations [Intel, e]. Additionally, the store buffer is automatically flushed by updating the `CR3` register [Minkin et al., 2019] which points to the top-level page table used for the virtual-to-physical address translation (Section 2.2 on Intel CPUs [Intel, e].

Cache flushing Flushing system state is an obvious solution to defend against attacks based on persistent state effects (e.g., cache footprint). For example, a system can mitigate cache-based attacks by flushing all level of caches [Godfrey and Zulkernine, 2013], including mitigating cross-VM side channels on core-private caches [Zhang and Reiter, 2013]. However, cache flushing cannot mitigate cross-core timing channels, where programs exploit cache contentions concurrently on different cores.

Many architectures provide cache flushing operations. For example, Arm offers operations for selectively flushing different levels of caches [ARM, 2008]. Intel architecture recently announced the L1-D cache flushing operation [Int, 2018b], as a defence mechanism for the Foreshadow and the Foreshadow-NG attacks [Van Bulck et al., 2018; Weisse et al., 2018]. However, Intel's L1-D cache flush operation has not been supported by all the hardware manufacturers, due to the unavoidable latency in the microcode updating process.

To mitigate the Spectre attack [Kocher et al., 2019], Intel recently launched a new interface between the processor and system software, the IBC mechanism [Intel, 2018c], allowing the OS to prevent an attacker from maliciously polluting the branch prediction history. The IBC mechanism includes the *indirect branch restricted speculation* across privilege modes, the *single thread indirect branch predictors* for preventing indirect branch predictions from being controlled by other hardware threads, and the *indirect branch predictor barrier* for preventing indirect branch predictions being controlled by previous history.

Lattice scheduling Initially proposed by Denning [1976], and implemented by Hu [1992] in the VAX/VMM security kernel, *lattice scheduling* amortises the cost of flushing all caches by only triggering those actions when switching from sensitive domains to untrusted ones. As a result, there is no timing information leakage from those sensitive domains. Cock [2013] verified the lattice scheduler on the seL4 microkernel for domain scheduling. This approach requires the system to have a hierarchical trust model. Therefore, a system with a mutual distrust model, such as cloud computing, cannot apply lattice scheduling.

Discussion Temporal partitioning (i.e., strict time multiplexing) is efficient for eliminating attacks that require the sequential use of shared resources. For a given partitioning scheme, its performance impact depends on the *direct cost* of the partitioning (e.g., the latency of cache flushing) as well as the *indirect cost* of the partitioning (e.g., the impact on system performance due to flushing). Here we use the cost of cache flushing as an example.

The cost of the cache flush includes direct cost, the cost of flushing operations, and the indirect cost, that is, the performance slowdown experienced by user-level programs while running on a cold cache. Flushing the top-level caches is affordable, as the size of an L1 cache is relatively small (e.g., 32 KiB on x86 platforms), resulting in less direct cost. For instance, Varadarajan et al. [2014] measured an 8.4 μs direct cost on flushing L1 caches that is conducted with a software cache flush, on a 6 core Intel Xeon E5645 processor. Moreover, they measured a 17% increase in the ping latency benchmark using a 1 ms interval.

Furthermore, the indirect cost is also small if the user-level programs are heavyweight, such as VMs hosted on the cloud, because a newly scheduled VM is unlikely to reuse any cache lines in the data or instruction cache. Therefore, conducting a L1 cache flush on a VM switch does not introduce much indirect cost. Lastly, the flushing is even more affordable if the content switching frequency is low, such as 30 ms on Xen for scheduling VMs [Zhang et al., 2012b].

By contrast, flushing lower-level caches is much more likely to introduce a significant performance degradation, because of their large capacity (e.g., 8 MiB on the Intel Core i7-4770). Additionally, flushing the LLC cannot prevent cross-core cache attacks, as the cache contention is exploited concurrently.

In summary, flushing top-level caches–caches shared by threads within a core–is an affordable operation for mitigating intra-core cache channels. However, flushing all levels of caches is an expensive operation, and cannot prevent cross-core cache attacks.

3.3.5 Partitioning hardware

Partitioning hardware can mitigate not only attacks relying on consecutive access, but also those relying on concurrent access to hardware resources.

Disable page sharing Disabling page sharing prevents FLUSH+RELOAD and its variations, because this types of attacks depend on shared memory frames. In practice, VMware Inc. [2014] recommends turning off the page sharing feature [Waldspurger, 2002] to prevent FLUSH+RELOAD cross-VM attacks. Similarly, Zhou et al. [2016] proposed the CacheBar, a copy-on-access mechanism, which duplicates a copy of a page while another security domain is accessing that page.

Disable speculative store forwarding To mitigate attacks on speculative store-to-load forwarding [Schwarz et al., 2019; van Schaik et al., 2019], Intel provided a microcode update for disabling the speculative store forwarding feature [Intel, 2019b]. The microcode update prevents the CPU from executing store-to-load forwarding speculatively (Section 2.3.4), hence can be applied during the context switch process for enhancing the memory protection boundary across privilege levels.

Hardware cache partitions Hardware-enforced cache partitioning provides a powerful mechanism against cache-based attacks. Percival [2005] recommended a L1 cache partitioning hardware design. Wang and Lee [2007] designed the partition-locked cache, PLcache, to assign locking attributes on cache lines.

Intel's cache allocation technology (CAT) provides a cache-way allocation service on the LLC [Intel, d], that prevents processes from evicting unallocated cache ways. CAT is designed for enhancing the QoS of a system, rather than security. Using CAT, Liu et al. [2016] demonstrate a software technique, CATalyst, to defeat the PRIME+PROBE and EVICT+TIME attacks on the LLC. Liu et al. created a secure cache partition that stores protected memory contents, acting as a pinned cache managed by software. To use the service, a user-level program allocates cache space, and loads protected pages into the secure partition by simply accessing them. CATalyst is effective in mitigating the PRIME+PROBE attack on the square-and-multiply algorithm in GnuPG 1.4.13 with insignificant performance overhead, only 0.7% for the SPEC benchmark and 0.5% for the PARSEC benchmark.

Arm v7 processors allow cache ways to be locked in L1-I or L1-D caches [ARM, 2008], which is used by Colp et al. [2015] to provide a small amount of safe on-chip storage for encryption keys.

Cache colouring As introduced in Section 2.4, cache colouring partitions a physically-indexed cache according to the memory frame numbers (i.e., physical addresses). Here we explain the related work that applied cache colouring to mitigate cache-based timing channels.

Shi et al. [2011] implemented a dynamic cache colouring solution for threads executing cryptographic algorithms hosted in a hypervisor. STEALTHMEM [Kim et al., 2012] provided a small amount of coloured memory, called stealth pages, for storing security-sensitive data without cache contention. The design assigned stealth pages with different colours to each core, and monitored the usage of memory frames having same colours as stealth pages through the page table alert mechanism. With help from page fault exceptions, STEALTHMEM monitors the number of pages being loaded on each cache colour, making sure none of those stealth pages being evicted. In other words, the STEALTHMEM pinned the stealth pages in the LLC. The overhead of STEALTHMEM is relatively small for the SPEC 2006 CPU benchmark: measured as 5.9% for the STEALTHMEM and 7.2% for handling extra page faults. Furthermore, using stealth pages introduced 2–5% overhead on three block ciphers: AES, data encryption standard (DES) and Blowfish.

Cock et al. [2014] evaluated the effectiveness of cache colouring by evaluating Shannon capacity [Shannon, 1948], which represents the average amount of information leaked by a channel assuming the receiver has an unbounded computation power. Cock et al. discovered that the cache colouring approach is more effective on simpler cores (Arm iMX.31, AM3358, or DM3730), compared with more complex cores (Arm Exynos4412, or Intel E6550). The residual channel in the latter cases were caused by TLB contention, which can be mitigated with TLB flushing.

Quasi-partitioning CacheBar [Zhou et al., 2016] prevents an attacker from monopolising a shared cache by actively evicting cache contents. To ensure each protection domain only consumes a limited number of cache lines, CacheBar assigns a budget to each domain, representing its cacheable allowances. The design also maintains a least recently used queue per domain for monitoring its cache occupancy. The method is essentially a software implementation of the countermeasure suggested by Domnister et al. [2012]: reserving cache lines in each L1 cache set for a hardware thread. However, reducing the capacity of L1 cache by partitioning is highly likely to produce much more overhead than flushing, as a smaller L1 cache will introduce an increasing number of L1 cache misses causing the pipeline to slow down.

Migrating VMs Moon et al. [2015] implemented a migration-as-a-service cloud computing service that periodically runs the VM placement algorithm with both the current and past VM assignments as inputs, mitigating information leakage across co-resident VMs. The replacement algorithm designed by Moon et al. achieves near-optimal information leakage subject to the migration overhead, which is also scalable to large-scale cloud platforms.

Discussion Hardware partitioning (i.e., spatial partitioning) is effective for eliminating attacks that exploit the use of shared resources both consecutively and concurrently. To make a fair evaluation, a system should evaluate the performance impact of a partitioning technique while the system is in a quiescent state (i.e., no other running programs), as well as while the system hosts other programs (i.e., potential competitors on system resources if they are shared).

Here we use cache colouring (Section 2.4) as an example, where each program can only run on its partition of the cache. To simplify the description, we assume that a system has only two programs, A and B, which each owns half of the cache. In other words, both A and B can only execute on half of the cache. Running on half of the cache is highly likely to reduce the performance of A if its working set is larger than its partition. By contrast, A may not observe any performance impact if its working set can fit into the partition. To demonstrate, previous work measured the cost of cache colouring in Xen's memory management module [Godfrey, 2013], and showed that the overhead is proportional to the working set given a small cache—a 50% performance cost for an Apache

[2013] macrobenchmark, and no significant penalty for benchmarks with small working sets [Godfrey, 2013].

However, the penalty can be different if *B* is also running. In the non-partitioned scenario, *B* can potentially compete for the cache, resulting in the slowdown of *A*. Thus, we need to compare the cost of static partitioning, allocating half of the cache to *A*, with dynamic partitioning, the cache line allocation performed by the hardware while both *A* and *B* are running. Therefore, a system should evaluate a partitioning scheme against the dynamic allocation carried by the hardware, before drawing any conclusion on its performance impact. Additionally, the performance of *A* can be more predictable with static partitioning than with dynamic partitioning, as *A* enjoys a dedicated share of the cache [Liedtke et al., 1997].

3.3.6 Summary

The constant-time technique (Section 3.3.1) is effective for mitigating timing channels in a particular program, by ensuring no secret-dependent resource consumption during program's execution. However, a constant-time implementation of a program may behave differently across hardware architectures. Furthermore, the constant-time technique makes the application responsible for security enforcement, which is only suitable in special circumstances, such as cryptographical software.

Noise injection (Section 3.3.2) corrupts the timing measured by attackers with noise, weakening the timing channel. However, achieving a complete closure on timing channels requires injecting a significant amount of anti-corrected "noise" [Cock et al., 2014], which can dramatically penalise system performance.

Completely virtualising all time sources (Section 3.3.3) is not suitable for timing channel mitigation on cloud platforms, as denying any access on real time clock is not feasible to cloud tenants [Garfinkel et al., 2007]. Hence, the cloud system would require resource partitioning for defending against timing channels.

Time partitioning (Section 3.3.4) mitigates timing channels based on a time-multiplexing usage on shared hardware resources. This technique resets related microarchitectural state before resuming the next running security domain, hence eliminating contention on corresponding hardware components. However, reseting microarchitectural state cannot mitigate cross-core timing channels due to the concurrent access on shared hardware components, such as the LLC.

Hardware partitioning (Section 3.3.5) distributes hardware components to security domains, eliminating timing channels replying on both consecutive and concurrent execution sequence. Applying the technique may require hardware assistance, depending on the feature of corresponding hardware components. For example, cache colouring (Section 2.4) is suitable for physically-indexed caches, but cannot be applied to virtually-indexed caches as the layout of virtual address space is outside the OS's control. For mitigating attacks on virtually-indexed caches (e.g., L1 caches or TLBs), the system has to either flush those caches or partition them with specific hardware support [Wang and Lee, 2007].

3.4 Multikernel

The multikernel [Baumann et al., 2009] system model consists of multiple kernel images running on a hardware platform without sharing a common knowledge base, such as kernel data structures, scheduling decisions, or number of available threads in the system. Each kernel image manages its own memory partition, and executes on separate cores. As a result, software modules that belong to different kernel images communicate through network messages. The multikernel model improves the system scalability on many-core platforms by abstracting any communication between cores as network traffic.

The idea of running multiple kernel images has been also applied on many-core systems. Corey [Boyd-Wickizer et al., 2008] allows applications to control the sharing of kernel data structures, enhancing the scalability of the system. Furthermore, Helios [Nightingale et al., 2009] simplifies the task of deploying applications on heterogenous platforms with satellite kernels, a kernel design that exports a uniform set of abstractions for providing OS services.

Barrelfish/DS [Zellweger et al., 2014] supports hot-plugging kernel images by providing kernel image saving and restoring services. Barrelfish/DS benefits the system performance on energy efficiency as well as supporting heterogeneous many-core systems.

3.5 Exokernel

Traditional monolithic kernels, such as UNIX, offer kernel services through fixed implementation, which cannot satisfy applications with different needs. To address that problem, the exokernel OS [Engler et al., 1995] offers low-level interfaces that are designed only to expose hardware resources. The design goal of the exokernel OS is to provide freedom to implement customised library OSes at user-level. As a result, the exokernel is responsible for resource protection but not management.

Engler et al. [1995] implemented a prototype of the exokernel, called Aegis, together with its library OS, ExOS. In their design, system services that are traditionally implemented in kernel are now moved to application-level libraries, including services used for virtual memory management or interprocess communication. Engler et al. demonstrated that these system primitives offered by the library OS are 5–40 times faster than their counterparts in Ultrix which is a mature monolithic UNIX kernel.

Later, Kaashoek et al. [1997] demonstrated that the exokernel architecture can improve end-to-end application performance compared to 4.4 BSD UNIX systems. Their exokernel, Xok, allows applications to take advantage of decentralised resource management, and achieved a high throughput on file systems and a HyperText Transfer Protocol (HTTP) server.

4 Threat Scenarios and the Targeted System Solution

The goal of our work is to provide time protection through kernel mechanisms that are suitable for preventing microarchitecture timing channels. A *security domain* contains a number of software components and processes, represented as a single unit in a system's security policy. For example, a trusted, secure hardware video compositor [Data61, 2017a] can support multiple security domains, each of which contains software components designed for processing video from an isolated network. Another example is a cloud computing platform in which each mutually distrusting VM is regarded as a security domain. Additionally, a language runtime environment, such as a web browser used for executing JavaScript, can be regarded as a security domain together with scripts that it executes at runtime. A system can choose to run multiple instances of the web browser in different domains, if its security policy requires isolating JavaScript executed in each browser.

The system only enforces *time protection* between domains. Within a domain, there is no restriction imposed by the security policy, granting flexibility on the internal structure of the domain. Therefore, the system can arrange processes that work together to deliver a system service in the same security domain, avoiding on internal context switches overhead resulting from time protection.

Our solution would be suitable for preventing microarchitectural timing channels on both single- and multi-core systems. We first define threat scenarios, based on our study of known microarchitectural timing channels (Section 4.1). Then, we analyse the hardware requirements for implementing mitigations on commodity hardware platforms (Section 4.2). Lastly, we define the *targeted system solution*—providing time protection as an OS service (Section 4.3).

4.1 Threat Scenarios

As noted in Section 3.1, our work does not cover the following types of timing channel attack:

- timing channels between hardware threads (Section 3.1.1),

- timing channels due to self-contention (Section 3.1.2),
- covert timing channels on bandwidth-limited interconnects (Section 3.1.3), or
- timing channels on DRAM (Section 3.1.4).

We require that the system either disables any hyperthreading (e.g., SMT) features, or the OS allocates all hardware threads of a core to the same security domain (Section 3.1.1). Sharing a core through hyperthreading not only prevents spatial partitioning, but also is vulnerable to timing channel attacks [Acıiçmez and Seifert, 2007; Percival, 2005; Yarom et al., 2016]. Because a core has much state that cannot be spatially partitioned, the system cannot prevent timing channels between hardware threads while allowing concurrent access. Hence, preventing these channels requires disabling hyperthreading or co-scheduling domains on hardware threads. These restrictions do not make the system setting unrealistic. Disabling the hardware threading feature is also aligned with common practice on public clouds [Marshall et al., 2010] and the OpenBSD [OpenBSD, 2018].

As our work regards time protection as a mandatory service by the OS, we assume that timing channels due to self-contention are mitigated by applying specific constant-time techniques (Section 3.1.2). However, our approach can prevent timing channels based on self-contention if the system configures a deterministic domain switching latency, a mechanism provided by time protection (Section 6.4.6).

We assume that the system either executes on one core or co-schedules a single domain across all cores [Ousterhout, 1982] while processing sensitive information. As stated in Section 3.1.3, covert channels on bandwidth limited hardware cannot be prevented on contemporary hardware, while side channels are infeasible. We therefore have to content ourselves with preventing side-channel attacks cross-core.

Moreover, we assume that attacks on the DRAM, such as the Rowhammer attack (Section 3.1.4), are mitigated by corresponding hardware or software designs. Lastly, the system mechanisms introduced by this work can also mitigate DoS attacks on related hardware components, because temporal isolation has much stronger system requirements compared to merely providing QoS (Section 3.1.5).

Based on existing microarchitectural attacks (Section 3.2), we define our targeted attacks as those that exploit capacity-limited resources. These attacks have been demonstrated on cache and cache-like components that contain microarchitectural states generated by recently executed programs (Section 3.2.4). Within a core, there are attacks (Section 3.2.2) on core-private data and instruction caches (e.g., L1 caches), caches used by branch prediction (i.e., the BTB and BHB), and caches used for virtual-to-physical address translations (i.e., the TLB). Between cores, there are attacks (Section 3.2.3) on the core-shared cache (i.e., the LLC).

We summarise threat scenarios in Figure 4.1, selected from the opposite ends of the system model spectrum.

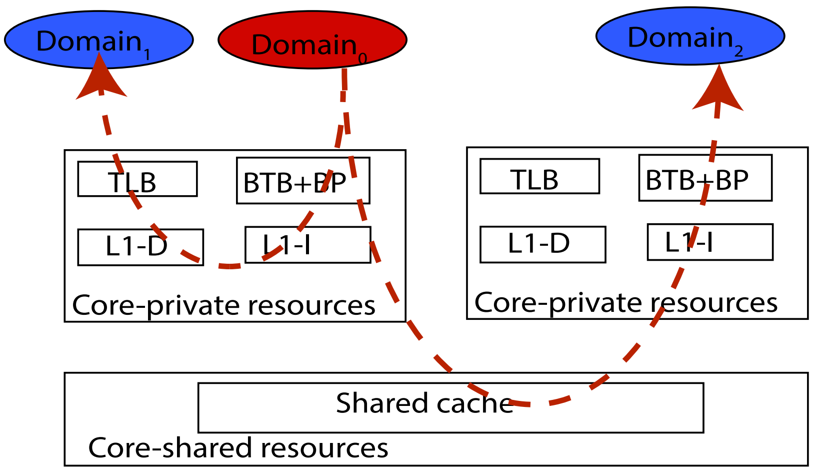


Figure 4.1: Threat scenarios: The arrow from Domain$_0$ to Domain$_1$ represents the confinement scenario of information leakage through intra-core covert channels, while the arrow from Domain$_0$ to Domain$_2$ indicates the cloud scenario of a cross-core side channel through a shared cache.

4.1.1 Confinement

In the confinement scenario, an untrusted program, such as an unverified library, third-party application, or web browser plugin, attempts to leak sensitive data which it holds. One classic example of such a scenario includes a military-grade cross-domain device that processes information at different classification levels [Denning, 1976]. An underlying assumption in this scenario is that the untrusted program is potentially malicious. As a result the OS must prevent information leakage through all possible channels, including any microarchitectural covert timing channels.

The threat is that Domain$_0$ in Figure 4.1, which holds the secret, seeks to leak that secret to Domain$_1$, which is not entitled to the secret, by using *intra-core covert timing channels*, hence violating the system security policy.

4.1.2 Cloud

In the cloud scenario, a cloud provider hosts mutually distrusted guest OSes, i.e., VMs, that run concurrently on a processor. Because the guest systems can communicate with the external world, the covert channel is not within the scope of the cloud scenario. Rather, we focus on preventing side channels where a malicious VM (Domain$_2$ in Figure 4.1) is stealing secrets owned by a victim VM (Domain$_0$), such as *cross-core side-channel attacks* through the LLC demonstrated by recent work [İnci et al., 2016; Irazoqui et al., 2015a, 2016; Liu et al., 2015].

As stated above, we assume that guest VMs cannot share a core concurrently through hyperthreading, due to the high level of resource sharing among hardware threads. Hence the system either disables hyperthreading completely or assigns all hyperthreads on a core to one guest VM. Nevertheless, we allow guest VMs time-multiplexing a core.

Cloud platforms are very performance sensitive, as the success of the business model of cloud is highly dependent on resource utilisation. Therefore, an ideal system solution must not introduce any significant performance degradation, nor prohibit efficient resource sharing among guest VMs.

4.2 Hardware Requirements

As we discussed in Section 3.3, mitigating timing channels requires preventing interference caused by competition on microarchitectural resources, which can be achieved by preventing any observable resource contention. A system can eliminate any resource contention either by temporal partitioning through resetting the microarchitectural state (Section 3.3.4) or by spatial partitioning of the hardware resources (Section 3.3.5).

Another possible solution is preventing any observation of timing channels by removing all possible time sources, for example the virtual time approach (Section 3.3.3). However, virtualising all kinds of system time is not only expensive [Li et al., 2013], but also has difficulties supporting cloud tenants [Garfinkel et al., 2007] and applications that require real time [Aviram et al., 2010b].

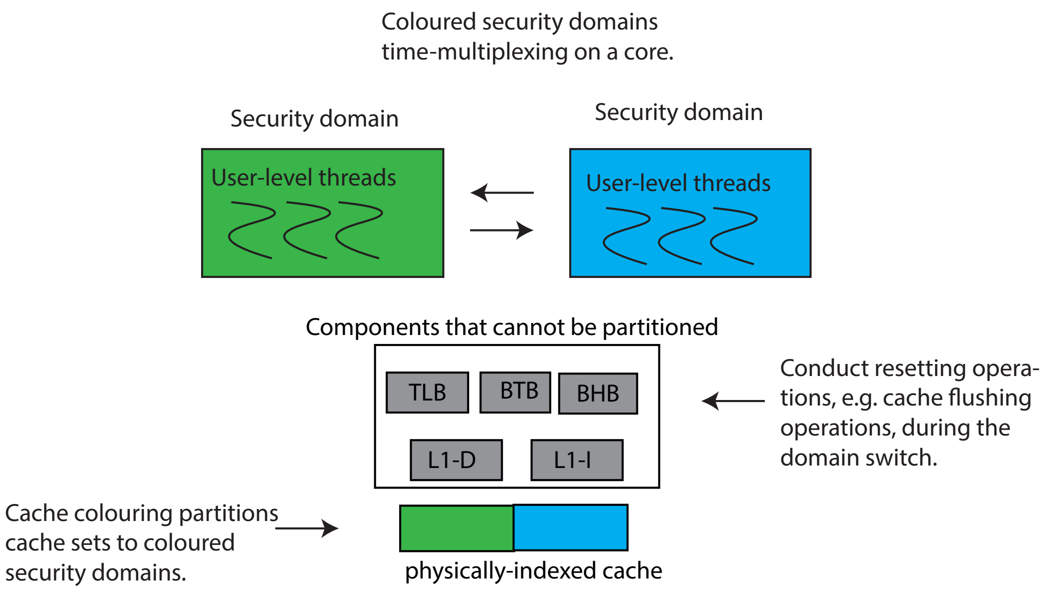

Coloured security domains
time-multiplexing on a core.
Security domain
Security domain
User-level threads
User-level threads
Components that cannot be partitioned
TLB
BTB
BHB
L1-D
L1-I
Conduct resetting operations, e.g. cache flushing operations, during the domain switch.
Cache colouring partitions cache sets to coloured security domains.
physically-indexed cache

Figure 4.2: An overview of mitigating covert timing channels between two security domains executing on a core.

4.2.1 Resetting on-core state

The only generally available spatial partitioning mechanism on caches is cache colouring (Section 2.4), which requires the OS to have control of address allocation. The OS cannot partition on-core resources that are indexed by virtual addresses, including L1 caches, the TLB, the BTB, and the BHB, as virtual addresses are under application control (Section 2.3.1). L1 caches are typically only have a single colour as they are typically very small. Hence L1 caches could not be coloured even if they are physically indexed. As a result, resetting stateful on-core resources is the only solution if resources cannot be easily partitioned without new hardware support. While such support, such as cache line locking, has been demonstrated [Wang and Lee, 2007], the technology is not available on commodity processors. Also, reducing the available L1 cache size can have a high performance impact.

The hardware must provide flushing operations for resetting those on-core resources that cannot be partitioned (Section 3.3.4), to allow the OS to reset state when switching between security domains, as shown in Figure 4.2. Theoretically, resetting on-core state is affordable, as those caches are relatively small, presenting a lower direct cost and indirect cost (Section 3.3.4).

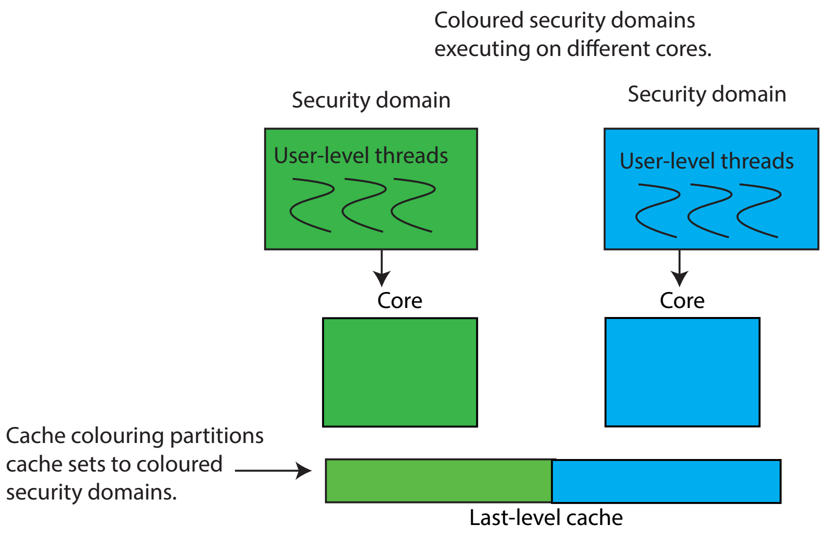

Coloured security domains executing on different cores.
Security domain
Security domain
User-level threads
User-level threads
Core
Core
Cache colouring partitions cache sets to coloured security domains.
Last-level cache

Figure 4.3: An overview of mitigating timing side channels between two security domains executing on different cores.

4.2.2 Partitioning

Partitioning is feasible where the OS has complete control over how shared infrastructure is distributed to security domains. The mechanism is effective at mitigating timing channels that require either time multiplexing (Section 3.2.3.1) or concurrent access (Section 3.2.3.2) to shared resources, such as the LLC.

Some hardware manufacturers provide assistance for locking a limited amount of secure data in cache ways (Section 3.3.5), which can effectively mitigate cache-based side channels on the protected data section. However, the efficient use of those secure cache ways requires the user-level application to identify secret data. We argue that time protection should be a black-box mechanism enforced by the OS, rather than a service relying on cooperation of applications. Hence, locking secure data in cache ways does not suit our targeted solution.

Partitioning physically-indexed caches using cache colouring [Kessler and Hill, 1992; Liedtke et al., 1997] (Section 2.4) can eliminate the PRIME+PROBE attack on single-core [Irazoqui et al., 2015a], as well as on multicore processors [İnci et al., 2016; Liu et al., 2015]. The cache colouring technique requires only that the OS controls the memory allocation of the system, assigning coloured memory frames to security domains.

For the example Intel processor given in Figure 2.2, cache colouring can partition both core-private L2 cache and core-shared LLC, as they are physically-indexed caches.

As demonstrated in Figure 2.7, cache colouring is a suitable technique for mitigating cache-based timing channel through partitioning physically-indexed caches. The system can construct security domains with frames of disjoint colours. As a result, each security domain can only occupy its partition on physically-indexed cache sets, as shown in Figure 4.2. Those coloured domains cannot create covert channels on partitioned caches by time-multiplexing on a core.

For resource sharing that is truly concurrent (e.g., accessing package-shared LLC), spatial partitioning is the only option, as resetting system state between time slices cannot mitigate channels based on concurrent accesses (Section 3.3.6). Figure 4.3 demonstrates a mitigation scenario for cross-core side channels on the package-shared LLC. In this case, security domains are executing on different cores and are built with coloured memory frames. Thus, both cores and the LLC are partitioned. As a result, the system is susceptible neither to the intra-core timing channel attacks of Section 4.1.1, nor the cross-core side channel threat of Section 4.1.2.

To summarise, our solution requires resetting on-core microarchitectural states with cache-flushing operations, mitigating intra-core timing channels on those components that cannot be partitioned (Section 3.3.4), and partitioning cache sets with the cache colouring technique (Section 2.4), mitigating timing channels on physically-indexed caches, including the package-shared LLC.

4.3 The System Solution – Providing Time Protection in the OS

We aim to achieve the goal of preventing intra-core covert channels (Section 4.1.1), as well as cross-core side channels (Section 4.1.2). We require that mitigation must not introduce significant system overhead. Additionally, the system needs to remain practical and capable of providing the service that applications expect of an OS. Our proposed solution can address many other system cases, if we can address the opposite ends of the system model spectrum demonstrated in Figure 4.1.

We argue that security enforcement is a core duty of the OS, as user-level applications cannot be trusted and the hardware cannot distinguish between a context switch (address-space enforcement) and a security domain switch (security enforcement). Hence, the most privileged system software, the OS, must provide time protection between security domains, as a black-box form of timing isolation, similarly to memory separation enforcement. Note that we use the term “OS” in a generic sense, referring to the most privileged software level that is responsible for security enforcement; it can represent a hypervisor, a microkernel, or a monolithic kernel such as Linux or Windows. Our targeted system offers time protection as a mandatory enforcement feature of the OS, a service provided by the kernel.

We propose time protection to target the threats shown in (Figure 4.1).

Definition: Time protection
A collection of OS mechanisms which jointly prevent interference between security domains that would make execution speed in one domain dependent on the activities of another.

Our ultimate goal is to obtain temporal isolation guarantees comparable to the spatial isolation proofs of the seL4 microkernel, but currently we focus on a system design that is suitable for any verifiable microkernel with minimal, general purpose, and policy-free mechanisms. One distinct advantage of policy-free mechanisms is that they can be used to prevent other timing channels even though these channels are not contention based, such as channels that exploit a non-constant time implementation (i.e., the execution time of an implementation is secret dependent) as introduced in Section 3.1.2.

Security domain abstraction To achieve our design goal, we require the kernel to provide an abstraction for security domains. In other words, the kernel must be able to build a connection between security domains and microarchitectural states used by them while executing, conducting any operations during domain switches necessary to mitigate related timing channels. Because seL4 abstracts system resources as kernel objects, our targeted system solution should be able to provide the same level of abstraction for representing security domains. In other words, the kernel regards domains as kernel objects–the unit for enforcing time protection schemes (e.g., conducting cache flushing operations during a domain switch (Figure 4.2).

Security domain creation and deletion The kernel should provide mechanisms for creating and destroying security domains. Importantly, a security domain can have more than one user-level thread executing inside their own address spaces (regarded as a user-level application in seL4 Section 2.6). Therefore, the kernel has to provide a method for consolidating threads into security domains. In a multi-level secure (MLS) system [DoD], a security domain represents a security classification level, each with a different security clearance. The time protection introduced by our work prevents domains with a different security clearance from obtaining unauthorised information through intra-core covert timing channels (Section 4.1.1). Once the security domain is deleted, the kernel has to offer a mechanism to release any memory used by that domain.

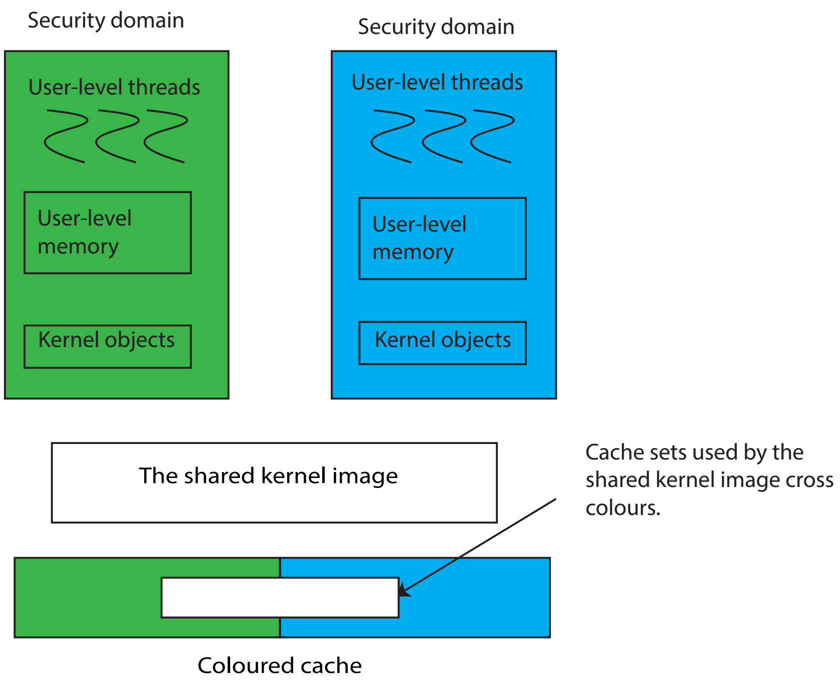


Figure 4.4: Coloured security domains share an un-coloured kernel image.

Deploying the cache colouring technique To partition shared physically-indexed caches (Section 2.3.1), the existing kernel mechanisms are sufficient for colouring user memory and kernel objects (the kernel metadata), as explained in Section 2.6.3. However, all user-level threads are still supported by a single kernel image that is created at boot (Figure 4.4). In other words, the kernel image is not coloured and is shared by all the domains, which can potentially cause a covert timing channel. For example, one coloured domain can probe cache sets used by the other domain by invoking kernel services (e.g., system calls), similarly to shared library code (Section 3.2.3). As cache sets used by the kernel image cross two partitions, the signal sent by the previously executed domain can be learnt by the other domain though a PRIME+PROBE attack (Section 3.2.1). To address that, the kernel has to provide mechanisms to both partition cache sets used by the kernel image (i.e., colour the kernel image) and make the execution time of the kernel deterministic enough to prohibit any timing channels.

Identifying domain switches While time-multiplexing a core, the kernel must be able to identify domain switches and conduct cache flushing operations (Figure 4.3) for mitigating intra-core covert timing channels. Domain switching is different than context switching because a domain can have multiple address spaces, and time protection is only necessary for a domain switch to prevent any information leakage between domains. It is important to identify the difference, and only conduct cache flushing operations during a domain switch to preserve system performance. Also, the domain switching latency must be irrelevant to activities generated by the previous running domain.

5 | Combating Microarchitectural Timing Channels by Resetting Hardware States

This chapter is the subject of the following paper of which I was the primary author: Qian Ge, Yuval Yarom, and Gernot Heiser. No security without time protection: We need a new hardware-software contract. In *Asia-Pacific Workshop on Systems (APSys)*, Korea, August 2018b. ACM SIGOPS. The analysis and experimentation of all published timing channel attacks and countermeasures was performed primarily by myself, with assistance from Yuval Yarom, under the supervision of Gernot Heiser.

The channel matrix measurement shown in this chapter were primarily done by myself. The original implementation of the tool chain is due to David Cock, as published in Cock et al. [2014]. I maintained the tool chain after David Cock finished his PhD in August 2014. We use this tool chain to analyse the correlation between resource contention created by the sender and timing measured by the receiver.

The tool chain for quantifying the information leakage of studied channels was primarily contributed by Tom Chothia, as published in Chothia et al. [2013]. The merging of the above mentioned two tool chains was performed by myself. We use this tool chain for calculating the mutual information (MI), as a measure of the size of the channel.

In this chapter, we present our investigation on the effectiveness of mainstream hardware for mitigating intra-core timing channels though temporal partitioning. We conduct our study on four popular processors and discover that modern hardware does not provide sufficient mechanisms to allow the OS to enforce full temporal isolation. To address that, we propose a new software-hardware contract, an augmented ISA, for providing mechanisms for supporting time protection.

5.1 Mitigating Intra-core Timing Channels with Resetting Operations

As discussed in Section 4.2.1, resetting the on-core state is the only method to mitigate timing channels created on microarchitectural components that cannot be spatially partitioned, except from virtualising all available time sources which has distinct drawbacks (Section 3.3.6). Therefore, the effectiveness of hardware resetting operations on mitigating those channels becomes important.

Hardware resetting operations, such as cache flushing operations, are provided by hardware manufacturers as part of the ISA, the hardware-software contract. Similar to the rest of the ISA, resetting operations are designed to enforce functional correctness rather than temporal partitioning.

Previous work demonstrated the use of cache flushing operations to mitigate cache-based side channels [Godfrey and Zulkernine, 2013; Zhang and Reiter, 2013]. However, there is no existing study on mitigating timing channels on other core-private resources, including TLB, BHB, nor BTB channels. To address this shortcoming, we conduct a systematic study on examining the effectiveness of hardware resetting operations on mitigating intra-core timing channels. In particular, we select four popular processors, two x86 processors (Haswell and Skylake) and two Arm processors (Arm Cortex-A9 and Arm Cortex-A53), to conduct a study on the effectiveness of manufacturer provided resetting operations on the mitigation of timing channels based on on-core cache and cache-like components.

5.2 Threat Model and Scope

This chapter focuses on a specific threat scenario, which is depicted in Figure 5.1: one security domain (high) leaks information to the other security domain (low) through intra-core covert timing channels. More specifically, the channels are based on resource contentions on components that cannot be partitioned without specific hardware support [Wang and Lee, 2007]. Based on our studies on known attacks (Section 3.2.4), these covert channels are based on core-private cache and cache-like components, including the L1-D cache, the L1-I cache, the TLB, the BTB, and the BHB.

We assume that an adversary tries to steal sensitive data, such as classified information owned by the security domain with high security clearance, through the above listed covert timing channels. The adversary manages to inject a Trojan program into a restricted security domain, possibly by compromising some system services first. Due to security policies enforced by the OS, the Trojan cannot directly leak any secret to the adversary from the confined secure environment, even though it has access to sensitive data.

To acquire the secret, the adversary controls a spy program that shares the same CPU core as the Trojan. The spy executes outside the restricted security domain. As they are

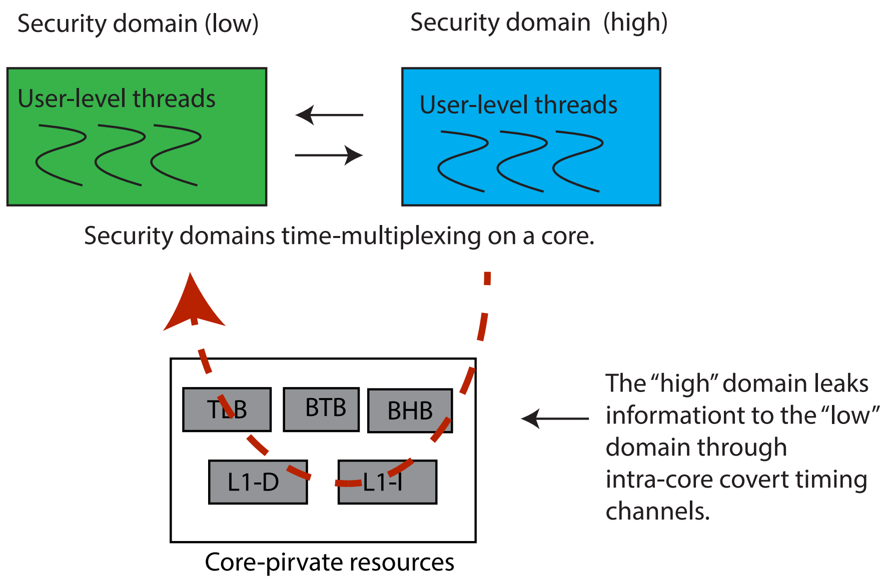

Security domain (low)
Security domain (high)
User-level threads
User-level threads
Security domains time-multiplexing on a core.
TLB
BTB
BHB
L1-D
L1-I
Core-pirvate resources
The "high" domain leaks informationt to the "low" domain through intra-core covert timing channels.

Figure 5.1: Covert timing channels on core-private cache and cache-like components.

executing in a security domain with low clearance, the spy program has more freedom to communicate with the external adversary. The adversary's aim is to exploit a covert timing channel, allowing the Trojan to send sensitive data to the spy through resource contention on the shared CPU core. Eventually, the adversary receives the sensitive data from the spy.

In this chapter, we investigate the degree to which the system can prevent the adversary from exploiting such covert channels by resetting hardware states during a domain switch.

5.3 Methodology

We investigate the hardware flush operations provided by manufacturers for their ability to contribute to closing intra-core timing channels. We perform this investigation by implementing all timing channels on core-private resources listed in Section 5.2 (Section 5.3.1), identify available hardware operations for closing those channels (Section 5.3.2), and analyse the channel leakage (Section 5.3.3) with and without mitigation techniques.

5.3.1 Implementing Intra-core Timing Channels

Similarly to previous work done by Cock et al. [2014], we consider a timing channel to have a set of *inputs* (I) that is selected by a sender (e.g., Trojan) and a set of *outputs* (O) that is observed by a receiver (e.g., spy). The specific input and output set depend on the channel and attack model.

We implement all the channels using the PRIME+PROBE technique on both x86 and Arm platforms, without any assumption about the virtual-address-space layouts, except the BHB channel which will be explained in Section 5.3.1.5. In other words, the Trojan and spy can allocate the probing buffer from anywhere in the virtual-address space for building the PRIME+PROBE attack. The main purposes of our implementations are to demonstrate the existence of a channel and to investigate the efficiency of mitigation techniques. We therefore do not design sophisticated encoding schemes to increase the channel throughput.

5.3.1.1 L1-D cache

To build the L1-D cache channel, we use the PRIME+PROBE attack from the Mastik toolkit [Yarom, 2016], which is a similar attack to that demonstrated by previous work [Osvik et al., 2006; Percival, 2005].

Figure 5.2 demonstrates the attack. Both Trojan and spy have a buffer that has the same size as the L1-D cache. By covering a contiguous range in virtual-address space, the buffer can cover all sets in the virtually-indexed L1-D cache (Section 2.3.1). Assuming each cache line has l bytes, the size of the buffer would be $l \times s \times w$ bytes, for covering a L1-D cache that has s sets and w ways. In that L1-D cache, each cache set contains $l \times s$ bytes.

To send a symbol S, the Trojan fills all ways on cache sets $0, 1, \ldots, s$ by reading lines $0, 1, \ldots, S-1$, $s, s+1, \ldots, s+S-1$, $\ldots$, $s \times (w-1) \ldots, s \times (w-1) + S - 1$, the first S number of cache lines in every cache ways. For example, to send a symbol 3, the Trojan reads the first 3 cache lines in every s cache lines, repeated w times in order to fill all cache ways.

To receive the symbol, the spy first primes all the cache sets by reading the entire buffer, then waits to be preempted by the system tick, and finally measures the time it takes to revisit its previously loaded cache lines, using the probing time as the output symbol.

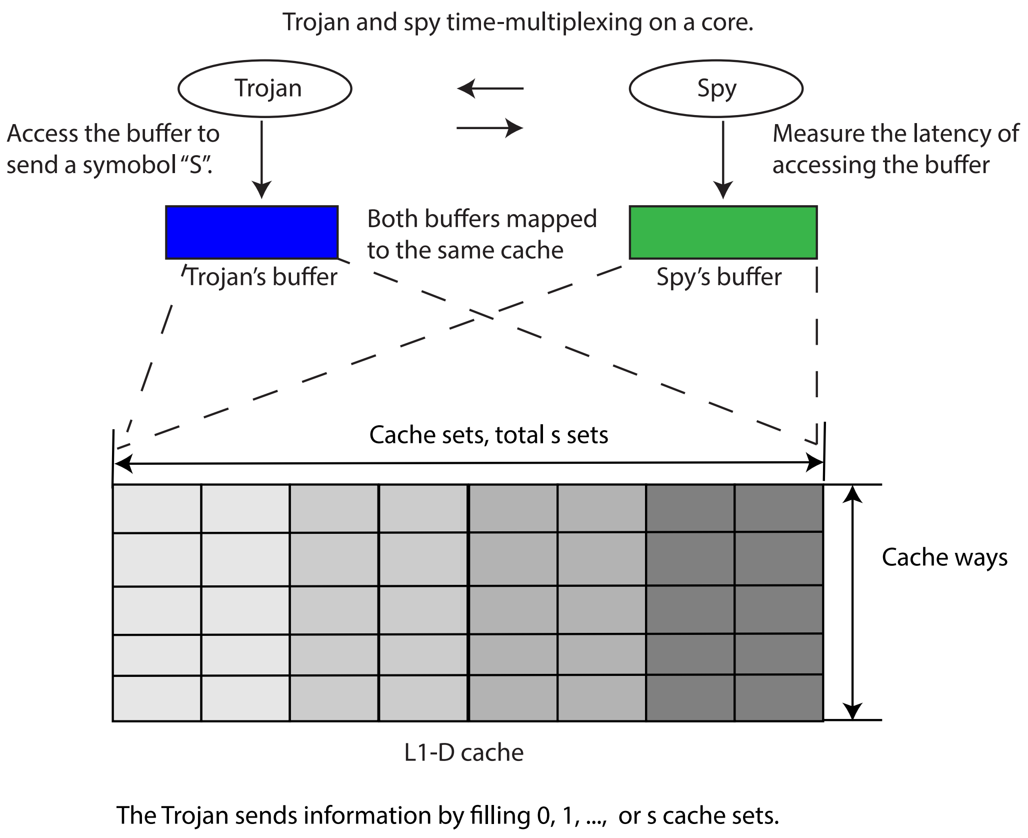
5.3.1.2 L1-I cache

The L1-I cache channel is identical to the L1-D cache channel, except that the PRIME+PROBE fills cache sets by executing code (jump instructions) rather than by reading data. Along with the L1-D cache channel, the Mastik toolkit [Yarom, 2016] contains the prototype of the L1-I cache channel, which is inspired by previous work [Acıiçmez, 2007].

The implementation probes the L1-I cache sets with a series of jumps, as shown in Figure 5.3. For probing the L1-I cache which has w ways, the first $w-1$ cache lines contain a jump instruction, to the next cache line in the same set. To exit, the last cache line in a set contains a return instruction. Hence, probing a cache set becomes calling the first cache line in a set, which returns at the last line in that set.

5.3.1.3 TLB

The TLB works as a virtually-indexed cache (Section 2.3.1), which contains most recently resolved virtual-to-physical mappings. Unlike previous work [Gras et al., 2018; Hund et al., 2013], we do not tailor our attack to target any individual associative TLB sets, but focus on the contentions created by simply probing TLB entries.



Trojan and spy time-multiplexing on a core.
Trojan
Spy
Access the buffer to send a symobol "S".
Measure the latency of accessing the buffer
Both buffers mapped to the same cache
Trojan's buffer
Spy's buffer
Cache sets, total s sets
Cache ways
L1-D cache
The Trojan sends information by filling 0, 1, ..., or s cache sets.

Figure 5.2: The covert timing channel on the L1-D cache.

To conduct the PRIME+PROBE, the Trojan and spy probe TLB entries by reading a single integer from a number of consecutive pages (Section 2.2). After the read, TLB entries are filled by the virtual-to-physical mappings triggered either by the Trojan or spy. For an n-entry TLB, the Trojan sends a symbol S by probing on $0, 1, \ldots, n$ entries. To receive a symbol, the spy measures the cost to access half of the TLB entries, to avoid any self-contention while executing the probe.

5.3.1.4 BTB

The BTB is a virtually-indexed cache, storing the destination of the recently executed branch (i.e., jump) instructions. To build the BTB channel, we chain jump instructions into a probing buffer. Here we focus on demonstrating a timing channel by contending on a set of BTB entries, rather than contending on few specific BTB entries as done by previous work [Evtyushkin et al., 2016a].

All jump instructions are cache-line aligned. The Trojan executes s jumps to send an input symbol S; the spy measures the cost of executing the entire buffer. On Arm platforms, the size of the buffer equals the known size of the BTB. On x86 platforms, however, the manufacturer does not publish the details of the BTB. We configure the Trojan to probe from 3,584 to 3,713 `jmp` instructions and the spy to probe on 4,096 `jmp` instructions, based

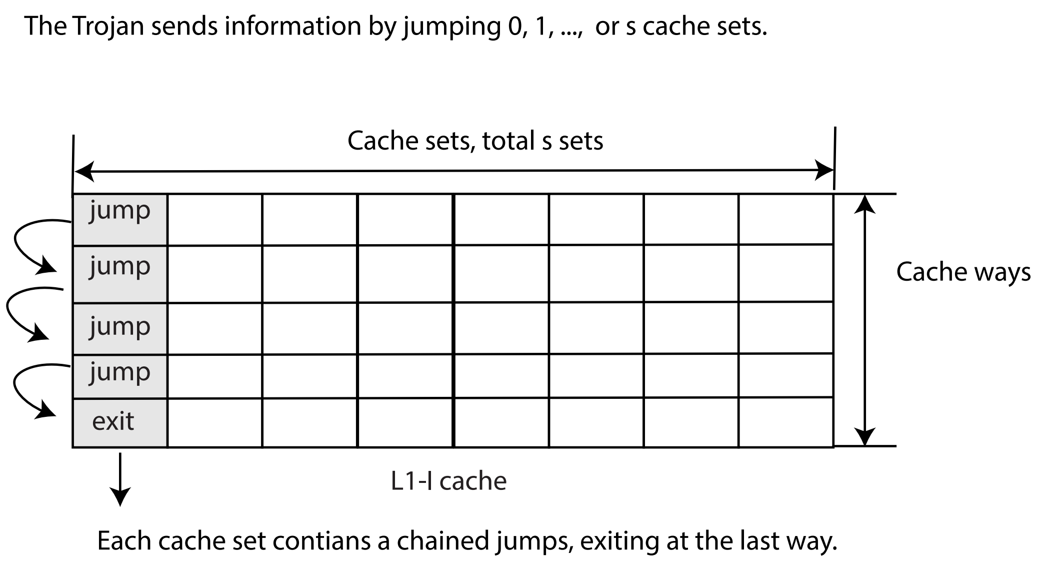

The Trojan sends information by jumping 0, 1, ..., or s cache sets.
Cache sets, total s sets
jump
jump
jump
jump
exit
Cache ways
L1-I cache
Each cache set contians a chained jumps, exiting at the last way.

Figure 5.3: The covert timing channel on the L1-I cache.

on previous work on reverse engineering the BTB structure [Godbolt, 2016; Milenkovic et al., 2004].

5.3.1.5 BHB

We build the BHB channel with inspiration from the residual state-based covert channel [Evtyushkin et al., 2016b]. The BHB is a virtually-indexed cache that contains outcomes (i.e., taken or not taken) of the most recently executed conditional branch instructions. To predict a coming conditional branch instruction, the speculative execution engine (Section 2.3.3) consults the past history stored in the BHB entry. Because the BHB only uses the virtual address for indexing, two conditional branch instructions that are located at the same virtual address but are from different virtual-address spaces can contend on a single BHB entry, influencing the predicted outcome of each other.

Unlike the above mentioned channels, the BHB channel only contends on one BHB entry rather than on a number of entries or sets: the Trojan sends a single-bit symbol, as "0" or "1", on each system tick. The Trojan and spy use the same code for sending and receiving. Here we present the implementation for the x86 architecture, shown in Listing 5.1, together with a detailed explanation. We present a more detailed explanation of the BHB channel than other channels, because the probing technique used here is distinctly different from the classical PRIME+PROBE techniques used in other channels (Section 3.2.1). The code is located at the same virtual address for both Trojan and spy. Therefore, both Trojan and spy contend on the same BHB entry by executing a conditional branch instruction, `jz` (line 13). In other words, the history of the Trojan can influence the predicted outcome of the same branch instruction while the spy executes.

The code starts with a sequence of conditional jump instructions that are always taken (line 8) to prime the branch history to a known state. Then, the attack (lines 10–17) measures the latency of the branch instruction (Line 13) that conditionally skips 256 nop instructions (line 14): the code takes the branch if the least significant bit on register `%edi` is 0. The output of this channel is the measured latency, which is contained in register `%eax`.

```
1   #define X_4(a) a; a; a; a
2   #define X_16(a) X_4(X_4(a))
3   #define X_256(a) X_16(X_16(a))
4
5   #define JMP   jnc 1f; .align 16; 1:
6
7   xorl %eax, %eax
8   X_256(JMP)
9
10  rdtscp
11  movl %eax, %esi
12  and $1, %edi
13  jz  2f
14  X_256(nop)
15  2:
16  rdtscp
17  subl %esi, %eax
```

Listing 5.1: BHB channel implementation on x86.

The conditional branch (line 13) is more expensive if the CPU mispredicts the outcome, aborting the speculatively executed computation (Section 2.3.3). To use branch prediction as a timing channel, the Trojan repeatedly calls the code with input “0” or “1” , which primes the history of the branch as taken or not taken. The spy always calls the code with input “0” (branch taken), sensing the influence of the branch history trained by the previously executed Trojan. A longer latency represents that the prediction conflicts with the actual outcome (i.e., Trojan inputs “1”), whereas a shorter latency represents that the prediction is correct, (i.e., Trojan inputs “0”).

5.3.1.6 Summary

The above listed channels create cache-like intra-core microarchitectural components, including the L1-D cache, L1-I cache, TLB, BHB, and BTB. These components work closely together, efficiently serving execution engines in the CPU. Because of interaction between those components, our evaluations cannot treat them separately. For instance, the attack that probes the L1-I cache (Section 5.3.1.2) using chained jump instructions also probes BTB entries (Section 5.3.1.4), thus an apparent L1-I channel may in reality be the result of a BTB contention.

Name		**Haswell**	**Skylake**	**Sabre**	**Hikey**
Architecture		**x86**	**x86**	**Arm v7**	**Arm v8**
Microarchitecture		**Haswell**	**Skylake**	**Cortex-A9**	**Cortex-A53**
Manufacturer		Intel	Intel	Freescale	HiSilicon
Processor		i7-4770	i7-6700	i.MX6	Kirin 620
Clock rate (GHz)		3.4	3.4	0.8	1.2
Year		2013	2015	2011	2015
Address size (bit)		64	64	32	32
Execution order		OoO	OoO	OoO	InO
Time slice (ms)		1	1	1	1
L1-D	size (KiB)	32	32	32	32
	line (B)	64	64	32	64
	sets×assoc.	64×8	64×8	256×4	128×4
L1-I	size (KiB)	32	32	32	32
	line (B)	64	64	32	64
	sets×assoc.	64×8	64×8	256×4	256×2
L2	size (KiB)	256	256	1024	512
	line (B)	64	64	32	64
	sets×assoc.	512×8	512×8	2048×16	512×16
L3	size (MiB)	8	8	N×A	N×A
	line (B)	64	64	N×A	N×A
	sets×assoc.	8192×16	8192×16	N×A	N×A
BTB	size	?	?	512	256
	sets×assoc.	?	?	256×2	?
Instruction-TLB	size	64	128	32	10
	sets×assoc.	8×8	16×8	32×1	10×1
Data-TLB	size	64	64	32	10
	sets×assoc.	16×4	16×4	32×1	10×1
L2-TLB	size	1024	1536	128	512
	sets×assoc.	128×8	128×12	64×2	128×4

Table 5.1: Experimental hardware, covering 2 generations of microarchitectures across x86 and Arm. Instruction-TLB and Data-TLB represent first-level TLBs, L2-TLB represents the second-level (unified) TLB. "?" indicates unknown values.

5.3.2 Mitigations

As mentioned in Section 5.1, the only mitigation strategy available for studied channels is providing a deterministic execution environment through hardware resetting operations, such as cache-flushing operations, or completely disabling features relying on hardware state, such as the prefetcher (Section 2.3.2). In this section, we list those hardware operations provided by manufacturers on experimental platforms. We select four experimental platforms, across two generations of both x86 and Arm architectures (Table 5.1).

5.3.2.1 Architectural support on x86

We list the architectural support on x86 for mitigating on-core timing channels in Table 5.2. The x86 architecture does not support selectively flushing caches, instead providing a privileged instruction, `wbinvd`, for flushing and invalidating all levels of caches [Intel, e]. After the instruction executes, all cache entries are invalidated, and subsequent accesses for both data and instructions suffer cache misses (Section 2.3.1); the data is instead loaded from the main memory. Flushing all levels of caches is too expensive to be used in practice, especially for the x86 architecture which has three levels of caches. Our experiments use the operation to demonstrate the maximum deterministic execution environment that can be archived by architectural means. When executing in 64-bit mode, we use the `invpcid` to flush and invalidate all TLB entries (both tagged and global entries) and paging structure caches.

The architecture separates instruction and data prefetching, each controlled by a dedicated prefetcher. As described in Section 2.3.2, the prefetcher predicts the instruction or data that will be visited in the near future, based on the most recent history. Thus, the prefetcher contains state that is influenced by the previously executed programs (i.e., the Trojan or spy). Instead of flushing, disabling the data prefetcher is the only option provided by the architecture to avoid any potential timing channel on this unit. The x86 architecture offers disabling the data prefetcher though updating the machine state register (MSR) `0x1A4` [Viswanathan, 2014]. However, there is no way to disable the instruction prefetcher.

Component	**Operation**
All levels of caches	`wbinvd`
TLB	`invpcid`
Data prefetcher	MSR `0x1A4`
Branch prediction	IBC

Table 5.2: Architectural support on x86 for mitigating on-core timing channels.

For mitigating the Spectre attack [Kocher et al., 2019], Intel provided a series of operations through a microcode update, called IBC [Intel, 2018c]. As introduced in Section 3.3.4, the IBC mechanism provides configurations for restricted speculations on indirect branch predictions across privilege modes or hardware threads, or based on previous history. IBC is designed to provide isolation on the branch prediction state, without claiming to be a flushing operation. To understand the effectiveness of the IBC, we apply the microcode patches on all of our x86 testing platforms [Intel, 2018a], and evaluate the timing protection with and without the IBC mechanism.

To summarise, for x86 we flush all caches and TLBs, disable the data prefetcher, and apply the IBC mechanism during a security domain switch. However, there is no operation to either flush or disable the instruction prefetcher.

5.3.2.2 Architectural support on Arm

We list the architectural support on Arm for mitigating on-core timing channels in Table 5.3. The Arm architecture supports selectively flushing caches. We use the `DCCISW` operation to flush and invalidate the L1-D cache, and `ICIALLU` for the L1-I cache. We flush L1 caches by instrumenting these two operations with the level of unification, which is the L2 cache. Similar operations on the L2 cache are platform dependent, as the L2 cache is normally implemented as a core-external cache. We therefore use the following specified operations offered on each platform: the clean and invalidate set operation on the external cache controller for the Sabre (Arm Cortex-A9) platform, and `DCCISW` and `ICIALLU` operations with level of coherency, the level for the main memory, for the Hikey (Arm Cortex-A53) platform.

Component	**Cortex-A9**	**Cortex-A53**
L1-D	`DCCISW`	`DCCISW`
L1-I	`ICIALLU`	`ICIALLU`
L2	clean and invalidate (external L2)	`DCCISW`, `ICIALLU`
TLB	`TLBIALL`	`TLBIALL`
Branch predictor (flush)	`BPIALL`	`BPIALL`
Branch prediction (disable)	`SCTLR`	N/A
Data prefetcher	`ACTLR`	auxiliary control

Table 5.3: Architectural support on Arm for mitigating on-core timing channels.

Both Arm testing platforms provide `TLBIALL` to flush and invalidate all TLB entries, and `BPIALL` for the branch predictor. On the Sabre platform, we disable the branch prediction by clearing the Z-field in the `SCTLR` and also the first-level data prefetcher by setting the `ACTLR` register. On the Hikey platform, we disable the data prefetcher by setting the CPU auxiliary control register.

In summary, on Arm we flush and invalidate all caches, TLBs, and branch prediction entries. On the Sabre platform, we also disable the first-level data prefetcher and the branch predictor for providing the most deterministic execution environment offered by hardware operations. On the Hikey platform, we disable the first-level data prefetcher. However, neither of the two testing platforms support flushing or disabling the instruction prefetcher.

5.3.3 Evaluating Channels

On all timing channels, the Trojan sends a pseudo-random sequence of input symbols, and the spy collects a large number of timing measurements, observing the Trojan's activity.

The channel can also be regarded as a pipe into which the Trojan sends secret values drawn from a set X, and from which the spy receives output from a set Y as measured timing variances. This attacking model can be used for leaking information in the confinement scenario introduced in Section 4.1.1. Using the cache attack as an example, the input is the number of cache sets accessed by the Trojan, and the output is the probing cost measured by the spy on revisiting the previously-loaded cache lines (a cache-sized buffer).

To analyse the timing channel, we show its channel matrix (Section 5.3.3.1) and quantify the channel leakage (Section 5.3.3.2) with a method for verifying zero-leakage channels (Section 5.3.3.3).

5.3.3.1 The channel matrix

For all studied timing channels in this thesis, we use the channel matrix, which represents the conditional probability of an observed output (spy probing time, y axis) given an input (cache sets accessed by the Trojan, x axis). The sample size n means the number of samples collected for each input symbol, and the total sample collected for generating a given matrix is $n \times S$, where S is the number of input symbols.

We present the channel matrix as a heat map, where colours indicate probability as per the scale on the right. In the heat graphs shown in this thesis, a brighter colour represents a higher probability.

If a channel exist, the matrix graph will show a correlation between the input and output, demonstrating that the selected input (number of cache sets accessed by the Trojan) impacts the output (probing cost measured by the spy). In other words, the probability of observing a particular output depends on the inputs, which in the channel matrix appears as a variation of colour along a horizontal line. The matrix graph would show no horizontal variation if no channel exists. We create the channel matrix using the technique developed by Cock et al. [2014].

Figure 5.4 demonstrates the channel matrix for the L1-I cache covert timing channel (Section 5.3.1.2) without any countermeasures on an Arm Cortex-A9 processor. In the graph, the input symbol represents the number of L1-I cache sets that the Trojan uses during each run, and the output symbol represents the cost of probing on all L1-I cache sets measured by the spy. The graph shows that the Trojan's activity (input x) is strongly correlated with the cost of probing measured by the spy (output y): a larger input symbol causes a larger output value, as a consequence of the cache contention created by the Trojan. In other words, the spy is much more likely to observe a higher probing cost when the Trojan has executed on more cache sets in its most recent run. In contrast, the spy cannot infer Trojan's activity while the channel is mitigated by resetting operations provided by hardware manufacturers. As shown in Figure 5.5, the correlation between x and y does not exist in the mitigated channel. We will explain the method used to analyse channel leakage in Section 5.3.3.3.

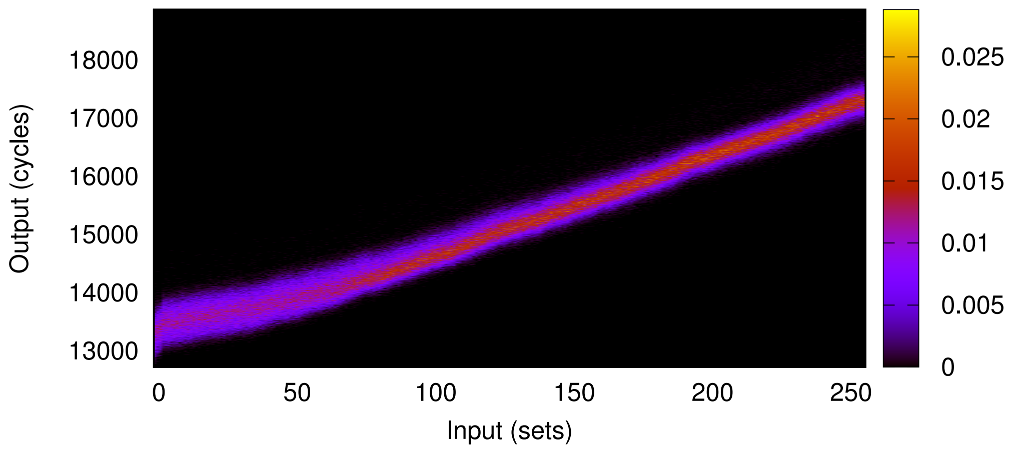


Figure 5.4: Channel matrix for the unmitigated L1-I covert channel on an Arm Cortex-A9. This example uses 3,808 samples for each input symbol and shows a channel capacity of 2500 mb (millibit) per usage, we summarise this as $\mathscr{M} = 2500\,\mathrm{mb}$, $n = 3808$.

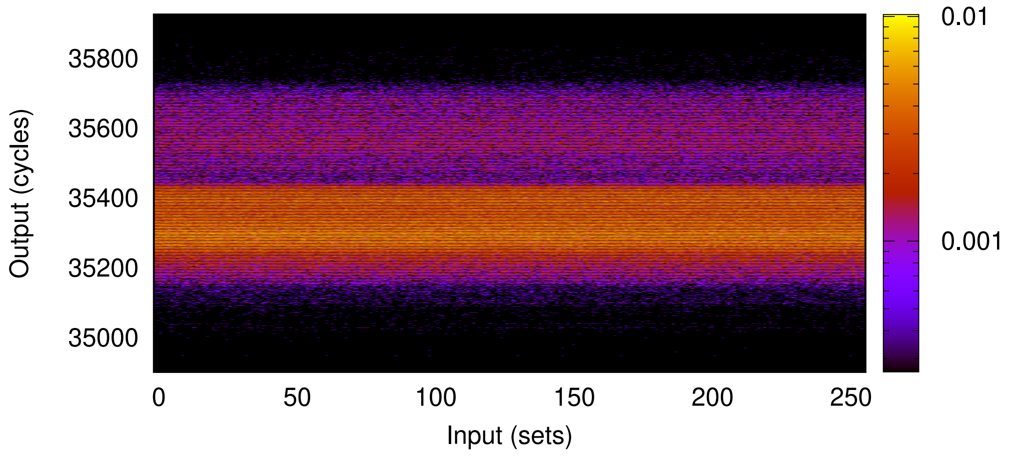


Figure 5.5: Channel matrix for the mitigated L1-I covert channel on an Arm Cortex-A9. This example uses 3,824 samples for each input symbol and shows a channel capacity of 0.7 mb (millibit) per usage, we summarise this as $\mathscr{M} = 0.7\,\mathrm{mb}$, $n = 3,824$.

5.3.3.2 Channel leakage

To quantify the leakage of timing channels, we use mutual information, MI, from Shannon information theory [Shannon, 1948], for measuring the size of a channel. MI quantifies the mutual dependency between an input and an output, representing the amount of information that the input variable passes to the output. A high MI value indicates a large reduction on the uncertainty about the input given the knowledge of the output. On the contrary, a zero MI means the two variables are independent.

In this work, MI represents the average number of bits of information that a computationally unbounded receiver can learn from each input symbol x by observing the output time y, representing how easily an input symbol can be guessed on average. Previous work [Cock et al., 2014] used discrete capacity [Shannon, 1948] for evaluating channels.

The discrete capacity treats each sample pair, (x, y), as unrelated separate values, hence can miss any patterns between those pairs. On the contrary, the MI treats the outputs as continuous, estimating the probability of a particular output y given some input x by examining every observation that resulted from x. For each of those observations, we calculate the distance from the observed $y\prime$ to y, and use that distance to estimate how that observation should affect our estimation at y.

We chose this model for the following three reasons. First, we treat all values as unordered and equivalent if we treat the output (timing measurement) as purely discrete. For example, a collection of unique particularly high values cannot be treated differently from a collection of unique uniformly distributed values by discrete capacity, therefore we might miss a leak. Second, a channel with continuous MI value of zero implies that the discrete capacity is also zero, for a uniform input distribution. Last, MI is easier to estimate reliably as an average function to compare against a maximum function (the discrete capacity), which makes the MI an effective metric for detecting channel leakage if there is any information leakage.

For all our experiments, we sample at least 1 million outputs from a pseudo-randomly generated input sequence. Then, we apply kernel density estimation [Silverman, 1986] to estimate the probability density function of outputs for each input. Finally, we use the rectangle method ([Hughes-Hallet et al., 2005] p. 340) to estimate the MI between a uniform distribution on inputs and the observed outputs, which we write as $\mathscr{M}$.

The channel bandwidth can now be calculated by multiplying the leakage ($\mathscr{M}$) by the input symbol rate, i.e., the frequency with which the pipe operates. For instance, Figure 5.4 has a leakage $\mathscr{M} = 2500\,\text{mb}$ (millibit), and $500 \times 2500 = 1,250,000\,\text{mb/s}$ throughput, if the Trojan and spy are scheduled 500 times per second (i.e., 1 ms time slice) on the same core (Figure 5.1). Although previous work achieved much higher bandwidth channels [Evtyushkin and Ponomarev, 2016; Gruss et al., 2016b; Liu et al., 2015; Maurice et al., 2017], our work does not focus on maximising the channel throughput, but rather demonstrates the existence of a channel to facilitate analysis.

5.3.3.3 Analysing timing channels

For all unmitigated and mitigated channels, we use both channel matrices and the channel leakage evaluation ($\mathscr{M}$) to analyse timing channels.

The zero-leakage test For each sampled data set, we conduct the MI evaluation. However, sampling can result in an apparent non-zero MI even when no channel actually exists due to noise. Therefore, we apply the following test to differentiate a significant leak from noise in the sampling process [Chothia and Guha, 2011; Chothia et al., 2013].

To perform the zero-leakage test, we simulate the measurement noise of a zero-leakage channel by shuffling the outputs in our dataset to randomly chosen inputs. The randomly assigned input and output pairs ensure that there is no correlation between input and output, representing a zero-leakage channel even though the new dataset has the same range of values as the sampled dataset. We repeat this process 100 times, and calculate $\mathscr{M}$ for each of the new datasets, resulting in 100 estimations for zero-leakage channels. Then, we calculate the mean and standard deviation of those 100 $\mathscr{M}$, and the exact 95% confidence interval for an estimate to be compatible with zero leakage, written as $\mathscr{M}_0$. Similar to calculating the MI for the original samples, the calculated $\mathscr{M}$ for each of the simulated channels can be non-zero as the simulation only removes any correlations between input and output but not the noise contained in the samples. In other words, $\mathscr{M}_0$ estimates the 95% confidence interval of a zero-leakage channel if noise exists: the calculated $\mathscr{M}_0$ is highly likely not a timing channel based on resource contentions, but noise.

If the $\mathscr{M}$ of a sampling dataset is larger than the $\mathscr{M}_0$ (outside the 95% confidence interval for zero leakage as $\mathscr{M} > \mathscr{M}_0$), we say the sampled data set is inconsistent with the MI being zero, and thus there is a leak on that sample set. The strict inequality is important here, because for very uniform data with no leakage $\mathscr{M}$ may equal $\mathscr{M}_0$: we consider that the dataset does not contain evidence of an information leak if the estimated MI is within, or equal to, the 95% confidence interval. Our present tool [Chothia et al., 2013] has a resolution of about 1 mb (millibit), and hence it cannot give conclusive evidence if $\mathscr{M} <$ 1 mb.

Confirming a timing channel For the unmitigated channels, we confirm the timing channel from matrices, and report the $\mathscr{M}$ value only, as the zero-leakage test is not relevant if the timing channel is obvious. For mitigated channels, we report both $\mathscr{M}$ and $\mathscr{M}_0$, and confirm any remaining channel by examining matrices obtained from repeated runs.

If we observe that $\mathscr{M} > \mathscr{M}_0$ for a mitigated channel, it is still possible that the evaluated $\mathscr{M}$ of a mitigated channel is not due to the timing channel, but some other side effects, such as hardware noise. In order to avoid any false claims, we repeat the experiments multiple times (five to eight times), and examine any consistent trend on the output distribution across the same range of inputs from channel matrices. A consistent trend on all repeated runs is solid evidence of a remaining timing channel, as the same distribution pattern is always related with the secret sent by the Trojan (inputs). Hence, we confirm that the evaluated MI is caused by residual channels, which are marked **bold** in Table 5.4. If there is no consistent pattern on matrices, we report the lower bound of the MI evaluation from repeated runs, as there is no evidence of timing channel on matrices.

Examining channel matrices obtained from repeated runs can also confirm any remaining channel if the leakage is less than the precision of the tool chain (1 mb), as well as detect any false negatives of the zero-leakage test.

5.4 Results

5.4.1 Evaluation platforms

We perform our study on the two most widely-used architectures, x86 and Arm, as listed in Table 5.1. For x86, we use the Haswell and Skylake microarchitectures; for Arm, we use the Sabre platform (Cortex-A9), an implementation of Arm v7, and the Hikey platform (Cortex-A53), an in-order (InO) implementation of Arm v8. We summarise their relevant features in Table 5.1, including caches, TLBs, and BTB. The information on the BTB is incomplete, because information is not published by the manufacturers.

The purpose here is to investigate the effectiveness of hardware operations on closing intra-core timing channels, which should be independent of the OS or hypervisor used. However, it becomes harder to preclude interference from software components on a more complex system, such as Linux, than when using a simple system, such as the seL4 microkernel [Klein et al., 2014; seL4]. We therefore use the seL4 microkernel (Section 2.6) as our base system.

In particular, we configure seL4 as a separation kernel [Rushby, 1984], where the Trojan and spy programs are scheduled by the system tick, and share nothing but the time source. This almost noise-free environment not only helps us in measuring small-capacity channels, but also simplifies the implementation of mitigation. Lastly, the formal verification of seL4 provides the absence of storage channels [Murray et al., 2013], which simplifies the analysis of results as any remaining channel must be a timing channel.

5.4.2 Overview of results

For each experiment, we collect more than 1 million input and output pairs. We summarise the result of unmitigated and mitigated MI analyses in Table 5.4. As introduced in Section 5.3.3.3, the $\mathscr{M}$ represents the MI calculated for the original samples whereas the $\mathscr{M}_0$ represents the 95% confidence interval of simulated zero-leakage channels. Both $\mathscr{M}$ and $\mathscr{M}_0$ can be non-zero due to noise even though the corresponding timing channel does not exist.

On x86 platforms, we also evaluate the mitigated channel without the IBC mechanism enabled [Intel, 2018c], to determine how effective IBC is in mitigating intra-core timing channels.

Raw channels The “none” rows in Table 5.4 demonstrate the unmitigated MI of all the channels on four testing processors. Most of the channels leak several bits per input symbol, except the BTB channels on the Skylake and Arm Cortex-A9, which have MI less than 100 mb. We suspect the reasons are replacement policies and the unknown BTB architecture on the x86 processors, as our simple attacks work best with a true least-recently used (LRU) replacement policy. Although we could try to enlarge the channel by reverse engineering

	Processor	**Haswell**	**Skylake**	**Cortex-A9**	**Cortex-A53**
Chan.	**Mitig.**	64-bit	64-bit	32-bit	32-bit
L1-D	none	4,053	4,458	2,070	1,540
	full	0.3 (0.3)	0.4 (0.4)	0.5 (0.6)	0.4 (0.4)
	no IBC	0.7 (0.9)	0.4 (0.4)	N/A	N/A
L1-I	none	260	2,088	2,489	4,137
	full	0.4 (0.4)	0.3 (0.3)	0.7 (0.7)	**18** (**0.8**)
	no IBC	**20.3** (**0.3**)	**9.7** (**0.4**)	N/A	N/A
TLB	none	2,564	2,106	559	544
	full	0.3 (0.3)	0.4 (0.4)	0.2 (0.3)	1.2 (1.2)
	no IBC	0.2 (0.2)	0.4 (0.5)	N/A	N/A
BTB	none	1,533	47	5.4	486
	full	0.4 (0.4)	0.8 (0.8)	2.0 (2.2)	**2.3** (**0.7**)
	no IBC	**307** (**0.7**)	**1.3** (**0.9**)	N/A	N/A
BHB	none	1,000	1,000	1,000	1,067
	full	**0.4** (**0.0**)	**0.8** (**0.1**)	0.0 (0.3)	**37** (**0.0**)
	no IBC	**555** (**0.0**)	**170** (**7.6**)	N/A	N/A

Table 5.4: Observed unmitigated ("none"), maximally mitigated ("full"), and mitigated without IBC ("no IBC", x86 only) MI (millibit) and the 95% confidence interval of simulated zero-leakage channels ($\mathscr{M}_0$). Value in parentheses is $\mathscr{M}_0$. Confirmed residual channels are marked in **bold**.

the replacement policy, the purpose of this work is to show timing channels on on-core microarchitectural components (Section 5.1).

As Table 5.4 demonstrates, all the microarchitectural components that contain recent execution history can be used for creating timing channels. As a result, the OS must consider the effect of those microarchitectural components when trying to enforce temporal isolation in the confinement scenario (Section 4.1.1). We will now examine these results in detail.

Mitigated channels The "full" rows in Table 5.4 show the MI with all mitigations enabled during the domain switch. Note the mitigation is more than flushing caches, rather enabling all hardware provided operations listed in Section 5.3.2, to provide a deterministic execution environment.

From Table 5.4, we observe that most of the channels are effectively mitigated by hardware functions for eliminating the microarchitectural state: the $\mathscr{M}$ of sampling datasets are inside the 95% confidence interval for zero-leakage channels ($\mathscr{M}_0$); and we cannot observe any consistent patterns on channel matrices obtained from multiple runs. Still, a few channels on Arm Cortex-A53 contain evidence of an information leak on matrices generated from repeated runs (Section 5.3.3.3). We highlight them in **bold**.

We also highlight the residual BHB timing channels on x86 processors with all mitigation enabled, as we observed a consistent distribution pattern from matrices generated from repeated runs, even though $\mathscr{M}$ values are beyond the precision of our tool chain (1 mb).

5.4.3 The effectiveness of hardware resetting operations

We generate all matrices for channels listed in Table 5.4, visualising the timing variance caused by microarchitectural contention if any.

5.4.3.1 L1-D cache

As mentioned in Section 5.3.1.1, the L1-D cache timing channel creates cache contentions though reading a buffer. Based on results listed in Table 5.4, the channel is completely closed on all four testing platforms, showing the cache flushing operations are very effective in closing the L1-D channel.

Figure 5.6 demonstrates matrices for the unmitigated (top), mitigated without the IBC mechanism enabled (middle), and maximally mitigated (bottom) L1-D cache channels on the Haswell processor.

From the matrix of the unmitigated channel, we can observe that the Trojan's activity (input), probing S cache sets on each system tick, is correlated with the probing cost measured by the spy (output). However, the trend disappears and the channel is mitigated when the kernel conducts hardware resetting operations (Section 5.3.2.1) during a domain switch, as the output is evenly distributed across all possible inputs, irrespective of the IBC mechanism.

Similarly, the L1-D channel is closed on Skylake (Figure 5.7), Arm Cortex-A9 (Figure 5.8), and Arm Cortex-A53 (Figure 5.9) platforms.

5.4.3.2 L1-I cache

The L1-I cache timing channel (Section 5.3.1.2) is an instruction-based channel, which probes L1-I cache sets with jumping instructions.

The channel is effectively closed on x86 platforms with the IBC mechanism enabled. As Figure 5.10 shows, there is a definite horizontal variation around input value 2 on the channel matrix (middle) without the IBC mechanism involved during the domain switch on the Haswell processor. However, the distribution of the output becomes evenly distributed across all possible input values once the IBC mechanism is introduced (bottom). Results for the MI evaluation are consistent with the observations from those matrices, the channel leakage is $\mathscr{M} = 20.3$ mb ($\mathscr{M}_0 = 0.3$ mb) while being mitigated without the IBC mechanism enabled, but $\mathscr{M} = 0.4$ mb ($\mathscr{M}_0 = 0.4$ mb) while being maximally mitigated, consistent with a fully closed channel.

We observe a similar phenomenon on the Skylake processor (Figure 5.11): the output is higher on average at inputs 0–10 without IBC (middle) whereas it is evenly distributed with

IBC enabled (bottom). In terms of the channel leakage, results of the MI evaluation are $\mathscr{M} = 9.7\,\text{mb}$ ($\mathscr{M}_0 = 0.4\,\text{mb}$) without enabling the IBC, which are down to $\mathscr{M} = 0.3\,\text{mb}$ ($\mathscr{M}_0 = 0.3\,\text{mb}$) with IBC enabled.

For Arm processors, the L1-I cache channel is fully closed on the Cortex-A9, but less so on the Cortex-A53. On the Cortex-A9 (Figure 5.12), the matrix for unmitigated channel (top) shows a clear correlation between inputs, number of cache sets probed by the Trojan using jumping instructions, and outputs, probing cost measured by the spy. With hardware resetting operations (bottom), there is no horizontal variation across all possible inputs. The unmitigated channel has $\mathscr{M} = 2,489\,\text{mb}$, which is completely closed with hardware resetting operations given the $\mathscr{M} = 0.7\,\text{mb}$ is inside the 95% confidence interval for zero-leakage channels, $\mathscr{M}_0 = 0.7\,\text{mb}$.

However, on the Cortex-A53 processor (Figure 5.13), the hardware resetting operations (bottom) are less effective, as the distribution of the output drops at inputs 0–50 then climbs gradually from input 50. Results of the MI evaluation are consistent with facts observed on those matrices: there is a small amount of residual channel while the hardware resetting operations are engaged, $\mathscr{M} = 18\,\text{mb}$ is outside of the 95% confidence interval for zero-leakage channels, $\mathscr{M}_0 = 0.8\,\text{mb}$.

5.4.3.3 TLB

The TLB timing channel (Section 5.3.1.3) uses the contention on TLB entires that are created by reading a number of consecutive pages. Similar to the L1-D cache channel (Section 5.4.3.1), the TLB channel is fully closed on all testing platforms.

Figure 5.14 shows channel matrices for unmitigated (top), mitigated without IBC (middle), and maximally mitigated channels (bottom) on the Haswell processor. On the matrix for unmitigated channel, we can observe the horizontal variation around a list of input values, 6, 20, 35–40, and 48. The variation disappears on matrices for the mitigated channels, regardless of the engagement of the IBC mechanism. Results of the MI evaluation also show that the channel is closed nonetheless of the engagement of the IBC mechanism, $\mathscr{M} = 0.2\,\text{mb}$ ($\mathscr{M}_0 = 0.2\,\text{mb}$) without enabling the IBC, and $\mathscr{M} = 0.3\,\text{mb}$ ($\mathscr{M}_0 = 0.3\,\text{mb}$) with the IBC.

Figure 5.15 demonstrates a similar situation on the Skylake processor: the horizontal variation that occurs on the matrix for unmitigated channel (top) vanishes on matrices for mitigated channels (middle and bottom).

Similar with x86 processors, the channel is fully closed on the Arm Cortex-A9 (Figure 5.16) and Cortex-A53 processors (Figure 5.17).

5.4.3.4 BTB

For the BTB timing channel (Section 5.3.1.4), the Trojan and spy probe BTB entries with jump instructions.

Similar to results for mitigating the L1-I channel (Section 5.4.3.2), the IBC mechanism is very effective on closing the BTB channel on x86 processors. Moreover, hardware resetting operations completely close the channel on the Arm Cortex-A9 processor but leave a small amount of residual channel on the Cortex-A53 processor.

Figure 5.18 shows matrices for the BTB channel on the Haswell processor. Without IBC mechanism involved, the matrix (middle) contains the curiously shaped pattern around inputs 3640–3660, which disappears on the bottom matrix, generated for the maximally mitigated channel. Results of the channel leakage evaluation also prove that there is a distinctive amount of residual channel without the IBC, $\mathscr{M} = 307\,\text{mb}$ ($\mathscr{M}_0 = 0.7\,\text{mb}$), which is fully closed with all hardware resetting operations, $\mathscr{M} = 0.4\,\text{mb}$ ($\mathscr{M}_0 = 0.4\,\text{mb}$).

On the Skylake processor (Figure 5.19), the IBC mechanism enhances the effectiveness of the hardware resetting operations, the evaluated channel leakage becomes $\mathscr{M} = 0.8\,\text{mb}$ ($\mathscr{M}_0 = 0.8\,\text{mb}$) compared to $\mathscr{M} = 1.3\,\text{mb}$ ($\mathscr{M}_0 = 0.9\,\text{mb}$) without the IBC mechanism being involved.

On the Arm Cortex-A9 processor (Figure 5.20), we can observe that the distribution of outputs slides slightly downwards around inputs 400–500 on the matrix for unmitigated channel (top). However, the matrix for the mitigated channel (bottom) shows that the outputs are evenly distributed across inputs. Results of the MI evaluation also shows the channel is completely closed with hardware resetting operations, $\mathscr{M} = 2.0\,\text{mb}$ ($\mathscr{M}_0 = 2.2\,\text{mb}$).

On the Arm Cortex-A53 processor (Figure 5.21), a small amount of channel leakage remains even though the channel is maximally mitigated with hardware resetting operations. The MI evaluation for the mitigated channel shows that the $\mathscr{M} = 2.3\,\text{mb}$ is outside the 95% confidence interval for zero-leakage channels, $\mathscr{M}_0 = 0.7\,\text{mb}$. By observing the matrix carefully, we identify that the distribution of outputs curve slightly downwards while the inputs are between 50–200, an evidence of a small amount of residual channel. We observe the same phenomenon on matrices generated for six repeated runs, confirming the residual timing channel.

5.4.3.5 BHB

The BHB timing channel (Section 5.3.1.5) is a single-bit channel, showing the timing effect of colliding on a BHB entry that is caused by probing on a condition branch instruction.

Results for the channel leakage evaluation (Table 5.4) demonstrate that the IBC mechanism improves the effectiveness of mitigating the channel on x86 processors.

For the unmitigated channel created on the Haswell processor, the top matrix in Figure 5.22, we see that the Trojan successfully influences the latency of the conditional branch instruction executed by the spy: the conditional branch instruction takes longer to execute when the CPU mispredicts the outcome (i.e., the input is “1”). Furthermore, the channel remains without the IBC mechanism, as shown on the middle matrix in Figure 5.22. However, the IBC mechanism does not fully eliminate the remaining channel, which can be observed on the bottom matrix in Figure 5.22— a tiny amount, less than 0.5% based on the data for generating the matrix, of outputs is smaller than 40 if input is “0”. The same trend exists in five repeated runs, confirming that the timing variance is not due to hardware noise but a residual channel. In terms of MI evaluations, the remaining channel is significant without the IBC involved, $\mathscr{M} = 555\,\text{mb}$ ($\mathscr{M}_0 = 0.0\,\text{mb}$), which is reduced to $\mathscr{M} = 0.4\,\text{mb}$ ($\mathscr{M}_0 = 0.0\,\text{mb}$).

Results are similar on the Skylake processor (Figure 5.23). The probing time clearly identifies the two input values—0 and 1 on the middle matrix in Figure 5.23, demonstrating a residual channel without the IBC enabled. The remaining channel has $\mathscr{M} = 170\,\text{mb}$ ($\mathscr{M}_0 = 0.0\,\text{mb}$) leakage. With IBC enabled, we can observe that a small number of output values, less than 2.6% based on our data, are smaller than 37 while the input is “0”. We also observe the same trend in five repeat runs, representing a residual channel, even though the calculated $\mathscr{M} = 0.8\,\text{mb}$ ($\mathscr{M}_0 = 0.1\,\text{mb}$) is beyond the the precision of our tool chain (1 mb).

For Arm processors the story is similar to that for the L1-I cache and BTB channels: hardware resetting operations fully close the channel on the Arm Cortex-A9 processor, but leaves a residual channel on the Arm Cortex-A53 processor $\mathscr{M} = 37\,\text{mb}$ ($\mathscr{M}_0 = 0.0\,\text{mb}$).

On the Arm Cortex-A9 processor (Figure 5.24), the channel is completely closed. The bottom matrix in Figure 5.24 shows that outputs are evenly distributed across inputs. The MI value for the mitigated channel is $\mathscr{M} = 0.0\,\text{mb}$ ($\mathscr{M}_0 = 0.3\,\text{mb}$).

The situation is different on the Arm Cortex-A53 processor. Based on our data for generating the bottom matrix in Figure 5.25, 20% of the input is smaller than 500 for input “0”, which demonstrates that the hardware resetting operations cannot fully prevent the contention on the BHB entry. The MI evaluation for the residual channel is $\mathscr{M} = 37\,\text{mb}$ ($\mathscr{M}_0 = 0.0\,\text{mb}$).

5.4.3.6 Summary

All studied x86 and Arm processors are very effective at mitigating L1-D (Section 5.4.3.1) and TLB (Section 5.4.3.1) channels. Those timing channels create contention on targeted microarchitectural components with data accesses (i.e., data-based).

For instruction-based timing channels, L1-I (Section 5.4.3.2), BTB (Section 5.4.3.4), and BHB (Section 5.4.3.5) channels, the IBC mechanism significantly reduces the MI on x86 processors, even though the manufacturer designed it as mitigation for the Spectre attack [Kocher et al., 2019]. Our results demonstrate that manufacturers are able to provide more effective operations to mitigate timing channels by resetting all possible microarchitectural state. However, we are able to observe a small amount of timing variance on channel matrices for the BHB channel on x86 processors, which helps to verify the small amount of residual channel while MI values are less than the precision of our tool chain.

For Arm processors, we observe that it is more difficult to completely close instruction-based timing channels with current available hardware operations on advanced processors, compared to data-based timing channels: the Arm Cortex-A53 contains residual channels on

L1-I cache, BTB, and BHB, which are all closed on the Arm Cortex-A9, an implementation of an earlier version of the Arm architecture. Still, the hardware provided operations largely reduces the MI for those channels on the Arm Cortex-A53, with maximum MI less than 40 mb.

The cause of the residual channels is that hidden microarchitecture state is preserved across a context switch. It is highly likely that the state accumulated in the instruction prefetcher contributes to the residual channels, due to the lack of operations for either flushing or disabling the instruction prefechter on all our testing platforms (Section 5.3.2).

5.5 Security Needs a New Hardware-Software Contract

We have seen that L1-I cache, BTB, and BHB timing channels cannot be removed on the Arm Cortex-A53 processors by using microarchitectural history resetting operations. Furthermore, a small BHB channel remains on both Haswell and Skylake processors, even after applying all resetting operations including the IBC mechanism. Therefore, there is a certain amount of residual microarchitectural state on those processors that cannot be reset. To address this problem, the hardware manufacturer should provide more information on resetting microarchitectural components that cannot be partitioned as part of the ISA.

5.5.1 The Augmented ISA: A New Hardware-Software Contract

The ISA, which defines the hardware-software contract, gives freedom to both hardware and software designs, with the expectation that everything works functionally correctly as long as everyone observes the contract. However, the ISA is not the right contract for ensuring timing channel protection, because it abstracts away the hardware state that accelerates system performance but leaves a great opportunity for microarchitectural timing channels. In other words, *we need a new hardware-software contract*, which must satisfy the following requirements:

1. It must provide the OS with sufficient mechanisms for supporting time protection;
2. it should be as simple as possible;
3. it should not reveal more architectural details than absolutely necessary; and
4. it should minimise restrictions imposed on architects.

Points 2–4 can be viewed as different aspects of the same principle, and are important in practice. The simplicity requirement helps both hardware and software on implementation. Additionally, manufacturers can choose to not reveal critical IP, retaining their freedom for microarchitectural innovations. Together these points argue for minimal augmentation to, rather than wholesale replacement of, the ISA. We therefore call the new contract the augmented ISA (aISA).

Based on this observation, we define the minimal properties the aISA should provide:

Property 1: Security enforcement
Any shared microarchitectural feature can either be spatially partitioned between security domains, or reset to a known state when required by privileged software.

Furthermore, resetting cannot close timing channels exploited between threads that are running concurrently on multiple cores or hardware threads, e.g., the LLC cross-core timing attack (Section 3.2.3.2). This lack of separation implies a second property:

Property 2: Secure concurrent sharing
Microarchitectural features accessed by concurrent execution streams must be partitionable; partitions must be completely static or controlled by a privileged software layer, such as the OS or hypervisor.

It is essential that the OS controls partitioning changes: in terms of information flow, dynamic partitioning by hardware is no different from a normal data or instruction cache, the dynamics can be exploited as a timing channel. However, the hardware can provide mechanisms for assisting partitioning, such as locking cache ways to threads.

As the OS cannot partition state accessed solely by virtual address (Section 2.4), we require:

Property 3: Secure virtually indexed state
Hardware state indexed solely by virtual address must not be concurrently accessible and must be resettable.

This condition does not apply to access using a combination of virtual address and (temporary unique) thread identification.

To allow the OS to partition microarchitectural components, the aISA must provide sufficient information. For instance, the OS can efficiently partition the physically-indexed hardware caches using cache colouring (Section 2.4), as long as the OS can determine the number of available colours. More recent Intel processors implement hashing functions on the LLC, which distributes cache lines to different LLC blocks [Yarom et al., 2015]. Although providing information on the hashing scheme can increase the number of available colours, the OS only needs to know an over-approximation of the number of colours.

Similarly, the aISA must clearly define reset mechanisms, especially for any aspect related with timing. For example, the execution history, such as the number of dirty cache lines, can affect the reset latency, which can be used as a timing channel. Therefore, the reset must be either a constant-time operations or padded to its worst-case latency.

Property 4: Specified mechanisms
The aISA must completely specify the mechanisms used to partition or reset microarchitectural features. Reset operations must be constant time or have a specified worst-case latency.

For resettable state, privileged software benefits from knowing the kind of information that is cached (data, instructions or addresses of data or instructions) to optimise system performance:

Property 5: State provenance
The aISA should specify whether a reset operation acts on state derived from data, instructions, data addresses or instruction addresses.

The aISA can abstract resetting of multiple features into a single operation, such as flushing all on-core microarchitectural states (L1 caches, TLB, branch predictor and prefetcher) by a single reset operation. However, a detailed abstraction can help the software to achieve potential performance advantages, as only the specified resetting operation is engaged to prevent a targeted attack.

5.5.2 Discussion

The augmented ISA that we proposed in Section 5.5.1 provides more than a functional specification of instructions, rather, the aISA ensures that the architecture provides a timing-secure microarchitecture. We now examine the implications and its potential cost.

5.5.2.1 Implications

Firstly, we note that the two options in Property 1, partitioning and reset, must not interfere with each other. For instance, the operation for flushing all levels of caches breaks partitioning of caches (i.e., cache colouring on the LLC), as a malicious program can invalidate all cache lines on the LLC including those that belong to other partitions. Therefore, the hardware should not provide that operation to user-level programs, but only as a privileged instruction for the OS or VMM. In other words, resetting instructions for partitionable features must be reserved for use by the privileged software layer, such as the OS or hypervisor, else the hardware fails to observe Property 1.

Secondly, Property 2 does not imply that all hardware threads on a core have to belong to the same security partition, but the microarchitecture would have to partition all state that is concurrently shared, including L1 caches, TLB, branch predictors, prefetchers, and any on-core state; this would seem to turn threads into full-blown cores.

Lastly, the requirement for resetting operations in Property 4, presenting a constant or defined latency, might seem restrictive at first glance. However, we argue that this is not the case: the resetting operation is required only for on-core resources that cannot be partitioned, such as pipeline states or core-private L1 caches. Resetting read-only data, such as instructions stored in the L1-I cache, generates little variance in operating latency, as the hardware only needs to invalidate those cache lines. Hence, resetting on-core state that contains solely read-only data, such as the L1-I cache, can be implemented as a constant-time operation.

Resetting caches that contain modified data is more expensive, as the hardware has to flush those modified cache lines down the memory hierarchy to maintain the cache coherency. Additionally, resetting operations may have multiple use cases, including maintaining the correctness of a system. For example, flushing the L1-D cache is useful not only for preventing the L1-D timing attack but also for maintaining the coherency between the instruction and data caches, a common case for implementing self-modifying code. The hardware may reveal only the worst-case latency of those resetting operations, because requiring the hardware to always enforce constant time execution may not be reasonable. As long as it knows the worst case, the privileged software layer can enforce time protection with software padding. Alternatively, the hardware might provide a parameterised flushing operation, which guarantees a minimum latency for the resetting operation.

5.5.2.2 Cost

In terms of the cost, the system pays not only the cost of flushing the microarchitecture state, the *direct* cost, but also the *indirect* cost of starting with a cold state (i.e., cold caches or predictors).

As discussed in Section 3.3.4, we argue that the direct cost is low for resetting a small amount of microarchitecture state, such as the 32 KiB L1-D cache on x86 processors. Previous work [Varadarajan et al., 2014] measured an 8.4 μs direct cost for flushing L1 caches manually on a 6 core Intel Xeon E5645 processor. In addition, the indirect cost is also small if security domains are not lightweight, such as VMs hosted on cloud computing platforms, because the cached content of previously running domains will be overwritten by the next running domain even without flushing. In other words, the small cache is normally cold anyway after a domain switch, thus flushing operations do not introduce much indirect cost. Moreover, flushing operations are only required when switching security domains, which are typically operated at a rate of 10–100 Hz. Such a long execution time implies that the relative cost of flushing L1 caches is quite affordable, unlike flushing all levels of caches.

Previous research has done in-depth studies of the cost of cache partitioning. For instance, Sanchez and Kozyrakis [2011] evaluated the cost of statically partitioning the LLC between cores, presenting a performance overhead of 7%. Additionally, STEALTH-MEM [Kim et al., 2012] measured the cost of providing pinned pages in the LLC as 5.9% for the SPEC 2006 CPU benchmark. This measured cost is far less than the cost of the Spectre [Kocher et al., 2019] or Meltdown [Lipp et al., 2018] defences. Recently, cache colouring has also been used to improve system performance [Han et al., 2018; Noll et al., 2018]. In the next chapter, we also evaluate the cost of cache colouring on seL4, which will be presented in Section 6.5.3.4.

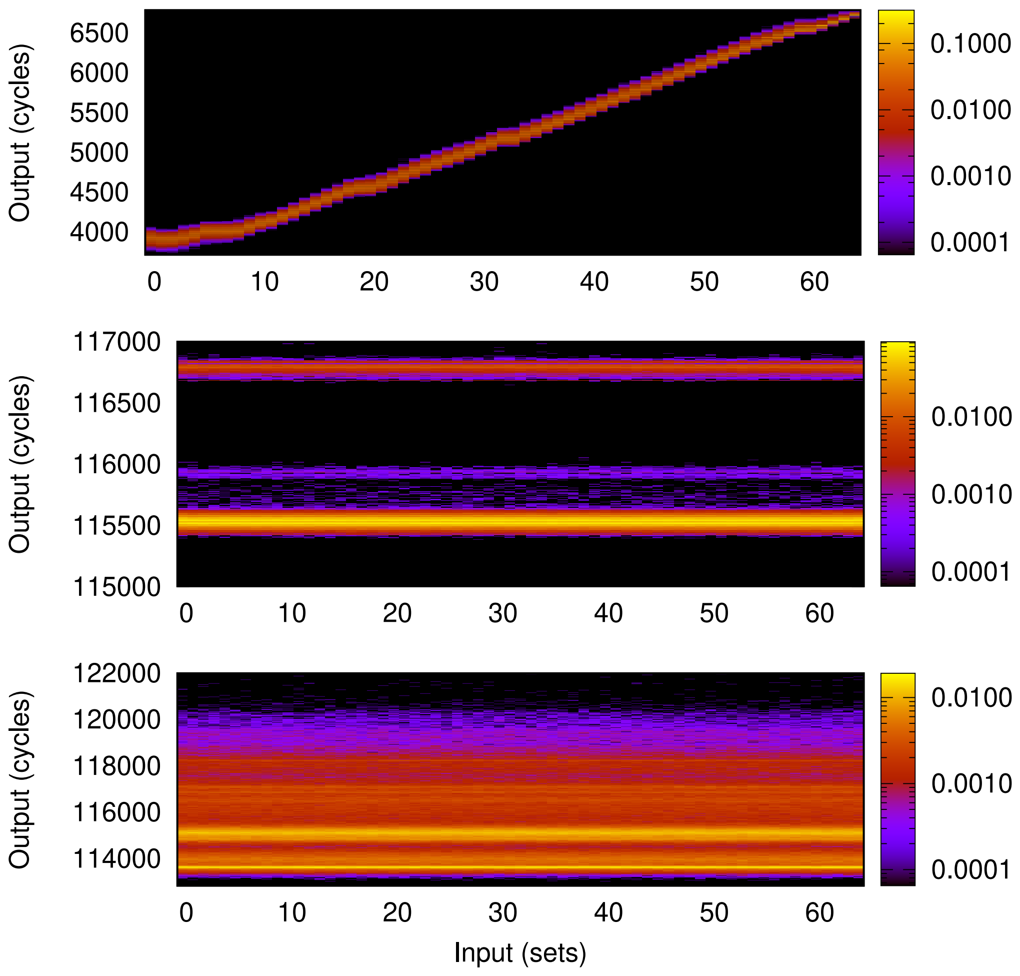


Figure 5.6: Channel matrix for the L1-D covert channel on Haswell. Top: without mitigation $\mathscr{M} = 4{,}053\,\mathrm{mb}$, $n = 15{,}436$, middle: mitigated without IBC $\mathscr{M} = 0.7\,\mathrm{mb}$, $\mathscr{M}_0 = 0.9\,\mathrm{mb}$, $n = 15{,}461$, bottom: maximally mitigated $\mathscr{M} = 0.3\,\mathrm{mb}$, $\mathscr{M}_0 = 0.3\,\mathrm{mb}$, $n = 15{,}444$.

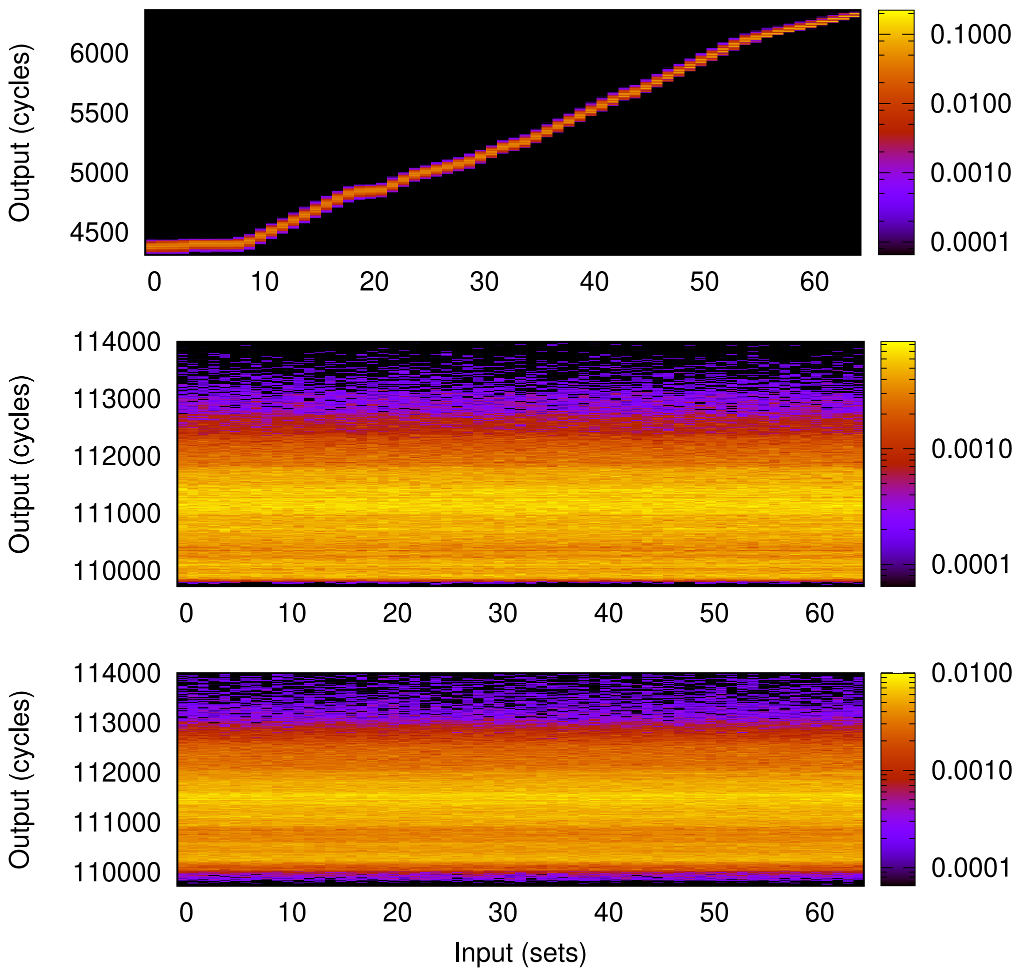


Figure 5.7: Channel matrix for the L1-D covert channel on Skylake. Top: without mitigation $\mathscr{M} = 4,458\,\mathrm{mb}$, $n = 15,366$, middle: mitigated without IBC $\mathscr{M} = 0.4\,\mathrm{mb}$, $\mathscr{M}_0 = 0.4\,\mathrm{mb}$, $n = 15,510$, bottom: maximally mitigated $\mathscr{M} = 0.4\,\mathrm{mb}$, $\mathscr{M}_0 = 0.4\,\mathrm{mb}$, $n = 15,341$.

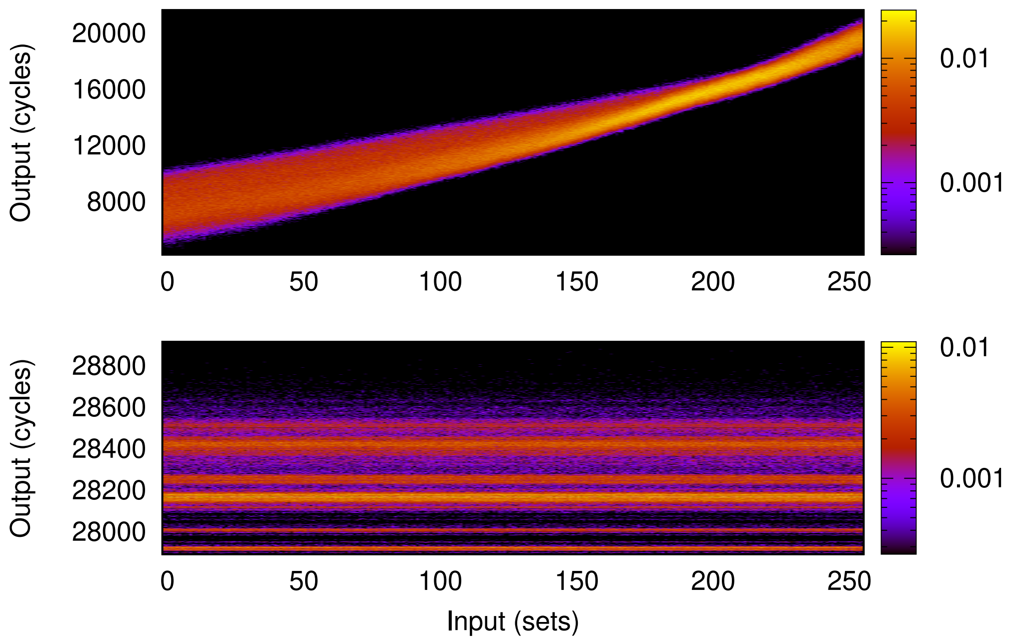


Figure 5.8: Channel matrix for the L1-D covert channel on Arm Cortex-A9. Top: without mitigation $\mathscr{M} = 2,070\,\text{mb}$, $n = 3,808$, bottom: maximally mitigated $\mathscr{M} = 0.5\,\text{mb}$, $\mathscr{M}_0 = 0.6\,\text{mb}$, $n = 3,810$.

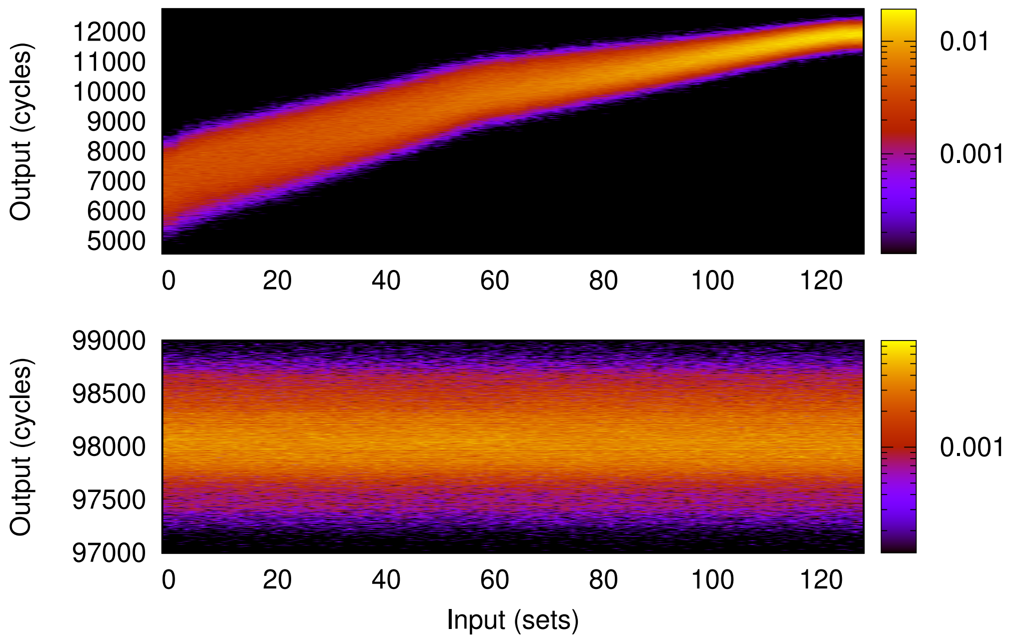


Figure 5.9: Channel matrix for the L1-D covert channel on Arm Cortex-A53. Top: without mitigation $\mathscr{M} = 1,540\,\text{mb}$, $n = 7,728$, bottom: maximally mitigated $\mathscr{M} = 0.4\,\text{mb}$, $\mathscr{M}_0 = 0.4\,\text{mb}$, $n = 7,693$.

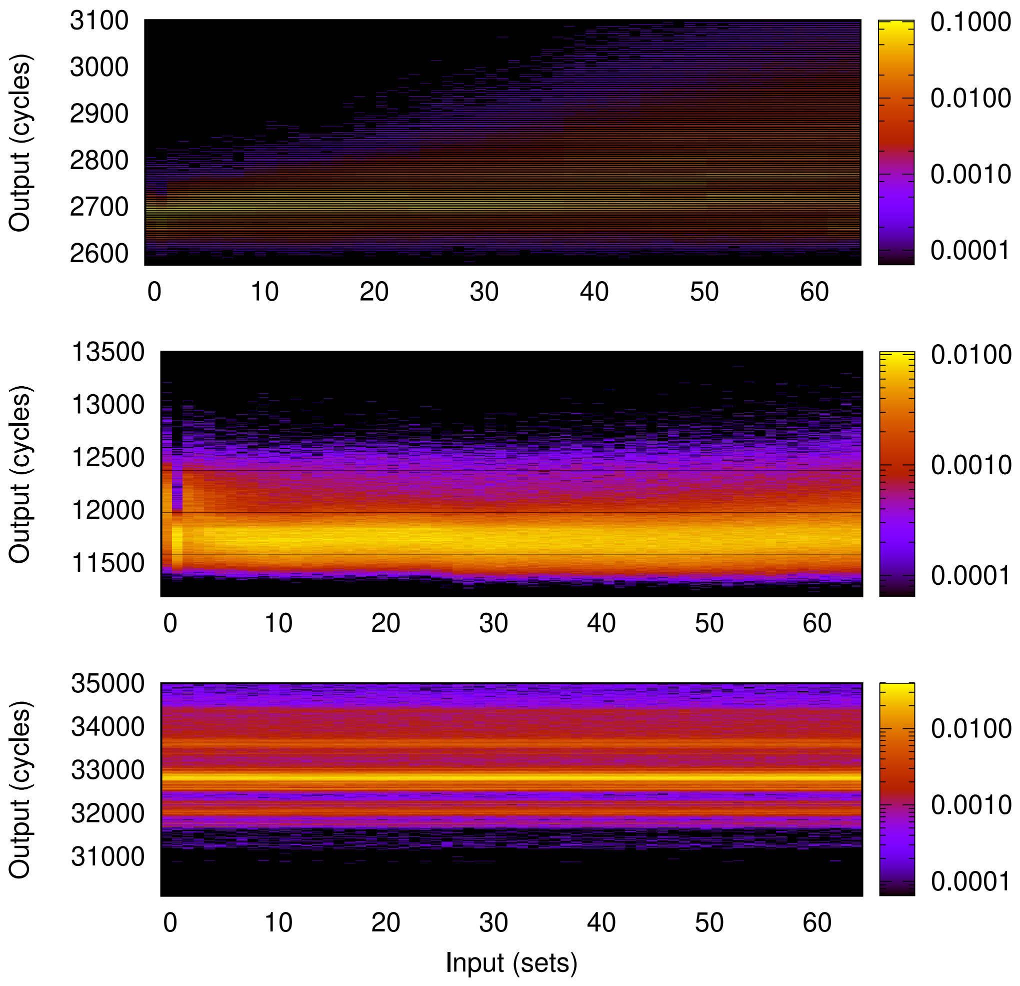


Figure 5.10: Channel matrix for the L1-I covert channel on Haswell. Top: without mitigation $\mathscr{M} = 260\,\mathrm{mb}$, $n = 15,532$, middle: mitigated without IBC $\mathscr{M} = 20.3\,\mathrm{mb}$, $\mathscr{M}_0 = 0.3\,\mathrm{mb}$, $n = 15,480$, bottom: maximally mitigated $\mathscr{M} = 0.4\,\mathrm{mb}$, $\mathscr{M}_0 = 0.4\,\mathrm{mb}$, $n = 15,482$.

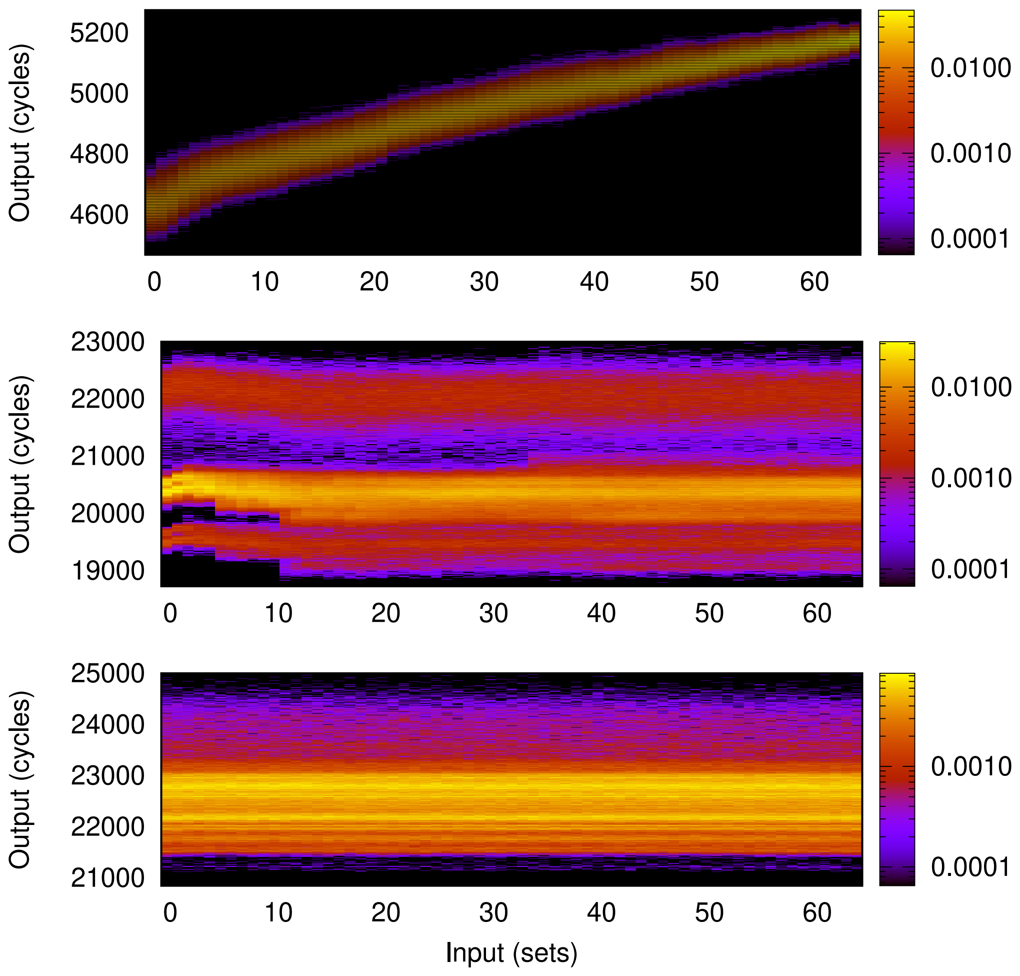


Figure 5.11: Channel matrix for the L1-I covert channel on Skylake. Top: without mitigation $\mathscr{M} = 2{,}088\,\text{mb}$, $n = 15{,}438$, middle: mitigated without IBC $\mathscr{M} = 9.7\,\text{mb}$, $\mathscr{M}_0 = 0.4\,\text{mb}$, $n = 15{,}472$, bottom: maximally mitigated $\mathscr{M} = 0.3\,\text{mb}$, $\mathscr{M}_0 = 0.3\,\text{mb}$, $n = 15{,}406$.

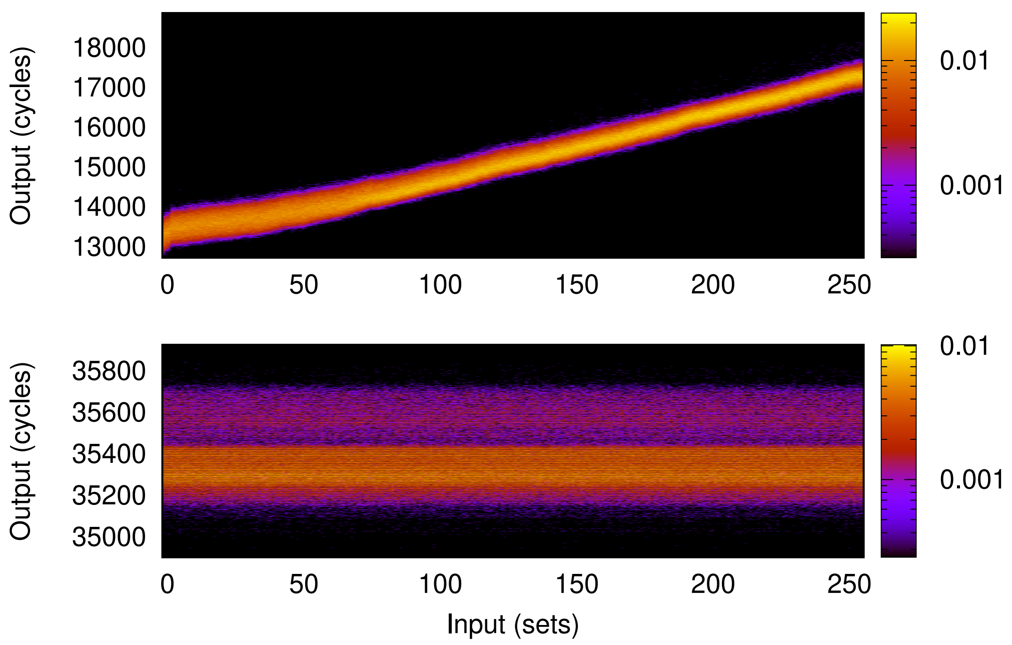


Figure 5.12: Channel matrix for the L1-I covert channel on Arm Cortex-A9. Top: without mitigation $\mathscr{M} = 2,489$ mb, $n = 3,808$, bottom: maximally mitigated $\mathscr{M} = 0.7$ mb, $\mathscr{M}_0 = 0.7$ mb, $n = 3,824$.

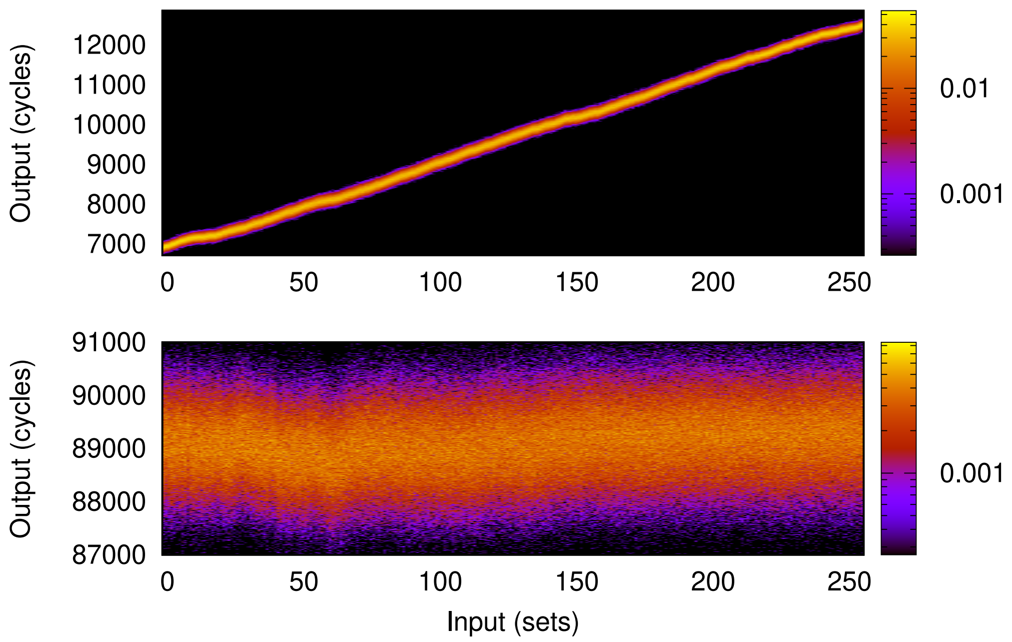


Figure 5.13: Channel matrix for the L1-I covert channel on Arm Cortex-A53. Top: without mitigation $\mathscr{M} = 4,137$ mb, $n = 3,822$, bottom: maximally mitigated $\mathscr{M} = 18$ mb, $\mathscr{M}_0 = 0.8$ mb, $n = 3,797$.

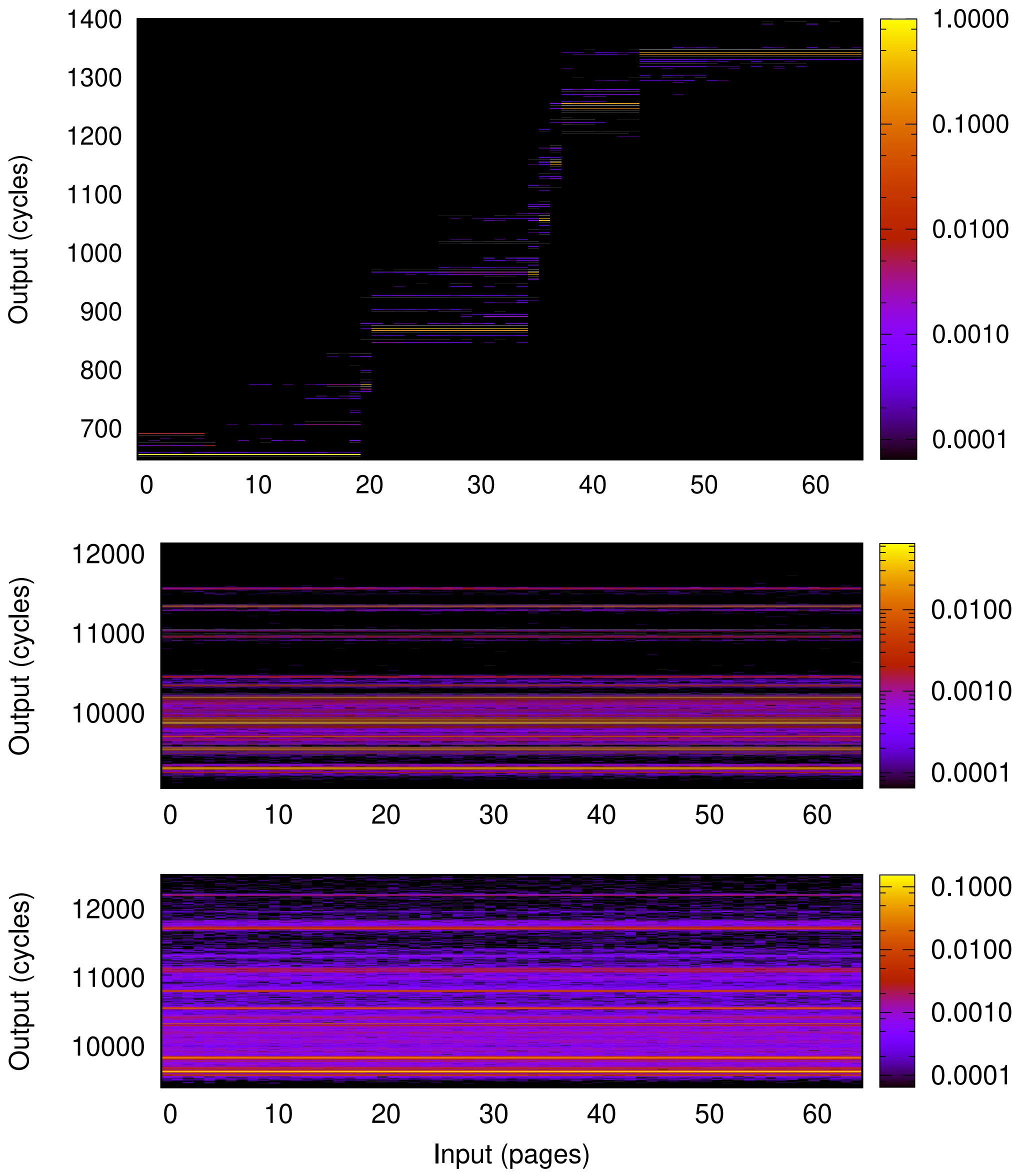


Figure 5.14: Channel matrix for the TLB covert channel on Haswell. Top: without mitigation $\mathscr{M} = 2{,}564\,\text{mb}$, $n = 15{,}511$, middle: mitigated without IBC $\mathscr{M} = 0.2\,\text{mb}$, $\mathscr{M}_0 = 0.2\,\text{mb}$, $n = 15{,}440$, bottom: maximally mitigated $\mathscr{M} = 0.3\,\text{mb}$, $\mathscr{M}_0 = 0.3\,\text{mb}$, $n = 15{,}529$.

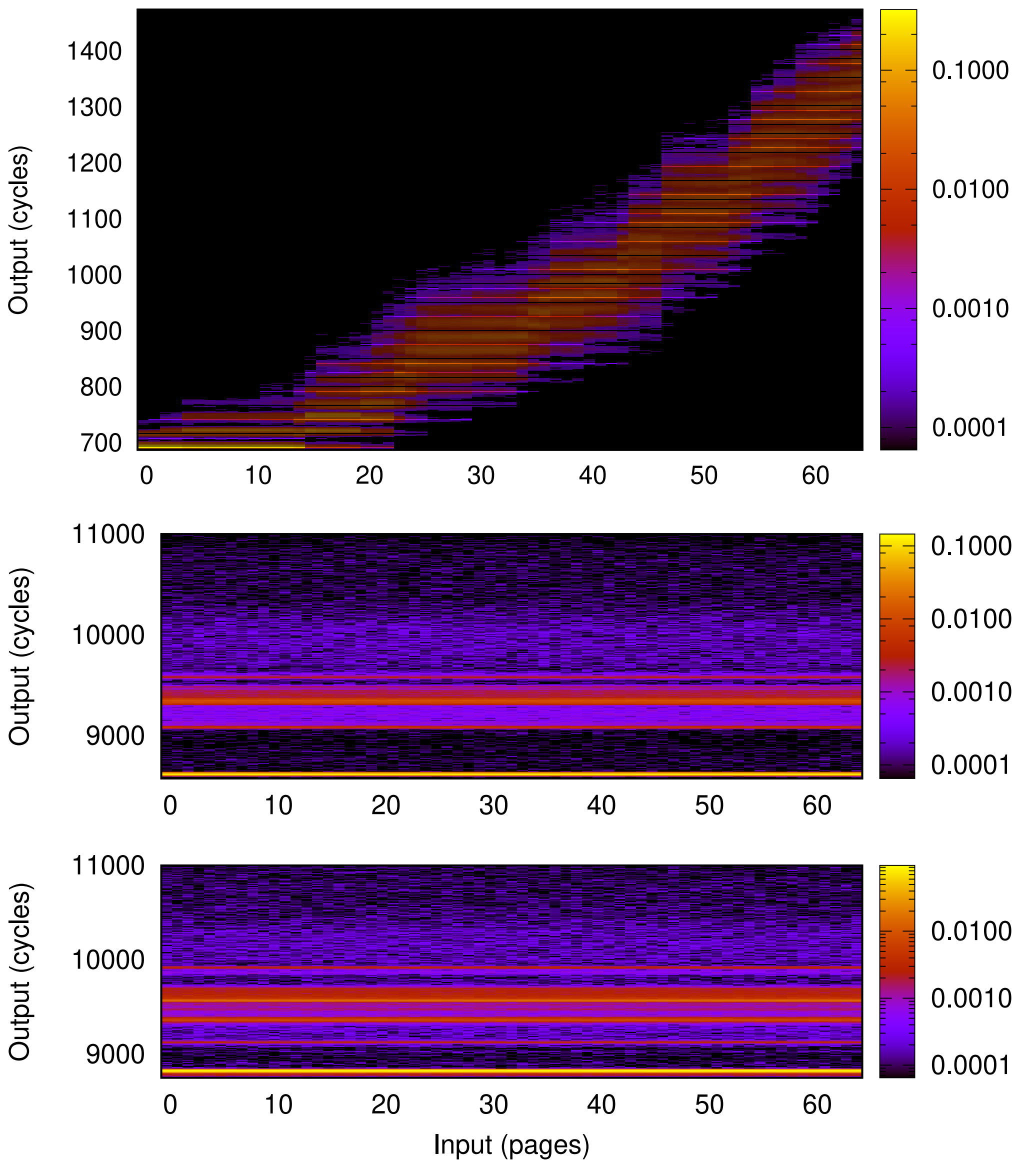


Figure 5.15: Channel matrix for the TLB covert channel on Skylake. Top: without mitigation $\mathscr{M} = 2{,}106\,\text{mb}$, $n = 15{,}444$, middle: mitigated without IBC $\mathscr{M} = 0.4\,\text{mb}$, $\mathscr{M}_0 = 0.5\,\text{mb}$, $n = 15{,}428$, bottom: maximally mitigated $\mathscr{M} = 0.4\,\text{mb}$, $\mathscr{M}_0 = 0.4\,\text{mb}$, $n = 15{,}494$.

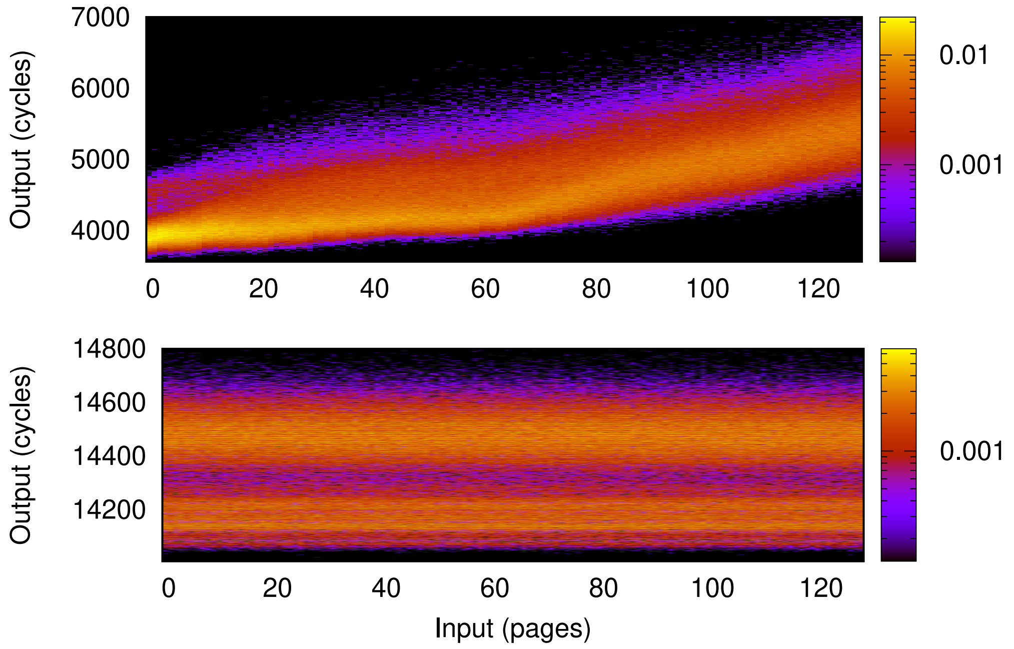


Figure 5.16: Channel matrix for the TLB covert channel on Arm Cortex-A9. Top: without mitigation $\mathscr{M} = 559\,\text{mb}$, $n = 7{,}680$, bottom: maximally mitigated $\mathscr{M} = 0.2\,\text{mb}$, $\mathscr{M}_0 = 0.3\,\text{mb}$, $n = 7{,}617$.

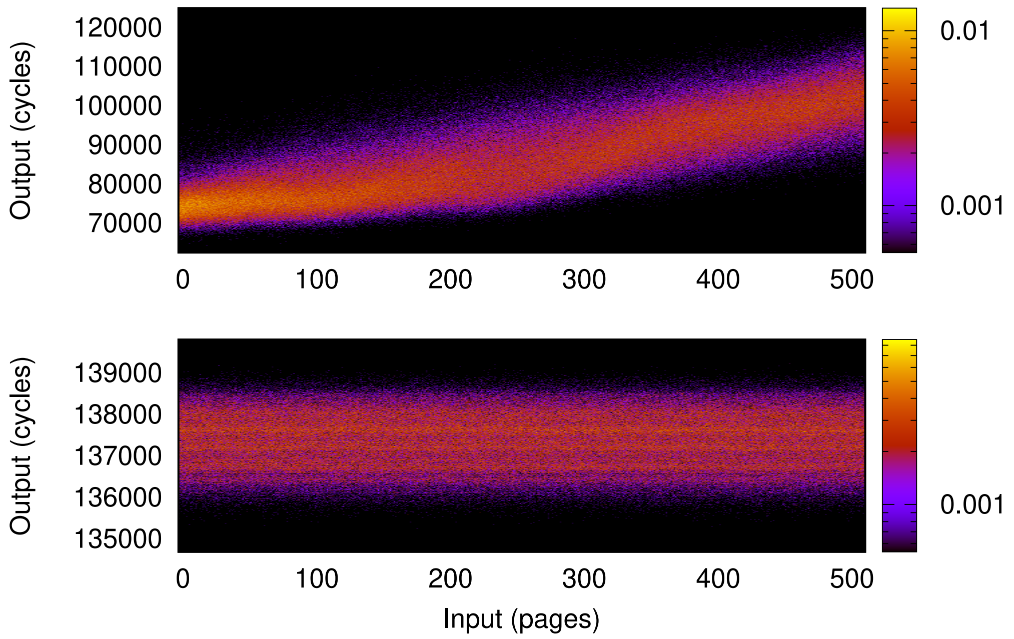


Figure 5.17: Channel matrix for the TLB covert channel on Arm Cortex-A53. Top: without mitigation $\mathscr{M} = 544\,\text{mb}$, $n = 1{,}859$, bottom: maximally mitigated $\mathscr{M} = 1.2\,\text{mb}$, $\mathscr{M}_0 = 1.2\,\text{mb}$, $n = 1{,}857$.

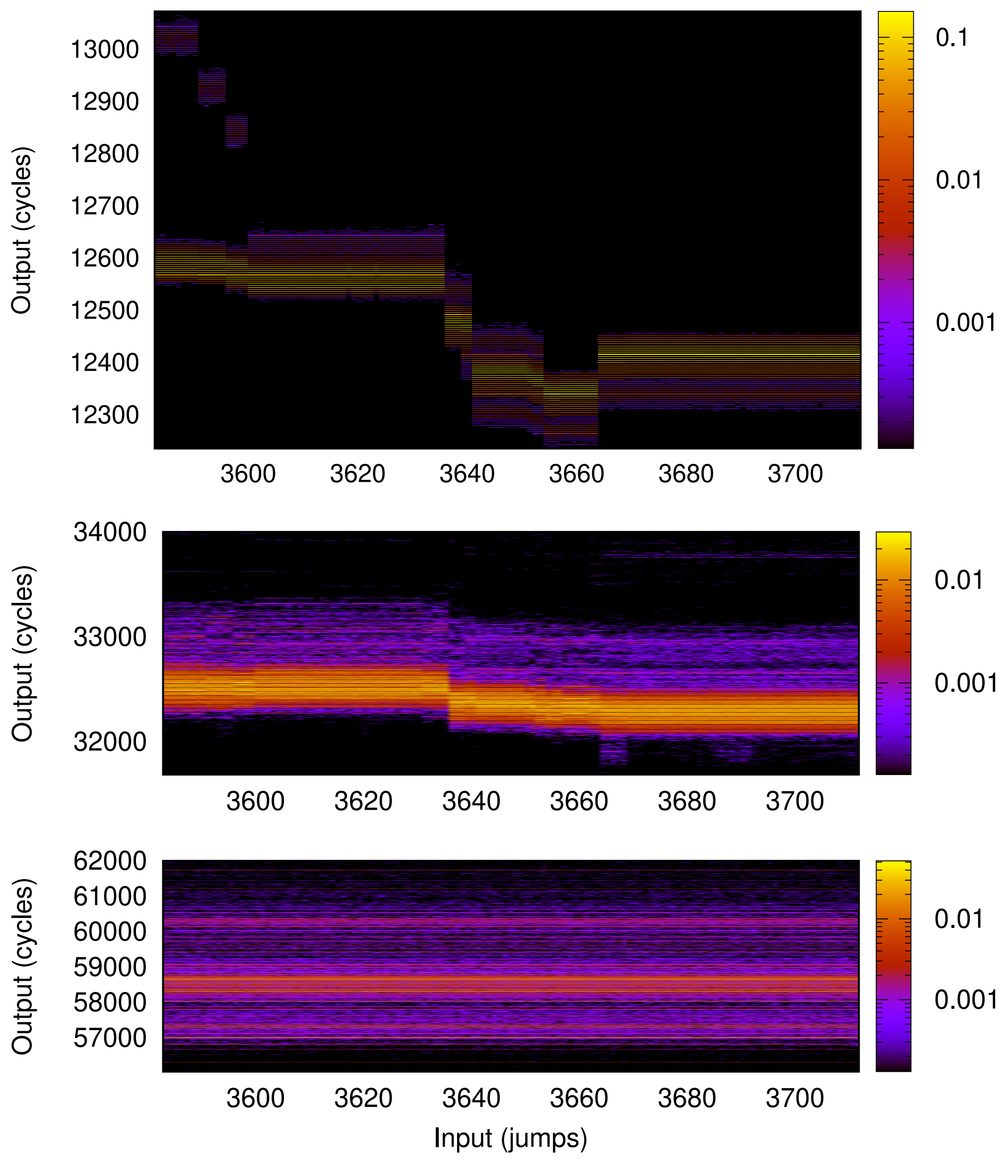


Figure 5.18: Channel matrix for the BTB covert channel on Haswell. Top: without mitigation $\mathscr{M} = 1,553\,\text{mb}$, $n = 7,663$, middle: mitigated without IBC $\mathscr{M} = 307\,\text{mb}$, $\mathscr{M}_0 = 0.7\,\text{mb}$, $n = 7,733$, bottom: maximally mitigated $\mathscr{M} = 0.4\,\text{mb}$, $\mathscr{M}_0 = 0.4\,\text{mb}$, $n = 7,748$.

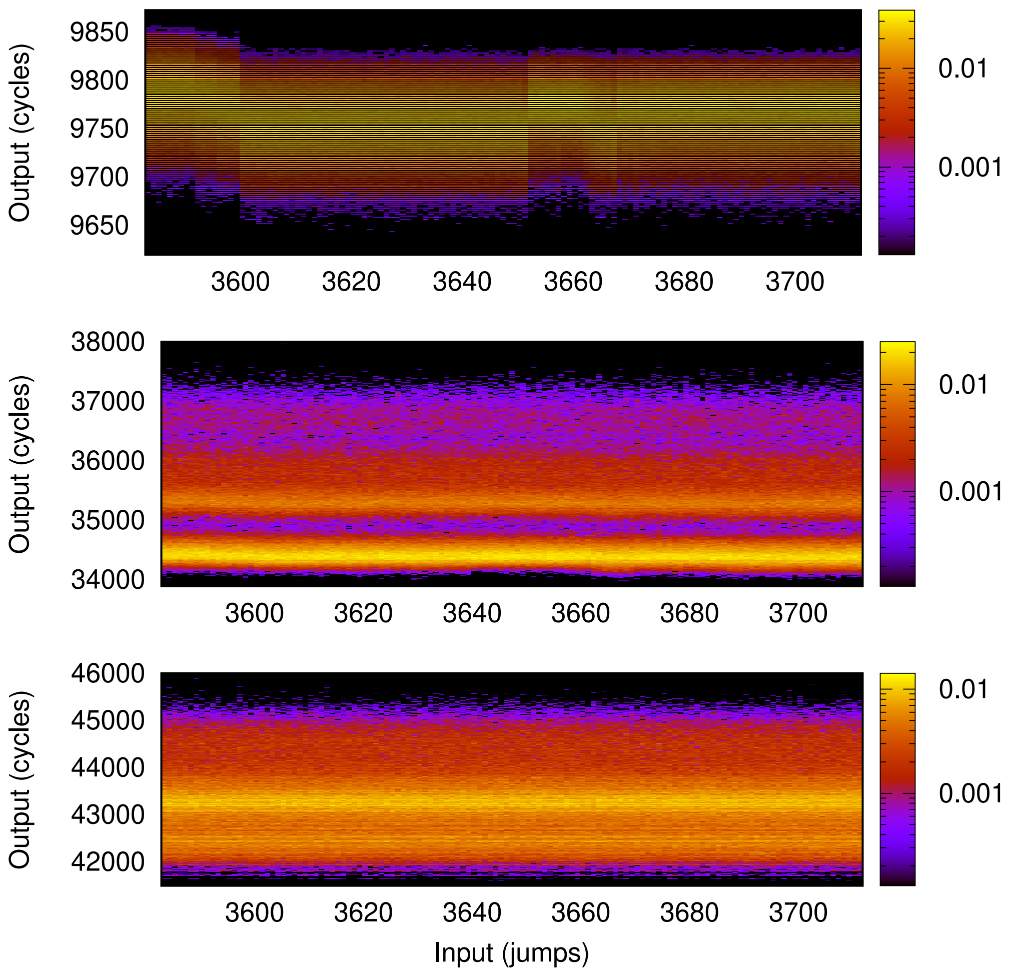


Figure 5.19: Channel matrix for the BTB covert channel on Skylake. Top: without mitigation $\mathscr{M} = 47\,\mathrm{mb}$, $n = 7,727$, middle: mitigated without IBC $\mathscr{M} = 1.3\,\mathrm{mb}$, $\mathscr{M}_0 = 0.9\,\mathrm{mb}$, $n = 7,713$, bottom: maximally mitigated $\mathscr{M} = 0.8\,\mathrm{mb}$, $\mathscr{M}_0 = 0.8\,\mathrm{mb}$, $n = 7,747$.

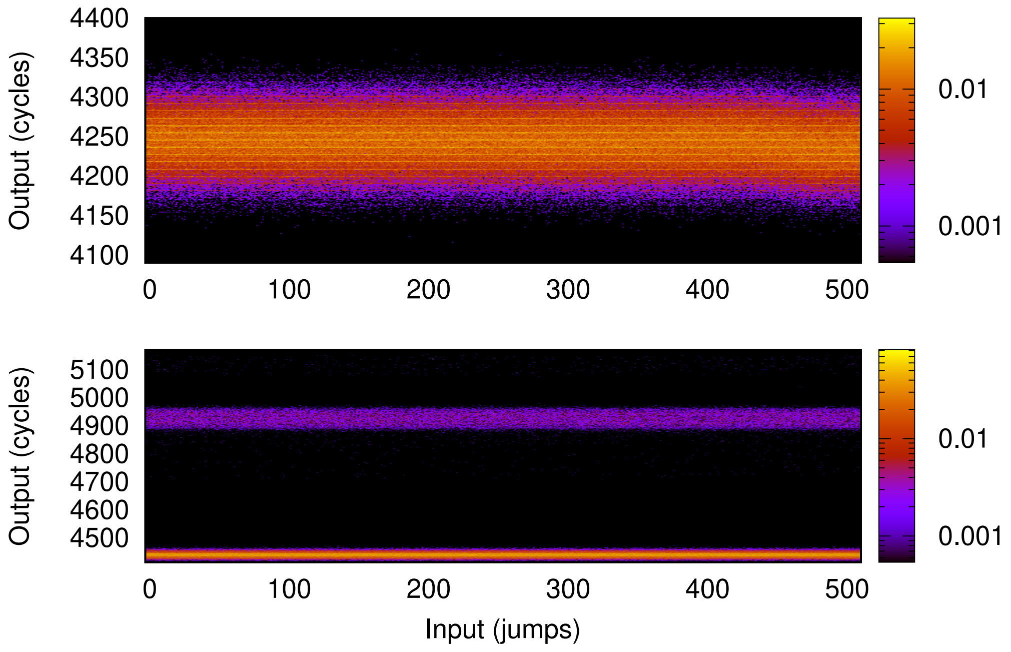


Figure 5.20: Channel matrix for the BTB covert channel on Arm Cortex-A9. Top: without mitigation $\mathscr{M} = 5.4\,\text{mb}$, $n = 1,857$, bottom: maximally mitigated $\mathscr{M} = 2.0\,\text{mb}$, $\mathscr{M}_0 = 2.2\,\text{mb}$, $n = 1,859$.

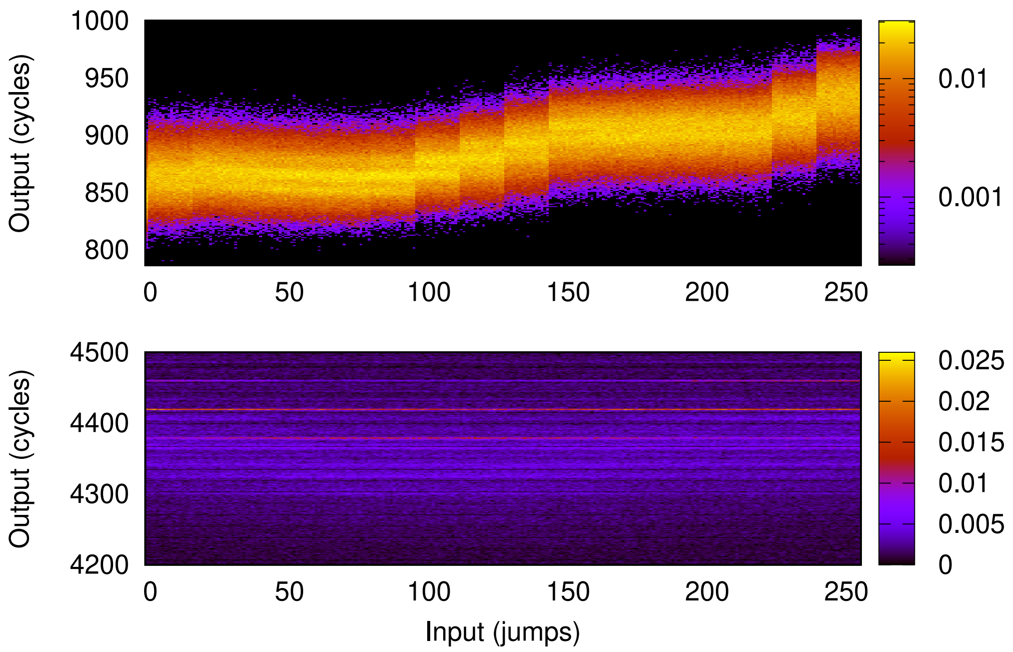


Figure 5.21: Channel matrix for the BTB covert channel on Arm Cortex-A53. Top: without mitigation $\mathscr{M} = 486\,\text{mb}$, $n = 3,771$, bottom: maximally mitigated $\mathscr{M} = 2.3\,\text{mb}$, $\mathscr{M}_0 = 0.7\,\text{mb}$, $n = 3,774$.

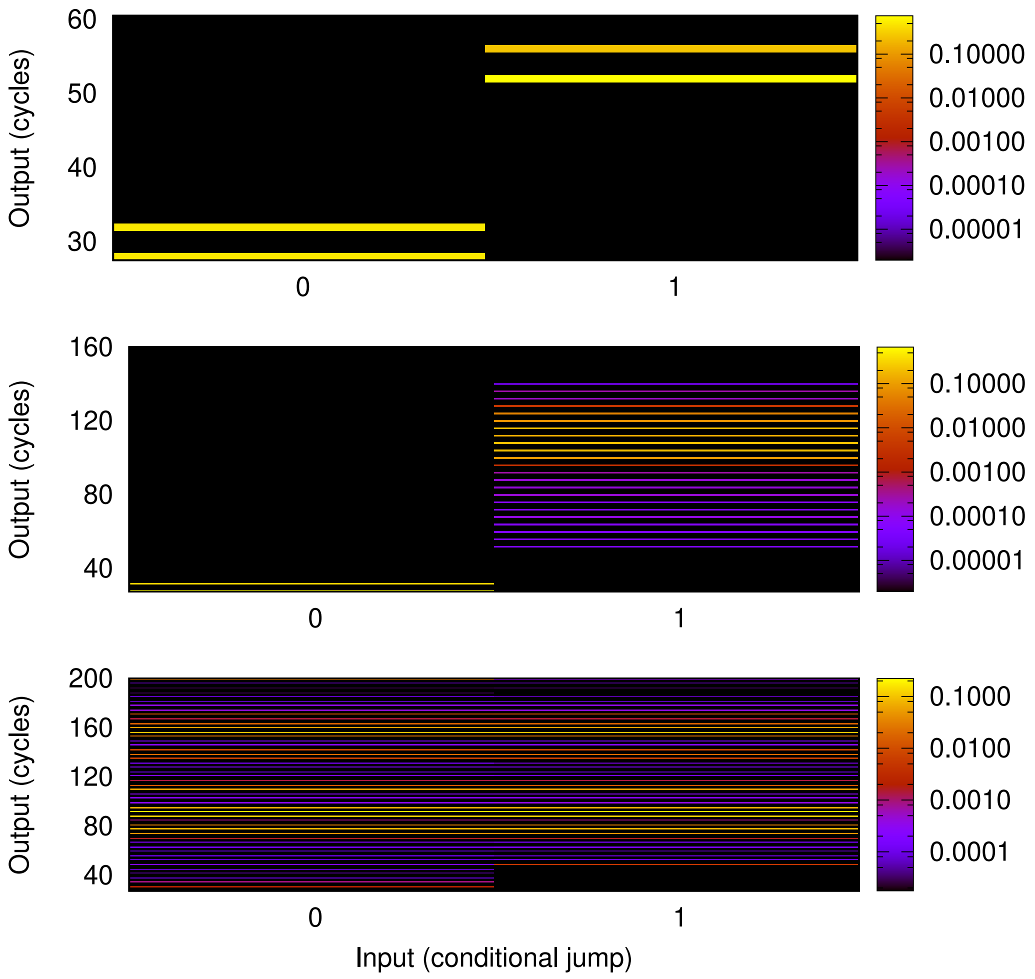


Figure 5.22: Channel matrix for the BHB covert channel on Haswell. Top: without mitigation $\mathscr{M} = 1{,}000\,\text{mb}$, $n = 511{,}765$, middle: mitigated without IBC $\mathscr{M} = 555\,\text{mb}$, $\mathscr{M}_0 = 0.0\,\text{mb}$, $n = 511{,}372$, bottom: maximally mitigated $\mathscr{M} = 0.4\,\text{mb}$, $\mathscr{M}_0 = 0.0\,\text{mb}$, $n = 511{,}852$.

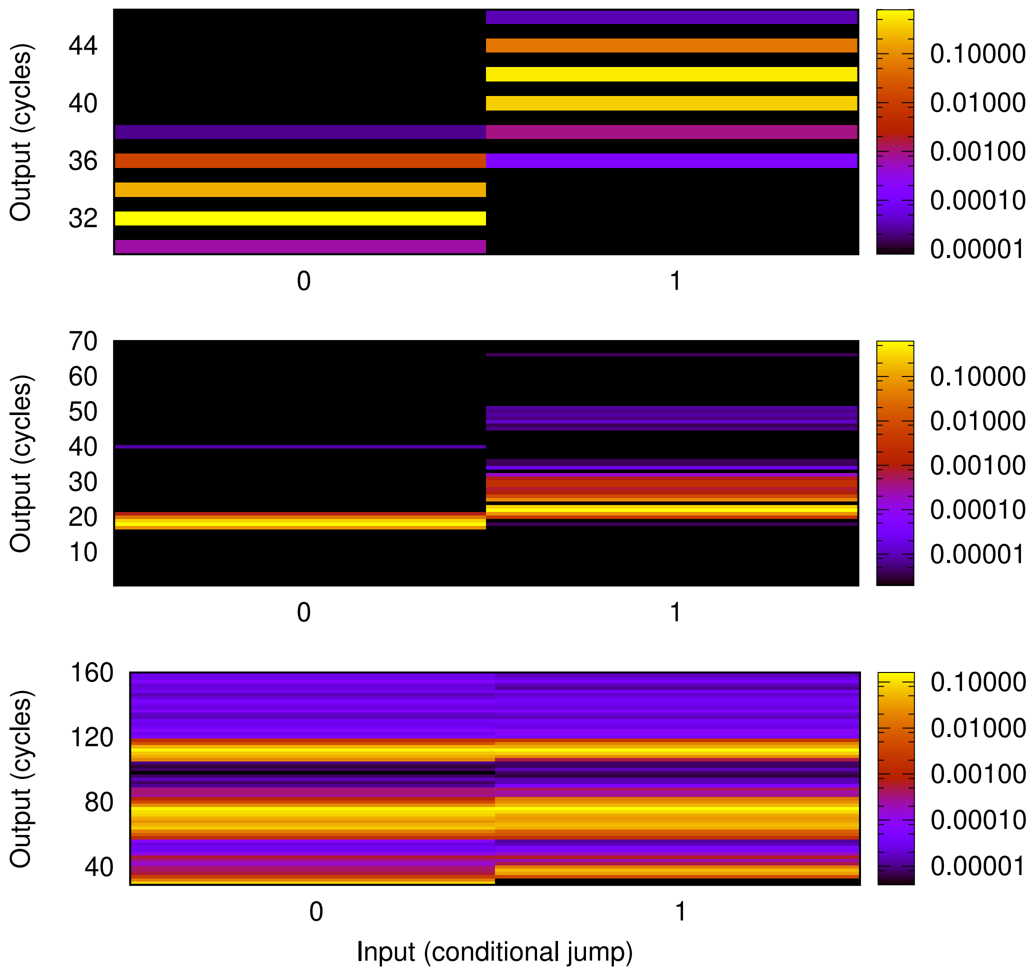


Figure 5.23: Channel matrix for the BHB covert channel on Skylake. Top: without mitigation $\mathscr{M} = 1{,}000\,\mathrm{mb}$, $n = 511{,}487$, middle: mitigated without IBC $\mathscr{M} = 170\,\mathrm{mb}$, $\mathscr{M}_0 = 7.6\,\mathrm{mb}$, $n = 511{,}219$, bottom: maximally mitigated $\mathscr{M} = 0.8\,\mathrm{mb}$, $\mathscr{M}_0 = 0.1\,\mathrm{mb}$, $n = 511{,}535$.

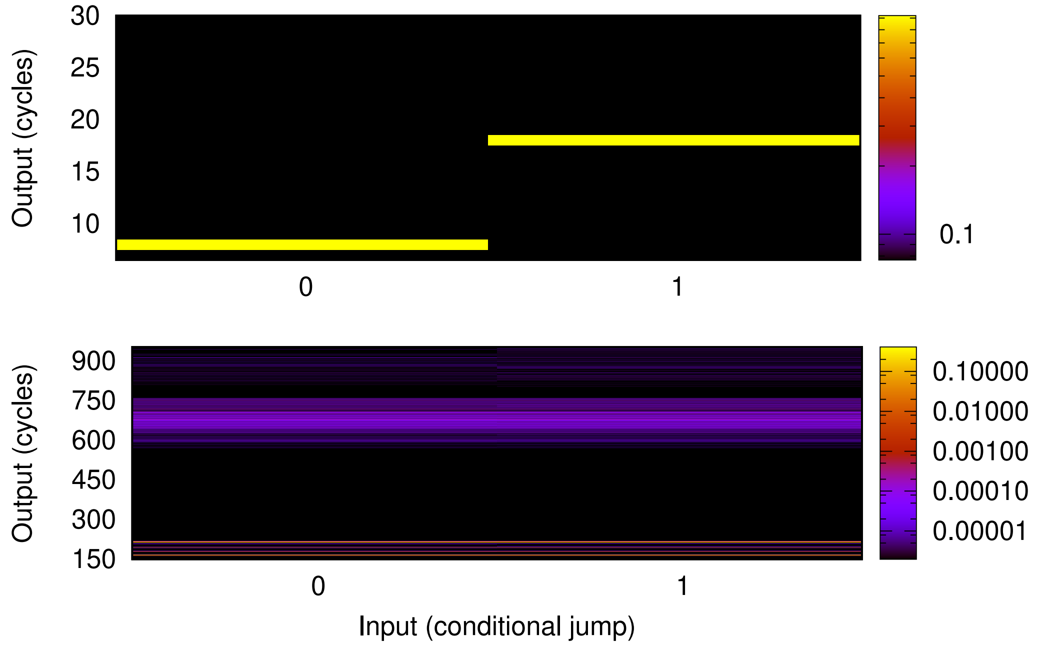


Figure 5.24: Channel matrix for the BHB covert channel on Arm Cortex-A9. Top: without mitigation $\mathscr{M} = 1{,}000\,\text{mb}$, $n = 511{,}434$, bottom: maximally mitigated $\mathscr{M} = 0.0\,\text{mb}$, $\mathscr{M}_0 = 0.3\,\text{mb}$, $n = 511{,}736$.

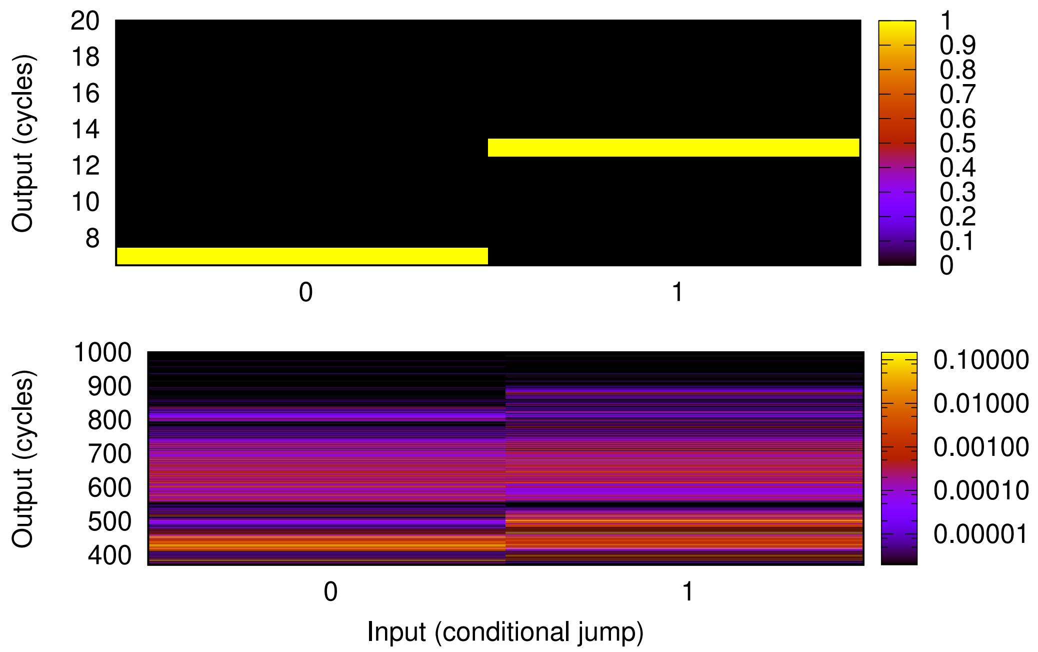


Figure 5.25: Channel matrix for the BHB covert channel on Arm Cortex-A53. Top: without mitigation $\mathscr{M} = 1{,}067\,\text{mb}$, $n = 511{,}872$, bottom: maximally mitigated $\mathscr{M} = 37\,\text{mb}$, $\mathscr{M}_0 = 0.0\,\text{mb}$, $n = 511{,}668$.

6 Time Protection Mechanisms and Their Implementation in seL4

This chapter is the subject of the following paper of which I was the primary author: Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: the missing OS abstraction. In *EuroSys Conference*, Dresden, Germany, March 2019. ACM. The design and implementation of kernel mechanisms were performed primarily by myself, under the supervision of Gernot Heiser. The evaluation of efficacy and performance impact of time protection were designed and performed mainly by myself, with assistance from Yuval Yarom and Tom Chothia, under the supervision of Gernot Heiser. The evaluation of fork and exec system calls on Linux was performed by Peter Chubb.

In Section 4.3, we outlined the targeted system solution that provides time protection in the OS. In this chapter, we examine the requirements for time protection in an OS, present a design in the seL4 microkernel with a proof-of-concept implementation, and evaluate its efficacy for time protection as well as the system performance overhead on both x86 and Arm processors.

6.1 Requirements for Providing Time Protection in the OS

In this section, we define the requirements for providing time protection in the OS. Our goal is to design a system integrating the solution discussed in Section 4.3.

As discussed in Section 4.2.1, resetting execution history by flushing is the only option for preventing timing channels on shared virtually-indexed on-core state as it cannot be partitioned. However, the existing ISA does not satisfy this requirement, as we demonstrate in Section 5.4.3.6. Hence we propose aISA which emphasises this requirement as part of its core property (Property 1, Section 5.5.1). Most importantly, the resetting operation must be performed not only on virtually-indexed caches and cache-like components (e.g., L1 caches or TLBs), but also on other on-core states that are shared. Recently published attacks [Minkin et al., 2019; Schwarz et al., 2019; van Schaik et al., 2019] demonstrated that timing channels are possible on line fill buffers, load ports, and store buffers (Section 3.2.2)—strong evidence that resetting all on-core states is compulsory for eliminating microarchitectural timing channels. Because our work focuses on providing time protection with system mechanisms, we assume that the hardware offers operations for resetting on-core state. These system mechanisms can easily adopt any future improvements to hardware, such as an implementation of the aISA. For this work, we require the OS to flush on-core caches, including L1 caches, the TLB, the BTB, and the BHB on a domain switch (Figure 4.3). Resetting more on-core states, such as the line fill buffer (Section 2.3.4) or prefetcher (Section 2.3.2), can be more expensive, however we leave the evaluation for future work.

Moreover, we assume that there is no concurrent resource sharing between hardware threads by either disabling hardware multithreading or allocating all hardware threads of a core to the same security domain (Section 3.1.1). As a result, timing channels on core-private microarchitectural components are only possible with time-multiplexed sharing, exploring microarchitectural state that contains execution history of previous running domains. Flushing on-core state is necessary to prevent these timing channels.

Requirement 1: Flush on-core state
When time-sharing a core, the OS must set on-core caches to a defined microarchitectural state on domain switch, unless the hardware supports spatially partitioning such state.

The OS can either partition other core-private caches (e.g., the physically-indexed L2 cache on Intel processors) or set those caches to a defined state on domain switch. However, the OS must prevent concurrent sharing among cores by partitioning (Section 4.2.2), in particular flushing the LLC does not help due to the concurrent access (Section 4.1.2). Furthermore, flushing the LLC introduces unacceptably high overhead, as we will show in Section 6.5.1.

Applying cache colouring to partition physically-indexed caches, such as the LLC, prevents memory frames from being shared among security domains, either explicitly or implicitly though page de-duplication [Bugnion et al., 1997; Miłoś et al., 2009], and hence may enlarge the memory footprint of the system. Nonetheless, prohibiting page sharing is necessary, as even sharing of read-only (code) pages has been demonstrated to be a source of side channels [Gullasch et al., 2011; Yarom and Falkner, 2014]. Cloud providers explicitly discourage the practice of cross-VM deduplication because of the risks it presents [VMware Knowledge Base, 2014].

The kernel image itself becomes the only partition shared by all domains while sharing any memory frames among security domains is prohibited. More specifically, sections in the kernel image, including code, data, stack, and heap, are shared. As described in Section 4.3, a shared kernel image can generate cache footprints due to invoking kernel services, such as system calls. Because those cache footprints are related to the security domain which invokes kernel services, the shared kernel image can be exploited as a timing channel in the same way as any shared memory frames (e.g., shared user-level libraries). We will demonstrate the timing channel in Section 6.5.2.3.

Requirement 2: Partition the OS
Each domain must have its private copy of OS text, stack and (as much as possible) global data.

Partitioning most kernel data is straightforward if the memory is managed from user space, such as the memory management model in seL4 as stated in Section 2.6: all dynamically allocated kernel memory is provided by user-level threads. Thus, partitioning all user memory using cache colouring automatically colours all dynamically allocated kernel data structures that are used for managing user-level threads (e.g., TCBs), as previously explained in Section 2.6.3. In contrast, other systems, such as Linux, would require a significant amount of work to achieve a similar level of kernel partitioning, although there is no fundamental barrier to accomplish it. This leaves a very small amount of uncoloured global data that is initialised at boot-up.

Requirement 3: Deterministic data sharing
Access to any remaining shared OS data must be sufficiently deterministic to avoid information leakage.

On-core cache flushing latency depends on the number of cache lines in those caches which can be used as a timing channel, as we will show in Section 6.5.2.2. For example, flushing the L1-D cache invalidates and flushes all used cache lines, which represents the activity of the previously running domain. Hence, the latency depends on the amount of used lines in the cache and equivalently on the execution history.

Requirement 4: Flush deterministically
State flushing must be padded to its worst-case latency.

Uncontrolled interrupts can also be a source of covert timing channels, as they allow a domain working as a Trojan to signal a spy thread in other domains by configuring those interrupts for preempting the spy's execution. These channels are applicable in the confinement scenario (Section 4.1.1), even though the bandwidth of such a channel would be bounded by the limited number of interrupts available on a platform. However, this type of channel is irrelevant to the cloud scenario (Section 4.1.2), because there is no evidence of leveraging interrupts as timing side channels and they are likely infeasible due to the limited data carried by an interrupt. Thus, our work regards interrupts as only a concern for intra-core covert timing channels.

Requirement 5: Partition interrupts
When sharing a core, the OS must disable or partition any interrupts other than the preemption timer.

The methodology of flushing on-core state and padding the corresponding latency are well understood. We now describe a system approach that partitions both kernel image (Requirement 2) and interrupts (Requirement 5), as well as simplify the sharing of remaining kernel data (Requirement 3) as a side effect. Overall, we are looking for a system solution that provides mechanisms without any built-in security policy in the kernel, in order to be suitable for verification purposes in the long run.

6.2 Partitioning the OS: Cloning the Kernel

Partitioning the kernel image (Requirement 2) demands that each partition has a copy of the kernel. A simple approach would be to create a defined number of kernel copies at boot time, so that each partition can run in a separate kernel text segment as done by some non-uniform memory access (NUMA) systems [RedHawk Linux]. In that case, the kernel has to handle the remaining global shared data section carefully, which is required by Requirement 3 (deterministic data sharing among all copies). Reducing the amount of shared global kernel data can simplify the handling process (Requirement 3), by replicating as much as possible among kernel instances, resulting in a multikernel approach similar to Baumann et al. [2009].

The main drawback of this approach is that it results in a completely statically partitioned system, where the security policy is hard-coded into the kernel boot image as the number of partitions. Any change of such a security policy would require a change to the kernel itself, which can reduce the degree of assurance or increase the cost of applying such an approach to real system scenarios. Moreover, the static approach would not suit the cloud scenario (Section 4.1.2): the number of security domains would be fixed, forcing the system to over-provision domains even though most of them would be idle during run time. In terms of verifying such a system, the verification project would need to consider parametrizing parts that are related with domain configurations, avoiding the need to re-verify the initialisation code.

We prefer a system design where the kernel is free from any specific security policy. In such a system, only one kernel configuration exists, and the security policy is defined by a privileged user-level process that allocates system resources (e.g., memory). The managing process can also create and delete domains dynamically at run time, in the same way as the process creation and deletion.

Our design introduces a policy-free *kernel clone* mechanism, which creates a copy of the kernel image in user-supplied memory frames. Each cloned kernel contains a replica of the kernel code and read-only data sections, as well as a dedicated kernel stack. The only section that cannot be cloned is the global data that is created at system initialization. Using this mechanism, the initial user process can partition the system to the extreme: partitioning user-level memory, kernel data, and even the kernel image, leaving a minimum amount of global data as shared (Figure 6.1).

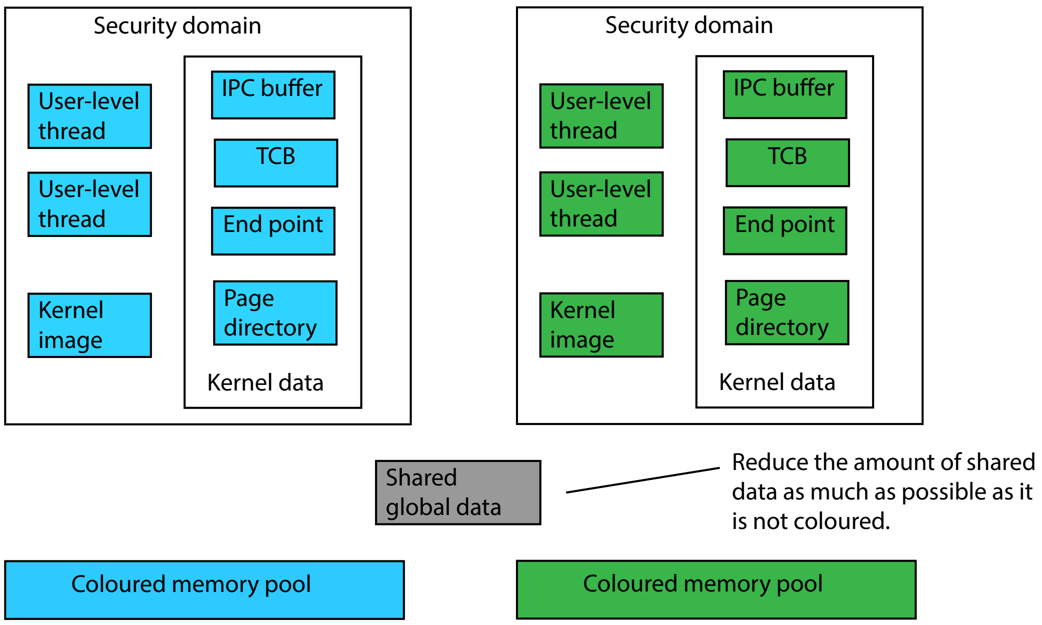


Figure 6.1: Coloured security domain.

To create such a partitioned system, the initial process can work as both resource and security manager that first groups all free memory into coloured pools, one per partition, then clones a kernel image into each coloured pool, and eventually starting child processes that are supported by the cloned kernel image from the same partition (i.e., coloured pool). During system runtime, the initial process can choose to clone more kernel images or delete any valid kernel images, depending on the runtime needs on scaling up or down the number of domains. An extreme use case would be that the initial process deletes itself, resulting in a completely and permanently partitioned system.

The existing memory management mechanisms in seL4 are sufficient to guarantee that the partitioned system will remain coloured for its lifetime. Additionally, cloning can also be undone as long as a process which has authority over a kernel image remains runnable. A partition can further divide itself into sub-partitions, each with new kernel clones, as long as it satisfies the requirements of managing coloured memory pools. Lastly, the kernel provides mechanisms to revoke a partition, for both cloned kernel images and other kernel data objects (e.g., user-level page table, or TCB). The only requirement for a thread to revoke a kernel object is to provide capabilities for the corresponding kernel images or objects, which are normally possessed by the initial thread.

6.3 Partitioning Interrupts

To control interrupts (Requirement 5), interrupts are associated with kernel images, and are only enabled while the corresponding kernel is running. As shown in Figure 6.2, cloned kernel images in each security domains have dedicated IRQ sources. The existing seL4 offers interrupt services through IRQ handling capabilities (IRQHandler) which allow user-level processes to subscribe and acknowledge any interrupt notifications (Section 2.6). We extend this model by associating interrupt capabilities with kernel images. As a result, each partition has dedicated IRQ sources, user-level threads, and a cloned kernel image.

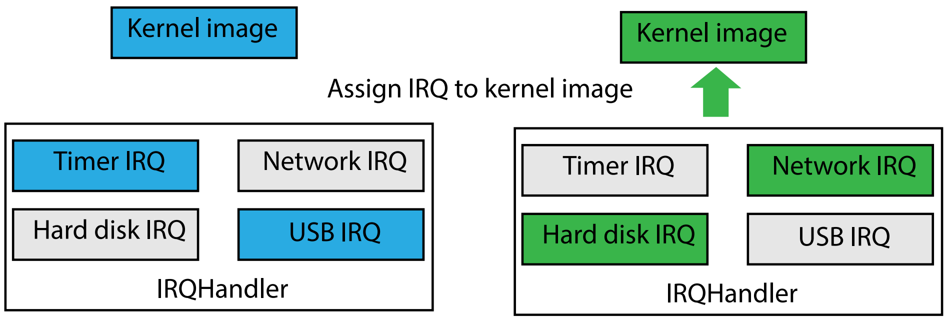


Figure 6.2: Partitioning IRQs sources by assigning IRQHandler capabilities to kernel images.

6.4 Implementation in seL4

We have implemented our design on both x86 and Arm systems, including multi-core support for all kernel mechanisms.

6.4.1 Kernel clone overview

The seL4 microkernel is a capability-based system (Section 2.6.2), where all kernel objects are referenced by capabilities, including the TCB, the VSpace (virtual address space of a user-level thread), or endpoints for IPC messaging. Capabilities represent objects as well as the authorisation to perform system operations on those objects. For example, an Endpoint capability with the read-only right only allows a thread to receive IPC messages but not send them. Moreover, a thread invokes system services by providing capabilities to system calls, such as calling `seL4_Send` on an Endpoint capability for sending an IPC message. Lastly, the kernel provides all unused memory to the initial thread as Untyped objects that are later retyped into other kernel objects, in order to create a system that suits the use case (e.g., a confinement system).

6.4.1.1 Representing a kernel image

To control the cloning mechanism, we introduce a new object type, KernelImage (Listing 6.1), to represent a kernel image. A holder of a capability with `clone` rights to a KernelImage object can clone the kernel image, if the holder has access to a sufficient amount of Untyped memory. Moreover, the cloned kernel can be destroyed in the same way as any other objects in the system, and revoking a KernelImage capability destroys all the other copies of that capability. The cloned kernel is independent of the kernel it is cloned from (the template kernel), and the template kernel can be destroyed without destroying the cloned kernel. The KernelImage object is similar to the existing PageDirectory object that represents the top-level page directory in the virtual address space used by a user-level thread (Section 2.6.2). The main difference is that the KernelImage object represents the top-level page directory as the virtual address space used by a kernel image.

```
block kernel_image_cap {
    field      capKIMappedASID   12    /*Assigned ASID*/
    field      capKIClone        1     /*The right for cloning*/
    field      capKIIsMapped     1     /*Mapped with an ASID*/
    padding                      50

    field      capType           6     /*Type: KernelImage*/
    field_high capKIBasePtr      48    /*Base address*/
    padding                      10
}
```

Listing 6.1: The format of KernelImage capability on the 64 bit x86 architecture.

6.4.1.2 Representing memory used by a kernel image

We introduce a second new object type, KernelMemory (Listing 6.2), to manage the memory used for cloning a kernel image. Having a separate object for kernel memory is necessary because the coloured memory is generally not contiguous (discussed in Section 2.4). The size of a kernel image, for example 300 KiB on a 4-core x86 machine, is lager than the size of a coloured frame, 4 KiB. Hence the system requires the cloning process to manage them separately: the process must create coloured frames used as kernel memory before it can clone a kernel image from those coloured frames. The KernelMemory object is similar to the existing Frame object that represents memory mapped into a user-level address space. In other words, KernelMemory represents memory mapped into the kernel-level address space, whereas Frame is used only to create user-level address space. The only difference between KernelMemory and Frame is that KernelMemory cannot be mapped as a shared frame between VSpace, a feature that is commonly used for creating shared buffers between processes.

```
block kernel_mem_cap {
    field      capKMMappedASID    12    /*The ASID of KernelImage*/
    field_high capKMMappedAddress 48    /*Address mapped to kernel window*/
    padding                        4

    field      capType             6    /*Type: KernelMemory*/
    field      capKMIsMapped       1    /*Mapped into a kernel image*/
    field_high capKMBasePtr       48    /*Base address*/
    padding                        9
}
```

Listing 6.2: The format of KernelMemory capability on the 64 bit x86 architecture.

6.4.1.3 Creating a partitioned system

For supporting kernel clone, seL4 introduces a new procedure at boot time: seL4 creates a master KernelImage capability with the `clone` right which references the initial kernel image. Once the boot stage is finished, the kernel passes the master capability to the initial user thread, as well as the size of the kernel image.

In each TCB, we add a new field, a copy of the KernelImage capability, that represents the kernel serving that thread. The creator of a TCB can assign a specific KernelImage to the thread with the existing `seL4_TCB_Configure` system call.

To partition the system, the initial thread first groups Untyped memory into coloured pools (Section 2.6.3). Then, it creates objects used by a coloured domain from the corresponding pool: retyping kernel objects from coloured Untyped memory, including creating the KernelImage, the KernelMemory, user-level threads, and all kernel objects used by those threads (e.g., endpoints for IPC messaging). Along with coloured KernelMemory objects, the thread clones a coloured kernel image which is represented by the KernelImage. Once kernel images are initialised, the initial thread associates user-level threads with the corresponding kernel image in the same pool (i.e., partition). At last, the initial thread makes all threads runnable, whereupon all threads are hosted by their respective kernel images.

For restricting the future cloning, the initial thread can choose to create copies of the master KernelImage capabilities, but without the `clone` right, preventing unauthorised cloning. As a result, any holder of the stripped KernelImage capabilities cannot clone new kernel images.

6.4.2 The process of cloning a kernel image

The process for cloning a kernel has three steps, as demonstrated in Figure 6.3. First, a user thread retypes Untyped memory into an uninitialised KernelImage and a sufficient amount of KernelMemory. The user-level thread can choose either to select the Untyped memory from a coloured memory pool (Section 2.6.3) or from a unified memory pool that is not aware of cache colours. The kernel is not aware of any colour contained in the Untyped memory, because cache colouring is a memory management policy implemented at user-level.

Secondly, the user thread allocates an ASID to the KernelImage with the ASID assigning method offered by the ASIDPool object, the object that represents the right to allocate an ASID to a virtual address space (Section 2.6.2). The user thread conducts the ASID assignment in the same way as allocating an ASID to a user-level page directory. By using the ASID, the hardware can identify the virtual-to-physical address mappings used for different kernel images (Section 2.2), which is also a requirement of both x86 and Arm processors.

Thirdly, the user thread invokes the `seL4_KernelImage_Clone` system call on the uninitialised KernelImage, passing its own KernelImage with `clone` right and KernelMemory capabilities as parameters, triggering the initialisation process on the KernelImage. The KernelImage with `clone` right can be the master capability (Section 6.4.1.3) owned by the initial thread, as well as a copy of the master capability that is distributed by the initial thread. After the cloning, the user thread obtains a cloned KernelImage, which is identical to the initial kernel image.

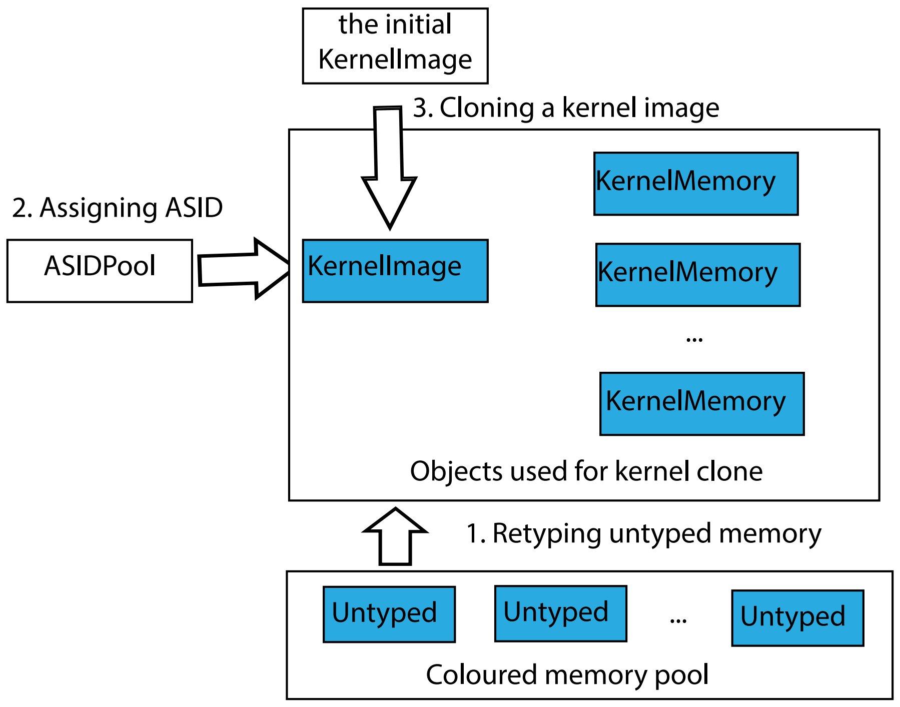


Figure 6.3: An overview of the kernel clone mechanism.

All cloned kernel images have the same virtual address layout (virtual pages) as the initial kernel image, as well as the contents stored in those virtual pages. However, virtual pages from different kernel images are mapped to different physical frames. We call the layout of the virtual address space used by a kernel image the *kernel window*, and the mapping of the address space the *kernel window mapping*. Cloning is the process of creating

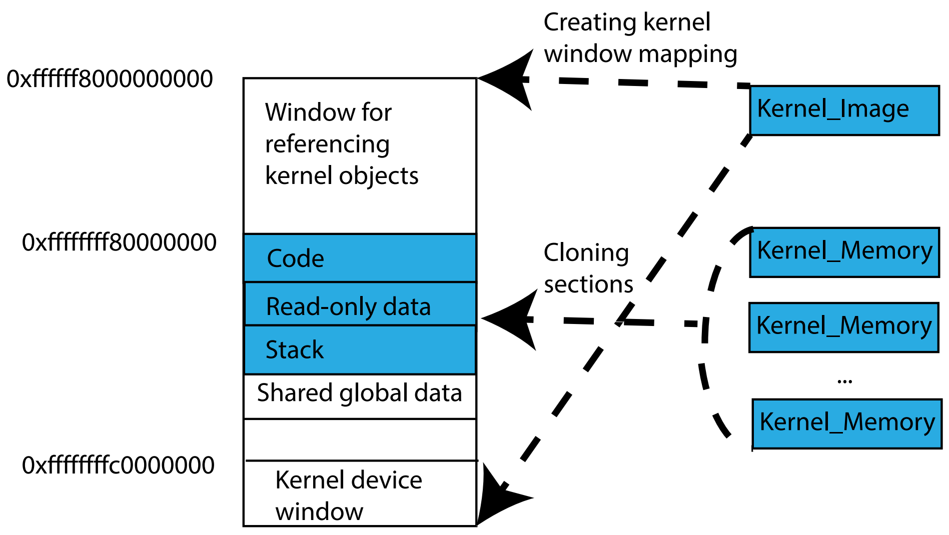


Figure 6.4: Cloning a kernel window mapping on x86.

the kernel window mapping of a kernel image using KernelImage and KernelMemory objects.

As mentioned in Section 6.4.1.1, the mapping for a kernel window is represented by KernelImage, the root of the kernel's address space that is equipped with an ASID. Cloning creates a valid mapping by populating the kernel window (i.e., KernelImage) with frames that are represented by KernelMemory objects.

Figure 6.4 demonstrates the cloning process implemented for x86 processors. On x86, the kernel window contains sections for referencing kernel objects (e.g., TCB objects) that are manged at user-level (Section 2.6.2); for executing a kernel image (code, ready-only data, stack, and shared global data); and for accessing hardware devices (e.g., interrupt controllers). Among those sections, only sections used for executing a kernel image are duplicated, except the shared global data section which is unchanged. The mapping of other non-kernel sections are unchanged. In other words, cloning creates a copy of the kernel's code, read-only data, and stack sections with memory referenced by KernelMemory objects, and populates the mapping in those sections accordingly.

As a result, each kernel image can execute on its own kernel window mapping for handling system calls, receiving interrupts, conducting security domain switches according to the system timer ticks, or staying idle when there is no thread can be scheduled on a core. The cloning process does not require any changes on the implementation for a multicore system, but inherits the existing multicore implementation from seL4 (Section 2.6).

6.4.3 Managing a cloned kernel image

Each cloned kernel image contains a metadata section for recording resources that belong to this kernel, including ASID, IRQs, kernel stacks, and cores for the current and past execution history. We use low addresses in the kernel window to store the metadata, as the kernel window mapping does not use those addresses. This design efficiently utilises areas in the KernelImage object that are not used for the kernel window mapping.

Figure 6.5 shows the layout of the metadata. This includes: the validation flag, the ASID and IRQs assigned to this kernel image, and addresses for kernel stacks. To support multicore systems, the metadata contains a bitmap representing cores that are currently executing this kernel image, and a bitmap for cores that have executed this kernel image.

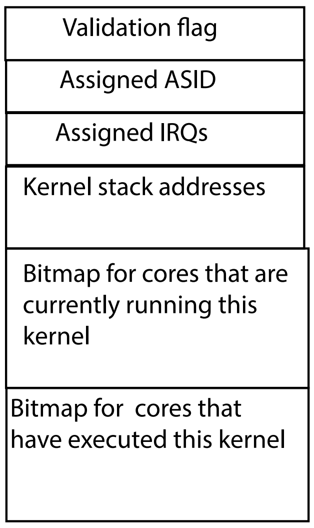

Validation flag
Assigned ASID
Assigned IRQs
Kernel stack addresses
Bitmap for cores that are currently running this kernel
Bitmap for cores that have executed this kernel

Figure 6.5: The structure of the kernel metadata.

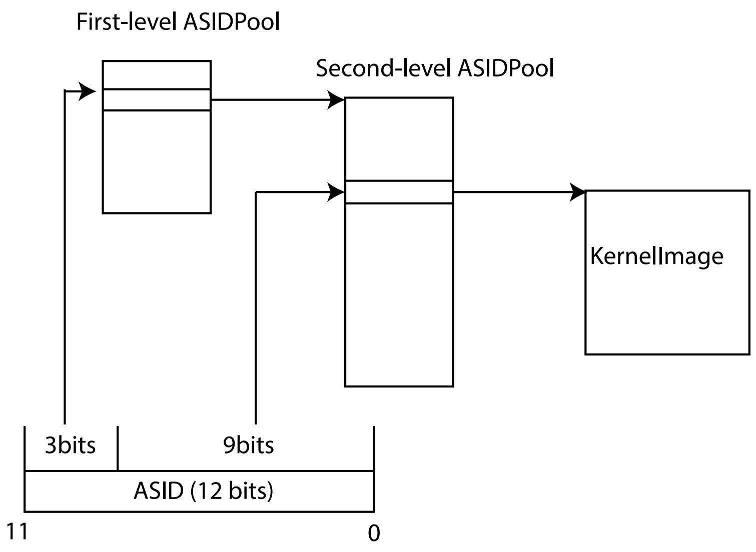


Figure 6.6: Addressing a kernel image with an ASID on x86.

The kernel stores the ASID assigned to a kernel image in the `capKIMappedASID` field of the KernelImage capability (Listing 6.1), as well as the `capKMMappedASID` field of all KernelMemory capabilities used by that kernel image (Listing 6.2). In other words, the

ASID sufficiently links all the objects belonging to a kernel image together, which is extremely useful for KernelMemory deletion (Section 6.4.7).

By reading the ASID stored in a KernelMemory capability, the kernel can locate its KernelImage through the existing ASID indexing process in seL4. As mentioned in Section 2.6.2, seL4 uses a two-level ASID mapping: the first-level ASIDPool is created at boot, and the second-level ASIDPool is managed by user-level threads. Figure 6.6 demonstrates the process of locating a KernelImage from its ASID on x86. The kernel first uses the upper three bits in the ASID to index the first-level ASIDPool, locating the second-level ASIDPool. Then, the kernel indexes the second-level pool with the lower nine bits, locating the KernelImage. Once the KernelImage is located, the kernel can mark the kernel image as invalid before deleting any KernelMemory from this image, hence avoiding any race conditions during the deletion process.

6.4.4 Audit of the shared global data section

Cloned kernel images only share the minimum amount of global data required for maintaining the correctness of the system, such as the scheduling queue, the IRQ states, or the current thread running on a core. The shared global data section is created at compile time as the static data section in a kernel image.

The cloned kernel images share only the following structures in the static global data section (numbers indicate size per core on x86, total of about 9.5 KiB):

1. the scheduler's array of head pointers to per-priority ready queues (4 KiB), as well as the bitmap used to find the highest-priority thread in constant time (32 B);
2. the current scheduling decision (8 B);
3. the tables of IRQ state and corresponding interrupt handlers (2×1.1 KiB);
4. the interrupt currently being handled, if any (8 B);
5. the first-level hardware ASID table (1.1 KiB);
6. the IO port control table (2 KiB, x86 only);
7. the pointers for the current thread, its capability space (Cspace), the current kernel, idle thread, and the thread currently owning the floating point unit (40 B);
8. the kernel lock for SMP (8 B); and
9. the barrier used for inter-processor interrupts (8 B).

To ensure that there are no possible cross-core side channels, we conduct an audit of the shared data on both x86 and Arm platforms. Specifically, we list all system circumstances under which the kernel will access those data structures, such as interrupt handling, context switching, or scheduling a runnable thread. We then conclude that none of those cache lines involved contain or are indexed by private user information (such as address-space layout). It is worth to mention that this conclusion will need proving when the kernel will be verified.

6.4.5 Partitioning interrupts

As mentioned in Section 6.2, we assign interrupt sources to KernelImages to partition interrupts (Requirement 5). In seL4, all interrupts except the system tick (i.e., the kernel's preemption timer) are controlled by IRQHandler capabilities, which allows a user-level thread to configure its corresponding interrupt sources. We introduce a `seL4_KernelImage_-SetInt` system call for associating an interrupt source, represented by the IRQHandler capability (Section 2.6.2), with a kernel image, represented by the KernelImage capability.

While executing, a kernel image can receive interrupts only from the preemption timer and associated interrupt sources; all the other interrupts are masked. Therefore, it is impossible for kernels (i.e., security domains) to trigger interrupts across partition boundaries if all interrupts are partitioned.

In the same way as the kernel clone mechanism, the system does not enforce any partitioning on interrupt sources, leaving the configuration of security policy as a user-level task. Rather than assigning dedicated interrupts to kernel images, a privileged user-level thread can choose to assign an interrupt to a group of kernel images, resulting in the interrupt being handled by any executing kernel in the group while triggered. In this case, the kernels in this group are not entirely isolated, as they are sharing an interrupt source. Furthermore, an interrupt cannot be handled if it is not associated with any kernel images. Lastly, the user-level device manager needs to ensure that all devices are correctly associated with corresponding security domains, to ensure that one domain cannot trigger interrupts that belong to another domain through accessing any hardware devices.

6.4.6 Domain-switch actions

The running kernel is mostly unaware of partitioning. As mentioned in Section 6.4.2, all valid kernel images are laid out identically in the kernel window (virtual pages), as well as the contents stored in those pages. Only the mapping for the kernel window is different for each kernel image. Hence, a kernel switch only requires switching between different kernel window mappings. Assuming that each domain is supported by its own kernel image, switching domains then becomes switching kernel window mappings. Also, a domain switch involves actions to satisfy requirement listed in Section 6.1, including flushing on-core state (Requirement 1), deterministic data sharing (Requirement 3), flushing deterministically (Requirement 4), and partitioning interrupts (Requirement 5).

6.4.6.1 Switching kernel window mapping

To perform a kernel switch (i.e., domain switch), the kernel switches the root page-directory pointer of the user-level address space as part of the normal context switch. Switching the page-directory pointer implicitly switches all the sections that are cloned (duplicated) for the next running domain. The mapping contains kernel code, read-only data, and stack sections that are mapped at fixed addresses in the kernel window from its own coloured memory (the KernelMemory used by this kernel image). This process is the same on both x86 and Arm processors.

The only explicit action needed for a kernel switch is preparing the kernel stack by copying the present stack to the new one before switching the page directory.

```
 1        asm volatile (
 2            "movq %%rsp, %%rax\n"
 3             /*copying from the bottom of the current stack*/
 4            "subq %1, %%rax\n"
 5            "1: \n"
 6            "movq (%%rax, %1, 1), %%rbx\n"
 7            "movq %%rbx, (%%rax, %0, 1)\n"
 8            "addq $8, %%rax\n"
 9            "cmpq $4096, %%rax\n"
10             /*copying until the top of the stack*/
11            "jl  1b\n"
12            "mfence\n"
13             /*updating the page directory pointer*/
14            "movq %2, %%cr3\n"
15             : "+r"(kernel_vaddr)
16             : "r" (esp_bottom), "r"(cr3)
17             : "rax", "rbx", "rsp", "memory"
18         );
```

Listing 6.3: Switching kernel image on x86.

Listing 6.3 lists the code used for kernel switches on x86: a sequence of stack copying followed by a page directory switch. The switching code first calculates the offset of the current stack pointer, by comparing the `rsp` with `esp_bottom`, which contains a reference to the bottom of the current stack (lines 2–3). Then, it copies the contents of the current stack to the corresponding point in the new stack (`kernel_vaddr`), one word a time, until the top of the stack (lines 4–9). Lastly, the kernel switch finishes by updating the `cr3` register, which refers the root of the page directory on x86. The kernel switch also automatically switches the idle thread, as the idle thread is also contained in the kernel window mapping as part of the code section.

The kernel detects the need for a stack switch (i.e., domain switch) by comparing the current KernelImage with the kernel image used by the destination thread through referencing the KernelImage capability stored in its TCB. In a properly partitioned system, the domain switch only occurs on a system tick, due to the fact that a domain cannot be preempted by interrupts owned by other domains (Section 6.4.5).

6.4.6.2 Flushing on-core state

To satisfy Requirement 1, the kernel flushes all on-core microarchitectural state after a domain switch but before resuming the destination thread in the next running domain.

Furthermore, the kernel releases the lock used by the multicore version of seL4 before flushing the state. As mentioned in Section 2.6.1, the current seL4 implementation uses a big lock which will be acquired before conducting any kernel services and released after the scheduling decision has been made. We release the kernel lock before flushing the on-core state, as flushing is not related with the consistency of any kernel data.

We summarise the operations for flushing on-core state in Table 6.1. To flush the on-core state on Arm (Sabre platform in Table 6.2), we flush the L1 caches, TLBs, and states contained in the BPU (BTB and BHB). On x86 (Haswell platform in Table 6.2), we flush the TLB and reset the BHB state with the IBC mechanism introduced recently for mitigating the Spectre attack [Intel, 2018c]. However, flushing the L1 caches is a challenging task on x86 as the hardware manufacturer only supports flushing the complete cache hierarchy (i.e., three levels of caches) but not selectively flushing any of the L1 caches. Hence, we implement a "manual" flush by sequentially traversing a cache-sized buffer. For flushing the 32 KiB L1-D cache, the kernel performs read operations on a 32 KiB buffer, one per cache line (64 B). Recently, Intel added support for flushing the L1-D cache [Int, 2018b], as a mitigation for the Foreshadow timing channel attack [Van Bulck et al., 2018]. Unfortunately, we cannot use this feature as a microcode update is yet to be available for our testing machine. Furthermore, Intel does not support L1-I flushing at the current stage.

Similarly, the kernel executes jump operations stored in a 32 KiB buffer for flushing the 32 KiB L1-I cache, one per instruction (8 B). In total, the flushing code executes 4,096 jump instructions. Jumping through these many instructions also flushes the BTB which stores the destination addresses of those jumps. However, the size of the BTB is not published by the manufacturer, jumping with a stride of 8 B is the best effort approach to conduct the manual flush on a 32 KiB buffer.

	Executing	
Flushing	**x86 (Haswell)**	**Arm (Sabre)**
L1-D	512 `mov` instructions on a 32 KiB buffer	`DCCISW`
L1-I	4,096 `jmp` instructions on a 32 KiB buffer	`ICIALLU`
TLBs	`invpcid`	`TLBIALL`
BHB	IBC	`BPIALL`
BTB	implicitly done by the L1-I cache flush	`BPIALL`

Table 6.1: Operations for flushing on-core states.

Obviously, our "manual" flush on x86 assumes that L1 caches replace cache lines according to the LRU or the first in, first out (FIFO) replacement policy. The "manual" flush can be potentially incomplete on future platforms or even on the current platforms, due to the lack of documentation of replacement policy used on those caches. However, the reliability of flushing operations could be increased if the hardware manufacturer either gives more exposure to corresponding hardware components or introduces new features for selectively flushing those caches. The hardware manufacturer should consider providing sufficient support on flushing on-core state.

6.4.6.3 Flushing deterministically

To ensure that domain switch has a deterministic latency (Requirement 4), the kernel can be configured with a domain tick length that is the sum of the latency of conducting a domain switch and the length of a domain time slice. When switching domains, the kernel defers returning to the user mode (i.e., resuming the next running domain) until the configured length has elapsed since the timer got reprogrammed at the previous switch.

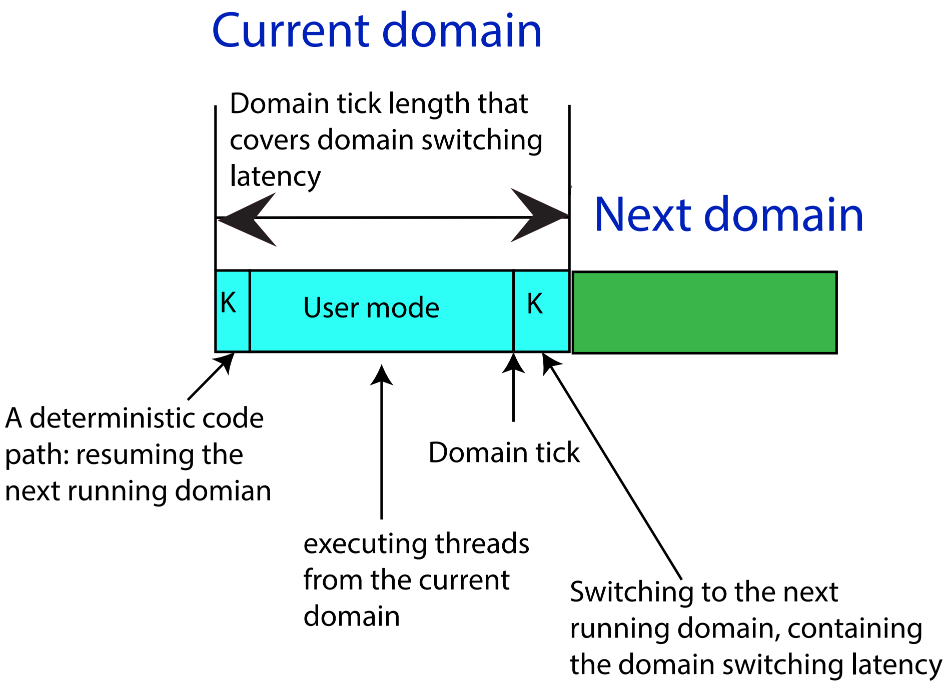

Current domain
Domain tick length that covers domain switching latency
Next domain
K
User mode
K
A deterministic code path: resuming the next running domian
Domain tick
executing threads from the current domain
Switching to the next running domain, containing the domain switching latency

Figure 6.7: The configured domain tick length, K represents kernel mode.

Figure 6.7 shows the mechanism of this configuration: the configured domain tick length covers the latency of resuming user-level thread, the length of a domain tick, and the latency of conducting a domain switch, including flushing any on-core state.

This mechanism allows the kernel to hide any timing variations from the moment a domain is entered until the next domain is scheduled. Possible timing variations include a delayed scheduling decision due to a postponed domain tick interrupt delivered by hardware, or the cost of flushing on-core microarchitectural states. A domain tick interrupt can be potentially delayed due to race conditions caused by other exceptions, such as, a system call exception or other interrupts.

6.4.6.4 Deterministic data sharing

The cloning mechanism simplifies the implementation of Requirement 3 by minimising the amount of shared data as much as possible. Our design achieves determinism by carefully prefetching the shared data section listed in Section 6.4.4 when executing a domain switch, loading every cache line in this section.

The kernel groups the shared data in a section, and identifies the start and end cache line in this section by the global symbol inserted at compile time. We use the layout of the kernel image compiled for x86 as an example (Listing 6.4). The kernel image contains four sections: `text` for the code section, `rodata` for the read-only data section, `data` for the section that only contains private data for this kernel, such as the kernel stack, and `bss` for the data section that is initialised during the boot up stage. In the `bss`, there is a subsection used for the idle thread, which is also kernel private. Hence, the shared global section that requires prefetching starting from the symbol `ki_share_start`, and ending at the symbol `ki_share_end`. The prefetching implementation strides on this section, on each cache line, loading the section into the on-core cache. As a result, the kernel's usage is deterministic for this section before returning to the next running domain.

```
.text . :
{
    *(.text)        /*code section*/
}
 .rodata . :
{
    *(.rodata)       /*read-only data section*/
}
 .data . :
{
    *(.data)      /*private data section, including kernel stack*/
}
.bss . :
{
    ki_idle_start = .;
    *(.bss.idle)  /*data section used for idle thread*/
    ki_idle_end = .;
     /*align to a frame size (4KiB), suitable for cloning*/
    . = ALIGN(4K);

    ki_share_start = .;
    *(.bss)        /*the global shared data section*/
    ki_share_end = .;
}
```

Listing 6.4: The layout of the kernel image on x86.

All the prefetching is done prior to padding of the domain-switching latency. Prefetching the shared data section also prevents the latency of restoring the next running domain from depending on the previous domain's execution history, as the kernel image and stack are already switched and the exit code path is deterministic.

6.4.6.5 Switching interrupts

As described in Section 6.4.5, each cloned kernel image has a set of assigned interrupt sources. Hence, the domain switching action must also switch interrupt sources: masking interrupts of the previous running domain and unmasking interrupts of the next running domain. Specifically, the kernel masks all interrupts before switching the kernel window mapping (Section 6.4.6.1), disabling all interrupts that belong to the previous running domain. Then, the kernel unmasks interrupts associated with the next running kernel image (i.e., security domain) before returning to user mode in the next domain.

The x86 platform has a hierarchical interrupt routing structure, where all interrupts at the bottom layer are routed to the top-level interrupt controllers on CPU cores. During execution of the domain switch, the kernel disables interrupts by configuring the interrupt controller on its core. However, bottom-level interrupt sources can still trigger interrupts before being masked off, creating a race condition. As demonstrated in Listing 6.5, we solve this problem by deliberately probing any possible pending interrupts (lines 6–12) after masking but before switching to the next kernel (lines 1–5), acknowledging any pending interrupt at the hardware level.

```
1         /*mask IRQs of the previous kernel image*/
2         irq_list = irq_kernel
3         for  irq on irq_list
4             if irq is enabled
5                 disable(irq)
6         /*ack any pending IRQ*/
7         apic_set_task_prority(USER_THRESHOLD)
8         do {
9             received = get_active_irq( )
10            if received != irqInvalid
11                ackInterrupt(received)
12        } while (received != irqInvalid)
```

Listing 6.5: The pesudo code for acknowledging any pending interrupts that belong to the previous running kernel image on x86.

To ensure that the kernel only acknowledges pending user-level interrupts, we temporarily raise the interrupt serving priority by configuring the task and processor priority feature provided by the advanced programmable interrupt controller (APIC) on each core. The task priority feature allows the OS to set a threshold for receiving interrupts, hence temporarily blocking low priority interrupts from interrupting the CPU while processing high-priority tasks. By setting the task priority register (TPR), the core can receive only interrupts which have an interrupt-priority class higher than the current priority class [Intel, a]. On x86 platforms, the priority class of each interrupt source is stored in bits 7–4 of the 8-bit interrupt vector value. On our testing platform, we reserve vector values that are higher than 64 for user-level interrupts, which have interrupt priority class higher than 3. Hence, the kernel can only receive the user-level interrupts while the interrupt threshold is configured as 3 (USER_THRESHOLD), line 7 in Listing 6.5.

There are two advantages with this design: first, it guarantees that all interrupts are served by their corresponding kernel image; second, interrupts cannot be lost if configured as level-triggered, hence domains can still receive them once execution is resumed. For interrupts that are configured as edge-triggered, the hardware will lose them if they are not being triggered at their partition. However, our design does not introduce additional limitations on top of those imposed by hardware.

The Arm platform has a much simpler, single-level interrupt control structure, hence there is no race involved.

6.4.6.6 Summary

To summarise, the kernel executes the following steps for handling a preemption tick; steps in bold are only performed on a kernel switch (i.e., domain switch).

1. save user-level context;
2. acquire the kernel lock;
3. process the timer tick normally;
4. **mask all interrupts** (Section 6.4.6.5);
5. **switch the kernel stack** (Section 6.4.6.1);
6. switch thread context (and implicitly the kernel image);
7. release the kernel lock;
8. **unmask interrupts of this kernel** (Section 6.4.6.5);
9. **flush on-core microarchitectural state** (Section 6.4.6.2);
10. **pre-fetch shared kernel data** (Section 6.4.6.4);
11. **poll the cycle counter to pad to the the configured latency** (Section 6.4.6.3);
12. reprogram the timer interrupt; and
13. restore the user stack pointer and return.

6.4.7 Deleting cloned kernel images

Our design supports destroying cloned images by using the delete operation on KernelImage or KernelMemory capabilities. Deleting the only existing copy of the KernelImage capability or any of its KernelMemory capabilities will trigger kernel image deletion. Additionally, revoking the Untyped capability that a KernelImage or KernelMemory object is derived from can also trigger a kernel destruction, as performing a revocation on a capability deletes all of its descendants, based on the revocation mechanism in the seL4 kernel.

The capability management model of seL4 requires that the kernel must be able to perform deletions on all valid capabilities. Hence our design separates the deletion of KernelImage and KernelMemory capabilities, as they are different types of object.

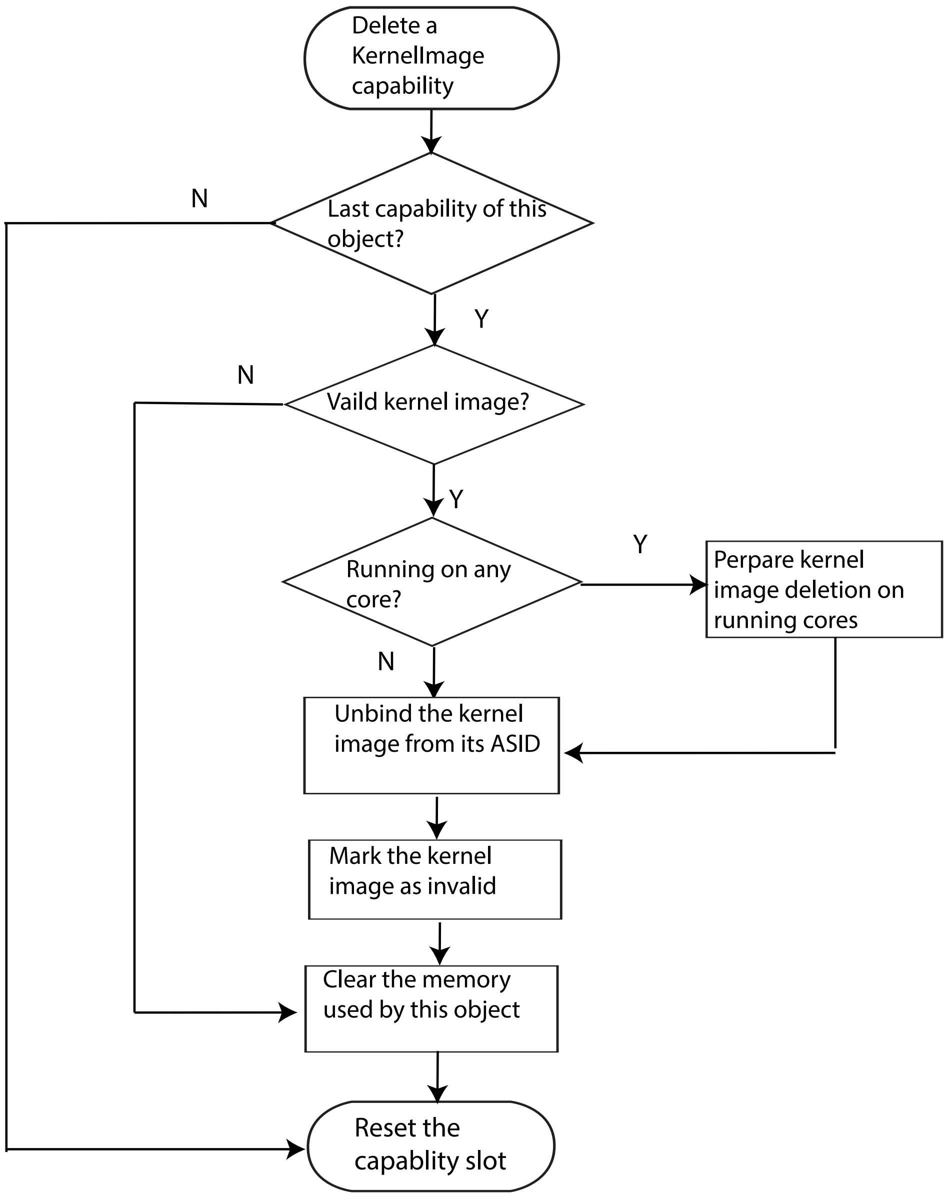

Delete a KernelImage capability
N
Last capability of this object?
Y
N
Vaild kernel image?
Y
Y
Running on any core?
Perpare kernel image deletion on running cores
N
Unbind the kernel image from its ASID
Mark the kernel image as invalid
Clear the memory used by this object
Reset the capablity slot

Figure 6.8: The flow chart of deleting a KernelImage capability.

The process of deleting a KernelImage capability is shown in Figure 6.8. Firstly, the kernel verifies that the capability is the last reference to the corresponding KernelImage object. If other copies of the capability exist, the kernel only deletes the capability by emptying its slot in the corresponding capability space. Otherwise, the kernel deletes the object if the kernel image is valid, i.e., the KernelMemory objects belonging to this kernel image have not been deleted yet.

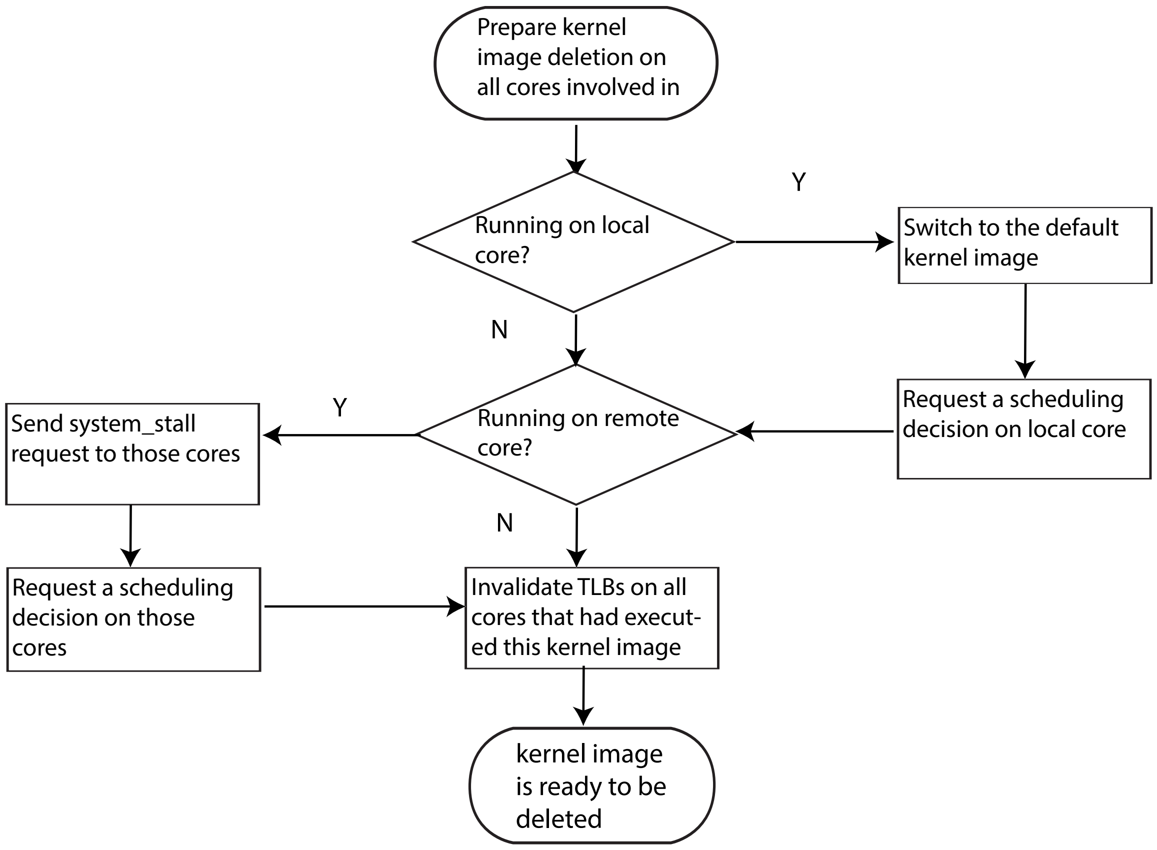

Prepare kernel image deletion on all cores involved in
Running on local core?
Y
Switch to the default kernel image
N
Request a scheduling decision on local core
Running on remote core?
Y
Send system_stall request to those cores
N
Request a scheduling decision on those cores
Invalidate TLBs on all cores that had execut-ed this kernel image
kernel image is ready to be deleted

Figure 6.9: The flow chart of preparing a kernel image deletion.

Additionally, the kernel cannot delete a kernel image that is currently in use. On a multicore system, destroying a kernel image can create a race condition if the kernel that is being destroyed is executing on other cores. Hence, the kernel prepares the deletion by switching to the default kernel image (i.e., kernel image created at boot time) on all the cores involved, including both local and remote cores (Figure 6.9).

For notifying all remote cores, the kernel sends IPIs to all cores on which the KernelImage is running. We call this type of IPI request `system_stall`. In order to precisely send `system_stall` messages, our design maintains a bitmap in each kernel image (Section 6.4.3), indicating cores on which the kernel is presently running. The kernel-switching process (Section 6.4.6.1) updates bitmaps in the current and next running kernel. When receiving the `system_stall` request, a core switches to the idle thread belonging to the default kernel image that is created at boot time. Likewise, the kernel sends a `TLB_invalidate` request to all cores that the kernel image had been running on, which is analogous to TLB shoot-down.

Then the kernel unbinds the KernelImage from its ASID, same as the PageDirectory object deletion. At last, the kernel marks the kernel image as invalid, and cleans up the memory used by the KernelImage object.

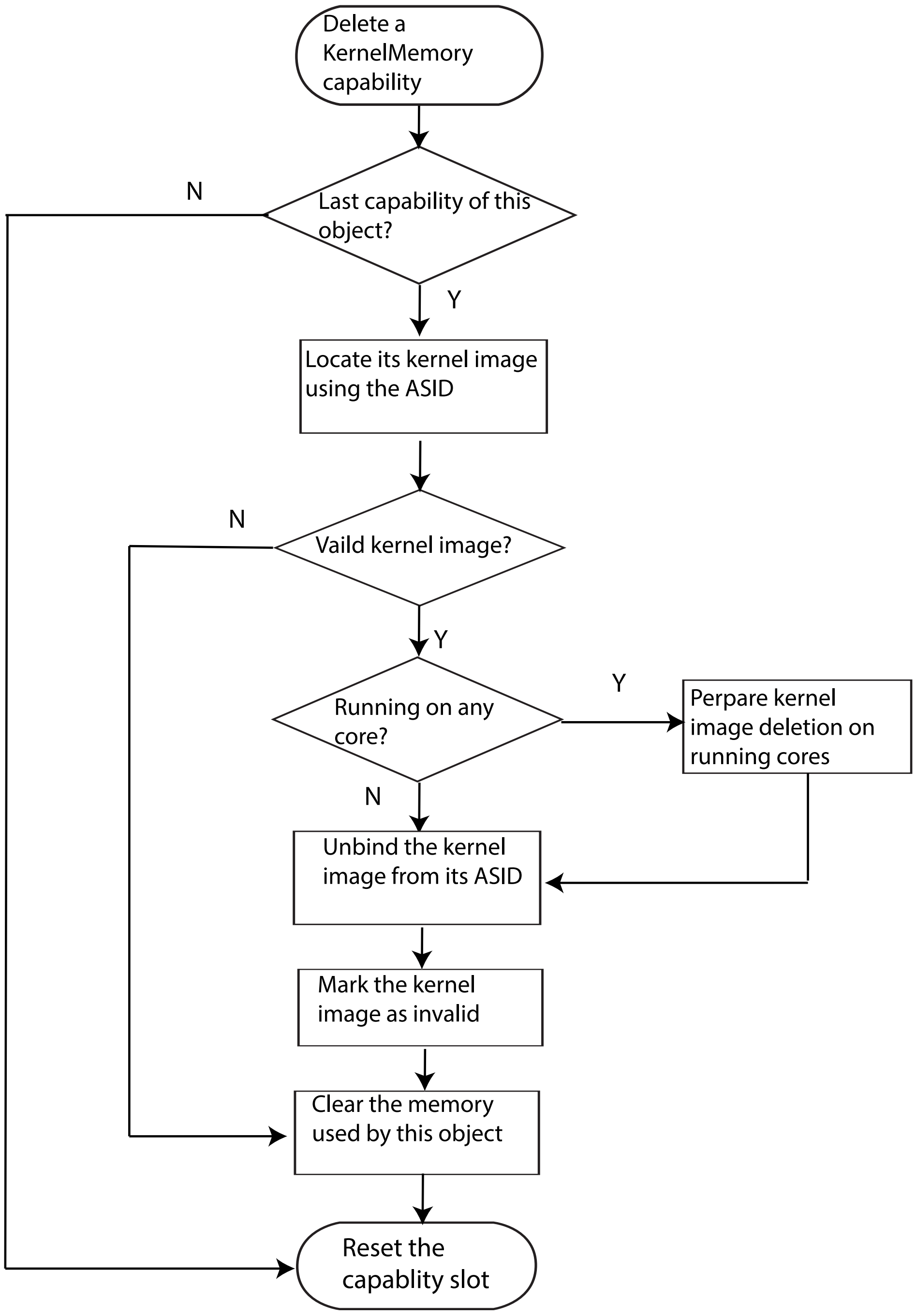

Delete a KernelMemory capability
N
Last capability of this object?
Y
Locate its kernel image using the ASID
N
Vaild kernel image?
Y
Y
Running on any core?
Perpare kernel image deletion on running cores
N
Unbind the kernel image from its ASID
Mark the kernel image as invalid
Clear the memory used by this object
Reset the capablity slot

Figure 6.10: The flow chat of deleting a KernelMemory capability.

Deleting a KernelMemory capability is similar to deleting a KernelImage capability, except that the kernel first locates the kernel image using this KernelMemory object by the

ASID (Section 6.4.3). As shown in Figure 6.10, destroying a KernelMemory that belongs to a valid kernel image also invalidates the corresponding kernel image, causing the same sequence of actions. Once a kernel image is marked as invalidated, destroying the remaining objects becomes easy as the kernel only needs to recycle their memory.

The seL4 kernel assumes that the system always has a runnable idle thread. To maintain this invariant, our design prevents the destruction of the initial kernel image by not providing its KernelMemory capability to the initial thread. Hence, our design guarantees that the default kernel and its idle thread will forever be available even though all the other kernel images can be dynamically created and destroyed. If no user-level thread is available, the system will do nothing but acknowledge the system timer interrupt.

Our current design does not support reusing the kernel memory used by the default kernel image, if the system decides that the initial kernel image is no longer useful once a strictly partitioned system is formed with cloned kernel images. The cost is affordable due to the small amount of dead memory. On x86, the default kernel image will only create 224 KiB of waste on a single core, and requiring an additional 4 KiB for the kernel stack of each additional core. The kernel image on x86 includes a 64 KiB of buffer for manually flushing the L1 caches. Arm sizes are 120 KiB on a single core, with extra 4 KiB for each core used as the kernel stack.

6.5 Evaluation

We evaluate our system design in terms of its efficacy in preventing timing channels, as well as its impact on system performance.

6.5.1 Hardware platforms

We conduct our experiments on both x86 and Arm platforms, as listed in Table 6.2.

We run a full set of evaluations on the Haswell processor (x86), and on the Arm Cortex A9 processor (Sabre platform). According to the results in Section 5.4.3.6, the flushing operations provided by manufacturers cannot fully mitigate the BHB channel on the Haswell processor, as it contains some hidden state. We acknowledge that the effectiveness of our "manual flush" on x86 (Section 6.4.6.2) could be worse than the result listed in Section 5.4.3.6 for mitigating intra-core timing channels, as the hardware does not provide operations for resetting L1 caches and the BTB.

Resetting on-core state that cannot be partitioned is one of hardware requirements listed in Section 4.2.1. which is not provided by Haswell, Skylake, nor Arm Cortex-A53 processors according to the results of our study conducted on four popular commodity processors (Section 5.4). Here we assume that our mechanism will be fully effective at preventing intra-core channels, if the processor provides a sufficient hardware-software contract for security as we proposed in Section 5.5. Since x86 and Arm are different architectures, we evaluate a full set of benchmarks on one x86 processor and one Arm processor.

System	Haswell	Sabre
Architecture	x86	Arm v7
Manufacturer	Intel	Freescale
Microarchitecture	Haswell	Cortex-A9
Processor/SoC	Core i7-4770	i.MX 6Q
Cores × threads	4×2	4×1
Clock	3.4 GHz	0.8 GHz
Cache line size	64 B	32 B
L1-D/L1-I cache	32 KiB, 8-way	32 KiB, 4-way
L2 cache	256 KiB, 8-way	1 MiB, 16-way
L3 cache	8 MiB, 16-way	N/A
I-TLB	64, 8-way	32, 1-way
D-TLB	64, 4-way	32, 1-way
L2-TLB	1024, 8-way	128, 2-way
BTB	?	512
RAM	16 GiB	1 GiB

Table 6.2: Hardware platforms.

For representing the x86 architecture, we select the Haswell processor; for representing the Arm architecture, we select the Arm Cortex-A9 processor, an implementation of Arm v7. Our implementation for the Arm architecture does not support the Arm v8 architecture yet, therefore we cannot evaluate on the Cortex-A53 processor.

For timing channel benchmarks, we evaluate the unmitigated channel, marked as **raw**, and the channel with time protection applied, marked as **protected**.

Our time protection deploys two coloured security domains using cache colouring. The cache colouring implementation splits available colours on the L2 cache into two partitions, 50% of colours for each partition unless specifically stated otherwise. On the Haswell platform, there are eight colours on the L2 cache, and colouring the memory according to colours of the L2 also colours the L3 cache. On the Sabre platform, there are 16 colours on the L2 cache. Each coloured domain has user-level threads and a dedicated kernel image which are created from the same partition. During a domain switch, the kernel conducts domain switch actions as described in Section 6.4.6.

For covert channels on components that cannot be partitioned (Section 5.3.1), we also list the result of performing a full flush of the microarchitectural state (marked as **full flush**) including the full cache hierarchy, as listed in Section 5.3.2. We list the effectiveness of the full flush to see whether minimal flush of only non-partitioned state leaves residual channels that could be closed by less discriminate flushing. On x86, the cache flushing operation `wbinvd` flushes and invalidates all three levels of caches, which is overkill for mitigating intra-core timing channels that only explore two levels of caches.

To establish a baseline of the cost of cache flushing operations, we measure the direct and indirect cost of flushing L1 caches as well as the complete cache hierarchy. The direct cost is the latency of performing corresponding cache flushing operations. We measure the latency of performing L1 or LLC cache flush operations. On x86, the L1 cache flush is implemented as a manual flush (Section 6.4.6). The indirect cost is the slowdown of an application whose working-set size equals the respective cache. Even though we measure the indirect cost for flushing L1 caches, this cost is somewhat academic, due to the limited capacity of L1 caches. A process would be highly unlikely to reuse any of the L1 cache lines after a domain switch, which is conducted once every 10–100 ms.

	Haswell (x86)			Sabre (Arm Cortex-A9)		
Cache	dir	ind	total	dir	ind	total
L1 (μs)	25.52	1.08	26.59	20	24.53	44.53
all (μs)	270	250	520	380	770	1,150

Table 6.3: Cost of cache flushes.

Table 6.3 summarises the result of these benchmarks. We report the mean from 320 runs. All relative standard deviations are less than 1% on Haswell and 3% on Sabre. The direct cost of the manual L1 cache flush on the Haswell platform is mainly contributed by the chained jumps for flushing the L1-I cache: there is only less than 0.5 μs consumed by the L1-D flush. The manual L1-I cache flush is expensive because of branch mis-predictions on chained jumps (Section 6.4.6), which could be largely reduced with a targeted flushing operation provided by the manufacturer. The indirect cost measures the slowdown of a user-level thread accessing a L1-D cache sized buffer with chained instructions which is the same used by the PRIME+PROBE attack on the L1-D cache (Section 5.3.1.1). Compared to only flushing L1 caches, flushing the entire cache hierarchy is much more expensive, costing 520 μs on Haswell and 1,150 μs on Sabre.

To put these figures into context, only flushing L1 caches introduces 0.3% overhead on Haswell and 0.4% overhead on Sabre, if the system tick is 10 ms. In contrast, flushing all caches can potentially introduce 5.2% overhead on Haswell and 11.5% overhead on Sabre with the same configuration. Clearly, only flushing L1 caches has a distinct advantage on system performance compared to flushing the entire cache hierarchy.

6.5.2 Timing channel mitigation efficacy

Our evaluations cover covert timing channels between two security domains by time-multiplexing a core, threats in the confinement scenario (Section 4.1.1), and a side channel on the LLC between domains running concurrently on different cores, which is the only threat in the cloud scenario (Section 4.1.2). For covert timing channels, we create channels on cache and cache-like components (Section 6.5.2.1), on domain switching latency (Section 6.5.2.2), on a shared kernel image (Section 6.5.2.3), and on a timer interrupt

(Section 6.5.2.4). For the timing side channel, we replay the cross-core side channel attack of Liu et al. [2015], a classical cross-core timing attack on the LLC.

To analyse covert timing channels, we use the method introduced in Section 5.3.3, including generating channel matrices (Section 5.3.3.1) and quantifying the channel leakage with MI calculations (Section 5.3.3.2). We rerun each experiment up to eight times, and analyse the results according to the method stated in Section 5.3.3.3.

6.5.2.1 Covert timing channels on cache or cache-like components

Platform	Cache	Raw	Full flush	Protected
Haswell (x86)	L1-D	4,053	0.3 (0.3)	0.3 (0.3)
	L1-I	260	0.4 (0.4)	0.6 (0.2)
	TLB	2,564	0.3 (0.3)	13.4 (22.9)
	BTB	1,533	0.4 (0.4)	0.2 (0.2)
	BHB	1,000	0.4 (0.0)	0.0 (0.0)
	L2	2,685	1.3 (1.3)	**49.1 (2.5)**
Sabre (Arm Cortex-A9)	L1-D	2,070	0.5 (0.6)	**29.7 (38.8)**
	L1-I	2,489	0.7 (0.7)	2.4 (2.6)
	TLB	559	0.2 (0.3)	1.1 (1.2)
	BTB	5.4	2.0 (2.2)	56.2 (66.2)
	BHB	1,000	0 (0.3)	0.0 (91.7)
	L2	1,929	11 (11)	0.7 (0.7)

Table 6.4: MI (millibit) of unmitigated (raw) covert timing channels on cache or cache-like components, mitigated with full cache flush (full flush) and time protection (protected). Value in parentheses is $\mathcal{M}_0$. Confirmed residual channels are marked in **bold**.

We implement a full set of covert channels that are exploitable on hardware components that behave like a cache, including L1-D, L1-I, and L2 caches, the TLB, the BTB, and the BHB. We use the PRIME+PROBE attack introduced in Section 5.3.1 on all above components, where the Trojan encodes secrets by touching a number of cache sets, and the spy receives by timing the cost of probing on a cache-sized buffer. The L2 cache channel has a similar implementation to the L1-D cache channel, only with an extended number of probing sets to touch all of the L2 cache sets. The Trojan and spy are running inside dedicated security domains, time-multiplexing a core.

The evaluated MI values of the protected scenario are generally higher than those of the full flush scenario. The reason is that the deterministic execution environment provided by the full flush contains much less hardware noise, thus smaller MI values contributed by noise. However, we confirm any residual channels by carefully examining matrices generated from up to eight repeated runs, as only reading the MI (millibit) numbers are not sufficient for confirm a timing channel as explained in Section 5.3.3.3.

Firstly, we successfully create timing channels in all cases, as we can observe a clear resource contention on channel matrices. Furthermore, the raw channels all have significant MI values. Secondly, the implementation of time protection can effectively mitigate timing channels on both Haswell and Sabre platforms. However, we confirm two residual channels, the L2 cache channel on Haswell, and the L1-D channel on Sabre, which we investigate further.

L1-D cache As described in Section 5.3.1.1, the L1-D cache timing channel uses the time to perform the attack on every cache set as the output symbol. The message sent by the Trojan is encoded as accessed cache sets, a number between 0 and 64 on Haswell (256 on Sabre).

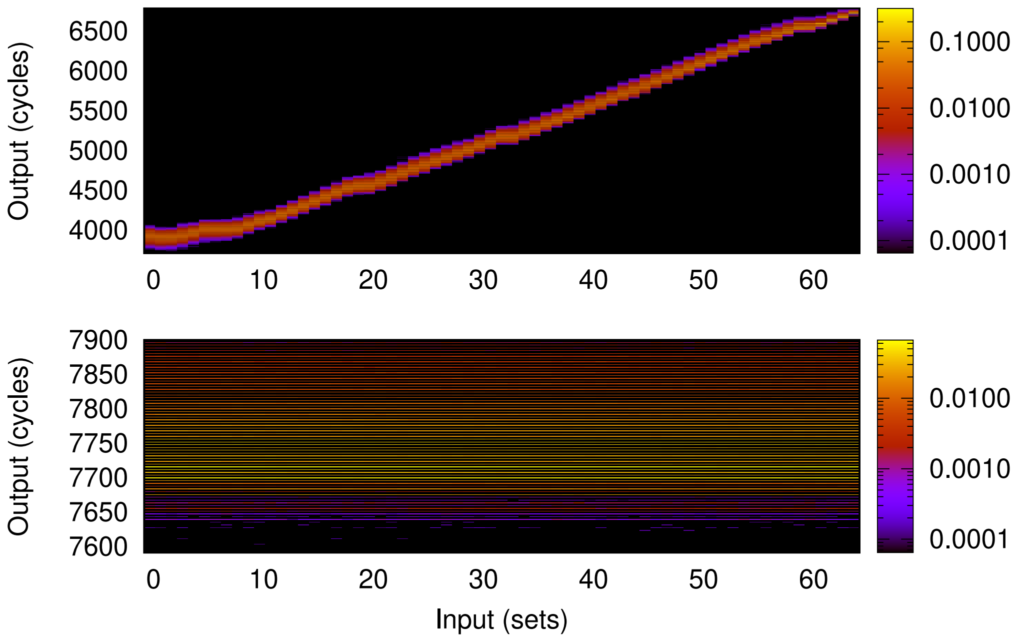


Figure 6.11: The L1-D cache covert channel on Haswell. Top: without mitigation, $\mathscr{M} = 4{,}053$ mb, $n = 15{,}436$, bottom: mitigated with time protection $\mathscr{M} = 0.3$ mb, $\mathscr{M}_0 = 0.3$ mb, $n = 15{,}501$.

The top matrix in Figure 6.11 shows the original L1-D channel on the Haswell, demonstrating a clear information leak ($\mathscr{M} = 4{,}053$ mb). The effectiveness of our time protection mechanism is shown in Figure 6.11 (bottom), where the channel is completely closed as there is no correlation between outputs (spy's probing cost) and inputs (Trojan's activity) ($\mathscr{M} = 0.3$ mb, $\mathscr{M}_0 = 0.3$ mb).

Figure 6.12 shows matrices for the same channel on the Sabre platform. On the top matrix, we can observe that the probing cost measured by the spy increases significantly while the Trojan accesses more cache sets, showing the original channel ($\mathscr{M} = 2{,}070$ mb). This trend disappears once time protection is enabled (bottom). However, we observe that the output has a 1% probability of observing 22,087 for inputs larger than 60, while it

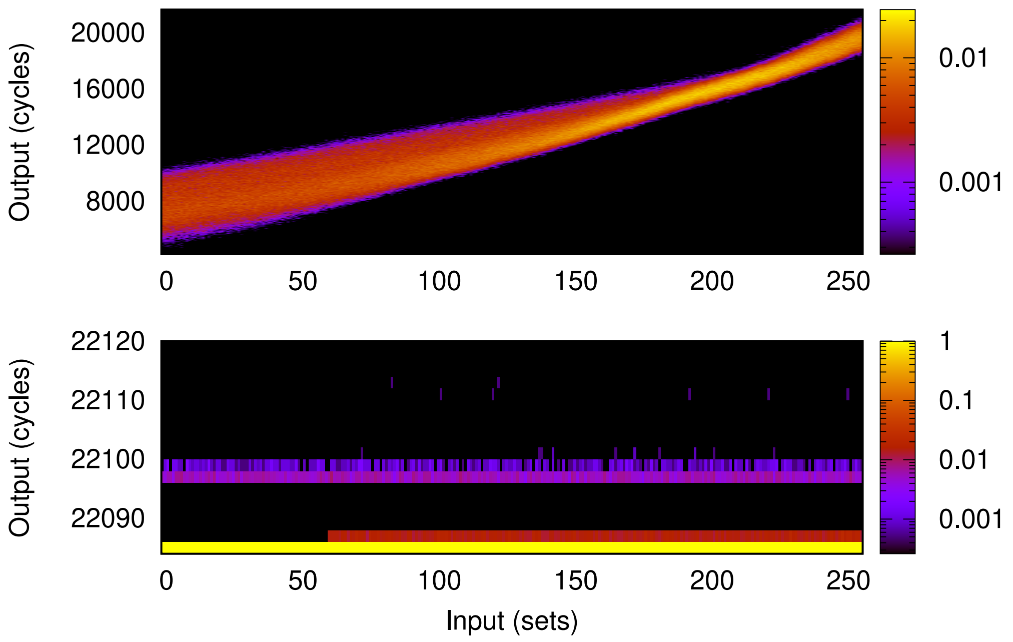


Figure 6.12: The L1-D cache channel on Sabre (Arm Cortex-A9). Top: without mitigation, $\mathscr{M} = 2,070\,\text{mb}$, $n = 3,808$, bottom: mitigated with time protection $\mathscr{M} = 29.7\,\text{mb}$, $\mathscr{M}_0 = 38.8\,\text{mb}$, $n = 3,842$.

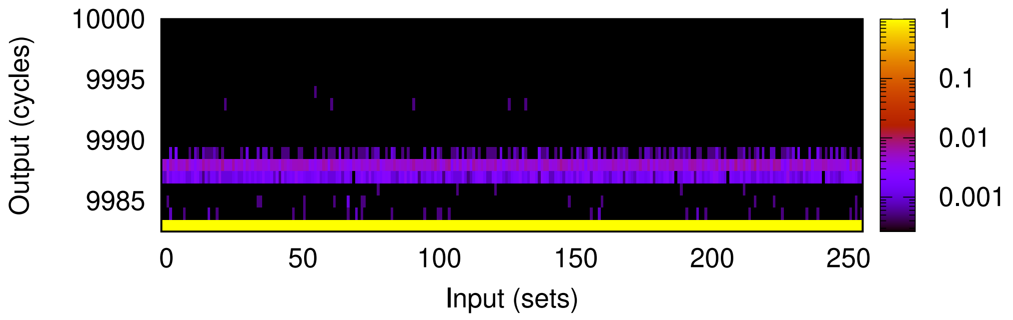


Figure 6.13: The L1-D cache covert channel on Sabre (Arm Cortex-A9), mitigated with time protection and disabling both prefetcher and branch prediction unit, $\mathscr{M} = 59\,\text{mb}$, $\mathscr{M}_0 = 61.8\,\text{mb}$, $n = 3,803$.

is essentially zero for smaller inputs, a clear channel. Furthermore, we repeat the same experiment five times and are able to observe the same distribution from all five repeated runs. Hence, the effect is highly unlikely to be caused by random hardware noise, but due to a small amount of on-core state that cannot be flushed with instructions targeting on-core state (Section 6.4.6.2), while a full flush closes the channel. To investigate the site of the channel, we rerun the mitigated channel with both the prefetcher and branch prediction unit disabled, the remaining two hardware operations we cannot included with the discriminating flush operations at our disposal (Table 6.1). Figure 6.13 shows the result—outputs are evenly distributed across inputs. The domain switch executes the `BPIALL` instruction to invalidate all branch prediction entries (Section 6.4.6.2). Hence, the residual channel that we observe on the bottom matrix in Figure 6.12 is due to the 2-level prefetcher (Section 2.3.3). This result also shows the importance of examining any residual channels using channel matrices even though the MI evaluation passes the zero-leakage test ($\mathcal{M} = 29.7\,\text{mb}$, $\mathcal{M}_0 = 38.8\,\text{mb}$), a false negative.

L1-I cache We run the L1-I cache timing channel as described in Section 5.3.1.2. The Trojan and spy conduct the PRIME+PROBE attack with jump instructions that map to corresponding cache sets. The input ranges are 0–64 on Haswell and 0–256 on Sabre.

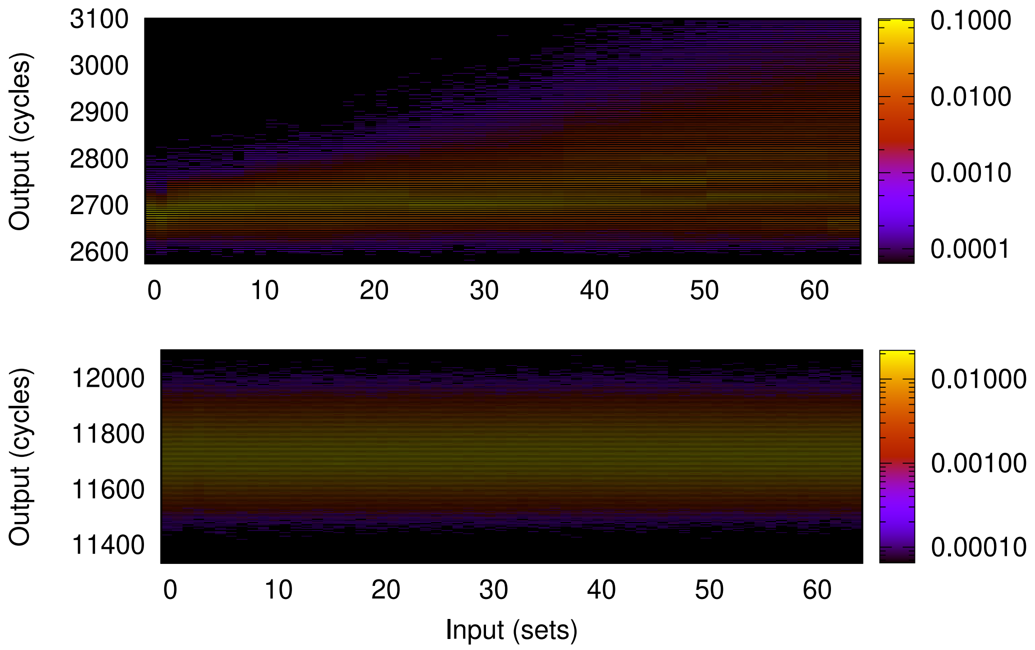


Figure 6.14: The L1-I cache covert channel on Haswell. Top: without mitigation, $\mathcal{M} = 260\,\text{mb}$, $n = 15{,}532$, bottom: mitigated with time protection $\mathcal{M} = 0.6\,\text{mb}$, $\mathcal{M}_0 = 0.2\,\text{mb}$, $n = 15{,}417$.

Figure 6.14 shows matrices for unmitigated (top) and mitigated with time protection (bottom) channels on Haswell. The top matrix presents the contention on L1-I cache sets that are utilised as a timing channel: the probing cost measured by the spy is higher while the Trojan jumps through more cache sets. With time protection (bottom), the Trojan's activity no longer affects the probing time of spy. We measure the MI of the mitigated channel as $\mathcal{M} = 0.6\,\text{mb}$ ($\mathcal{M}_0 = 2\,\text{mb}$), and we cannot observe any consistent trend with matrices generated for eight repeated runs. Hence, we conclude that the channel is closed.

On Sabre, the channel is closed with time protection as outputs are evenly distributed across inputs (bottom in Figure 6.15). The MI evaluation for unmitigated channel is $\mathcal{M} = 2{,}489\,\text{mb}$, and mitigated channel is $\mathcal{M} = 2.4\,\text{mb}$ ($\mathcal{M}_0 = 2.6\,\text{mb}$).

TLB As specified in Section 5.3.1.3, we implement the TLB covert timing channel as a PRIME+PROBE attack on TLB entries. The Trojan probes TLB entries by reading an integer

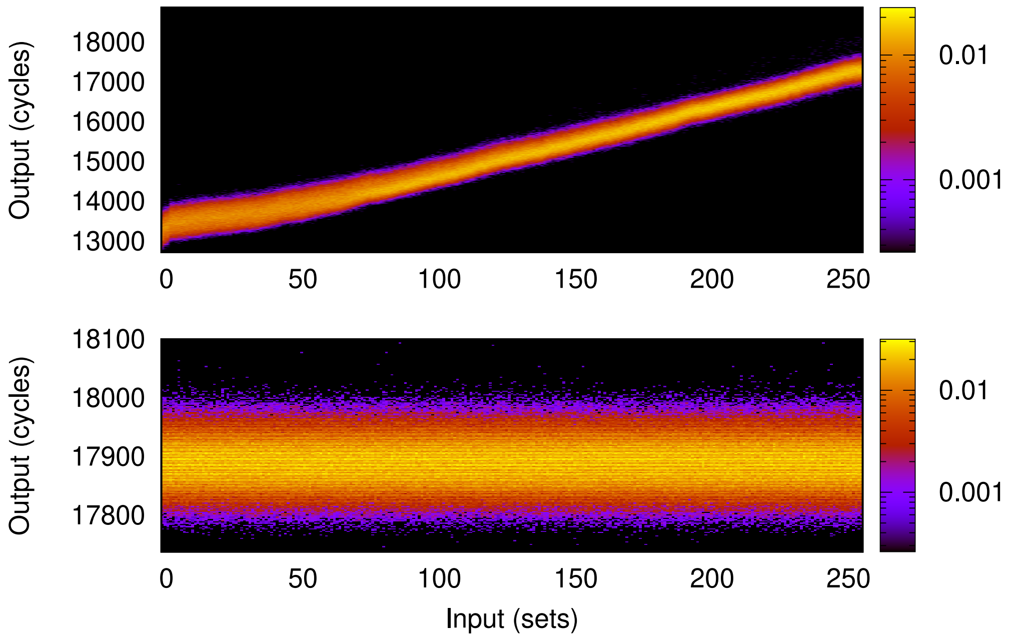


Figure 6.15: The L1-I cache covert channel on Sabre (Arm Cortex-A9). Top: without mitigation, $\mathcal{M} = 2{,}489\,\mathrm{mb}$, $n = 3{,}808$, bottom: mitigated with time protection $\mathcal{M} = 2.4\,\mathrm{mb}$, $\mathcal{M}_0 = 2.6\,\mathrm{mb}$, $n = 3{,}835$.

from a number of consecutive pages, whereas the spy measures the cost of accessing half of the TLB entries.

Figure 6.16 presents matrices for the TLB covert timing channel on Haswell. We see that the stepping effect on the distribution of outputs for the unmitigated channel (top) disappears once time protection is enabled (bottom). There is a correlation between the input ranging in 3–37 and output value 1713: 0.01% of outputs are 1713. However, we cannot observe the same trend from eight repeated runs. Hence the distribution is highly likely due to hardware noise. Results for the MI evaluation are $\mathcal{M} = 2{,}564\,\mathrm{mb}$ for the original channel and $\mathcal{M} = 13.4\,\mathrm{mb}$ ($\mathcal{M}_0 = 22.9\,\mathrm{mb}$) for time protection, a closed channel.

On Sabre, the effect on the distribution of outputs due to the TLB contention (top) completely disappears in the bottom matrix generated for the time-protection mitigated channel, as shown in Figure 6.17. The MI evaluation also shows that the original channel ($\mathcal{M} = 559\,\mathrm{mb}$) is completed closed ($\mathcal{M} = 1.1$ mb, $\mathcal{M}_0 = 1.2$ mb).

BTB The BTB covert timing channel uses chained branch instructions as the probing buffer, as described in Section 5.3.1.4. Same as the attack implemented in Section 5.3.1.4, the Trojan probes from 3584 to 3712 branch instructions on Haswell (0 to 512 on Sabre).

As demonstrated on the bottom matrix in Figure 6.18, time protection effectively removes the strip shaped pattern contained on the matrix for the unmitigated channel (top). Results for MI evaluation are $\mathcal{M} = 1{,}553\,\mathrm{mb}$ for the original channel, and $\mathcal{M} = 0.2\,\mathrm{mb}$ ($\mathcal{M}_0 = 0.2\,\mathrm{mb}$) for the mitigated channel.

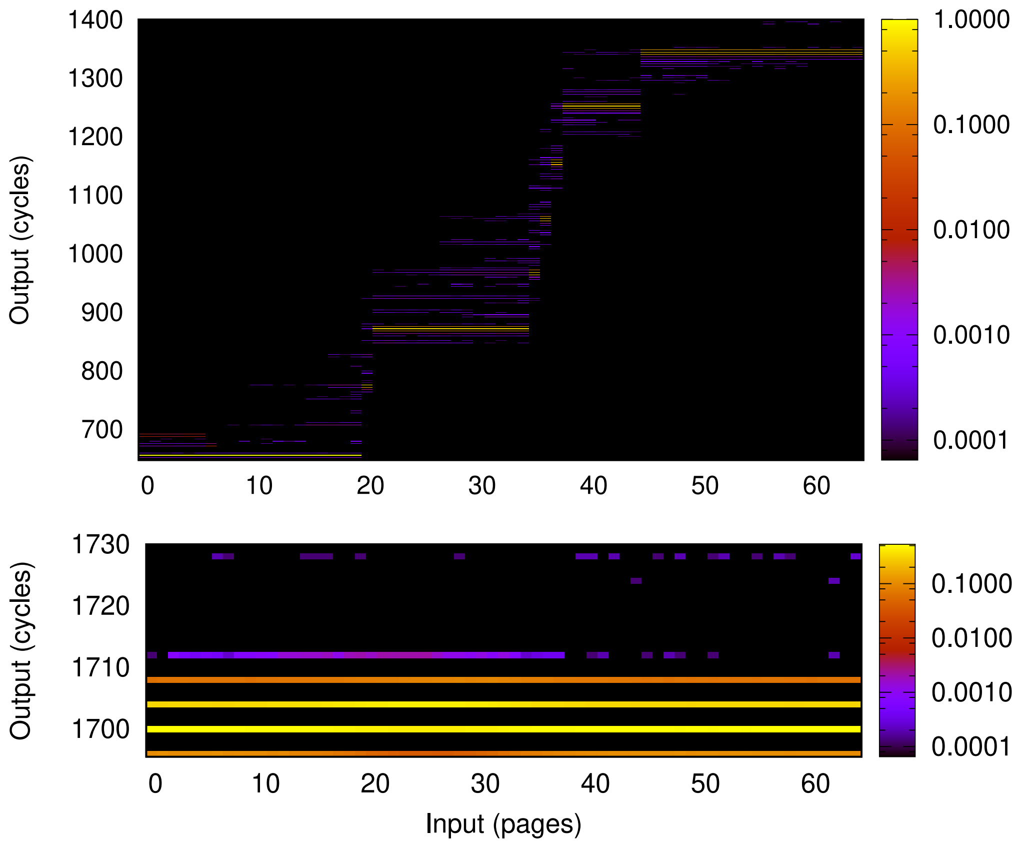


Figure 6.16: The TLB covert channel on Haswell. Top: without mitigation, $\mathscr{M} = 2{,}564\,\text{mb}$, $n = 15{,}511$, bottom: mitigated with time protection $\mathscr{M} = 13.4\,\text{mb}$, $\mathscr{M}_0 = 22.9\,\text{mb}$, $n = 15{,}470$.

We present matrices for the unmitigated (top) and time protection mitigated (bottom) channels on Sabre in Figure 6.19. On the bottom matrix, outputs are evenly distributed across inputs, demonstrating a fully closed channel ($\mathscr{M} = 56.2\,\text{mb}$, $\mathscr{M}_0 = 66.2\,\text{mb}$).

BHB We run the BHB covert timing channel as described in Section 5.3.1.5. The Trojan sends information by either taking (input “0”) or skipping (input “1”) a conditional jump instruction, whereas the spy measures the latency on taking a similar conditional jump instruction, sensing any speculative execution caused by the Trojan’s history.

On Haswell, the channel ($\mathscr{M} = 1{,}000\,\text{mb}$) is closed ($\mathscr{M} = 0.0\,\text{mb}$, $\mathscr{M}_0 = 0.0\,\text{mb}$) by our time protection implementation, as shown on the bottom matrix in Figure 6.20. The distribution of outputs is independent of inputs on the matrix generated for the mitigated channel (bottom). It is interesting to note that time protection is more effective at closing the BHB channel, comparing with the result for the full flush scenario (Section 5.4.3.5). This fact is mostly likely due to our manual L1-I cache flush being effective at flushing the BHB as a side effect: a large number of jump instructions (4,096) are included in the manual L1-I cache flush Section 6.4.6.2. However, we cannot confirm the actual situation on hardware due to the lack of documentation provided by the manufacturer.

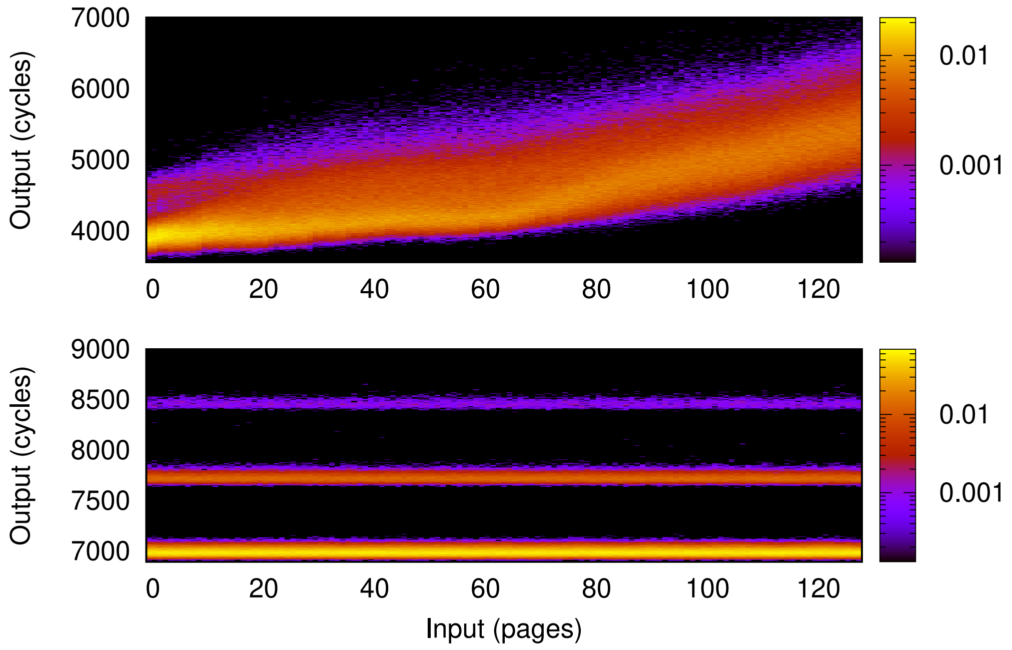


Figure 6.17: The TLB covert channel on Sabre (Arm Cortex-A9). Top: without mitigation, $\mathscr{M} = 559\,\text{mb}$, $n = 7,680$, bottom: mitigated with time protection $\mathscr{M} = 1.1\,\text{mb}$, $\mathscr{M}_0 = 1.2\,\text{mb}$, $n = 7,685$.

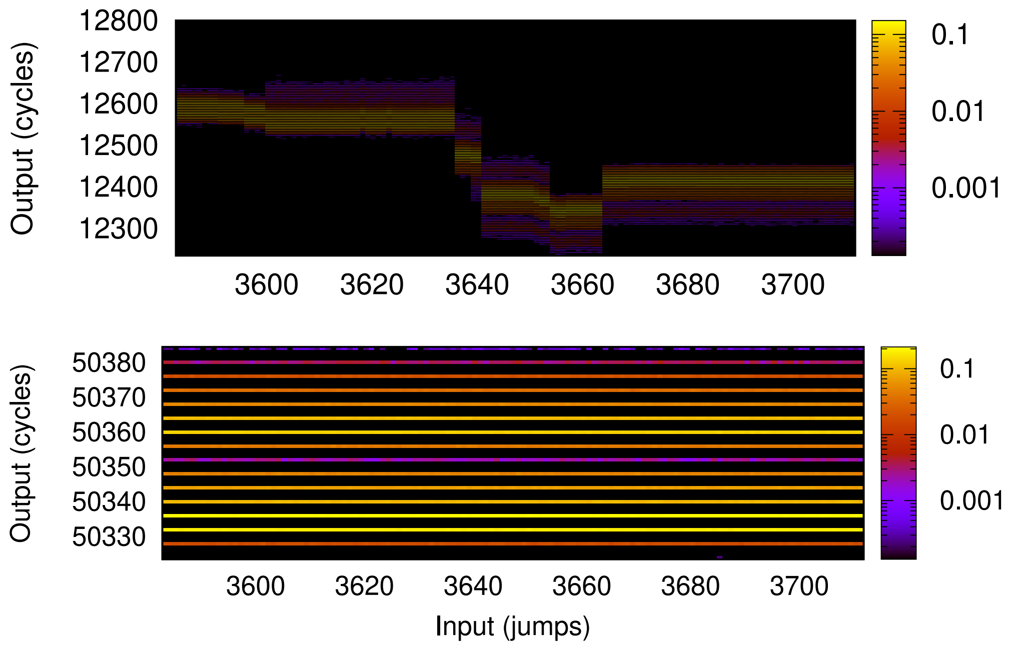


Figure 6.18: The BTB covert channel on Haswell. Top: without mitigation, $\mathscr{M} = 1,553\,\text{mb}$, $n = 7,663$, bottom: mitigated with time protection $\mathscr{M} = 0.2\,\text{mb}$, $\mathscr{M}_0 = 0.2\,\text{mb}$, $n = 7,769$.

Similarly, the channel ($\mathscr{M} = 1,000\,\text{mb}$) is closed ($\mathscr{M} = 0.0\,\text{mb}$, $\mathscr{M}_0 = 91.7\,\text{mb}$) on Sabre (Figure 6.21) by time protection.

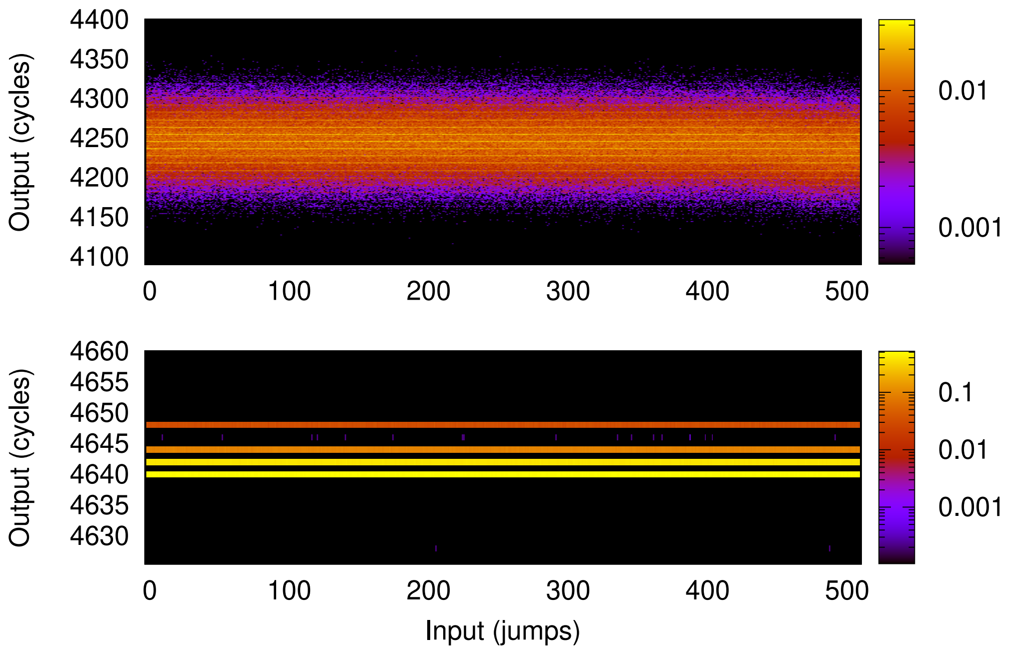


Figure 6.19: The BTB covert channel on Sabre (Arm Cortex-A9). Top: without mitigation, $\mathscr{M} = 5.4\,\mathrm{mb}$, $n = 1,857$, bottom: mitigated with time protection $\mathscr{M} = 56.2\,\mathrm{mb}$, $\mathscr{M}_0 = 66.2\,\mathrm{mb}$, $n = 9,680$.

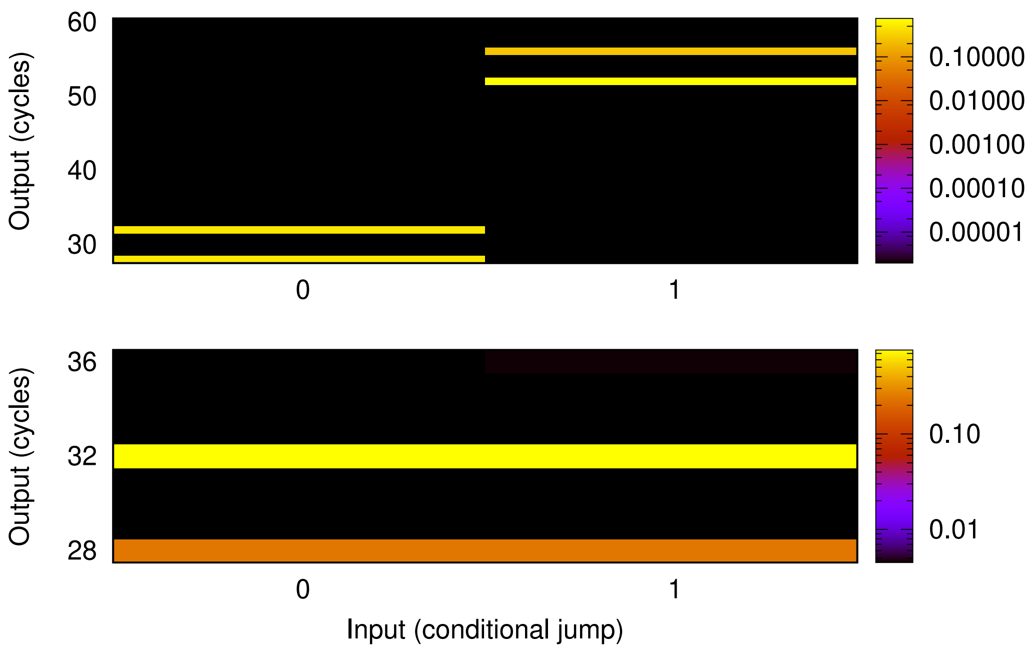


Figure 6.20: The BHB covert channel on Haswell. Top: without mitigation, $\mathscr{M} = 1,000\,\mathrm{mb}$, $n = 511,765$, bottom: mitigated with time protection $\mathscr{M} = 0.0\,\mathrm{mb}$, $\mathscr{M}_0 = 0.0\,\mathrm{mb}$, $n = 511,538$.

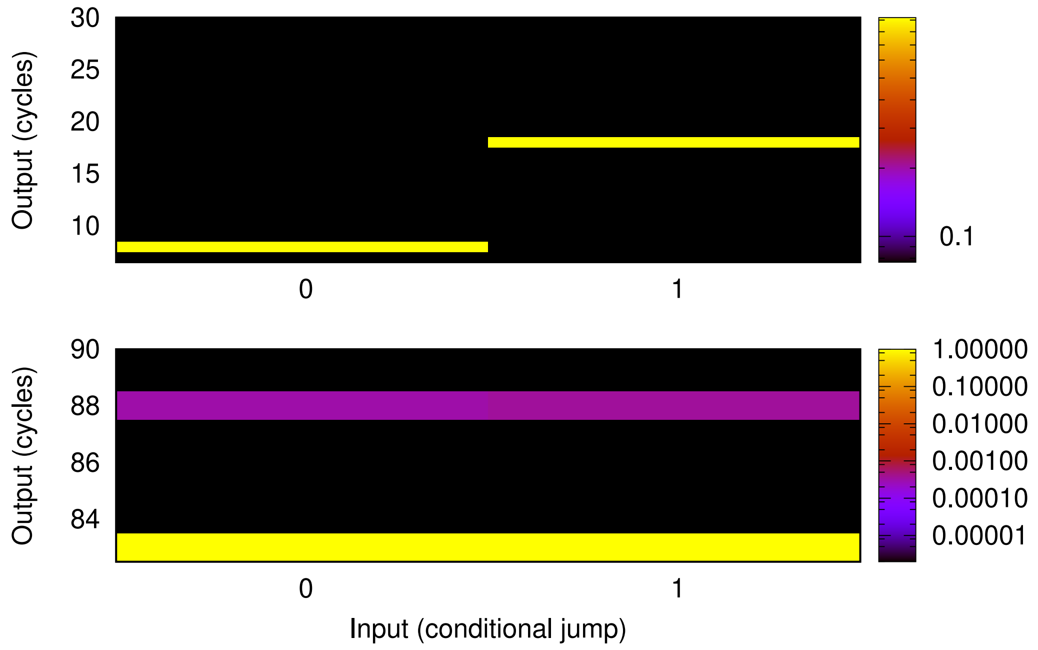


Figure 6.21: The BHB covert channel on Sabre (Arm Cortex-A9). Top: without mitigation, $\mathscr{M} = 1{,}000$ mb, $n = 511{,}434$, bottom: mitigated with time protection $\mathscr{M} = 0.0$ mb, $\mathscr{M}_0 = 91.7$ mb, $n = 511{,}550$.

L2 cache The L2 cache channel uses a similar implementation to the L1-D cache channel, but with a larger probing set to cover all the L2 cache sets. We run the L2 covert channel as a representation of the covert channel on a physically-indexed cache.

Figure 6.22 shows matrices for both the original (top) and mitigated (bottom) channels with time protection on the Haswell processor. On the matrix generated for the original channel, we can observe that the output, probing cost measured by the spy, surges upwards while the input, number of cache sets visited by the Trojan, increases. The MI evaluation for the original channel is $\mathscr{M} = 2{,}685$ mb, which is reduced by time protection, $\mathscr{M} = 49.1$ mb ($\mathscr{M}_0 = 2.5$ mb). However, we can still observe the wave on the bottom matrix, indicating that the distribution of outputs is still correlated with input values even though the timing mitigation is enabled. Moreover, the phenomenon is consistent across eight repeated runs, confirming a remaining channel. The calculated MI values for those eight repeated experiments range from $\mathscr{M} = 49.1$ mb to $\mathscr{M} = 351.1$ mb. In Table 6.4, we use the lower bound because calculated MI values can also be caused by other effects, such as the hardware noise.

We suspect that some remnant state cannot be fully reset, because the hardware does not comply with our requirements, as it does not provide resetting operations on all on-core states that cannot be partitioned (Section 4.2.1). One possible remaining microarchitectural state is the prefetcher, which prefetches data streams based on previous cache misses (Section 2.3.2). To investigate, we disable the data prefetcher (Section 5.3.2.1) in the time protection scenario. The result shown in Figure 6.23 justifies our hypothesis, as the

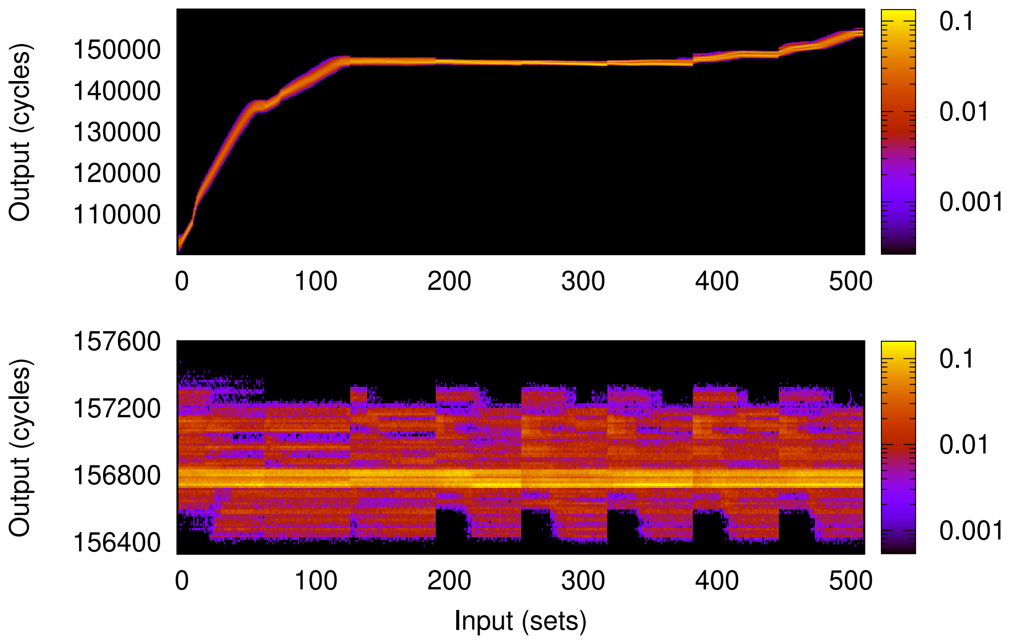


Figure 6.22: The L2 cache covert channel on Haswell. Top: without mitigation, $\mathscr{M} = 2{,}685$ mb, $n = 3{,}817$, bottom: mitigated with time protection $\mathscr{M} = 49.1$ mb, $\mathscr{M}_0 = 2.5$ mb, $n = 1{,}867$.

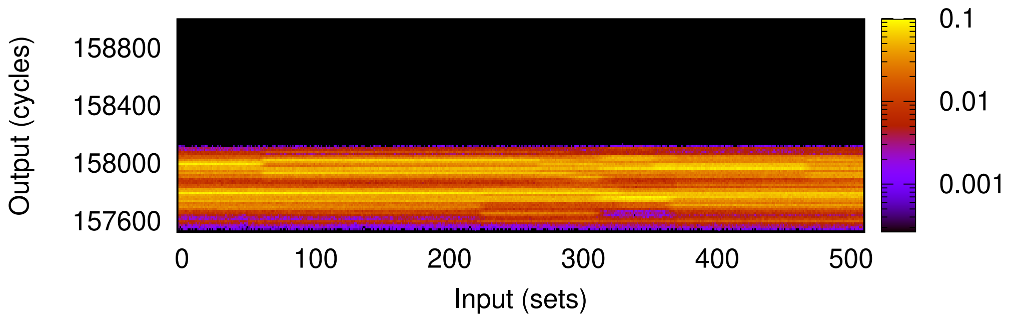


Figure 6.23: The L2 cache covert channel on Haswell, mitigated with time protection and disabling prefetcher $\mathscr{M} = 5.2$ mb, $\mathscr{M}_0 = 2.7$ mb, $n = 3{,}807$.

waving effect on the output distribution disappears once the data prefetching is disabled ($\mathscr{M} = 5.2$ mb, $\mathscr{M}_0 = 2.7$ mb). Moreover, we cannot observe any consistent trend on matrices for five repeated runs, demonstrating that the calculated MI is highly likely due to another side effect, such as hardware noise. This is more evidence than a need for a better software-hardware contract that controls all hidden microarchitecture states (Section 5.5).

Time protection is effective on Sabre ($\mathscr{M} = 0.7$ mb, $\mathscr{M}_0 = 0.7$ mb) for mitigating the L2 channel. Figure 6.24 demonstrates channel matrices for the L2 cache channel on Sabre without (top) and with the time protection (bottom). The result shows that partitioning the L2 cache is effective in preventing the covert channel on the L2 cache, as there is no L2 flushing operation involved in time protection. As the L2 is the external cache on Sabre,

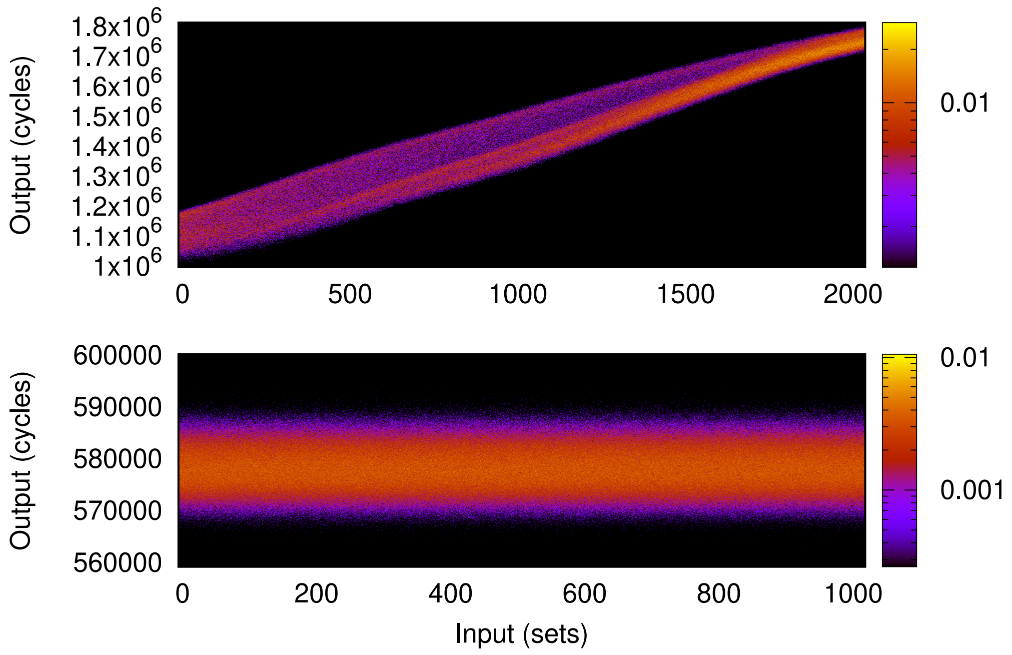


Figure 6.24: The L2 cache covert channel on Sabre (Arm Cortex-A9). Top: without mitigation, $\mathcal{M} = 1{,}929$ mb, $n = 424$, bottom: mitigated with time protection $\mathcal{M} = 0.7$ mb, $\mathcal{M}_0 = 0.7$ mb, $n = 3793$.

the spy can only probe half of the L2 cache due to cache colouring conducted in the time protection scenario.

6.5.2.2 Covert timing channel on domain switching latency

As Requirement 4 stated, the state flushing must be padded to its worst-case latency, because the latency of flushing on-core caches depends on the number of used lines in those caches. Since the flushing invalidates and flushes all those lines, the cost of flushing increases with the increasing number of cache lines. Hence, the latency of flushing can reflect the number of cache lines used by previous running domain, forming a timing channel.

To demonstrate this channel, we create a Trojan that alters the number of cache sets it accesses in each time slice, manipulating the cost of the on-core cache flushes (Section 6.4.6.2), and hence the domain switching latency.

Additionally, we establish a spy that measures its *online time* and *offline time*. The online time is the spy's observed length of its time slice, the uninterrupted execution time; whereas the offline time is the latency between its time slice.

Figure 6.25 shows the system scenario. Both Trojan and spy are executing in their own security domains, sharing a core. Each security domain is created with coloured memory containing user-level processes and a kernel image. To demonstrate a channel, the kernel conducts all the actions listed in Section 6.4.6 during a domain switch, except for padding the configured domain tick length which also covers the on-core flushing latency (Section 6.4.6.3). The cost of flushing on-core states during a domain switch varies

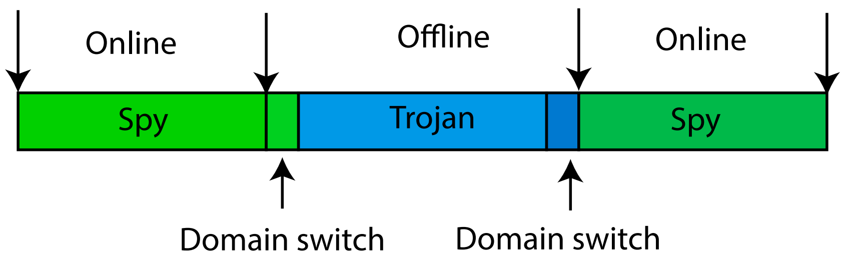


Figure 6.25: The system scenario of the cache-flush channel.

according to the number of cache sets accessed by the Trojan, and therefore the spy's online or offline time.

```
while(1) {
    current = timestamp_counter( );
    //detected a large jump, the begining of the current tick
    if (current - previous > THRESHOLD) {
        online = previous - start;
        offline = current - previous;
        //recording the start of this tick
        start = current;
    }
    previous = current;
}
```

Listing 6.6: The online and offline time measurement.

To detect a preemption, the spy reads repeatedly the time-stamp counter, waiting for a large jump that indicates a preemption. Listing 6.6 contains the pseudo code for the spy's measurements. Online time measures the uninterrupted period, while offline time is the length of the jump.

The top matrix in Figure 6.26 shows the channel created on Haswell. We see that the measured offline time (output) increases slightly while the Trojan enlarges its working set size. In other words, the Trojan successfully modulates the offline time by polluting L1 caches. The channel is effectively mitigated by the configured domain tick length, covering the timing variation in on-core cache flush (bottom in Figure 6.26). Figure 6.27 shows matrices for the original (top) and mitigated (bottom) online channel on Haswell. On the top matrix, we see that the online time of the spy shortens while the Trojan probes on a larger working set, representing the increased latency for switching from Trojan to spy. The online time becomes stable once the configured domain tick length is enabled (bottom), as the latency of domain switch can no longer shorten the domain tick length enjoyed by the spy.

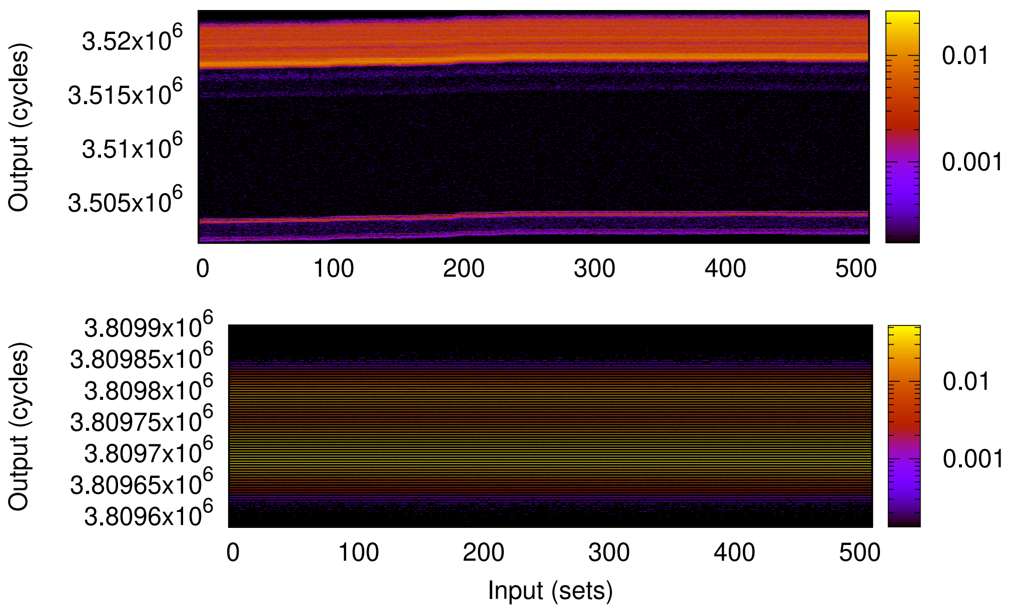


Figure 6.26: The offline time observed by the spy vs. the Trojan's cache footprint, which is caused by a variation of domain switching latency (top). The channel is mitigated by the configured domain tick length (bottom) on Haswell (x86). The original channel has $\mathscr{M} = 7.7\,\text{mb}$, $n = 1,844$, which is mitigated to $\mathscr{M} = 0.3\,\text{mb}$, $\mathscr{M}_0 = 0.3\,\text{mb}$, $n = 7,653$.

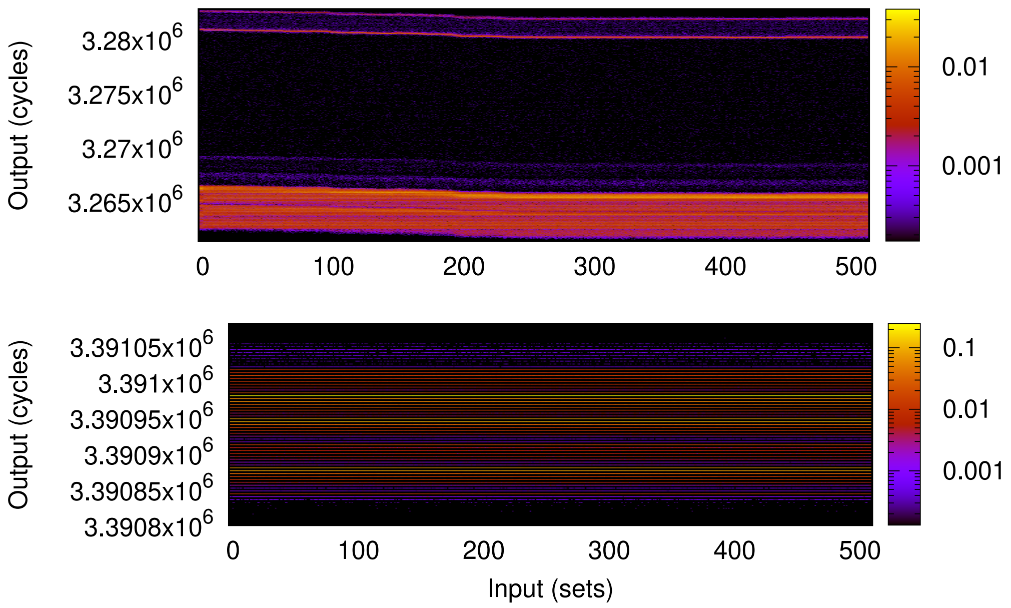


Figure 6.27: The online time observed by the spy vs. the Trojan's cache footprint, which is caused by a variation of domain switching latency (top). The channel is mitigated by the configured domain tick length (bottom) on Haswell (x86). The original channel has $\mathscr{M} = 7.8\,\text{mb}$, $n = 1,844$, which is mitigated to $\mathscr{M} = 0.3\,\text{mb}$, $\mathscr{M}_0 = 0.3\,\text{mb}$, $n = 7,654$.

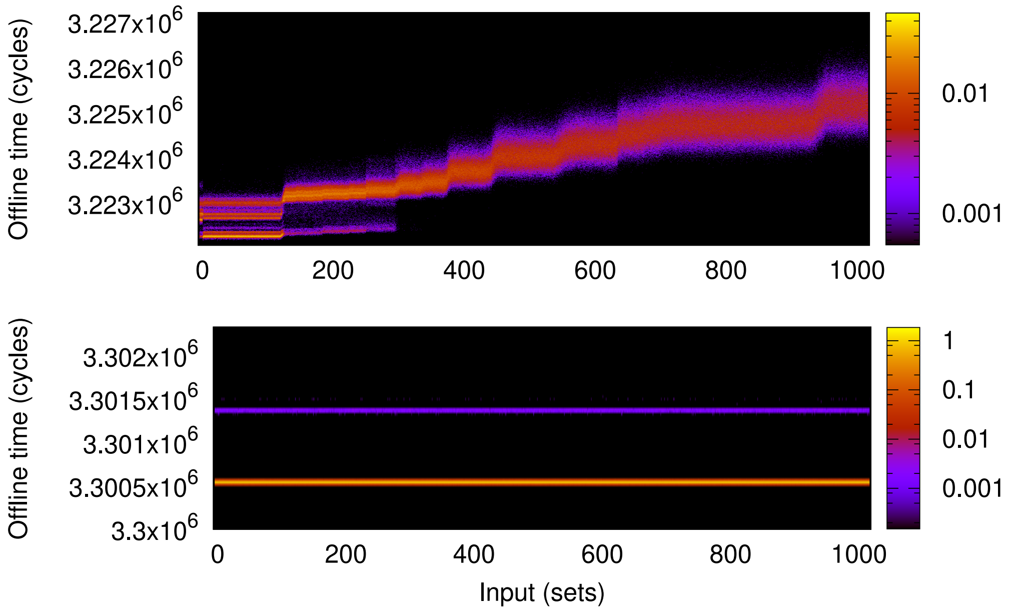


Figure 6.28: The offline time observed by the spy vs. the Trojan's cache footprint, which is caused by a variation of domain switching latency (top). The channel is mitigated by the configured domain tick length (bottom) on Sabre (Arm Cortex-A9). The original channel has $\mathscr{M} = 1,406\,\mathrm{mb}$, $n = 1,828$, which is mitigated to $\mathscr{M} = 256\,\mathrm{mb}$, $\mathscr{M}_0 = 342\,\mathrm{mb}$, $n = 6,719$.

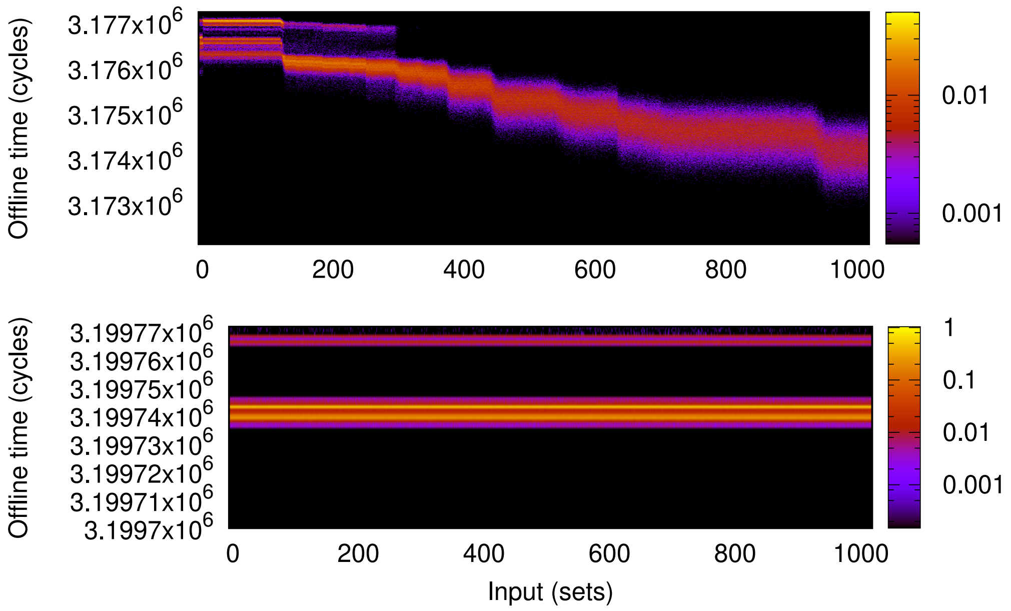


Figure 6.29: The online time observed by the spy vs. the Trojan's cache footprint, which is caused by a variation of domain switching latency (top). The channel is mitigated by the configured domain tick length (bottom) on Sabre (Arm Cortex-A9). The original channel has $\mathscr{M} = 1,406\,\mathrm{mb}$, $n = 1,828$, which is mitigated to $\mathscr{M} = 194\,\mathrm{mb}$, $\mathscr{M}_0 = 239\,\mathrm{mb}$, $n = 6,719$.

Similarly, both offline and online channels (the top matrix in the corresponding figure) is mitigated (the bottom matrix in the corresponding figure) by the configured domain tick length on Sabre, as shown in Figure 6.28 and Figure 6.29.

Table 6.5 summarises the result for MI evaluation. The domain tick length is configured as the sum of the length of a domain tick and the padded domain switch latency. The padded domain switch latency is 58.8 μs on Haswell and 62.5 μs on Sabre. We configure the length of the padding according to the cost of domain switch cost measured in Table 6.8, about twice the cost measured on corresponding platforms. We have only configured the padding time as an upper bound of the cost, which can be further optimised.

Platform	**Timing**	**No pad**	**Protected ($\mathscr{M}_0$)**
Haswell (x86)	Online	7.8	0.3 (0.3)
pad = 58.8 μs	Offline	7.7	0.3 (0.3)
Sabre (Arm Cortex-A9)	Online	1,406	194 (239)
pad = 62.5 μs	Offline	1,406	256 (342)

Table 6.5: Channels resulting from cache-flush latency (mb) without and with time protection.

6.5.2.3 Covert timing channel via a shared kernel image

As stated in Requirement 2, the shared kernel image can be used as a timing channel, even though all security domains, including user-level threads and kernel data (e.g., TCBs or page tables), are partitioned on physically-indexed caches with cache colouring (Section 2.6.3). The shared kernel image can be exploited as a timing channel in a similar way to user-level libraries shared between domains. Note, this attack scenario already prevents the attack demonstrated by van Schaik et al. [2018], since page tables are automatically coloured as part of the kernel data on seL4.

We implement the shared kernel image attack on seL4 with a user-level cache-colouring memory allocator. All other time protecting mechanisms (flushing and padding) are in place. The initial thread creates two threads, a Trojan and a spy, with disjoint cache colours. The two threads share nothing but a kernel image. Then, the initial thread suicides, leaving the two attacking threads as the only runnable threads in the system. The system tick length is 1 ms.

The attack explores contention on LLC between the Trojan and spy, by triggering kernel services. Figure 6.30 demonstrates the attack scenario. The shared kernel image uses all cache colours, including those owned by the spy. Hence, the Trojan can create contention on the spy's cache sets, by calling into the kernel. In each system tick, the Trojan, acting as a sender, sends information by triggering seL4 system calls, whereas the spy, acting as a receiver, measures the probing cost on the coloured cache sets that the kernel uses for serving system calls. To clarify, the spy can only probe on the cache sets used by the kernel that are located in its own share of the cache. Both the Trojan and spy detect a system tick by probing on the time-stamp counter, waiting for a large jump which represents a preemption, as shown in Listing 6.7.

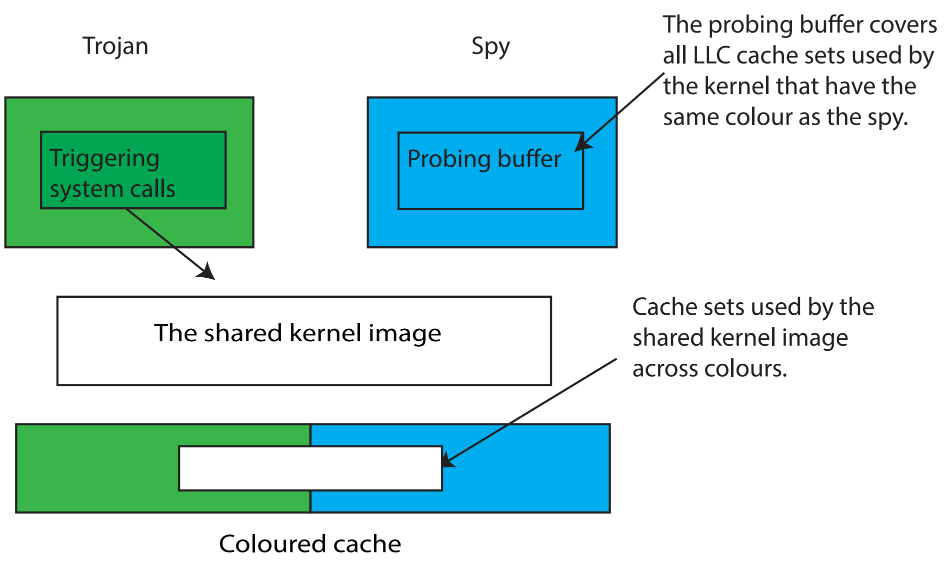

Trojan
Spy
The probing buffer covers all LLC cache sets used by the kernel that have the same colour as the spy.
Triggering system calls
Probing buffer
The shared kernel image
Cache sets used by the shared kernel image across colours.
Coloured cache

Figure 6.30: The Trojan sends information to the spy via a shared kernel image, even though both of them are coloured.

```
        newTimeSlice( ) {
             /*read time-stamp counter*/
            previous = timestamp_counter( );
           for ( ; ; ) {
               current = timestamp_counter( );
               /*detected a large jump, the begining of the current tick*/
               if (current - previous > TS_THRESHOLD)
                   return;
               previous = current;
           }
        }
```

Listing 6.7: Detecting a system tick by probing on the time-stamp counter.

Before conducting the attack, the spy first creates a probing buffer with the PRIME+PROBE technique [Liu et al., 2015; Osvik et al., 2006; Percival, 2005]. The spy identifies cache sets used by the kernel from comparing cache misses on each cache set in its own share of the cache, before and after executing a system call. If the number of cache misses increases, the selected caches are also highly likely used by the kernel. With this method, the spy gradually establishes a probing buffer that covers all cache sets potentially used by the kernel locating on its share of the cache.

When a new time slice is detected, the Trojan encodes a random sequence of symbols from the input set I, by triggering the kernel services.

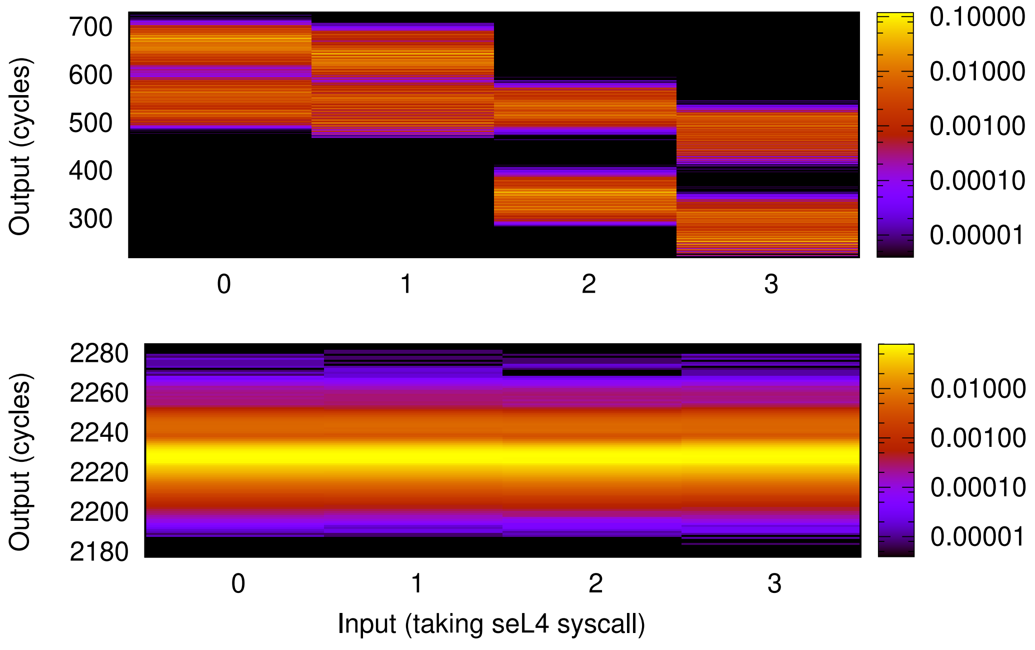


Figure 6.31: Kernel covert timing channel on Haswell (x86), with coloured userland (top) and full time protection (bottom). The original channel has $\mathcal{M} = 800\,\text{mb}$, $n = 255,790$, which is closed to $\mathcal{M} = 0.3\,\text{mb}$, $\mathcal{M}_0 = 0.0\,\text{mb}$, $n = 255,790$.

On the Haswell platform, the Trojan encodes information from $I = 0,1,2,3$: `seL4_Signal` for 0, `seL4_TCB_SetPriority` for 1, `seL4_Poll` for 2, and being idle for 3. Moreover, the spy conducts probing on each time slice, returning the cost of probing the buffer as the output.

The Figure 6.31 (top) shows the channel matrix resulting from the attack on Haswell. From the channel matrix, we can clearly observe the cost of probing (output symbols) being affected by the usage of kernel services (input symbols). As the brighter colours indicate a higher probability, the spy is more likely to take 475–729 cycles on probing while the Trojan called `seL4_Signal` (input 0), 462–724 for `seL4_TCB_SetPriority` (input 1), and 284–595 for `seL4_Poll` (input 2). Quantifying the leakage gives us the MI estimation as $\mathcal{M} = 800\,\text{mb}$.

The time protection mechanism defeats the channel (bottom in Figure 6.31), as the kernel image is no longer shared between the Trojan and spy. For creating this scenario, the initial thread clones two kernel images using coloured memory, and assigns each coloured partition with the corresponding kernel image. Once scheduled, threads within a partition can only invoke their own kernel, and hence are completely separated on cloned kernel sections. Moreover, the kernel deterministically visits the remaining global data shared between kernel images during a domain switch (Section 6.4.6.4), hence leaving a deterministic cache footprint (not related to the previous running domain). With the kernel clone, the MI evaluation is $\mathcal{M} = 0.3\,\text{mb}$ ($\mathcal{M}_0 = 0.0\,\text{mb}$). Moreover, we repeat the test five times, and cannot observe a consistent trend on channel matrices. Thus, the evaluated MI is highly likely due to hardware noise.

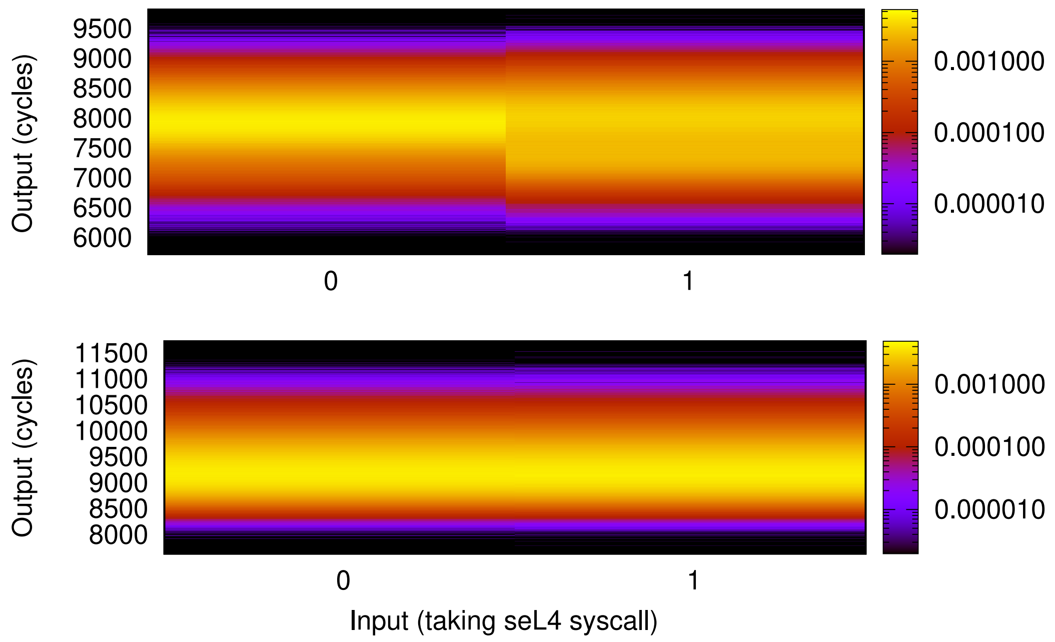


Figure 6.32: Kernel covert timing channel on Sabre (Arm Cortex-A9), with coloured userland (top) and full time protection (bottom). The original channel has $\mathscr{M} = 20$ mb, $n = 511,151$, which is closed to $\mathscr{M} = 0.0$ mb, $\mathscr{M}_0 = 0.0$ mb, $n = 511,531$.

In addition, we implement a similar channel on the Sabre platform, shown in Figure 6.32. On the Sabre platform, the Trojan encodes information from $I = 0, 1$: being idle for 0, and invoking kernel services with three system calls (`seL4_Signal`, `seL4_TCB_SetPriority`, and `seL4_Poll`) for 1. Compared to the channel created on Haswell, the encoding scheme is different on the Sabre platform because of the microarchitectural differences: we first implement the same attack as the one on Haswell, the Trojan invoking one system call in each send, but notice a weak signal. The reason for the weak signal is the small cache footprint left by the shared kernel image being hidden from the aggressive 2-level data prefetcher (Section 2.3.3). The Spy's probing buffer for detecting a system call is prefetched before accessed; hence the cost of probing the buffer cannot fully reveal the actual cache misses. Therefore, we enhance the channel by making the Trojan invoke all three system calls in each system tick, in order to enlarge the kernel footprint left on the L2 cache. We did not try to improve the channel leakage with more sophisticated encoding schemes, due to our main focus on only demonstrating a channel.

In the top matrix in Figure 6.32, we can observe that the probing time measured by the spy is more densely distributed on 7500–8500 while the Trojan sends "0" than sending "1", showing the timing channel. With time protection, the distribution disappears as shown on the bottom matrix. The MI evaluation for the unmitigated channel is $\mathscr{M} = 20$ mb while the kernel image is shared between coloured partitions, which is completely mitigated by time protection ($\mathscr{M} = 0.0$ mb, $\mathscr{M}_0 = 0.0$ mb).

6.5.2.4 Covert timing channel on a shared Interrupt

We evaluate interrupt partitioning (Section 6.4.5) with a covert timing channel based on a timer interrupt. Figure 6.33 shows the attack scenario. The Trojan and spy execute on the same core, with a 10 ms system tick. To send information, the Trojan programs the timer to fire while the spy is running. The timer interrupt breaks the system tick in which the spy executes into two parts, and the length of each part is related to the configuration of the timer. To receive information, the spy measures its online time, the uninterrupted execution time, which is correlated with the timer.

```
while(1) {
    newTimeSlice( );
    //polling until received the pending IRQ notification
    do {
        seL4_Poll(timer, &message);
    }  while(!message);
    handle_timer_irq( );
    //timer latency: 11-19 (ms)
    latency = random_latency( );
    //set the timer
    set_timeout(timer, latency * NS_IN_MS);
}
```

Listing 6.8: The Trojan programs the timer interrupts in every system tick.

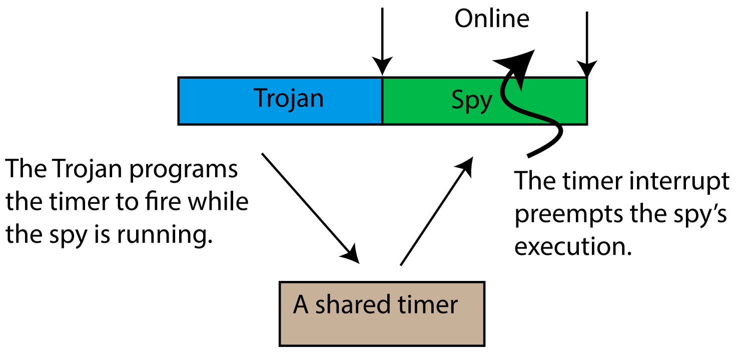

Online
Trojan
Spy
The Trojan programs the timer to fire while the spy is running.
The timer interrupt preempts the spy's execution.
A shared timer

Figure 6.33: The Trojan sends information to the spy via a shared timer interrupt.

Listing 6.8 shows the pseudo code for the Trojan to program the timer interrupt. In each system tick, the Trojan first acknowledges any pending timer interrupt. Then, it programs the timer with a latency between 11–19 ms, greater than the system tick (10 ms). Hence, the timer will be triggered at the next system tick while the spy is running. When the timer is triggered, the spy can detect changes on its online time (Listing 6.6), as the kernel has to handle the interrupt before resuming the spy's execution. In other words, the system tick of the spy breaks into two parts by being interrupted, resulting in two measurable online periods in a system tick.

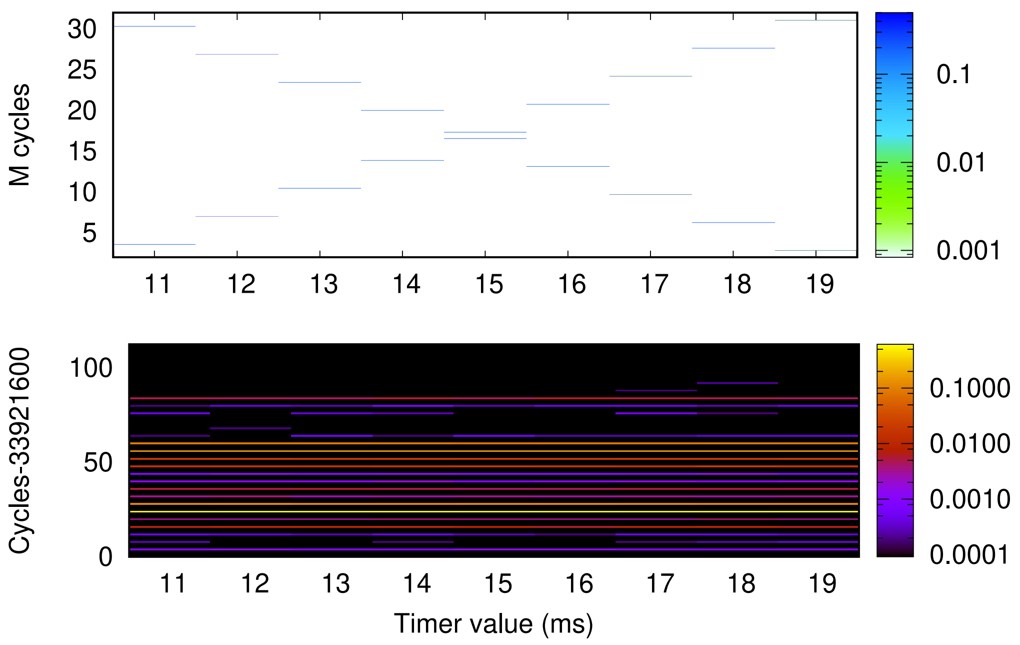


Figure 6.34: The online time observed by the spy vs. the timer interrupt configured by the Trojan, resulting in observing interrupted timer tick (top) which is mitigated interrupt partitioning (bottom) on Haswell (x86). The original channel has $\mathscr{M} = 902$ mb, $n = 10,860$, which is mitigated to $\mathscr{M} = 0.5$ mb, $\mathscr{M}_0 = 0.7$ mb, $n = 11,029$.

Platform	**Shared**	**Partitioned ($\mathscr{M}_0$)**
Haswell (x86)	902	0.5 (0.7)
Sabre (Arm Cortex-A9)	444	42 (68)

Table 6.6: MI (mb) of the interrupt channel (shared), mitigated with time protection (IRQ partitioning).

Table 6.6 summarises results of MI evaluation for original and mitigated channels. Original channels are $\mathscr{M} = 902$ mb on Haswell, and $\mathscr{M} = 444$ mb on Sabre, which are mitigated by time protection (IRQ partitioning) as demonstrated by the corresponding channel matrices and MI evaluation, $\mathscr{M} = 0.5$ mb ($\mathscr{M}_0 = 0.7$ mb) on Haswell, and $\mathscr{M} = 42$ mb ($\mathscr{M}_0 = 68$ mb) on Sabre.

6.5.2.5 Cross-core LLC side channel

For cross-core attacks in the cloud scenario (Section 4.1.2), our threat scenario only considers the side channel on core-shared LLC. We reproduce the cross-core side channel attack (Section 3.2.3.2) of Liu et al. [2015] on GnuPG version 1.4.13 on the Haswell platform, in order to test our time protection mechanism on preventing LLC-based cross-VM attacks.

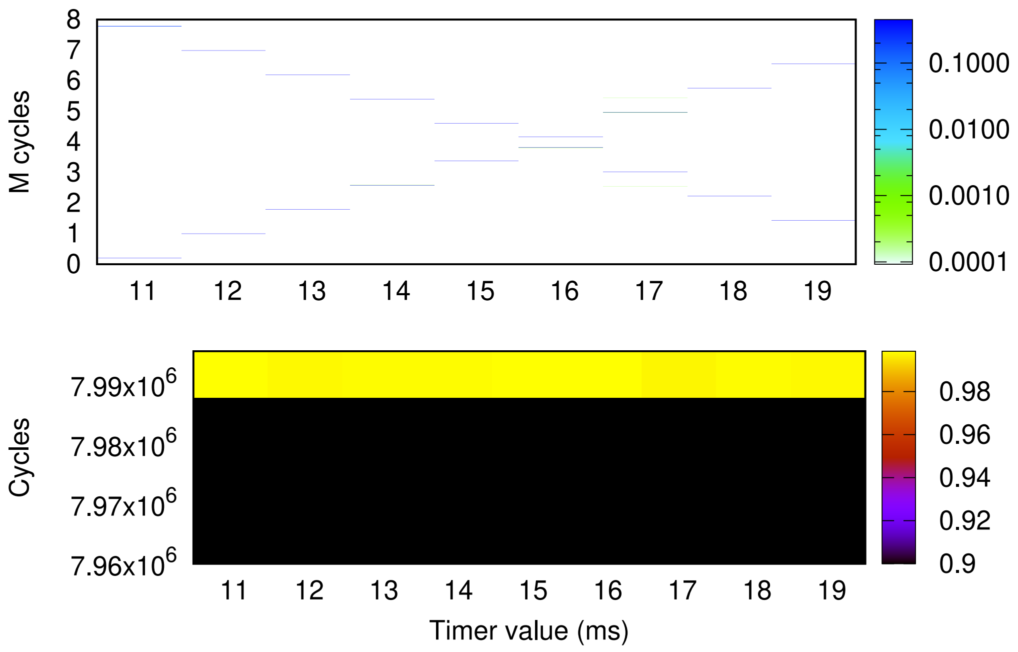


Figure 6.35: The online time observed by the spy vs. the timer interrupt configured by the Trojan, resulting in observing interrupted timer tick (top) which is mitigated interrupt partitioning (bottom) on Sabre (Arm Cortex-A9). The original channel has $\mathcal{M} = 444$ mb, $n = 10,902$, which is mitigated to $\mathcal{M} = 42$ mb, $\mathcal{M}_0 = 68$ mb, $n = 10,976$.

The original attack created by Liu et al. is a cross-VM attack: an attacking VM conducts the PRIME+PROBE attack on the core-shared LLC, in order to break the private key used in an ElGamal server hosted in the victim VM. The attack, originally performed in a clean lab environment, was later demonstrated on an Amazon EC2 cloud by İnci et al. [2016].

The attack targets the square-and-multiply implementation for the modular exponentiation in the decryption process of the ElGamal. In this attack, an attacking thread probes cache sets on the LLC, learning the cache usage of the ElGamal decryption from a victim thread concurrently running on the other core. A successful attack reveals the secret key through the victim's cache activity captured by the probing.

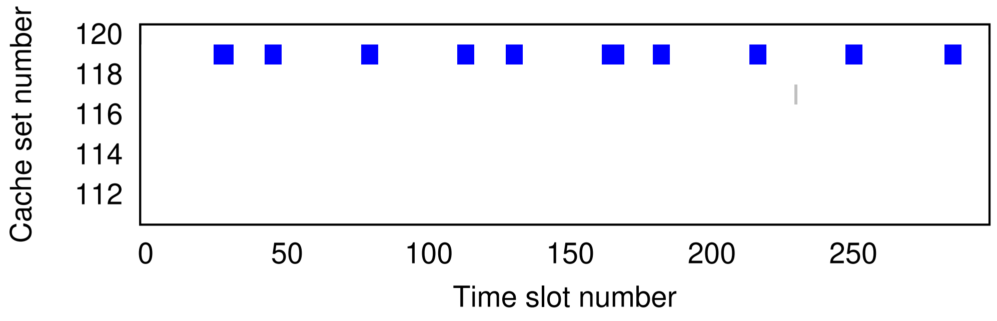


Figure 6.36: Unmitigated concurrent LLC side-channel attack on Haswell (x86). The pattern in blue shows the victim's cache footprint detected by the spy.

To create this attack, we use two processes that are currently executing on different cores on the Haswell platform. The victim process iteratively decrypts a file, whereas the spy process conducts the PRIME+PROBE attack implemented by the Mastik toolkit [Yarom, 2016]. After each round of probing, the spy examines its record, the number of cache misses in each cache set, searching for patterns corresponding to the use of the square function. The Figure 6.36 demonstrates the cache activity captured by PRIME+PROBE attack from the spy. On cache set number 119, the sequence of blue dots represents invocations of the square function, which are separated by intervals of varying lengths, encoding the secret key. The long interval encodes bit "1", while the short interval encodes bit "0".

Our time protection mechanism closes the channel. As a result, the spy cannot detect any cache usage patterns of the victim. We only implement this attack on the Haswell platform, as the Sabre (Arm Coretex-A9) is not relevant in the cloud scenario. However, Section 6.5.2.1 already demonstrates that time protection is effective at preventing the covert channel on the L2 cache on the Sabre platform, due to cache partitioning.

6.5.3 Performance

To understand the impact on system performance, we evaluate time protection with an IPC microbenchmark (Section 6.5.3.1), a microbenchmark of the cost of domain switching (Section 6.5.3.2), a microbenchmark of the cost of kernel cloning and destruction (Section 6.5.3.3), a system benchmark of the cost of cache colouring (Section 6.5.3.4), and a system benchmark of the impact of domain switching (Section 6.5.3.5).

6.5.3.1 IPC microbenchmark

We evaluate the impact of time protection by measuring the cost of cross-address-space message-passing IPC. The IPC operation is one of the most important performance tests for microkernels, as the highly optimised IPC path can easily reflect any performance impact introduced by the new kernel feature. We use an IPC microbenchmark to examine the baseline cost of our mechanisms, even though the cross-domain IPC is an artificial scenario in a strictly partitioned system as it does not use a fixed time slice or time padding (which would defer IPC delivery to the partition switch).

	Haswell (x86)		**Sabre (Arm Cortex-A9)**	
Version	**Cycles**	**Overhead**	**Cycles**	**Overhead**
original	381	-	344	-
colour-ready	386	1%	391	14%
intra-colour	380	0%	395	15%
inter-colour	378	-1%	389	13%

Table 6.7: IPC performance microbenchmarks.

Table 6.7 summarises the results. The *colour-ready* refers to a kernel that supports time protection without using it. The *intra-colour* measures IPC between threads within a domain, while the *inter-colour* represents the cost of IPC between two threads across different coloured security domains, thus different kernel images. The inter-colour test does not use a fixed time slice or cache flushing, which would deliberately defer the IPC delivery until the end of padded partition switch. Although it is an artificial configuration, the test gives a concrete estimation of the cost of switching between coloured kernel images, representing the baseline cost of the kernel clone mechanism. We report the mean from 30 runs, all relative standard deviations are less than 1%. We are not using the mainline kernel [seL4, 2019], hence results are not comparable with results listed on the seL4 web site [seL4].

The time-protection mechanism introduces negligible overhead on Haswell, as IPC costs are within 1% of the baseline for all tests cases. However, supporting the kernel clone mechanism introduces 14% overhead on the Sabre platform. The reason is that the baseline kernel uses global mappings to map the kernel's virtual address space, and those mappings are never evicted from TLB as locked entries. With kernel clone, there is no default mapping for the kernel's virtual address space, and thus the kernel cannot use global mappings. This creates pressure on TLB entries.

The effect is more pronounced on the Sabre platform when compared to more advanced Arm processors, such as the Arm Cortex-A53, as TLBs on the Arm Cortex-A9 processor have smaller associativities. There are two 1-way L1 TLBs and a 2-way L2 TLB on the Arm Cortex-A9 processor. Hence, the cross-address-space IPC test suffers from an increasing number of conflict misses on TLBs. However, there is no overhead from using per-partition kernel images, beyond this architectural limitation that poorly supports multiple kernel mappings. A more advanced Arm processor would have a higher associativity on TLBs, such as the 4-way L2 TLB on the Arm Cortex-A53 processor (Table 5.1). Thus, we expect this overhead to be significantly reduced on more recent processors.

6.5.3.2 Domain switching cost

With time protection, we expect that the cache flushing cost (Table 6.3) dominates the domain switching latency. To verify this, we evaluate the domain switching latency for cache-based attack workloads. Specifically, we select three attack scenarios from Section 6.5.2.1, including attacks on L1-D, L1-I, and L2. For the L3 test on Haswell, we use the same test as Table 6.3 for measuring cost of switching from a domain whose working-set size equals the size of the L3 cache. We measure the time taken to switch from the probing conducted by the spy to an idle domain. For all tests, spy and idle domains are partitioned on caches by coloured security domains, and time protection is fully enabled except for padding the length of domain ticks. We use this setup to represent realistic defence scenarios.

Platform	**Mode**	**Idle**	**L1-D**	**L1-I**	**L2**	**L3**
Haswell	Raw	0.18	0.19	0.22	0.23	0.5
(x86)	Full flush	271	271	271	271	271
	Protected	30	30	30	30	30
Sabre	Raw	0.7	0.8	1.2	1.6	N/A
(Arm Cortex-A9)	Full flush	414	414	414	414	N/A
	Protected	27	27	27	31	N/A

Table 6.8: Cost (μs) of switching away from a domain running various cache probing (spy).

The results are summarised in Table 6.8, for measurements on the original seL4 (raw), seL4 with time protection (protected), and seL4 with all hardware provided microarchitectural resetting operations (full flush) listed in Section 5.3.2. For the L3 test on Haswell and the L2 test on Sabre, the measurements on the original seL4 have bimodal distributions with relative standard deviations of 18% for Haswell and 25% for Sabre, hence we report median values. The reason of the bimodal distribution is that the domain switch execution either hits a fastpath or a slowpath on hardware, depending on state of the CPU (e.g., pipeline, and caches). For all the other tests, we report the mean for 320 runs, where all relative standard deviations are less than 3% for Haswell and 1% for Sabre.

By examining the result in Table 6.8, we first observe that the latency depends on the workload for the orginial seL4: the larger the probing buffer, the higher the cost. However, the protected system has no such dependency, as a result of all the determinism provided by our time protection mechanisms. Secondly, the latencies for the full flush scenario match the direct flushing cost measured in Table 6.3. For time protection, the switching latency is only slightly higher than the cost of direct L1-flushes listed in Table 6.3, which confirms the domination of the L1-flushing latencies in a domain switch.

Our implementation of time protection introduces significantly less overhead than the hardware provided resetting operations (full flush), even though it is just as effective at mitigating timing channels, except for cases resulting from the lack of targeted prefetcher flushing operations on Haswell and Sabre, as discussed in Section 6.5.2.1. For a system configured with 10 ms time slice, the relative overhead of a full flush would be about 3% on Haswell, and 4% on Sabre, whereas time protection introduces only about 0.3% relative overhead on both processors.

6.5.3.3 Kernel cloning and destruction cost

To understand the overhead of managing cloned kernel images, we evaluate the cost of cloning and destroying a kernel image created from coloured frames on seL4. The memory overhead of cloning a kernel image is 224 KiB on Haswell (x86) and 120 KiB on Sabre

(Arm Cortex-A9). The memory requirement is for a single-core system, and requires extra kernel stacks to support added cores with the cost of 4 KiB per core.

Table 6.9 summarises the cost of cloning and destroying a cloned kernel image for both Haswell and Sabre platforms.

To measure the cost of cloning, we measure the latency of conducting the kernel clone system call (`seL4_KernelImage_Clone`) by the initial thread. As mentioned previously (Section 6.4.2), the system call clones a kernel image (the KernelImage object) with memory (KernelMemory objects) provided as parameters of the system call. All KernelImage and KernelMemory objects are created from coloured memory, owning 50% of colours on the L2 cache, in order to simulate the system scenario for creating a partitioned system. We measure the cost on a single-core system; adding support for an extra core only requires creating the mapping for the extra kernel stack, which is just adding a translation in the kernel window. The cost of cloning is 79 μs on Haswell and 608 μs on Sabre, which represents the total cost of creating a kernel window mapping on the KernelImage, the root page directory of the kernel's address space, and cloning kernel sections into kernel memory, KernelMemory objects.

We measure the cost of the frequently executed fork and exec system calls on Linux using the LMbench [McVoy and Staelin, 1996]. The fork and exec system calls are always used together for launching a program on Linux. We run this benchmark to have a relative comparison with the cost for kernel clone. Table 6.10 listed the results. On Haswell, executing the fork system call costs 53 μs, and executing the exec system call costs 204 μs. The total cost of executing the two system calls is 257 μs on Haswell. On Sabre, executing the two system calls cost $4,298$ μs in total: fork 1,099 μs, exec 3,199 μs. On both platforms, cloning a kernel image costs much less than executing the fork+exec.

Platform	**clone**	**destroy**			
		KImage	KMemory	num. KM	total
Haswell	79 (0.3%)	0.59 (10.18%)	0.17 (23.60%)	55	10
Sabre	608 (4.1%)	67.45 (1.68%)	1.67 (21.14%)	29	116

Table 6.9: Cost of cloning (μs) on seL4 with coloured memory. Numbers in parentheses denote relative standard deviation. Reporting mean for 10 runs if the relative standard deviation is less than 4.1%, otherwise, reporting the median due to the bimodal distributions caused by executing on a fast path and a slow path on hardware while testing.

As discussed in Section 6.4.7, our design handles deletion of KernelImage and KernelMemory objects separately, in order to be consistent with the existing capability management model on seL4. Therefore, we measure the cost of deleting the representation of a kernel image, the KernelImage object, as well as the average cost of deleting KernelMemory objects used by that kernel image. The total cost of deleting a kernel image is the cost of deleting all objects used by the kernel image, including a KernelImage object and all of its KernelMemory objects.

Platform	fork	exec	total
Haswell	53 (6%)	204 (0.5%)	257
Sabre	1,099 (3%)	3,199 (7%)	4,298

Table 6.10: Cost of fork and exec system calls on Linux (μs). Numbers in parentheses denote standard deviation values. Reporting mean for 4 runs on Sabre and 6 runs on Haswell.

On Haswell, performing object deletion costs 0.59 μs for the KernelImage object, and 0.17 μs for each KernelMemory object. The total cost of deleting all objects used by the kernel image is the cost of deleting the KernelImage object and 55 KernelMemory objects used for cloning kernel sections (code, read-only and stack sections), which is 10 μs on Haswell. On Sabre, performing a kernel image deletion costs 116 μs, which is the total of deleting its KernelImage object (67.75 μs) and 29 KernelMemory objects (1.67 μs for deleting each object). As with the cloning cost, the total cost for deletion is calculated based on the memory requirement for a single-core machine; adding an extra core requires deleting an extra KernelMemory object used as the kernel stack on that core, which costs 0.17 μs on Haswell and 1.67 μs on Sabre.

6.5.3.4 The cost of cache colouring

We evaluate the cost of cache colouring (Section 2.4), as time protection uses this technique for partitioning physically-indexed caches. Cache colouring replaces the dynamic partitioning (allocation of cache sets) performed by hardware with a static partitioning (assigning cache sets to coloured frames) that is controlled by the OS. It is expected that performing cache partitioning with cache colouring can lead to less optimal usage of the cache, hence introducing performance degradation, a well-understood tradeoff. However, static partitioning also leads to more predictable performance [Kessler and Hill, 1992; Liedtke et al., 1997; Lynch et al., 1992], as each application runs on their own share of the cache. Recently, cache colouring has also been proposed as a method for improving system performance [Han et al., 2018; Noll et al., 2018].

Nonetheless, partitioning physically-indexed caches as part of mandatory security enforcement can cause some performance degradation on average, especially if a cache is shared between one application with a large footprint and another application with a small footprint. To test the effect of owning different portions of the cache, we test a single benchmark, Splash-2, with two configurations on cache colouring, 50% and 75% of colours of the L2 cache.

The Haswell platform contains three levels of caches where L1 and L2 caches (256 KiB) are core-private, and the L3 cache (8 MiB) is an external cache. We colour the benchmarking thread according to the L2 cache colours, because it also implicitly colours the L3 cache (Section 6.5.1). The alternative option would be to only colour the L3 cache, and conduct a flushing operation on the L2 cache during a domain switch by paying the increased domain-switching latency. However, the x86 architecture does not support a targeted L2 cache flush; nor does it provides any documentation on the implementation of cache line replacement policies, hence we cannot test the alternative option.

There are two levels of caches on Sabre, core-private L1 caches, and an external L2 cache (1 MiB). We conduct the cache colouring according to the L2 cache colours.

Time protection on seL4 is a research prototype, which lacks a complete implementation of the portable operating system interface (POSIX) that is expected by most system benchmarks, such as the SPEC benchmark. Hence, we port the easily portable Splash-2 benchmark [Woo et al., 1995] to the seL4 system, except for *volrend* due to its Linux dependencies on processing tagged image file format (TIFF) files. The Splash-2 benchmark is obviously quite dated, but for our purpose we only need a system benchmark that exercises the coloured cache. To achieve this, we configure the Splash-2 benchmark with running parameters to consume 220 MiB of heap and 1 MiB of stack, which is a large memory consumption for running on half of the L2 cache (128 KiB on Haswel, and 512 KiB on Sabre).

The overhead of cache colouring is summarised in Figure 6.37 for Haswell and Figure 6.38 for Sabre. We evaluate the cache colouring with and without the kernel clone mechanism, to demonstrate the extra cost of supporting the benchmarking thread with a coloured kernel image if any. For all tests, we report the mean of 10 repeated single-threaded runs, with relative standard deviations less than 3%, as well as the geometric mean across the suite. During the test, the benchmarking thread is the only runnable user-level thread in the system.

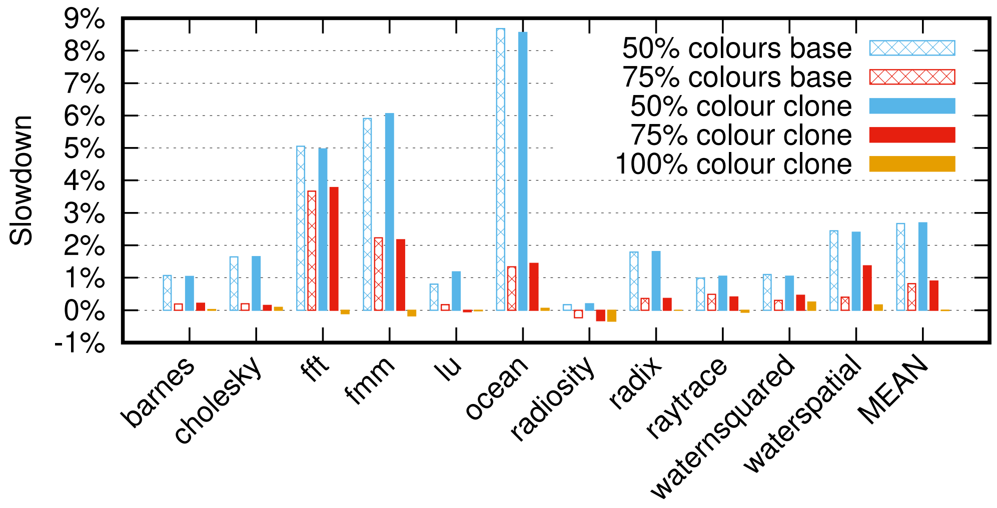


Figure 6.37: Slowdowns of Splash-2 benchmarks against baseline kernel without partitioning for Haswell (x86) and geometric mean. Benchmarks are run as the only process on the system. The "base" cases use the standard kernel with reduced cache. The "cloned" cases run the benchmark on a cloned kernel with reduced cache, the "100% colours clone" case uses an unpartitioned cache like the baseline.

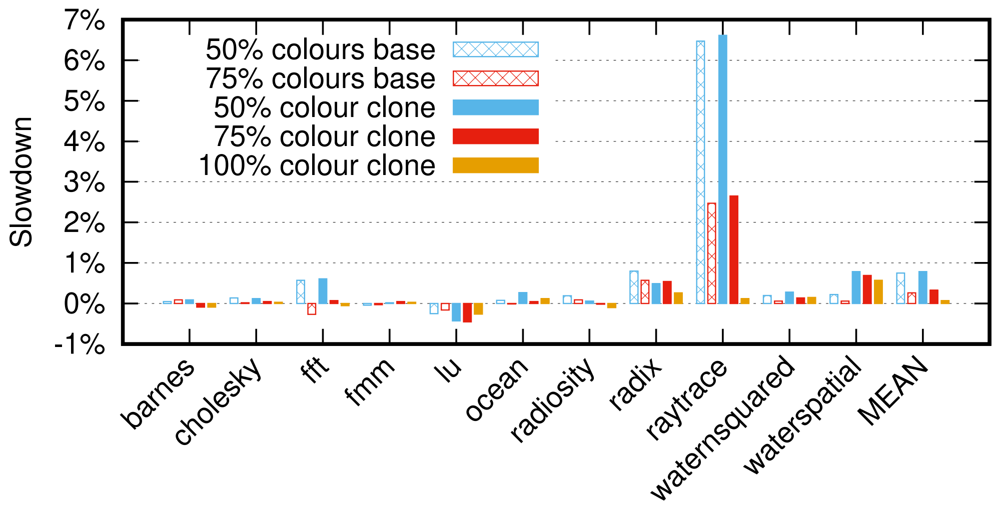


Figure 6.38: Slowdowns of Splash-2 benchmarks against baseline kernel without partitioning for Sabre (Arm Cortex-A9) and geometric mean. Benchmarks are run as the only process on the system. The “base” cases use the standard kernel with reduced cache. The “cloned” cases run the benchmark on a cloned kernel with reduced cache, the “100% colours clone” case uses an unpartitioned cache like the baseline.

On Haswell (Figure 6.37), having 50% of the L2 cache only slows down the majority of Splash-2 tests by less than 3%. Furthermore, having an increased share of the L2 cache (75%) improves the performance, limiting the overhead to less than 3.5%. Most importantly, the kernel clone mechanism introduces close to zero overhead. The geometric mean is 2.7% slowdown for running on 50% of the L2 cache with cloning, and 0.9% for running on 75% of the L2 cache with cloning.

On Sabre (Figure 6.38), executing with 50% of the cache colours introduces less than 1% of slowdown for most of the Splash-2 benchmarks, except for *raytrace*, which incurs a 6.5% slowdown due to its large working set size. Running on 75% of the colours improves the performance of *raytrace*, with only 2.5% slowdown. Overall, kernel cloning introduces almost no performance penalty, except for *waterspatial* with less than 0.5% overhead, showing the overhead of cloning on a memory intensive workload that does high performance computing. As discussed in Section 6.5.3.1, cloning increases the number of conflict misses on TLBs that have small associativities, like for the Sabre platform. Still, the geometric mean is 0.8% for running on 50% of the L2 cache with cloning, and 0.3% for running on 75% of the L2 cache with cloning.

6.5.3.5 The impact of domain switches

The above evaluation of cache colouring and cloned kernel images does not reveal the effect of increased domain-switching latency resulting from flushing on-core state (Section 6.4.6.2). To evaluate the impact of increased domain-switching latency, we run the Splash-2 benchmark in a security domain sharing the processor with an idle domain. The two domains are partitioned on physically-indexed caches as stated in Section 6.5.3.4. The system tick length is 10 ms. During a domain switch, the kernel conducts all actions listed in Section 6.4.6.6. This benchmark scenario measures the impact of decreased CPU bandwidth from the increased domain-switching latency.

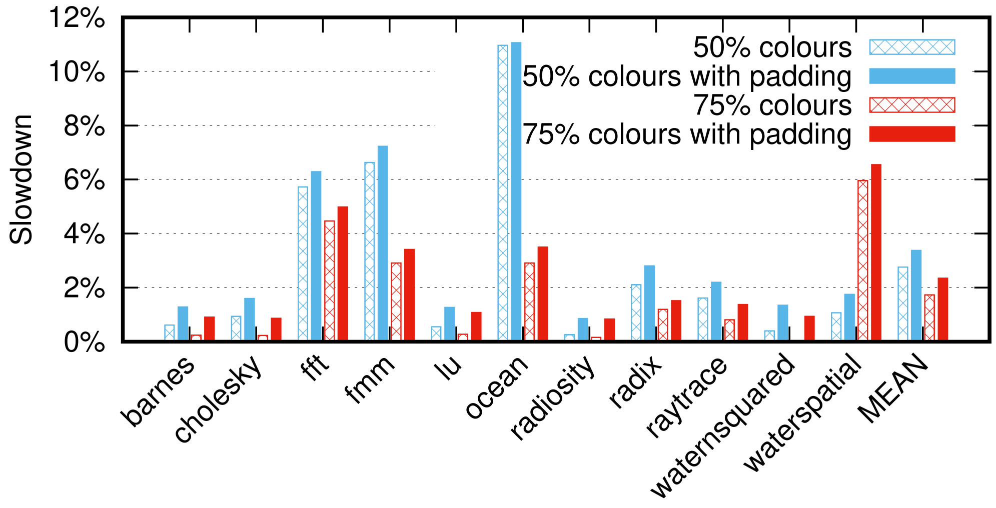


Figure 6.39: Slowdowns of Splash-2 benchmarks of time protection, including the increased domain-switching latency in a time-shared setup, against baseline kernel without partitioning for Haswell (x86) and geometric mean. Benchmarks share the processor with an idle domain. The “colours” cases run the benchmark on a security domain with reduced cache without padding the domain switch latency to the configured length. The “colours with padding” cases pad the domain switch latency to the configured length while running on a reduced cache.

To identify the cost of padding the configured domain tick length (Section 6.4.6.3), we evaluate slowdown resulting from time protection with and without padding. For padded domain ticks, we configure the system to use the tick length configured to prevent the timing channel resulting from domain-switching latency (Section 6.5.2.2). The padded domain switch latency is 58.8 μs on Haswell and 62.5 μs on Sabre, which is the same as configured for preventing the covert channel on the domain switching latency (Section 6.5.2.2). Results are summarised in Figure 6.39 for Haswell, and Figure 6.40 for Sabre. We report the mean of 10 repeated time-shared runs where relative standard deviations are less than 2%. On Haswell, results for all *lu* tests and the *water-spatial* test on unmodified seL4 have 5%–6% relative standard deviations caused by bimodal distributions, hence we report the median. We also show the geometric mean of the slowdown across the suite.

On Haswell, deploying time protection with 50% of the L2 cache introduces less than 3% of slowdown on the majority of Splash-2 tests. The overhead ranges from the lowest, 0.26%, to the highest, 10.96%. Padding the domain tick length introduces a little extra overhead which causes the overhead ranging from 0.86% for the lowest to 11.06% for the highest, adding around 0.6% overhead on top of time protection. Geometric means of having 50% of the L2 cache are 2.8% without padding, and 3.4% with padding. Increasing

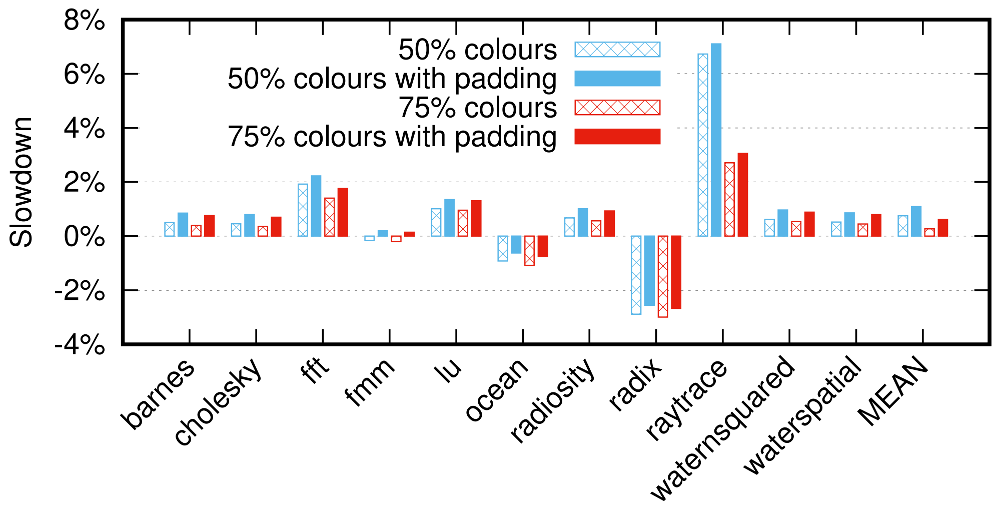


Figure 6.40: Slowdowns of Splash-2 benchmarks of time protection, including the increased domain-switching latency in a time-shared setup, against baseline kernel without partitioning for Sabre (Arm Cortex-A9) and geometric mean. Benchmarks share the processor with an idle domain. The "colours" cases run the benchmark on a security domain with reduced cache without padding the domain switch latency to the configured length. The "colours with padding" cases pad the domain switch latency to the configured length while running on a reduced cache.

cache share to 75% limits the slowdown to below 6% without padding and 6.5% with padding. Geometric means for running on 75% of the L2 cache are 1.7% without padding and 2.4% with padding. For *waterspatial*, having 75% of the L2 cache introduces more slowdown compared to only having 50%, this is an example of timing anomalies where having more cache can potentially cause longer execution time [Lundqvist and Stenström, 1999].

On Sabre, time protection with 50% of the L2 cache introduces slowdown from the lowest, −2.9%, to the highest, 6.7%. For the *radix* test, having less cache improves the performance, which is another example of timing anomalies. Padding the domain switch latency introduces less than 0.4% overhead, causing the slowdown ranging from −2.6% to 7.1%. Geometric means of having 50% of the colours are 0.8% without padding and 1.1% with padding. With 75% of the cache colours, time protection introduces −3.0% to 2.7% slowdown without padding and −2.7% to 3.0% slowdown with padding. Geometric means of having 75% of the colours are 0.3% without padding and 0.6% with padding.

6.5.4 Summary

Our evaluation shows that time protection is generally highly effective at preventing all studied timing channels, including covert channels between threads that are time-sharing a core, and a cross-core side channel between concurrently executing threads. Our studied timing channels cover threats on both confinement (Section 4.1.1) and cloud (Section 4.1.2) scenarios. However, we also observe that present hardware does not provide enough support for resetting on-core resources that cannot be partitioned, leaving the prefetcher as a source of residual channels (Section 6.5.2.1). This finding supports our earlier claim that the current ISA is not sufficient to provide timing security, and that we need an improved hardware-software contract to address this shortcoming (Section 5.5).

Our implementation of time protection in seL4 is low-cost, particularly for running a cloned kernel, except on Sabre (Arm Cortex-A9) due to its low TLB associativity (Section 6.5.3.1). The memory overhead of cloning is also low (Section 6.4.7): 224 KiB on Haswell (x86) and 120 KiB on Sabre. Moreover, kernel image creation and destruction is fully dynamic and affordable, 79 μs on Haswell and 608 μs on Sabre for creation, and 10 μs on Haswell and 116 μs on Sabre for destruction, much less than process creation costs on Linux. Time protection also introduces low overhead on memory intensive benchmarks (Section 6.5.3.5), by only slowing down the Splash-2 benchmark up to 3.4% on Haswell and 1.1% on Sabre, according to the geometric mean across the suite.

Although our prototype is implemented on the seL4 microkernel, we expect that the time protection can be adopted by other kernels, as the idea of dynamically creating and destroying kernel images has no dependency on seL4. Our prototype does not depend on any specific hardware features (e.g., customised hardware caches), hence the evaluation can be a point of reference for a similar implementation on other systems. Additionally, creating or destroying kernel images is only necessary during system initialisation or reconfiguration, which may make it affordable in monolithic kernels such as Linux, even though cloning a monolithic kernel requires populating a larger kernel window (Section 6.4.2) than seL4. Lastly, the cost of flushing on-core caches during a domain switch should also be affordable on monolithic kernels because these caches are small and are highly likely to contain non-reusable cache lines after a switch, as discussed in Section 3.3.4. In other words, the cost of flushing on-core caches is independent of the OS, which should be the same in a monolithic kernel such as Linux.

7 | Conclusion

Microarchitectural timing channels exploit timing variance due to the shared use of caches or other cache-like hardware components. These channels result from contention on hardware resources (e.g., caches) that are functionally transparent to software and designed to maximize average-case performance, generally by exercising the well-established principles of temporal and spatial locality.

The existence of microarchitectural timing channels challenges security enforcement conducted by the OS, and has eluded a comprehensive solution to date. The importance of preventing those channels has been highlighted by recent attacks, including the attack on cryptographic encryption keys between VMs on cloud system [İnci et al., 2016; Liu et al., 2015], as well as a the Spectre attack, which targets the speculative execution engine in the CPU's pipeline [Kocher et al., 2019] and uses timing channel to extract information.

Our work addresses this challenge by providing *time protection* as an OS abstraction, providing temporal isolation analogously to the spatial isolation provided by the established memory protection. Rather than preventing a particular timing channel, we propose a design of OS mechanisms that prevents unauthorized timing information flow between security domains, the first attempt of providing a principled solution for preventing microarchitectural timing channels in the OS. Additionally, our work shows the need of a new, security-oriented, hardware-software contract, to address the shortcoming of insufficient support for preventing timing interference on existing hardware.

7.1 Contributions

To summarise, this work makes the following contributions:

- We design kernel mechanisms for providing time protection for mandatory temporal isolation enforcement in the OS;
- we demonstrate that time protection introduces low overhead and is effective in preventing studied timing channels;
- we observe the lack of support for the complete prevention of timing channels on current hardware, causing uncloseable channels associated with hidden microarchi-

tectural state. We therefore propose an improved hardware-software contract, the augmented ISA, to address this shortcoming.

We propose time protection, a mandatory, black-box kernel mechanism implemented in the seL4 microkernel for preventing microarchitectural timing channels. Time protection combines spatial and temporal partitioning to prevent interference on shared hardware components. Our work provides a new kernel mechanism, kernel image clone, to almost entirely duplicate kernel images with user managed memory. In combination with the memory model in seL4, a privileged user-level thread can deploy cache colouring policies for creating security domains which are supported by dedicated kernel images. Time protection also offers an option of partitioning hardware interrupts by assigning interrupts to security domains. One distinct advantage of this design is that the kernel has no built-in security policy, giving the freedom for any user-level security deployment.

Our evaluation shows that time protection is effective in preventing studied timing channels with small to negligible performance overhead. However, we also observe that the current hardware does not provide enough support for resetting on-core state that cannot be spatially partitioned, thus preventing effective temporal partitioning. This discovery supports our earlier claim that a new software-hardware contract is necessary for providing true security by preventing microarchitectural timing channels [Ge et al., 2018b]. In particular, the hardware needs to support flushing all virtually-addressed state, and provide mechanisms for partitioning concurrently-accessed resources.

Despite hardware limitations, we demonstrate time protection as a general OS abstraction, and an implementation that can easily adapt to any improved hardware changes that provide better security.

7.2 Strength and Limitations

Time protection provides a set of mechanisms for deploying security policies according to the threat scenario. For example, preventing all possible covert channels (Section 4.1.1) in a confinement scenario requires padding the domain switching latency to the worst-case, one of the most expensive operations. In contrast, a cloud platform may only consider deploying cache colouring policy for partitioning security domains on the core-external cache, which is enough for preventing cross-core side channels.

The security domain is a collection of software components and processes which are treated as a single unit by the system's security policy. Consequently, within a domain, there are no restrictions imposed by the security policy, and its internal structure is solely a matter of software-engineering convenience.

Therefore, the overall cost of security is a matter of system structure, particularly the granularity of domains and the amount of useful work performed in a time slice. An extreme example is an interactive program that performs little work between events: each activation would be padded to the full time slice length. On the other hand, compute-bound programs will only observe a small overhead resulting from the increased domain-switching latency.

A system does not need to create and destroy kernel images as frequently as user-level processes. While it is good that the cost of creating and destorying kernel images is very small in seL4, this will not be the case for a monolithic system such as Linux. However, the system can clone as many images as the maximum number of security domains during initialisation, leaving those images as stand-by system resources which will never be destroyed.

Our implementation of time protection is orthogonal to other countermeasures for preventing the recent, speculation-based attacks, such as Spectre [Kocher et al., 2019], Meltdown [Lipp et al., 2018], and Foreshadow [Van Bulck et al., 2018] attacks. Other countermeasures can be easily deployed together with time protection. However, our implementation is efficient for preventing cross-domain Spectre attacks, as time protection includes the IBC mechanism [Intel, 2018c], Intel's hardware mitigation for the Spectre attack, during a domain switch.

The number of available cache colours is a potential bottleneck, limiting the number of supportable security domains. A system can choose to only colour the LLC which offers a much larger number of colours (32 colours on the Haswell platform) than the core-private L2 cache (8 colours on the Haswell platform), especially for the cloud scenario. For the LLC that contains multiple slices [Yarom et al., 2015], the actual number of colours is much higher than only colouring according to the indexing scheme within a slice as described in Section 2.4. For these LLCs, knowledge of the hash function used for mapping cache slices can increase the number of cache colours.

Re-allocating coloured memory between security domains is possible, but with the cost of moving (and copying contents) allocated memory between affected partitions. However, this is inevitable as there is a lack of hardware support for more fine-grained partitioning, which can be improved with better help on hardware. In comparison, partitioning the cache using Intel's CAT mechanism [Intel, d], allocating cache ways to domains, introduces less overhead for re-allocating, but has higher runtime overhead as it reduces associativity.

7.3 Discussion and Future Work

To implement time protection in other OS, the system needs to provide a solution for duplicating kernel images (Section 6.4.1). In particular, cloning requires partitioning kernel data, which can be a challenge for a kernel with a dynamic heap and far more static global data. The advantage of cloning a microkernel, such as seL4, is that its small size makes the memory consumption more affordable than a monolithic kernel, such as Linux, and this is particularly helped by seL4's unique memory management model.

Still, there is no fundamental reason why the same approach could not be done in other systems. For example, the mechanism for cloning kernel images can be easily adopted by multikernels (Section 3.4) such as Barrelfish [Baumann et al., 2009], since the Barrelfish kernel already provides mechanisms for dynamic kernel swapping [Zellweger et al., 2014], which is a clear sign of its capability to decouple the kernel state from the rest of the kernel. Similarly, cloning should also be applicable in exokernels(Section 3.5) which are designed to be lightweight kernels that are mainly responsible for exposing hardware resources to application-level library OSes. Implementing cloning the kernel image in a monolithic kernel, such as Linux, requires supporting the dynamic kernel heap in all cloned kernel images. Possible solutions include preserving a kernel heap region during initialisation, and acknowledging any kernel heap growth in all cloned kernel images at runtime. However, there are other possibilities, such as relocating all kernel services that cause heap growth in user-level library OSes.

Additionally, the two partitioning schemes in time protection, temporal partitioning and spatial partitioning, have no dependency on either hardware or software systems. Therefore, a system can introduce these partitioning schemes in a suitable format. For instance, cache colouring used as spatial partitioning can be implemented in a system's memory allocator, which has already been demonstrated by previous work [Bershad et al., 1994; Cock et al., 2014; Kessler and Hill, 1992; Kim et al., 2012; Liedtke et al., 1997; Shi et al., 2011].

One potential challenge of colouring a kernel image is colouring the kernel heap used for kernel metadata. On seL4, kernel metadata are abstracted as objects that can be created and destroyed at user-level. This feature undoubtedly simplifies the design of colouring kernel heap in seL4: a user-level resource manager creates kernel objects from coloured memory (Section 2.6.3), hence no kernel modification is required. Colouring a kernel image with a dynamic heap can be a challenging task. One potential solution is applying other spatial partitioning schemes to the kernel heap, such as reserving cache ways through hardware provided mechanisms (Intel's CAT, Section 3.3.5).

One possible extension of this work is to explore other usages of the cloning mechanism, for selectively cloning sections in the kernel image. For time protection, the system can configure to clone code and read-only sections but share the static global data section to maintain the consistency of system states (e.g., the scheduling queue). For multikernel systems [Baumann et al., 2009; von Tessin, 2012], each kernel image manages a partition of the system, including CPUs and main memory, thus only requiring cloning the static global data section for managing their system states. We have not yet found other convincing system scenarios for cloning other combination of sections, but they may be worth exploration and future study.

Bibliography

Products formerly Conroe, 2018a. URL https://ark.intel.com/products/codename/2680/Conroe.

Deep dive: Intel analysis of L1 terminal fault, 2018b. URL https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault.

Products formerly Skylake, 2018c. URL https://ark.intel.com/products/codename/37572/Skylake.

Jeffery M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, and Paul D Madland. Method and apparatus for performing load operations in a computer system, December 1997. US Patent 5,694,574.

Jeffery M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, and Paul D Madland. Method and apparatus for blocking execution of and storing load operations during their execution, March 1999. US Patent 5,881,262.

Jeffrey M Abramson, David B Papworth, Haitham H Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, and Paul D Madland. Out-of-order processor with a memory subsystem which handles speculatively dispatched load operations, October 1995. US Patent 5,751,983.

Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Technical Conference*, pages 93–112, Atlanta, GA, US, 1986.

Onur Acıiçmez. Yet another microarchitectural attack: exploiting I-cache. In *ACM Computer Security Architecture Workshop (CSAW)*, pages 11–18, Fairfax, VA, US, 2007. ACM.

Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Proceedings of the 2007 Crytographers' track at the RSA Conference on Topics in Cryptology*, pages 225–242. Springer, 2007a.

Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Asia Conference on Computer and Communication Security (ASIA CCS)*, Singapore, 2007b.

Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 110–124, Santa Barbara, CA, US, 2010. Springer.

Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Crytographers' track at the RSA Conference on Topics in Cryptology*, pages 256–273, San Francisco, CA, US, 2008.

Onur Acıiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 80–91, Vienna, AT, 2007. IEEE.

Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. Cryptology ePrint Archive, Report 2018/1060, 2018. https://eprint.iacr.org/2018/1060.

Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 526–540, San Francisco, CA, May 2013. IEEE.

Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Annual Computer Security Applications Conference*, pages 422–435, Los Angeles, CA, US, December 2016.

AMD. AMD FX processors. URL http://www.amd.com/en-us/products/processors/desktop/fx. Accessed on 12.2018.

Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, Tel-Aviv, Israel, June 2013.

Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *Proceedings of the IEEE Symposium on Security and Privacy*, San Jose, CA, US, May 2015.

Apache. Apache http server benchmarking tool, 2013.

ARM. ARMv8 instruction set overview. URL https://www.element14.com/community/servlet/JiveServlet/previewBody/41836-102-1-229511/ARM.Reference_Manual.pdf. Accessed on 10.2016.

ARM Architecture Reference Manual, ARM v7-A and ARM v7-R. ARM Ltd., April 2008. URL https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf. ARM DDI 0406B.

Cortex-A9 Technical Reference Manual. ARM Ltd., r2p2 edition, 2010. URL http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388f/DDI0388F_cortex_a9_r2p2_trm.pdf. ARM DDI 0388F.

Arm Cortex-A53 MPCore Processor Technical Reference Manual. ARM Ltd., 2016. URL http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500d/DDI0500D_cortex_a53_r0p2_trm.pdf.

Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 520–538, Chicago, IL, US, 2010.

Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determinating timing channels in compute clouds. In *ACM Workshop on Cloud Computing Security*, pages 103–108, Chicago, IL, US, 2010a. IEEE.

Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, Vancouver, BC, 2010b.

Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th International Symposium on Computer Architecture*, pages 48–59, Saint-Malo, France, 2010.

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *ACM Symposium on Operating Systems Principles*, pages 29–44, Big Sky, MT, US, October 2009. ACM.

Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh aah…, just a little bit": A small amount of side channel can go a long way. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 75–92, Busan, KR, September 2014.

Daniel J. Bernstein. Cache-timing attacks on AES, 2005. URL https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

Daniel J Bernstein. Curve25519: new Diffie-Hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228, 2006.

Daniel J. Bernstein and Peter Schwabe. A word of warning. Workshop on Cryptographic Hardware and Embedded Systems'13 Rump Session, August 2013.

Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *Proceedings of the 2nd Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 159–176, Santiago, CL, October 2012.

Brian N. Bershad, D. Lee, Theodore H. Romer, and J. Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, San Jose, CA, US, October 1994. ACM.

Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–35, 2008.

Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, US, 2003.

Milind Bodas, Glenn J Hinton, and Andrew F Glew. Mechanism to improved execution of misaligned loads, December 1998. US Patent 5,854,914.

Alan C. Bomberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*, pages 95–112, Seattle, WA, US, April 1992. USENIX Association.

Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *IEEE Security and Privacy*, pages 987–1004, San Jose, CA, USA, May 2016.

Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 43–57, San Diego, CA, US, December 2008. USENIX.

Benjamin A. Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. *arXiv preprint arXiv:1506.00189*, 2015.

Ernie Brickell. Technologies to improve platform security. Workshop on Cryptographic Hardware and Embedded Systems'11 Invited Talk, September 2011. URL https://www.iacr.org/workshops/ches/ches2011/presentations/Invited%201/CHES2011_Invited_1.pdf.

Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006:52, 2006.

Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *Proceedings of the 15th Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 667–684, Tokyo, JP, December 2009.

David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, pages 1–14, Washington, DC, US, 2003. USENIX.

Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15:412–447, 1997.

Yuriy Bulygin. CPU side-channels vs. virtualization malware: the good, the bad or the ugly. In *ToorCon: Seattle*, Seattle, WA, US, April 2008.

Carlos Cardenas and Rajendra V Boppana. Detection and mitigation of performance attacks in multi-tenant cloud computing. In *1st International IBM Cloud Academy Conference*, Research Triangle Park, NC, US, 2012.

Sanguhn Cha, O Seongil, Hyunsung Shin, Sangjoon Hwang, Kwangil Park, Seong Jin Jang, Joo Sun Choi, Gyo Young Jin, Young Hoon Son, Hyunyoon Cho, Jung Ho Ahn, and Nam Sung Kim. Defect analysis and cost-effective resilience architecture for future DRAM devices. In *Proceedings of the 23rd IEEE Symposium on High-Performance Computer Architecture*, pages 61–72, Austin, TX, USA, February 2017.

Tom Chothia and Apratim Guha. A statistical test for information leaks using continuous mutual information. In *IEEE Computer Security Foundations Symposium*, pages 177–190, Cernay-la-Ville, FR, 2011. IEEE.

Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. A tool for estimating information leakage. In *International Conference on Computer Aided Verification*, pages 690–695, Saint Petersburg, RU, 2013. ACM.

David Cock. Practical probability: Applying pGCL to lattice scheduling. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, pages 1–16, Rennes, France, July 2013.

David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of some timing channels on seL4. In *ACM Conference on Computer and Communications Security*, pages 570–581, Scottsdale, AZ, USA, November 2014. ACM.

Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ECC memory against Rowhammer attacks. In *IEEE Security and Privacy*, San Francisco, CA, USA, May 2019.

Patrick J. Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Protecting data on smartphones and tablets from memory attacks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 177–189, Istambul, TK, March 2015. ACM.

Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 45–60, Oakland, CA, US, May 2009.

Data61. Cross domain desktop compositor, 2017a. https://ts.data61.csiro.au/projects/TS/cddc.pml.

Data61. *seL4 Reference Manual, Version 7.0.0*, September 2017b.

Dorothy. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19:236–242, 1976.

Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–155, 1966.

DoD. *Trusted Computer System Evaluation Criteria*. Department of Defence, 1986. DoD 5200.28-STD.

Valgrind developers. Valgrind. URL http://valgrind.org/. Accessed on 9.2018.

Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX. In *Proceedings of the 26th USENIX Security Symposium*, pages 51–67, Vancouver, BC, Canada, August 2017.

Leonid Domnister, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization*, 8(4), January 2012.

DotCloud. DotClod developer cloud platform. URL https://www.dotcloud.com/. Accessed on 10.2018.

Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security*, 18(1):4, June 2015.

James D Dundas. Repair of mis-predicted load values, March 2002. US Patent 6,883,086.

George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, US, 2002.

George Washington Dunlap, III. *Execution replay for intrusion analysis*. PhD thesis, University of Michigan, 2006.

Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology*, Santa Barbara, CA, US, 1985.

Kevin Elphinstone and Gernot Heiser. From L3 to seL4 – what have we learnt in 20 years of L4 microkernels? In *ACM Symposium on Operating Systems Principles*, pages 133–150, Farmington, PA, USA, November 2013.

Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, US, December 1995.

Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, pages 843–857, Vienna, AT, October 2016.

Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *Proceedings of the 49th ACM/IEE International Symposium on Microarchitecture*, Taipei, Taiwan, October 2016a.

Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization*, 13(1):10, April 2016b.

Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs, 2018. URL https://www.agner.org/optimize/microarchitecture.pdf.

Bryan Ford. Plugging side-channel leaks with timing information flow control. In *Proceedings of the 4th USENIX Workschop on Hot Topics in Cloud Computing*, pages 1–5, Boston, MA, USA, 2012. USENIX.

Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the GPU. In *IEEE Security and Privacy*, pages 195–210, San Francisco, CA, USA, May 2018.

Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: VMM detection myths and realities. In *Workshop on Hot Topics in Operating Systems (HotOS)*, San Diego, CA, US, 2007.

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

Qian Ge, Yuval Yarom, and Gernot Heiser. No security without time protection: We need a new hardware-software contract. In *Asia-Pacific Workshop on Systems (APSys)*, Korea, August 2018b. ACM SIGOPS.

Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: the missing OS abstraction. In *EuroSys Conference*, Dresden, Germany, March 2019. ACM.

Matt Godbolt. The BTB in contemporary Intel chips, February 2016. URL http://xania.org/201602/bpu-part-three.

Michael Godfrey. On the prevention of cache-based side-channel attacks in a cloud environment. Master's thesis, Queen's University, Ont, CA, September 2013.

Michael Godfrey and Mohammad Zulkernine. A server-side solution to cache-based side-channel attacks in the cloud. In *Proceedings of the 6th IEEEInternational Conference on Cloud Computing*, Santa Clara, CA, US, 2013.

Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Christiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, US, February 2017.

Ben Gras, Kaveh Razavi, Herbert Bos, and Christiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *Proceedings of the 27th USENIX Security Symposium*, pages 955–972, Baltimore, MD, US, August 2018. USENIX.

Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Security Symposium*, pages 897–912, Washington, DC, US, August 2015.

Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Vienna, Austria, October 2016a.

Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *Proceedings of the 13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, San Sebastián, Spain, July 2016b.

Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of Rowhammer defenses. In *IEEE Symposium on Security and Privacy*, pages 489–505, San Francisco, CA, US, May 2018. IEEE.

Shay Gueron. Intel's new AES instructions for enhanced performance and security. In *Fast Software Encryption*, pages 51–66. Springer, 2009.

Shay Gueron. Intel advanced encryption standard (AES) new instructions set, 2010. URL https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf.

David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 490–505, Oakland, CA, US, May 2011. IEEE.

Myeonggyun Han, Seongdae Yu, and Woongki Baek. Secure and dynamic core and cache partitioning for safe and efficient server consolidation. In *International Symposium on Cluster Computing and the Grid*, pages 311–320, Washington, DC, US, May 2018. IEEE.

Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *ACM Symposium on Operating Systems Principles*, pages 66–77, St. Malo, FR, October 1997.

Christopher A. Healy, Robert D. Arnold, Frank Müller David B. Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48:63–70, January 1999.

Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Transactions on Computer Systems*, 34(1):1:1–1:29, April 2016.

Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

Wei-Ming Hu. Reducing timing channels with fuzzy time. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 8–20, Oakland, CA, US, 1991. IEEE Computer Society.

Wei-Ming Hu. Lattice scheduling and covert channels. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–61, Oakland, CA, US, 1992. IEEE.

Deborah Hughes-Hallet, Andrew M. Gleason, Guadalupe I. Lonzano, et al. *Calculus: Single and Multivariable*. Wiley, New York, NY, US, 4 edition, 2005.

Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy*, pages 191–205, San Francisco, CA, May 2013. IEEE.

Andy A Hwang, Ioan A Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, London, UK, March 2012.

Mehmet Sinan İnci, Berk Gülmezoğlu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 368–390, Santa Barbara, CA, US, August 2016. Springer.

Intel. Microcode revision guidance, July 2018a. URL https://www.intel.com/content/dam/www/public/us/en/documents/sa00115-microcode-update-guidance.pdf.

Intel. Deep dive: Introduction to speculative execution side channel methods, 2018b. URL https://software.intel.com/security-software-guidance/insights/deep-dive-introduction-speculative-execution-side-channel-methods.

Intel. Speculative execution side channel mitigations, May 2018c. URL https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.

Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, 3C, and 3D: System Programming Guide*, November 2018a. URL https://software.intel.com/sites/default/files/managed/a4/60/325384-sdm-vol-3abcd.pdf.

Intel. Intel Core i7-4770 processor, 2019a. URL https://ark.intel.com/products/75122/Intel-Core-i7-4770-Processor-8M-Cache-up-to-3-90-GHz-.

Intel. Microarchitectural data sampling advisory, 2019b. URL https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00233.html.

Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, April 2012b.

Intel. *Intel 64 and IA-32 Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel Corporation, August 2015c. http://www.intel.com.au/content/www/au/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html.

Intel. *Improving Real-Time Performance by Utilizing Cache Allocation Technology*. Intel Corporation, April 2015d.

Intel. *Intel 64 and IA-32 Architecture Software Developer's Manual Volume 2: Instruction Set Reference, A-Z*. Intel Corporation, September 2016e. http://www.intel.com.au/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-VM attack on AES. In *Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 299–319, Gothenburg, Sweden, September 2014.

Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *IEEE Symposium on Security and Privacy*, pages 591–604, San Jose, CA, US, May 2015a. IEEE.

Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in Intel processors. In *Euromicro Conference on Digital System Design*, Funchal, Madeira, Portugal, August 2015b.

Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *Asia Conference on Computer and Communication Security (ASIA CCS)*, pages 353–364, Xi'an, CN, May 2016.

Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with Intel TSX. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Vienna, Austria, October 2016.

JEDEC. Standard no. 79-3F. *DDR3 SDRAM Specification*, 2012. URL https://www.jedec.org/sites/default/files/docs/JESD79-3F.pdf.

M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Brice no, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on Exokernel systems. In *ACM Symposium on Operating Systems Principles*, pages 52–65, Saint Malo, FR, October 1997.

Paul Karger. *Improving Security and Performance for Capability Systems*. PhD thesis, University of Cambridge Computer Laboratory, 1988. Technical Report No. 149.

R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10:338–359, 1992.

Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium*, pages 189–204, Bellevue, WA, US, August 2012. USENIX.

Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceedings of the 41st International Symposium on Computer Architecture*, pages 361–372, June 2014.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

Gerwin Klein, Toby Murray, Peter Gammie, Thomas Sewell, and Simon Winwood. Provable security: How feasible is it? In *Workshop on Hot Topics in Operating Systems (HotOS)*, page 5, Napa, USA, May 2011. USENIX.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In *Proceedings of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Montreal, Canada, April 2019.

Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Haburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwartz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy*, pages 19–37, San Francisco, May 2019. IEEE.

Jingfei Kong, Onur Acıiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Proceedings of the 15th IEEE Symposium on High-Performance Computer Architecture*, Raleigh, NC, US, 2009.

Boris Köpf and Markus Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*, New York, NY, US, 2009.

Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *Proceedings of the 24th International Conference on Computer Aided Verification*, pages 564–580, Berkeley, CA, US, 2012. Springer.

Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16: 613–615, 1973.

Adam Langley. ctgrind, 2010. URL https://github.com/agl/ctgrind.

Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86, Palo Alto, CA, US, March 2004.

Roy Levin, Ellis S. Cohen, William M. Corwin, Fred J. Pollack, and William A. Wulf. Policy/mechanism separation in HYDRA. In *ACM Symposium on Operating Systems Principles*, pages 132–140, Austin, TX, US, 1975. ACM.

Peng Li, Debin Gao, and Michael K Reiter. Mitigating access-driven timing channels in clouds using StopWatch. In *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, Budapest, HU, 2013. IEEE.

Jochen Liedtke. Improving IPC by kernel design. In *ACM Symposium on Operating Systems Principles*, pages 175–188, Asheville, NC, USA, December 1993.

Jochen Liedtke. On μ-kernel construction. In *ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 213–223, Montreal, CA, June 1997. IEEE.

Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *Proceedings of the 25th USENIX Security Symposium*, pages 549–564, Austin, TX, US, August 2016.

Moritz Lipp, Michael Schwartz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, Baltimore, MD, USA, August 2018. USENIX.

Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *Proceedings of the 47th ACM/IEE International Symposium on Microarchitecture*, Cambridge, UK, December 2014.

Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, San Jose, CA, US, May 2015. IEEE.

Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE Symposium on High-Performance Computer Architecture*, pages 406–418, Barcelona, Spain, March 2016. IEEE.

Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *IEEE Real-Time Systems Symposium*, Phoenix, AZ, US, December 1999.

William L. Lynch, Brian K. Bray, and M. J. Flynn. The effect of page allocation on caches. In *ACM/IEE International Symposium on Microarchitecture*, pages 222–225, Portland, OR, US, 1992. IEEE.

Jack A Mandelman, Robert H Dennard, Gary B Bronner, John K DeBrosse, Rama Divakaruni, Yujun Li, and Carl J Radens. Challenges and future directions for the scaling of DRAM. *IBM Journal of Research and Development*, 46(2.3):187–212, 2002.

Andrew Marshall, Michael Howard, Grant Bugher, and Brian Harden. Security best practices for developing Windows Azure applications, 2010. URL https://docs.microsoft.com/en-us/azure/security/security-best-practices-and-patterns.

Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 118–129, Portland, OR, US, June 2012. ACM. URL http://doi.acm.org/10.1145/2366231.2337173.

Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Kyoto, Japan, November 2015.

Clémentine Maurice, Manuel Weber, Michael Schwartz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Kay Römer, and Stefan Mangard. Hello from the other side: SSH over robust cache covert channels in the cloud. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, US, February 2017. USENIX.

Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, San Diego, CA, US, January 1996.

Vineeth Mekkat, Oleg Margulis, Jason M Agron, Ethan Schuchman, Sebastian Winkel, Youfeng Wu, and Gisle Dankel. Method and apparatus for recovering from bad store-to-load forwarding in an out-of-order processor, December 2015. US Patent 9,996,356.

Microsoft. What is IaaS?, 2018. URL https://azure.microsoft.com/en-au/overview/what-is-iaas/.

Milena Milenkovic, Aleksandar Milenkovic, and Jeffrey Kulick. Microbenchmarks for determining branch predictor organization. *Software: Practice and Experience*, 34(5): 465–487, April 2004.

Grzegorz Miłoś, Derek G. Murray, Steven Hand, and Michael A. Fetterman. Satori: Enlightened page sharing. In *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, US, June 2009.

Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. Fallout: Reading kernel writes from user space. *arXiv preprint arXiv:1905.12701*, 2019.

Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

Soo-Jin Moon, Vyas Sekar, and Michael K. Reiter. Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 1595–1606, Denver, CO, US, October 2015.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013. IEEE.

Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *13th International Workshop on Selected Areas in Cryptography*, Montreal, CA, 2006.

Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *ACM Symposium on Operating Systems Principles*, pages 221–234, Big Sky, MT, US, October 2009. ACM.

Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. Accelerating concurrent workloads with CPU cache partitioning. In *International Conference on Data Engineering*, pages 437–448, Paris, FR, April 2018. IEEE.

NXP. RDIMX6SABREPLAT: SABRE platform for smart devices based on the i.MX 6 series, 2019. URL https://www.nxp.com/support/developer-resources/evaluation-and-development-boards/sabre-development-system/sabre-platform-for-smart-devices-based-on-the-i.mx-6-series:RDIMX6SABREPLAT.

OpenBSD. Disabling the SMT (Simultanious Multi Threading) feature on OpenBSD, June 2018. URL https://www.mail-archive.com/source-changes@openbsd.org/msg99141.html.

ORACLE. SPARC T4 processor. URL http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/t-series/sparc-t4-processor-ds-497205.pdf. Accessed on 11.2018.

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

Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 Crytographers' track at the RSA Conference on Topics in Cryptology*, pages 1–20, San Jose, CA, US, 2006. Springer.

John Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd IEEE International Conference on Distributed Computing Systems*, pages 22–30, October 1982.

Daniel Page. Defending against cache-based side-channel attacks. *Information Security Technical Report*, 8(1):30–44, 2003.

Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, CA, 2005. URL http://css.csail.mit.edu/6.858/2014/readings/ht-cache.pdf.

Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *Proceedings of the 25th USENIX Security Symposium*, Austin, TX, US, August 2016.

Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. For a microkernel, a big lock is fine. In *Asia-Pacific Workshop on Systems (APSys)*, Tokyo, JP, July 2015. ACM.

Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a needle in the software stack. In *Proceedings of the 25th USENIX Security Symposium*, pages 1–18, Austin, TX, USA, August 2016.

RedHawk Linux. An overview of kernel text page replication in RedHawk Linux 6.3, September 2012. URL https://www.concurrent-rt.com/wp-content/uploads/2016/11/kernel-page-replication.pdf.

Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 199–212, Chicago, IL, US, 2009.

John Rushby. A trusted computing base for embedded systems. In *Proceedings of 7th DoD/NBS Computer Security Conference*, pages 294–311, September 1984.

Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *International Symposium on Computer Architecture*, pages 57–68, 2011.

Marvin Schaefer, Barry Gold, Richard Linde, and John Scheid. Program confinement in KVM/370. In *Proceedings of the Annual ACM Conference*, pages 404–410, Atlanta, GA, US, 1977. ACM.

Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. *arXiv preprint arXiv:1905.05726*, 2019.

Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer bug to gain kernel privileges. *Black Hat*, 15, 2015. URL https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf.

seL4. seL4 microkernel web site. URL sel4.systems. Accessed on 10.2018.

seL4. The mainline seL4 kernel, 2019. URL https://github.com/seL4/seL4.

Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 471–481, Seattle, Washington, USA, June 2013. ACM.

Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 1948. Reprinted in SIGMOBILE Mobile Computing and Communications Review, 5(1):3–55, 2001.

Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *ACM Symposium on Operating Systems Principles*, pages 170–185, Charleston, SC, USA, December 1999. ACM.

Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 194–199, HK, June 2011. IEEE.

Bruno R. Silva, Diego Aranha, and Fernando M. Q. Pereira. Uma técnica de análise estática para detecção de canais laterais baseados em tempo. In *Brazilian Symposium on Information and Computational Systems Security*, pages 16–29, Florianópolis, SC, BR, December 2015.

Bernard W. Silverman. *Density estimation for statistics and data analysis*. Chapman & Hall, London, UK, 1986.

Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *Proceedings of the 18th European Symposium On Research in Computer Security*, pages 718–735, Egham, UK, September 2013. Springer.

Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42nd ACM/IEE International Symposium on Microarchitecture*, New York, NY, US, 2009.

Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *Proceedings of the 38th International Symposium on Computer Architecture*, San Jose, CA, US, 2011.

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

Jo Van Bulck, Marina Minkin, Ofir Weiss, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Stracks. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, pages 991–1008, Baltimore, August 2018. USENIX.

Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In *Proceedings of the 2015 Crytographers' track at the RSA Conference on Topics in Cryptology*, pages 3–21, San Francisco, CA, USA, April 2015.

Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious management unit: Why stopping cache attacks in software is harder than you think. In *Proceedings of the 27th USENIX Security Symposium*, pages 937–954, Baltimore, MD, US, August 2018. USENIX.

Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *IEEE Symposium on Security and Privacy*, San Francisco, USA, May 2019.

Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based defenses against cross-VM side-channels. In *Proceedings of the 23rd USENIX Security Symposium*, San Diego, CA, US, 2014.

Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen. In *ACM Workshop on Cloud Computing Security*, pages 41–46, Chicago, IL, October 2011. ACM.

Vish Viswanathan. Disclosure of H/W prefetcher control on some Intel processors, September 2014. URL https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.

VMware Inc. Security considerations and disallowing inter-virtual machine transparent page sharing, October 2014. URL http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=2080735.

VMware Knowledge Base. Security considerations and disallowing inter-virtual machine transparent page sharing, October 2014. URL http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=2080735.

Michael von Tessin. The clustered multikernel: An approach to formal verification of multiprocessor OS kernels. In *2nd Workshop on Systems for Future Multi-core Architectures*, pages 1–6, Bern, Switzerland, April 2012.

Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 181–194, Boston, MA, US, 2002. USENIX.

Yao Wang and G Edward Suh. Efficient timing channel protection for on-chip networks. In *Proceedings of the 6th ACM/IEEE International Symposium on Networks on Chip*, pages 142–151, Lyngby, Denmark, 2012. IEEE.

Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *Annual Computer Security Applications Conference (ACSAC)*, Miami Beach, FL, US, 2006.

Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 494–505, San Diego, CA, US, June 2007. ACM.

Hassan M. G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. SurfNoC: A low latency and provably non-interfering approach to secure networks-on-chip. In *Proceedings of the 40th International Symposium on Computer Architecture*, pages 583–594, 2013.

Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, pages 87–104, Monterey, CA, USA, October 2015.

Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. https://foreshadowattack.eu/foreshadow-NG.pdf, August 2018.

Dong Hyuk Woo and Hsien-Hsin S. Lee. Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, Phoenix, AZ, US, 2007.

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, S. Margherita Ligure, IT, 1995. ACM.

John C. Wray. An analysis of covert timing channels. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 2–7, Oakland, CA, US, May 1991. IEEE.

Weiyi Wu, Ennan Zhai, David Isaac Wolinsky, Bryan Ford, Liang Gu, and Daniel Jackowitz. Warding off timing attacks in Deterland. In *Conference on Timely Results in Operating Systems*, Monterey, CS, US, October 2015.

Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium*, pages 159–173, Bellevue, WA, US, 2012. USENIX.

Leslie Xu. Securing the enterprise with Intel AES-NI, 2010. URL http://www.intel.com/content/www/us/en/enterprise-security/enterprise-security-aes-ni-white-paper.html.

Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *ACM Workshop on Cloud Computing Security*, pages 29–40. ACM, 2011.

Yuval Yarom. Mastik: A micro-architectural side-channel toolkit, 2016. URL http://cs.adelaide.edu.au/~yval/Mastik/Mastik.pd.

Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. IACR Cryptology ePrint Archive, Report 2014/140, February 2014.

Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium*, pages 719–732, San Diego, CA, US, 2014. USENIX.

Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel last-level cache. https://eprint.iacr.org/2015/905, September 2015.

Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In *Conference on Cryptographic Hardware and Embedded Systems 2016 (CHES 2016)*, pages 346–367, Santa Barbara, CA, US, August 2016. Springer.

Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 42–53, Atlanta, Georgia, USA, May 1999.

Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling cores, kernels, and operating systems. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 17–31, Broomfield, CO, US, 2014. USENIX.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 99–110, Beijing, CN, 2012a.

Rongting Zhang, Xin Su, Jiacheng Wang, Chingyue Wang, Wenxin Liu, and Rynson WH Lau. On mitigating the risk of cross-VM covert channels in a public cloud. *IEEE Transactions on Parallel and Distributed Systems*, 26:2327–2339, 2014a.

Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Memory DoS attacks in multi-tenant clouds: Severity and mitigation. *arXiv preprint arXiv:1603.03404v2*, 2016.

Yinqian Zhang and Michael K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, pages 827–838, Berlin, DE, November 2013.

Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, pages 305–316, Raleigh, NC, US, October 2012b. ACM.

Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant side-channel attacks in PaaS clouds. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, Scottsdale, AZ, US, 2014b.

Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Vienna, Austria, October 2016.