Barbara Gigerl

Automated Analysis of Speculation Windows in Spectre Attacks

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieurin

Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor: Daniel Gruss

Institute for Applied Information Processing and Communication

Graz, May 2019

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Datum	Unterschrift

Abstract

Speculative execution is a feature integrated into most modern CPUs. Although introduced as a way to enhance the performance of processors, the release of Spectre attacks showed that it is a significant security risk. Since CPUs from various vendors, including Intel, AMD, ARM, and IBM, implement speculative execution, all different kinds of devices are affected by Spectre attacks, for example, desktop PCs and smartphones. Spectre attacks exploit the branch prediction mechanisms of the CPU and then use a cache covert channel to leak secret data. Several attack variants have been discovered since the release, including Spectre-PHT which targets the Pattern History Table of the CPU. The success rate of the attack primarily depends on the size of the speculation window. A larger speculation window implies a higher likelihood for the attack to succeed. The window size can be influenced by the choice of the condition in Spectre-PHT attacks.

This thesis explores the limits of Spectre-PHT attacks. We present a large-scale test framework for CPUs of four different manufacturers, Intel, AMD, ARM, and IBM. The framework tests different conditions and their influence on the speculation window and detects the size of the speculation window for a spyecific condition automatically. We developed methods for robust measurements and evaluated the success rate of the attack and the throughput for each condition. The attacker has two possibilities to extend the size of the speculation window. First, one can use instructions in the condition which take a long time to execute. This includes simple integer or floating point instructions arranged in a dependency chain and accesses to DRAM, which are slow when the data is uncached, or the corresponding TLB entry is missing. On Intel and AMD platforms, this also includes AVX instructions executed when the AVX unit is disabled. Our analysis shows that dependency chains give the best results on almost all platforms. Accessing a flushed variable or a variable without a present TLB entry in the condition maximizes the speculation window on all platforms. Second, the attacker can use fast instructions in the condition and make them slow. Our tests cover this scenario by combining port contention and instruction dependency chains. We demonstrate that this has a positive effect on the success rate of the attack.

We suggest how our analysis results can be combined with Foreshadow, another transient execution attack. Foreshadow has not yet been shown to work using speculative execution as an exception suppression mechanism. We show that the reason for this cannot be a too small speculation window.

Kurzfassung

Speculative Execution ist eine Funktionalität, die in die meisten modernen CPUs eingebaut ist. Man konnte dadurch die Geschwindigkeit und Effizienz von Prozessoren erhöhen, aber die Veröffentlichung von Spectre Attacken hat gezeigt, dass sie auch ein signifikantes Sicherheitsrisiko darstellt. CPUs verschiedener Hersteller wie Intel, AMD, ARM und IBM integrieren Speculative Execution in ihre Prozessoren, weshalb zahlreiche Geräte, beispielsweise Desktop PCs und Smartphones, von Spectre Attacken betroffen sind. Spectre Attacken nutzen Ressourcen zur Sprungvorhersage in Prozessoren aus und verwenden danach eine Seitenkanalattacke, basierend auf dem Cache, um geheime Daten zu lesen. Diverse Varianten von Spectre Attacken wurden seit der Veröffentlichung entdeckt. Dazu zählt auch Spectre-PHT, welche die Pattern History Table der CPU ausnutzt.

Diese Arbeit zielt darauf ab, Grenzen einer Spectre-PHT Attacke bezüglich der Erfolgsrate zu finden. Wir präsentieren ein Test Framework für CPUs vier verschiedener Hersteller, Intel, AMD, ARM und IBM. Das Framework testet verschiedene Conditions und deren Einfluss auf die Größe des Speculation Windows. Wir entwickeln Methoden für robuste Mesungen und evaluieren die Erfolgsraten der Attacke und den Throughput für jede Condition. Der Angreifer oder die Angreiferin hat zwei verschiedene Möglichkeiten um das Speculation Window größer zu machen. Erstens kann er oder sie Instruktionen verwenden die eine lange Ausführungszeit haben. Das sind beispielsweise Instruktionsabhängigkeitsketten und Zugriffe auf den DRAM, welche langsam sind, wenn die Daten nicht im Cache liegen oder der zugehörige TLB Eintrag nicht vorhanden ist. Unsere Analyse zeigt dass Instruktionsabhängigkeitsketten die besten Resultate auf fast allen Plattformen geben. Zweitens kann er oder sie schnelle Instruktionen verwenden und diese dann langsam machen. Unsere Tests decken dieses Szenario ab, indem Port Contention und Instruktionsabhängigkeitsketten kombiniert werden. Wir zeigen, dass dies einen positiven Effekt auf die Erfolgsrate der Attacke hat.

Wir zeigen wie unsere Analyseresultate mit Foreshadow, einer anderen Transient Execution Attack, kombiniert werden können. Es wurde noch nicht gezeigt, dass Foreshadow mit Speculative Execution als Fehlerunterdrückungsmechanismus funktioniert. Wir beweisen, dass dies nicht an einem zu kleinen Speculation Window liegen kann.

Acknowledgments

First, I want to thank my supervisor Daniel Gruss for his exceptional support, all the effort he put into answering my questions and the helpful reviews of the drafts of this thesis.

Furthermore, I want to thank my colleagues at IAIK for the meaningful discussions and the help and support.

I also want to thank my parents, Erna and Rupert, for all the wise advices and never-ending support throughout my studies. A big thank also belongs to the rest of my family for their continuous encouragement and help.

I am very thankful for my friends, in particular my best friend David, for all the emotional support. Finally, I want to express my gratefulness to have Alex in my life, for all his love and patience and for keeping me motivated and positive-minded while working on this thesis.

Contents

1 Introduction **1**
1.1 Outline . . . 3

2 Background **4**
2.1 CPU Architectures . . . 4
2.2 CPU Caches . . . 10
2.3 CPU Optimization Mechanisms . . . 17
2.4 Out-of-Order Execution . . . 21
2.5 Speculative Execution . . . 23
2.6 Transient Execution Attacks . . . 24
2.7 Port Contention . . . 27

3 Attack Analysis Setup and Framework **29**
3.1 Automated Framework . . . 29
3.2 Test Scenarios . . . 35

4 Real-world Analysis Results **38**
4.1 Intel . . . 38
4.2 AMD . . . 50
4.3 ARM . . . 58
4.4 IBM Power . . . 66
4.5 Comparison . . . 73

5 Enhancing Real-World Attacks **75**
5.1 Leaking more data . . . 75
5.2 Foreshadow using Speculative Execution . . . 76

6 Conclusion **79**

Chapter 1

Introduction

Transient execution attacks came up with the discovery of Meltdown [50] and Spectre [44] and have been shown to be very powerful. The basis of these attacks are instructions which are executed transiently by the CPU. *Transient* means that the CPU executes the instructions, but for some reason, their result must not be made visible on an architectural level, and they are rolled back [44]. An example for this might be that the CPU speculates on taking a branch and predicts the wrong result: it executes the instructions first, but later undoes their architectural effects, but the microarchitectural side effects are not rolled back, which leaves traces in various parts of the CPU, most importantly, the cache.

It is possible that these traces depend on secret data. The attacker can then leak this secret data using a cache side-channel attack. Cache side-channel attacks are timing attacks and based on the observation that accessing cached memory is faster than accessing uncached memory. There exist various techniques for mounting such attacks, including Flush+Reload [92], Flush+Flush [27], Evict+Reload [28] and Prime+Probe [80].

The two most common cases where a CPU executes transient instructions are out-of-order execution and speculative execution [12]. Both are essential performance features of modern CPUs, mainly to handle the delay caused by busy execution units. Out-of-order execution means that instructions are not executed in the same order in which they are committed [50]. For example, consider a scenario where the CPU fetches two mutually independent instructions, a load from memory and the addition of two registers. It can first issue the load and send it to the proper execution unit. After that, it handles the addition while the load is being processed. It might happen that the addition completes before the load, even though the execution of the load started first. Out-of-order execution yields a significant performance improvement since the load will take longer than the addition. However, out-of-order executed instructions might sometimes cause exceptions, e.g., divide-by-zero. In this case, the CPU has to rollback all the instructions which were already executed before the erroneous instruction, causing tran-

sient execution. Speculative execution improves the performance of programs containing branches. When the CPU locates a branch, it must wait until all involved operations are finished to decide on the outcome of the branch. This would be a huge bottleneck and inefficient resource usage because the CPU would simply stall. Instead, it relies on the information of the branch predictor, suggesting whether or not to take the branch [12], and speculatively continues execution on the suggested location. However, the branch predictor might make an incorrect prediction, causing the CPU to rollback the changes.

Spectre [44] is an example of a transient execution attack which by now exists in numerous variants, each targeting a different branch prediction mechanism: Spectre-PHT uses the Pattern History Table, Spectre-BTB the Branch Target Buffer, Spectre-STL the CPUs memory disambiguation prediction and Spectre-RSB the Return Stack Buffer [12]. Several defenses against Spectre attacks have been published [91, 42, 44, 41, 83]. One idea is to make it harder to set up cache covert channels [44], which often comes with a heavy performance loss and does not prevent the attacker from using another side-channel to leak. One could also use serializing instructions like memory fences after taking a branch, for example the `lfence` instruction. `lfence` guarantees that all memory loads preceding the instruction are finished before `lfence` is executed [38], preventing also speculative loads from memory.

In this thesis, we evaluate Spectre-PHT attacks on different platforms using a fully-automated framework. This framework evaluates the performance in terms of success rate and throughput of the attack using different conditions and automatically determines the ideal size of the speculation window for each condition. It is ported to all our test devices, which are a Intel CPUs (Skylake), a AMD CPU (AMD Ryzen), a ARM CPU and a IBM Power9 CPU. Our analysis shows that the success rate of the attack depends on the condition which checks if the array index is out of bounds in a Spectre-PHT attack.

In a real-world setting, we are sometimes confronted with code situations differing from artificial Spectre attacks. For example, the array bounds check might not only check whether the index is smaller than the array size, or the attacker might be unable to flush the bounds check variable. Therefore, we want to look at different forms of conditions in Spectre-PHT attacks and their influence on the performance of the attack. The condition can either use instructions which take a long time to execute or instructions which are slow because of some external influence. Instructions which take a long time to execute are instruction chains, loading uncached data from DRAM, loading data without an existing TLB entry or AVX instructions on Intel and AMD. Instruction chains are formed by executing the instructions multiple times in succession. Each instruction depends on the previous one, which is why they are also called dependency chains. A condition depending on uncached data takes long to evaluate because data must be loaded from DRAM. If one has to load data which does not have an existing TLB entry, a whole page table walk has to be done in order to find out the physical address to load from. This is a very costly operation and therefore conditions depending on data without a TLB entry take long to evaluate. It has been shown that AVX units are disabled in

Intel CPUs when they are unused due to power saving. Using an AVX instruction when the unit is disabled requires additional power-up time.

An attacker can also slow down the execution of instructions by exploiting port contention. In this scenario, the attacker starts a second process on a different logical core, but on the same physical core and executes instructions which will contend a specific execution port. One important performance measurement for Spectre-PHT attacks is the size of the speculation window, *i.e.*, how many instructions are executed speculatively. The size of the speculation window differs from platform to platform and depends heavily on the condition in use, which we also show with this thesis.

Foreshadow, another transient execution attack, was shown to work using userspace exception handlers as an exception handling mechanism and TSX as an exception suppression mechanism. However, there is no proof-of-work using speculative execution as an exception suppression mechanism. We demonstrate successfully that the reason for this is not due to a too short speculation window, by altering the condition in a way that the speculation window is extended to the maximum.

1.1 Outline

This thesis is structured as follows: Chapter 2 provides detailed background information on CPU architectures, explains how CPU caches work and discusses the most important CPU optimization mechanisms. Additionally, we give an overview of transient execution attacks and more details on Spectre attacks. In Chapter 3, we describe the automated attack analysis framework. Chapter 4 discusses the results of our analysis. Chapter 5 applies the insights of our analysis in real-world settings. Finally, Chapter 6 concludes and summarizes our work.

Chapter 2

Background

In this chapter, we discuss background information required to understand the remainder of the thesis. In Section 2.1, we give an overview of the currently most widely used CPU architectures. In Section 2.2, we explain how CPU caches work, how they are structured and to which extent they form a security risk. Modern CPUs make use of several techniques to optimize for performance, which we explain in Section 2.3. Sections 2.4 and 2.5 will give more details on two optimization techniques, out-of-order execution, and speculative execution. Section 2.6 addresses why these concepts are also a security threat in the form of transient execution attacks. In Section 2.7, we explain the concept of port contention and how it can be exploited.

2.1 CPU Architectures

When talking about the architecture of buildings, we describe how the building is constructed, how it looks like and how it is designed. The term *CPU architecture* is aligned with these definitions and describes the structure and design of a CPU. One must distinguish between the *microarchitecture* and the *instruction set architecture (ISA)* of a CPU [68]. Generally speaking, the instruction set architecture defines which instructions and concepts, for example, out-of-order execution or speculative execution, are supported by a processor while the microarchitecture gives the concrete implementation [66]. Two different processors which implement the same ISA might still have a different microarchitecture. For example, the microarchitecture of an Intel Skylake processor is different from the microarchitecture of an Intel Haswell processor.

The machine can either be a reduced instruction set computer (RISC) or a complex instruction set computer (CISC) [81, 57, 16, 39]. RISC focuses on simple instructions [66]. Each instruction does exactly one operation and typically takes one clock cycle to execute. RISC provides many general-purpose registers but only a few addressing modes. The ARM Cortex-A57 CPU and the IBM Power9 CPU both are RISC. This has the

advantage that the processor design is simpler and the execution time per instruction is lower. The disadvantage is that the number of instructions per program is higher compared to CISC. CISC optimizes for a low number of instructions needed to perform one operation [66]. Each instruction typically takes more than one clock cycle. CISC is characterized by fewer general-purpose registers but several addressing modes. The Intel and AMD CPUs we use as test devices are CISC. The advantage is that fewer instructions are needed per program. The disadvantage is that more complexity is introduced to the CPU and one single instruction takes longer to execute.

2.1.1 Intel

Intel, as one of the biggest CPU manufacturers, produces not only processors for desktop PCs and laptops, but also for other application areas like smartphones and low-power computing. Since we use desktop PCs for our tests, we want to describe the microarchitecture of those processors, in particular of Intel Skylake processors, since this is our test device. Their desktop processors implement the x86 instruction set architecture. Simultaneous multithreading (SMT), or hyper-threading as Intel calls it, is a fundamental concept which was introduced by them in 2002 [56]. Roughly speaking, this technology provides a possibility to nearly double CPU cores without doubling hardware resources. The introduction of logical cores enables this: One physical core has two or more logical cores, which share hardware resources such as the execution engine, the branch predictor or the L1 and L2 caches. We extensively describe this term in Section 2.3. Figure 2.1 shows a sketch of a logical core in an Intel Skylake CPU.

In general, the structure of a logical core can be divided into three areas: the front-end, the back-end or execution engine, and the memory subsystem [56, 52]. The task of the front-end is to fetch instructions from memory. Since we have a CISC architecture, instructions are decoded into μops in the next stage. All these steps are performed fully in-order. The branch predictor influences which instructions are fetched and decoded. The decoded instructions are queued up in the allocation queue which serves as the interface between front-end and back-end. The μops are handed over to the back-end via the reorder buffer [89, 37]. This buffer serves as a central element of the back-end because it keeps track of the state of the operation, *i.e.*, it tracks if the instruction is already ready to be executed, retired or even discarded. Once μops arrive in the reorder buffer, the scheduler assigns them to a suitable execution port. Each execution port connects to one or more execution units. Each execution unit has its own task. For example, on Intel Skylake CPUs, execution port 6 has two associated execution units, one for performing arithmetic and logical operations on integers and one for handling branches. Instructions which manipulate memory, *i.e.*, load and store instructions, are sent to the memory subsystem by the scheduler. The back-end design is constructed in an out-of-order fashion: it is not guaranteed for all the operations to be executed in-order. If possible, the scheduler assigns the operations in a way that the execution ports are all occupied optimally. It might happen then, that for μops o_1 and o_2, which

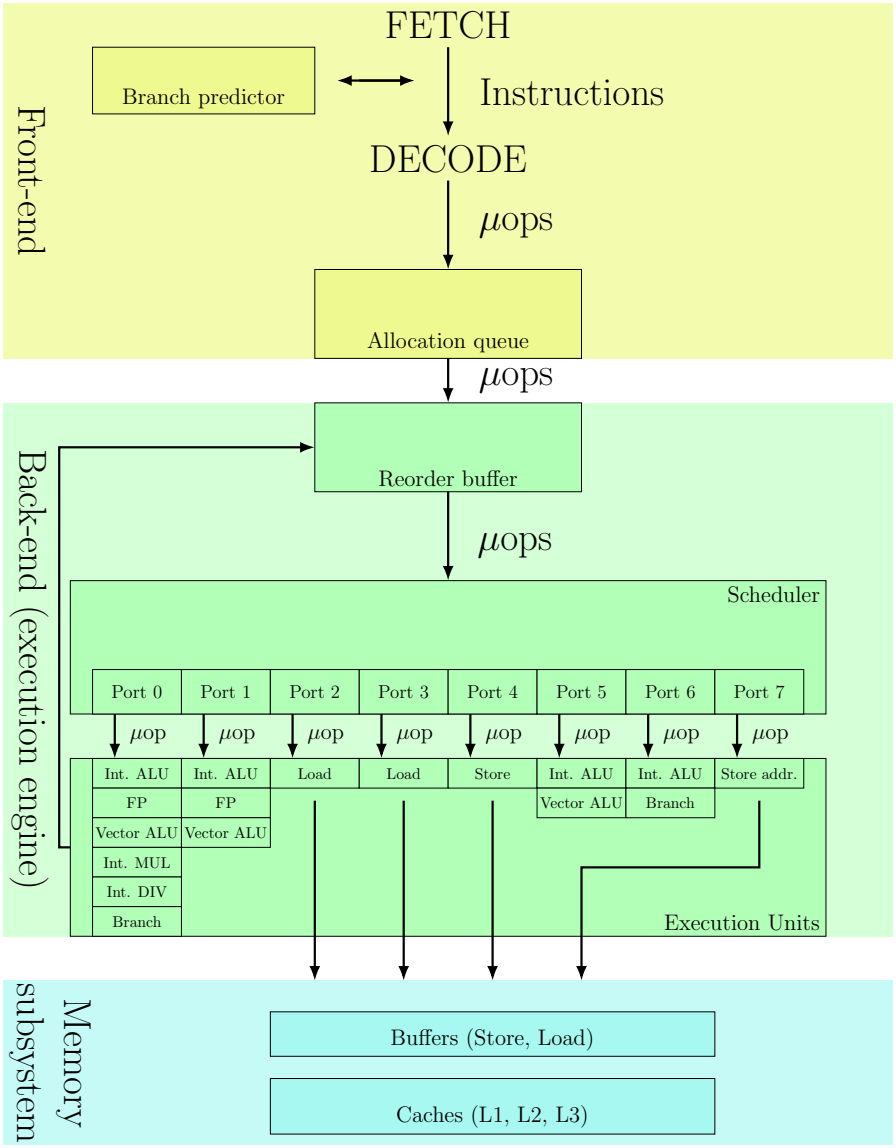


Figure 2.1: Sketch of a logical core in an Intel Skylake CPU. The front-end fetches and decodes instructions into μops, influenced by the branch predictor. The μops are handed over to the back-end and stored in the reorder buffer. The scheduler assigns them to a suitable execution unit (Integer ALU unit, floating point operations, integer vector operations, etc.) on a specific execution port. Loads and stores to memory are forwarded to the memory subsystem. Once an operation is executed, it returns to the reorder buffer, where it eventually retires [52, 20].

were executed in exactly this order, μop o_2 finishes before o_1. However, to keep the execution functionally correct, operations have to retire in-order, *i.e.*, o_1 before o_2.

For our tests, we use an Intel i7-6700K (Skylake). Skylake was released in 2016 and represents the 8th generation of Intel CPUs [55].

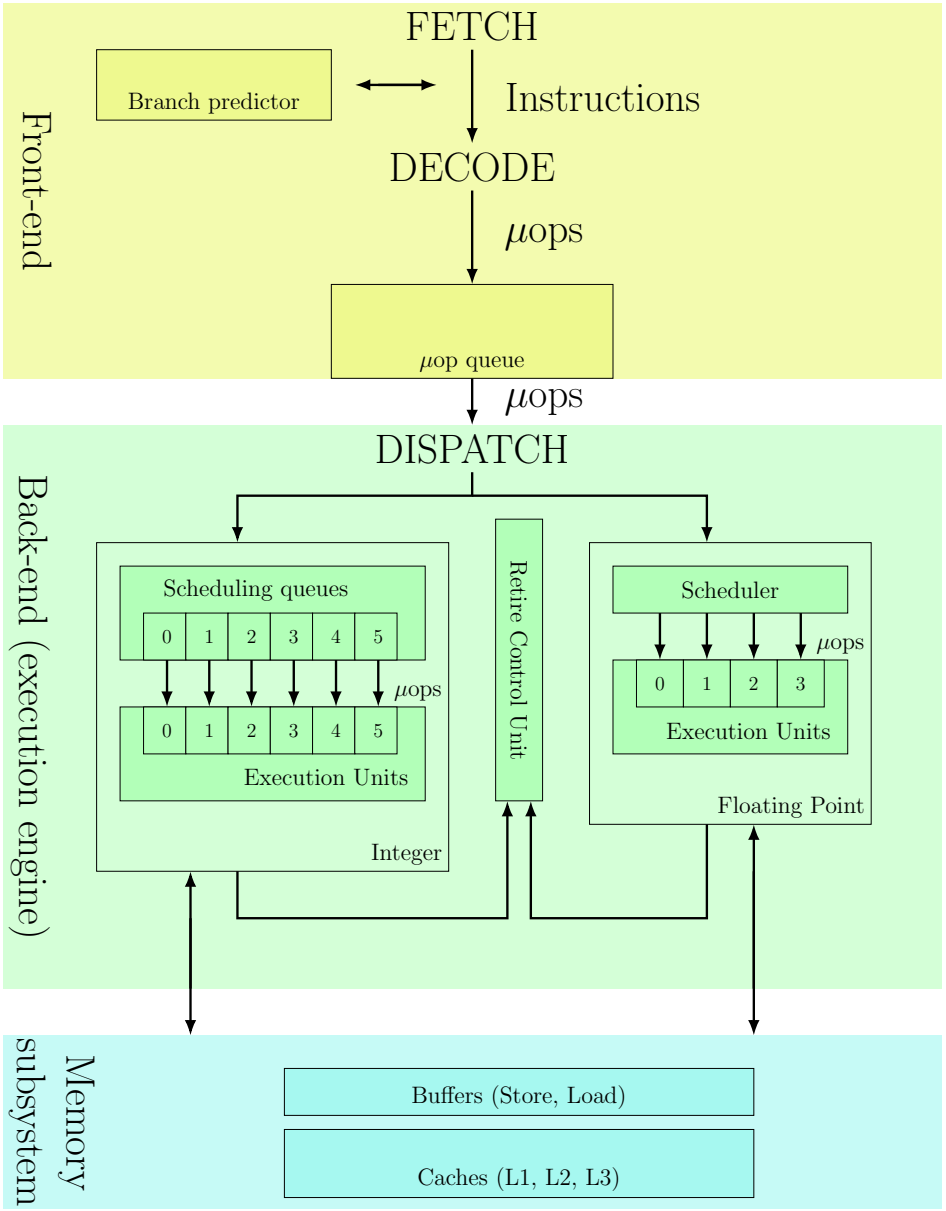


Figure 2.2: Sketch of a logical core in an AMD Zen CPU. The front-end fetches and decodes instructions into μops, influenced by the branch predictor. The μops are handed over to the back-end via the μop queue. Then they are dispatched to either the integer or FP execution engine. Schedulers located inside the engines and issue μops to suitable execution units. For memory manipulating instructions, the memory subsystem, which contains caches, is used. The results are gathered in the RCU (Retire Control Unit) which makes sure the instructions retire in the correct order.

2.1.2 AMD

AMD also offers a wide variety of products. Again, we focus on Desktop systems which implement the x86 instruction set architecture. Just like Intel, they use SMT as an optimization mechanism. The first CPU supporting SMT was introduced much later than Intel did with the AMD Zen architecture in 2017 [17]. The following section describes the structure of a single logical core in a physical core of our test CPU, an AMD Ryzen Threadripper 1920X, as illustrated in Figure 2.2.

A single logical core comprises the parts front-end, back-end, and memory subsystem [1, 3], as we already saw it in the previous section for Intel processors. The most significant difference in design between Intel Skylake and AMD Zen CPUs is the split back-end: floating point and integer operations are executed independently from each other in separate parts of the execution engine. Both the integer and the floating point execution resources have their own scheduler distributing the instructions.

The front-end's task is to fetch and decode instructions in-order [53, 1]. The branch prediction unit influences this process. The μops are further passed to the μop queue which serves the same purpose as the allocation queue in Intel CPUs. The next step is to dispatch these instructions from the μop queue. In the dispatch-step, the CPU decides if the operation is an integer instruction or a floating point instruction to send it to the correct part of the back-end.

In AMD Zen microarchitectures, the back-end can be furthermore divided into three parts, the integer execution unit, the floating-point execution unit and the retire control unit (RCU) [1, 2]. The integer execution unit has six scheduling queues (instead of one big scheduler on Intel systems) which manage the μop-execution-port assignment. Two of those six queues are for memory operations only. These schedulers issue μops to execution ports, just like in Intel systems. There are four ALUs for integer operations and two for address generation [1]. The floating point execution unit is structured similarly to the integer execution unit. The only difference is that there is only a single scheduling queue which distributes the instructions to one out of four execution ports. Between those two parts lies the retire queue, managed by the RCU (Retire Control Unit). It collects all possibly out-of-order executed operations and makes sure they retire in-order. The third part of the overall core, the memory subsystem, does not differ significantly from Intel platforms. It consists of a load buffer, a store buffer and the caches.

2.1.3 ARM

Compared to the manufacturers we described in previous sections, ARM focuses more on mobile and low-power computing. Their processors are generally RISC. Our test device is an Nvidia Jetson TX1 board which internally uses in total four CPU cores (ARM Cortex-A57). The ARM Cortex-A57 implements the ARMv8-A instruction set architecture. The whole CPU has a simple structure, and simultaneous multithreading is not implemented which means four physical cores are operating, to a certain degree,

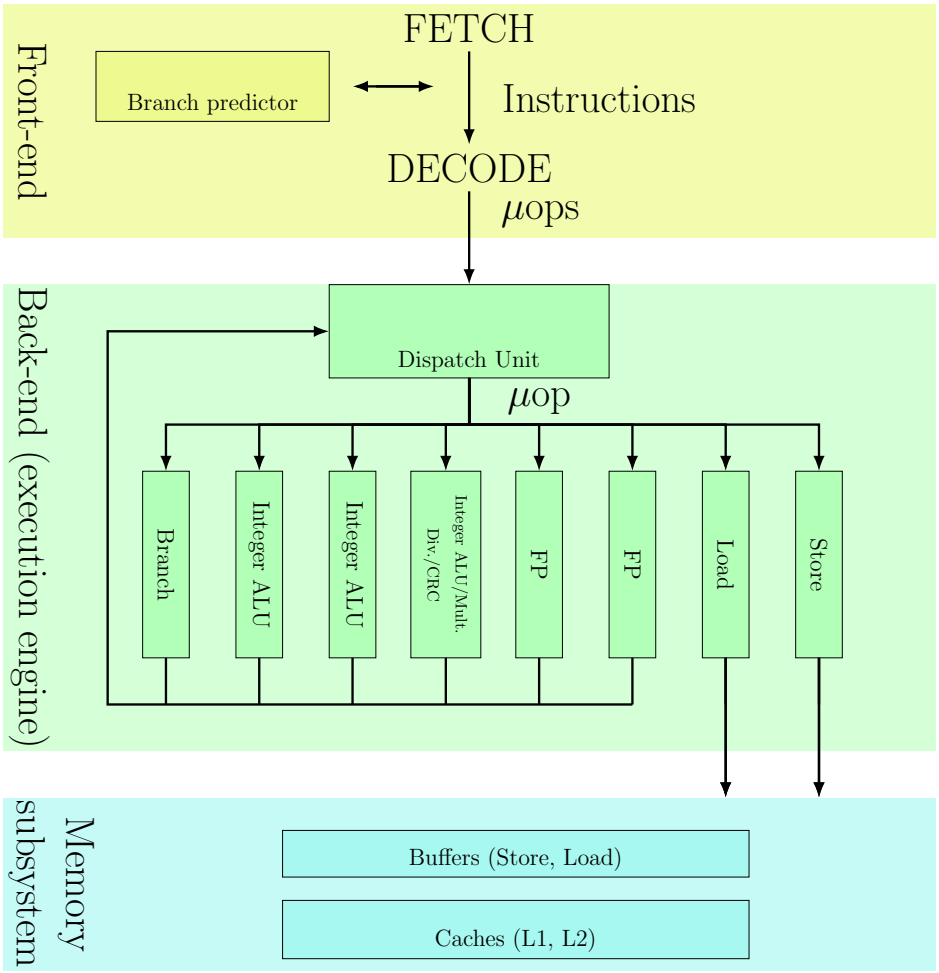


Figure 2.3: Sketch of a physical core in an ARM Cortex-A57 CPU. The front-end fetches and decodes instructions into μops, influenced by the branch predictor. The μops are handed over to the back-end via the dispatch unit. Then they are distributed to one of the eight execution ports. For memory manipulating instructions, the memory subsystem is used. The results are gathered in the Dispatch Unit, which makes sure the instructions retire in the correct order.

independently from each other, not sharing caches or execution ports as it is done in the Intel Skylake processor and the AMD Zen processor. Figure 2.3 shows a sketch of a physical core of an ARM Cortex-A57 CPU.

In a physical core of the ARM Cortex-A57 CPU, we have three parts, the front-end, back-end or execution engine, and the memory subsystem [24, 7, 6]. The front-end fetches instructions and decodes them. They are then placed in the instruction dispatch unit where they wait to be dispatched to the suitable execution unit. The dispatch unit consists of a reorder buffer and a scheduler. The out-of-order back-end consists of eight execution units. There is one dealing branches exclusively, two for integer

ALU operations, one for integer ALU combined with division, multiplication and CRC computation, two for floating-point operations, one for load operations and one for store operations. Finally, instructions retire in-order in the reorder buffer. The memory subsystem consists of buffers for load and store as well as the L1 and L2 caches.

2.1.4 IBM Power

IBM Power9 is the only system we investigate which is purely used for servers nowadays. The microarchitecture implements the Power ISA, an instruction set architecture developed by the OpenPOWER Foundation, which is heavily supported by IBM. The Power ISA is a RISC design.

Since the IBM Power9 is a server CPU, it focuses on performance a lot. The CPU works with 16 logical cores, grouped in four physical cores (SMT4). However, the design of Power9 also supports SMT8 cores [78, 51].

One notable thing about the IBM Power9 architecture is the sliced design of the processor [78, 31]. Each physical core comprises four slices; a slice is a logical core. A slice is as a basic computation block, consisting of an execution unit for integer operations, a unit for floating point operations and one for address generation coupled together with units for load and store. Two slices are combined into a super-slice. There is only one unit for integer divisions per super-slice. One physical core consists of two super-slices. There is only one branch slice per physical core.

Figure 2.4 shows the sketch of an SMT4 core of an IBM Power9 CPU. In general, the division into front-end, back-end and memory subsystem can also be recognized in an IBM Power9 CPU. The front-end is as usually responsible for fetching and decoding instructions. The dispatch unit is the entry point of the back-end, which operates out-of-order. The dispatch unit is responsible for assigning the instruction to either the branch slice or one of the four other slices. The slice processes the instruction and - if necessary - interacts with the memory subsystem.

2.2 CPU Caches

CPUs have been optimized a lot in recent years in terms of speed and performance. This performance is not only determined by the internal components of the CPU, but also by the time it takes to interact with external peripherals. One of these peripherals is the main memory, a volatile storage medium used to store all kind of data needed for operations of a computer. The CPU has the possibility to store data internally in registers, but this space is limited. Therefore, the CPU needs to swap out data to the main memory sooner or later. For example, the main memory stores parts of the OS kernel while the OS is running or parts of the program binary for a specific process. Unfortunately, this memory is not as fast as the CPU, and considerable latency

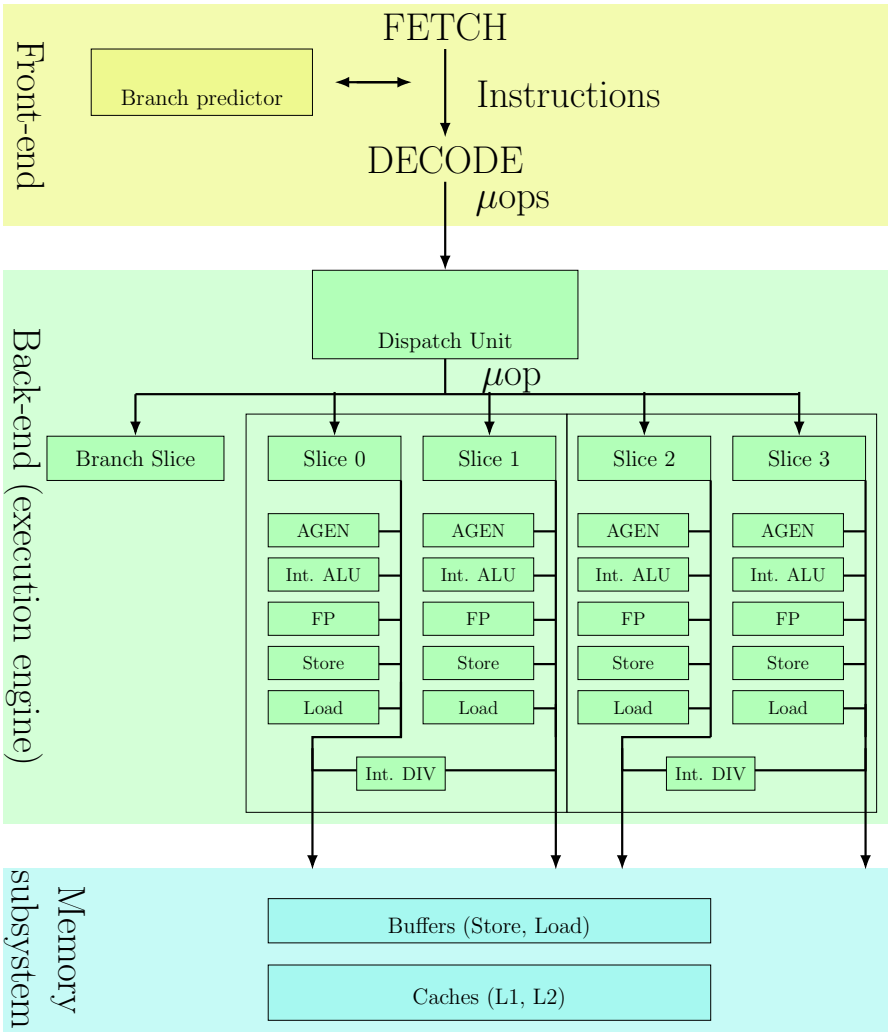


Figure 2.4: Sketch of a physical core in an IBM Power9 CPU. The front-end fetches and decodes instructions into μops, influenced by the branch predictor. The μops are handed over to the back-end via the dispatch unit. Then they are distributed to either the branch slice or one of the other slices, depending on the instructions. The results are gathered in the Dispatch Unit which makes sure the instructions retire in the correct order.

is introduced when loading or storing data. Caches aim to close this gap between CPU and memory performance and are a part of every modern processor.

Caches are small banks of fast memory, storing small portions of data from the main memory. Whenever the CPU needs to access data from DRAM, it first looks for the data in the cache. If it finds the data there, the slow memory access can be skipped, and data is retrieved much faster. However, we cannot just remove DRAM entirely because caches are limited in size. Increasing their size would again lead to higher access times.

Table 2.1: Size of caches (L1 data cache, L1 instruction cache, L2 cache, L3 cache) of our test devices. The ARM Cortex-A57 does not have an L3 cache.

	L1d	**L1i**	**L2**	**L3**
Intel i7-6700K	32 KB	32 KB	256 KB	8192 KB
AMD Ryzen Threadripper 1920X	32 KB	64 KB	512 KB	8192 KB
ARM Cortex-A57	32 KB	48 KB	2048 KB	-
IBM Power9	32 KB	32 KB	512 KB	10 240 KB

Caches were introduced because of observations based on the *principle of locality* [32, 75]. It says that programs tend to use data, instructions, and addresses not only once, but multiple times during execution in spatial or temporal proximity. The program likely will need this data again soon and ideally will be cached by then. Locality comes in two dimensions, temporal locality, and spatial locality [32, 30]. Temporal locality talks about the observation that data is used over time repeatedly and in temporal proximity. Spatial locality means that multiple data blocks needed by the program tend to be near to each other in memory.

2.2.1 Structure of a Cache

Caches are organized hierarchically [32]. In modern CPUs, these hierarchies are called *levels* , and all of our test devices have three of them (L1, L2, and L3), except the ARM Cortex-A57, which has two, as shown by Table 2.1. The L1 cache is the smallest and fastest cache which is closest to the CPU. Typically it is divided into instruction cache and data cache. As the name suggests, the instruction cache is used to store instructions, *i.e.*, whenever the CPU fetches instructions from memory, it first searches the L1 instruction cache. If it finds the code there, it can circumvent the load from memory, which is costly. Respectively, the L1 data cache is searched before loading data from memory. The cache which is on the last level of the cache hierarchy is called the Last Level Cache (LLC). For our Intel, AMD and IBM test processors the LLC is the L3 cache, for the ARM processor, it is the L2 cache.

Cache sizes differ between CPUs. Table 2.1 gives an overview of cache sizes on our test CPUs. As we can see, the L1 caches of our test CPUs are between 32 KB and 64 KB. The L2 caches are larger with a size between 265 KB and 512 KB. Only the ARM Cortex-A57 has a larger L2 cache (2048 KB) because it does not have an L3 cache. L3 caches of the Intel, AMD and IBM test CPUs have a capacity of 3-10 MB.

As mentioned in the previous section, our test CPUs are multi-core CPUs, and some of them implement simultaneous multithreading as an optimization. For reasons of space and efficiency, caches are often not exclusive for one specific logical core. For example, our Intel and AMD test CPUs share L1, and L2 caches between logical cores and the LLC is shared between physical cores.

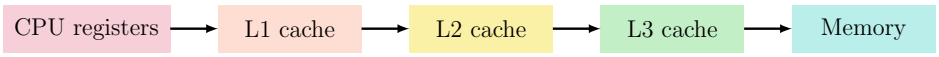


Figure 2.5: Memory hiearchy

Cache levels can be *inclusive*, *non-inclusive* or *exclusive* [62, 32]. If cache level A is inclusive to cache level B then A contains all data which is also contained in B. For example, Intel CPUs usually use inclusive L3 caches [32], which means that everything that can be found in the L1 or L2 cache is also contained in the L3 cache, but there might be data which is solely in the L3 cache. Inclusiveness makes cache coherence easier, with the trade-off of wasting memory due to duplicate data storage [62]. If cache level A is exclusive to cache level B then A does not contain any data which is contained in B. The advantage of this is that cache space is not wasted. The disadvantage is that enforcing cache coherency is more difficult. If cache level A is non-inclusive to cache level B then A is neither inclusive nor exclusive to B. For example, Intel's L2 caches are non-inclusive, which means that data residing in the L2 cache may or may not exist in the L1 cache.

To make sure that multiple cached copies of data that reside in different cores are up-to-date, so-called cache coherence protocols are defined. Cache coherence is an issue related to multi-core processors and the fact that not all caches are shared between cores [94, 5]. Imagine a situation between cores A and B of a multi-core CPU with three cache levels where both A and B have a block of a specific memory address in their L1 cache. Cache coherency problems can occur when A writes data to the address. The data might be updated in the A's private L1 cache, but when B attempts to read the data, it will not read the most recent version unless it is notified about the change of the value in some way. The cache coherence protocol defines a consistent cache update policy. It ensures that all caches contain the most recent version of data at any given point in time.

Frequently, modern caches are divided into *sets* or *ways*. Each cache set is again divided into multiple cache *lines*, also known as *blocks*. Each cache line or block is *tagged* to indicate which memory address it belongs to [32]. Additionally, each line has a *valid* bit. The valid bit indicates if the data is still up-to-date or if a write to memory has already happened earlier and invalidated the cache line. It is also used to determine if the cache line is empty. If there are n cache lines in a cache set, the cache is said to be *n-way set associative* [33]. If addresses a and b are congruent, the values at addresses a and b are stored in the same cache set.

Directly-mapped caches are caches where data from one specific address can be stored at exactly one location in the cache. Fully-associative caches are caches where data from one specific address can be stored anywhere in the cache. Respectively, n-way set associative caches are a solution between directly-mapped and fully-associative caches.

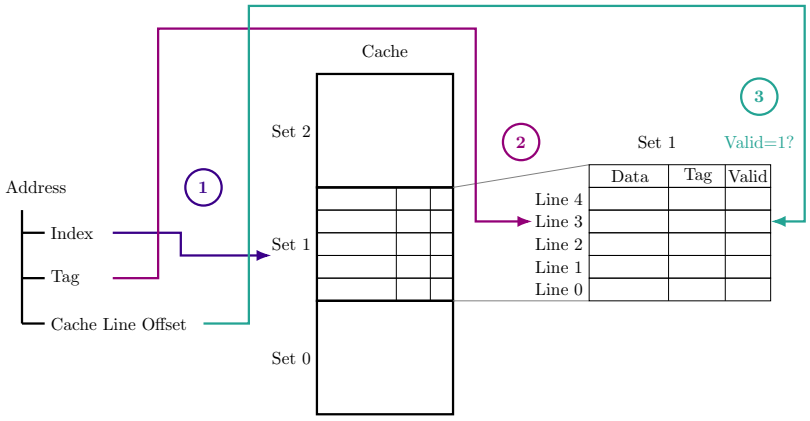


Figure 2.6: Basic operation of a 5-way set-associative cache. From the virtual and/or physical address, index, tag, and cache line offset are retrieved. Based on the index, the correct cache set is selected (1). Then the cache set is searched for data with a matching tag (2). If this data exists, the valid bit is checked (3). If the valid bit is 1, the data can be retrieved.

2.2.2 Operation of a Cache

In case the CPU wants to read data from memory, it follows the memory hierarchy shown in Figure 2.5. That is, it starts at the L1 data cache and looks for the data in there. Three information pieces are needed to search a cache: tag, index and offset. First, the correct cache set is selected by the index. This cache set contains multiple lines (n lines in an n-way set associative cache), and now one must find out which line contains the requested data. This is done using the tag, which is first computed from the data address and which must be compared with the tags in the cache. If a tag matches the computed one, the CPU found the correct cache line. Nevertheless, it might still be that an old version of the data was found or that the cache line is empty. Therefore, the CPU also checks if the valid bit is set. If all these requirements are fulfilled, the correct data is found and does not have to be loaded from memory (*cache hit*). If one of these steps fails, the CPU repeats the procedure for the L2 and L3 caches as well. In case there is no cache hit on any one of these levels, the CPU has to load data from memory (*cache miss*). This process is illustrated in Figure 2.6.

There are different possibilities which data to choose for tags and indices. Generally, either the physical or virtual address is used [49]. Virtual addresses are known instantly when the memory access happens, but physical addresses must first be retrieved from the address translation process. This process is quite costly, and therefore virtually indexed caches are faster in general. On the other hand, virtual addresses are only unique per process, and therefore multiple copies of the same data might be in the cache simultaneously.

One can distinguish between four different methods for cache tagging and indexing: VIVT (virtually indexed, virtually tagged), PIVT (physically indexed, virtually tagged),

VIPT (virtually indexed, physically tagged) and PIPT (physically indexed, physically tagged). VIVT caches are fast because there is no need to retrieve the physical address. However, the virtual tag is not unique, and the same piece of data might be in the cache twice, for example when processes share memory. PIVT caches have no advantage, for small caches since, on the one hand, latency is introduced to retrieve the index and collisions happen because of the virtual tag. VIPT caches are often used because the cache set can be selected while the physical address for the tag is retrieved, leading to an overall performance gain due to parallelism. Additionally, there will not be duplicate data in case shared memory is used. PIPT caches have the major drawback of slowness because of the physical address in use. There are no collisions in connection with shared memory. Today, most L1 caches are VIPT whereas PIPT is used for L2 and L3 caches [26]. The L1 cache set selection and the TLB lookup to retrieve the physical address can be done in parallel. Once the physical address is available, the L1 cache line can be slected. In case an L1 cache miss is encountered, the L2 cache is searched. Since the physical address is already available from the last step, it is convenient to use it for both tag and index.

As already mentioned above, it will happen that the cache is full and data, which is required by the program, cannot be found in the cache. To make space for new, more recent data, parts of the cache must be evicted. The cache replacement policy is used to decide which data to evict. There are several strategies [69, 72]. For example, the Least-Recently Used (LRU) strategy tracks the age for each cache line and replaces the one with the highest age. Whenever data is accessed, the age of the line is reset and incremented for the other lines. Pseudo-random policies select a random cache line to evict. A pseudo-random number generator is used as a source of randomness. Research showed that in modern CPUs, adaptive replacement policies are frequently in use [88]. They try to tune the cache replacement policy in a way such that the best results for the programs currently being executed are achieved [70]. Often, this involves observing accesses to cache lines and trying to figure out access patterns.

2.2.3 Cache Attacks

There are various forms of cache attacks, including Flush+Reload [92], Flush+Flush [27], Evict+Reload, [28] and Prime+Probe [80]. Researchers showed various attacks involving caches to build efficient covert channels. For example, in 2017, Maurice et al. [58] proposed an error-correcting, high-throughput covert channel using caches which can transmit more than 45 KBps. Other microarchitectural attacks like Spectre or Meltdown rely on cache attacks in order to transmit the leaked data to the attacker process.

Cache attacks exploit the timing difference which occurs between loading cached and loading uncached data. As mentioned in the previous section, loading data from memory takes a long time. Caches speed up this process, but the difference in timing between a cache hit and a cache miss is so significant that it can be exploited, as shown in Figure 2.7. In a typical cache attack, there are two processes, a victim and an attacker process.

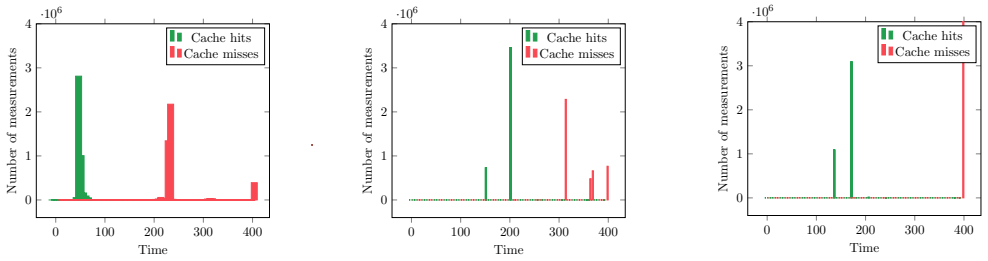


Figure 2.7: Differences between timing of cache hits and misses on Intel, ARM, nd AMD

The attacker process wants to spy on the victim process. Both processes run on the same machine.

Flush+Reload is a classic cache side-channel attack which requires shared memory between the attacker and the victim [92, 29]. It exploits the observation that attacker and victim share the LLC. Therefore, if either the attacker or victim accesses data, it is loaded into the LLC. If either the attacker or victim flushes a cache line, it is thrown out of the LLC. The first step the attacker takes is to map shared memory into his address space, for example a shared library [85]. The attacker wants to learn when the victim accesses address a. Therefore, the attacker flushes a from the cache and waits until the victim has finished executing the victim program. Now, if the victim ever accessed a, it will most likely be in the cache; otherwise, if the victim never accessed a, it will not be in the cache. In the last step, the attacker measures the time it takes to access a. If a is in the cache, the attacker gets a cache hit and the access time is low. If a is not in the cache, the attacker gets a cache miss and the access time is high. There are several ways how one can evict cache lines: x86 platforms provide the unprivileged *clflush* instruction, ARM provides several management instructions for their data cache (e.g. *dc csw*) and IBM Power offers *dcbf* (data cache block flush). Time is measured using the *rdtsc* instruction on x86, ARM provides a monolithic clock source, and IBM Power provides a special purpose register which can be read out to retrieve a stable timing source.

Flush+Flush [27] is a variant of Flush+Reload which relies on differences in the execution time of the *clflush* instruction. It is based on the observation that if data is not cached, *clflush* is faster. For example, on Intel CPUs *clflush* starts at the LLC. Uncached data is not in the LLC, which is inclusive to the L2 and L1 cache. Therefore, the L2 and L1 cache do not have to be checked, which saves time.

Evict+Reload [28] is also a variant of Flush+Reload. It is useful on platforms where a *clflush*-like instruction is either not available or privileged. To evict data at address a from the cache, the attacker fills the cache with addresses that are congruent to a until a is evicted from the cache. How many addresses must be accessed before a is evicted depends on the replacement policy in use.

Prime+Probe [65] has the advantage of not requiring shared memory. As a downside, it requires accurate knowledge about the cache replacement policy, which is only docu-

mented by manufacturers in rare cases, and the physical address. The attack consists of two phases, called *prime* and *probe* respectively. Assume the attacker wants to know if a specific cache set s was accessed by the victim. In the first step, the *prime* phase, the attacker fills the cache set s, according to the cache replacement policy. Then, the victim process is executed, and if it accesses data in s, it evicts data from the attacker. The attacker enters the *probe* phase in which he measures the access times to the memory addresses which he used in the *prime* phase. If he encounters a cache miss, the victim's program has accessed s.

2.3 CPU Optimization Mechanisms

In recent years, there have been numerous tweaks applied to CPUs to gain the most performance from it. Instruction pipelining is probably the oldest and most fundamental optimization concept. It takes advantage of parallelism introduced to the fetch-decode-execute action. Manufacturers have also done small modifications on a much lower level, the instruction level, including move elimination, zero idioms, and one idioms. Speculative execution and branch prediction are amongst the more sophisticated optimization techniques, where the CPU tries to make assumptions about future program execution by guessing. Out-of-order execution aims at reordering instructions in a way such that they can be executed in the most efficient way (for example, execute a load from memory before an arithmetic instruction). Since speculative execution and out-of-order execution are significant for this thesis, they are described detailed in Section 2.5 and Section 2.4. On a higher level, there is simultaneous multithreading, which takes advantage of the concept of logical CPU cores.

2.3.1 Instruction Pipelining

The IBM 7030 Stretch is considered the first pipelined machine [67], constructed in 1959. Today, pipelining is the key technique used for building efficient CPUs.

The main idea of pipelining is to divide the execution of an instruction into multiple steps [23]. By parallelizing the processing of these steps, multiple instructions can be executed in parallel. Each of these steps is called a *stage* in a pipeline [32, 74]. The term *pipeline* comes from the topological structure of this system since stages are chained one after each other. Instructions enter at the one end, go through all the stages and leave at the other end. The basic RISC pipeline covers five stages: fetch, decode, execute, memory access and write-back [32]. Today, CPUs use more than just five stages: Intel Skylake and Haswell CPUs have 14-19 [52], AMD Zen CPUs have 19 [53], and IBM Power9 CPUs have 12-16 pipeline stages [51]. Although, the total execution time of all instructions is reduced, the execution time of a single instruction is slightly increased because the pipeline introduces complexity to the overall CPU architecture which must be managed [32].

```
1    sub eax, eax
2    xor eax, eax
```

Listing 2.1: Zero idioms

```
1 add ecx, 2
2 mov eax, ecx
```

Listing 2.2: Example of move elimination. The first instruction can be reformulated to `add eax, 2`, the second can be eliminated.

```
1 mov eax, 10
2 mov ebx, 10
3 cmp eax, ebx
```

Listing 2.3: Example where the CPU does not apply move elimination, but could.

During the execution of instructions, pipeline *hazards* might occur which cause the pipeline to *stall*. Generally, hazards occur whenever it is not possible to execute all the instructions which are currently in the pipeline in the given order. They are categorized into three types: data hazards, control hazards and structural hazards [32, 79, 15]. Data hazards occur when the current instruction depends on results which are not yet available, for example, when the result will be computed by an instruction earlier in the pipeline. Control hazards arise from instructions changing the value of the program counter, for example, a branch instruction. Structural hazards occur when the execution of an instruction requires a specific resource, an execution port or similar, which is currently exhausted. Hazard handling is done by stalling the pipeline, *i.e.*, waiting until the hazard resolves.

2.3.2 Move Elimination, Zero Idioms, and Ones Idioms

Optimization can also happen on the instruction-level by filtering out instructions which technically should not require any resources. Although they have to wander the whole pipeline, they do not require execution resources like execution units.

Zero idioms are data-independent instructions which will result in setting a register to zero. The CPU recognizes these instructions and handles the execution in the register allocation phase by allocating an empty register [20, 10]. The instruction never requires any execution resources. Since those instructions are independent of each other, they are called dependency-breaking instructions [20]. Listing 2.1 shows examples of zero idioms.

Ones idioms are data-independent instructions which will result in setting a register to one. Unlike zero idioms, they will need an execution unit [20]. Intel CPUs only apply this to *pcmpeq* instructions, which compare two MMX registers. If the registers are equal, the destination register is set to all ones, otherwise to all zeros [38].

Move elimination applies to register-to-register moves and chained register-to-register moves [52, 20]. Register-to-register moves are *mov* instructions which have registers as operands. If they are preceded by any other instruction, it can be eliminated, and the instruction can be rewritten to store the results into the desired register directly. An example is given in Listing 2.2. Here, the preceding instruction is the *add* instruction in the first line. The CPU can eliminate the *mov* in the second line and directly use `eax`

as the destination register of the *add* instruction.

Move elimination is not applied in all possible cases, as shown in Listing 2.3. The *cmp* instruction is not modified to work with immediate values directly. The reason for this is that adding more logic for move elimination would also add more complexity to the pipeline. On the AMD Zen platform, move elimination can also be done for floating point operations [53].

2.3.3 Branch Prediction

Branch Prediction is a feature most modern CPUs implement. It is as old as pipelining and also very powerful. The IBM 7030 Stretch is considered the first machine to implement it [76].

As already mentioned in the previous section, control hazards happen when a branch instruction enters the CPU's pipeline. Theoretically, the pipeline must stall until the branch is fully evaluated because it is not known before whether to take the branch or not. Therefore, the CPU consults the branch prediction unit to guess whether to take the branch or not. Following the recommendation of the branch predictor, the branch which is most likely taken is fed into the pipeline, and the instructions are executed speculatively until the result of the branch instruction is finally known. Section 2.5 gives more details on speculative execution. If it turns out in the end that the guess was right, execution time was saved. If the opposite is the case, the other branch will be fed into the pipeline, and everything which was executed speculatively will be discarded. However, in the end, the cycles which were wasted due to a misprediction are not significantly more than what would have been needed without branch prediction [20].

We distinguish between two basic branch prediction methods, static and dynamic prediction. While static prediction happens at compile time, the CPU does dynamic prediction at runtime. Dynamic prediction has the big advantage that it can adapt *dynamically* to the program's behavior [14, 46]. The major downside is that additional hardware resources are needed which not only consumes chip area but also adds complexity to the total chip architecture [79]. In the following, we focus on dynamic prediction.

Modern CPUs apply branch prediction to conditional branches, but prediction also happens for unconditional branches in terms of predicting the jump target address. Unconditional branches are always taken, for example, returning from a function or calling a function. Conditional branches might or might not be taken, depending on the outcome of the branch instruction, for example, an if-then-else-statement [20].

The two central microarchitectural elements for performing those predictions are the Pattern History Table (PHT) and the Branch Target Buffer (BTB) [79, 20, 93]. The PHT maps a branch address to one or more bits, depending on the prediction strategy, which indicates whether the branch was taken in the past or not. In contrast to that, the BTB extends this prediction by providing knowledge about the target of the branch. The BTB maps a branch address to the correct target address. Whenever an unconditional

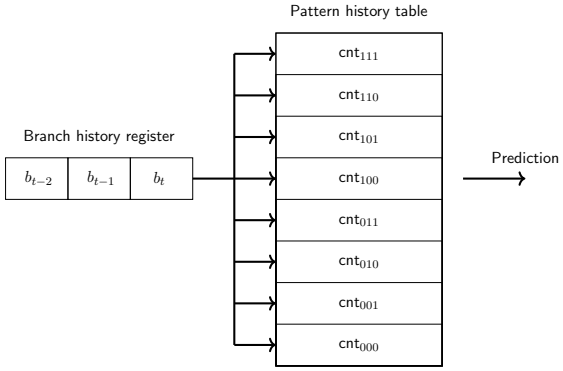


Figure 2.8: Example implementation of branch prediction in a processor. The branch history register stores the three previous outcomes of a branch, *i.e.*, if it was taken or not. Based on that, the selection in the pattern history table is made, which contains a counter. This counter finally predicts whether to take the branch or not.

branch is executed, or a conditional branch is taken, the BTB is searched for the target address. If it does not contain the target address, for example when the branch is taken for the first time, the target address is inserted into the table as soon as it is known. The second time the branch is taken, the target address can easily be fetched from the BTB instead of having to wait for the PC to increment and deliver the correct target address [20]. However, the BTB provides only predictive information, also due to its size limit. There is no guarantee that the branch target provided by it is the correct branch target, especially for conditional branches. Very often, modern CPUs combine PHT and BTB into one single hardware component [20].

There are various methods how to implement branch prediction. The simplest way is to equip the BTB with a two-bit counter [20]. The counter is incremented if the branch is taken and decremented if it is not taken. It can never exceed 3 or fall below 0. If the BTB is asked to predict if the branch is taken, a counter value of 0 or 1 means to not take it, a counter value of 2 or 3 means to take it.

A second approach is to combine the PHT with a branch history register, as illustrated in Figure 2.8. The branch history register, a shift register, stores the history of the last n outcomes of the branch (0 for not taken and 1 for taken). In Fiugre 2.8, n equals 3. For example, if the branch was taken and then not taken twice, the value in the branch history register would be 100. For each of the possible 2^n combinations, a counter, working as described in the previous section, is stored in the PHT [20]. According to Fog [20], this method is called *two-level adaptive predictor*.

Modern CPUs apply the two concepts described above in some form. However, manufacturers keep secret the exact structure and operations of the branch prediction units. Almost all information which is currently known has been reverse-engineered [53, 52, 51, 24, 20].

```
1 add ebx, eax
2 mov ebx, byte[ebx]
3 add edx, ecx
```

Listing 2.4: Execution sequence as given by binary

2.3.4 Simultaneous Multithreading

Simultaneous Multithreading (SMT) is a technique that introduces logical cores to run on a single physical CPU core [82]. These logical cores have their own architectural state, but actually they run on the same physical core. The underlying assumption which justifies why SMT pays up is that a single process or thread most of the times cannot utilize the whole CPU core, but two processes might. The hardware execution resources are either completely shared or statically partitioned [56]. Shared means that all hardware threads can use a particular resource. For example, AMD Zen CPUs share the μop cache and schedulers as well as execution units. Statically partitioned means that the component is divided into as many parts as logical cores and each hardware thread only accesses his part of the resource. AMD Zen processors statistically partition components like the retire queue and the μop queue.

Most modern CPUs use a form of SMT today. Intel has been using *hyperthreading* for quite a few years now, but also AMD provides SMT technology. IBM Power9 provides CPUs having 4 or 8 hardware threads per core. ARM introduced SMT in their Cavium line and in 2018, they also started the ARM Cortex-A65AE CPUs which also implement SMT [47].

SMT provably has several security issues. Since logical cores share multiple microarchitectural components, an attacker running on one logical core can spy on the victim running on the other logical core. This also creates several side channels, various attacks exploiting SMT have been published, including PortSmash [4], TLBleed [25], and several Spectre variants [12].

2.4 Out-of-Order Execution

Out-of-order execution is another powerful optimization technique applied by pipelined processors. The first machine implementing out-of-order execution was the IBM System/360 Model 91 in 1966 [36], but today most modern CPUs implement it.

On the hardware level, the *reorder buffer* is the central element supporting out-of-order execution in the CPU. It contains μops, as they are waiting to be scheduled to one or more execution units. The μops might be reordered there and sent to the scheduler. After their execution, the μops return to the reorder buffer. The order in which μops are retired is often not the same as the order in which μops are executed, which is the

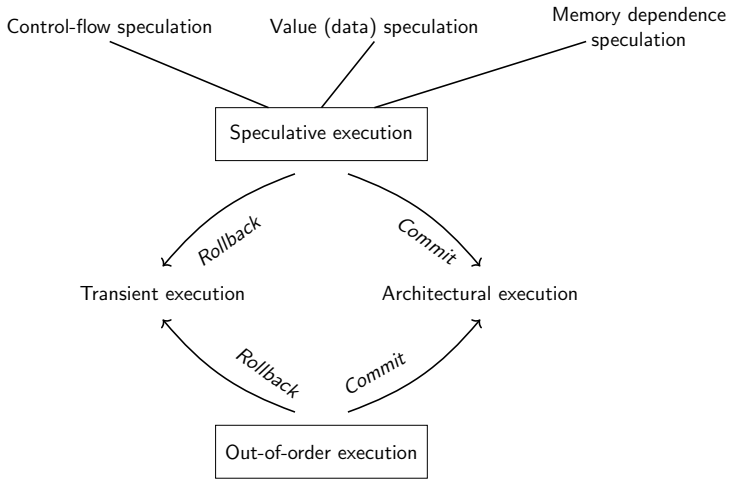


Figure 2.9: Control-flow speculation, value or data speculation and memory disambiguation lead the CPU to speculatively execution. Whenever speculative execution or out-of-order execution is not successful, the CPU does a rollback, and we talk about transient execution. Whenever speculative execution or out-of-order execution is successful, the CPU commits the results into the architectural state, and we talk about architectural execution.

reason why this concept is called out-of-order execution. In the reorder buffer, the μops wait for their *retirement*. Retirement refers to making changes (caused by the execution of an instruction) architecturally visible. Retirement has to happen in-order; otherwise, this destroys the programmer's illusion of programs being executed in-order.

The reason why CPUs apply out-of-order execution is performance [32]. For example, the CPU might not be able to execute a specific instruction because operands of that instruction are not ready yet. Then, the CPU can execute another, later, instruction first, for which the input is already known, instead of waiting for the first instruction to be ready to execute. However, the later instruction, which is executed while the operands of the first instruction become available, must be independent from the first one [20]. Independent means that no operand of the instruction depends on the previous one.

Consider the example in Listing 2.4, which shows a sequence of three instructions as given by the binary. The *mov*-instruction in line 2 depends on the *add*-instruction in line 1. The CPU cannot execute the *mov* without the *add* being completed. The *dec*-instruction in line three is independent of the previous instructions. The CPU can execute it any time. Now, assume that the CPU wants to execute line 2 and realizes it needs the result of the *add* to do this. It can execute the *dec*-instruction in line three out-of-order while it is waiting for the *add* to be finished.

2.5 Speculative Execution

As already mentioned in earlier sections, *speculative execution* is applied whenever the processor has to execute a branch instruction. Due to out-of-order-execution, the CPU might not yet know whether to take the branch or not. Theoretically, a pipeline stall happens, and the CPU has to wait until the result is computed. This is a huge bottleneck, and, therefore, the CPU uses branch prediction and continues executing instructions speculatively [40]. This means that the results of these instructions are not retired and are not architecturally visible unless the result of the branch is known [20]. Once it is known for sure by computing the branch instruction whether to take the branch or not, the speculatively executed instructions are either rolled back, in case the prediction was wrong, or committed if the prediction was correct [12]. However, one must not forget that branch prediction or formally control-flow speculation is only one possibility to induce speculative execution on a CPU, as shown in Figure 2.9.

Value or *data speculation* relies on the concept of value locality, which means that the same value results from the same instruction multiple times in a row [48, 60]. Respectively, the CPU speculates on the result of operations when - for some reason - the result cannot be known immediately.

Another option for the CPU to speculate is *memory dependence speculation* [90, 61]. To overcome memory latency, the CPU internally uses a store buffer [53, 52]. Whenever the CPU has to perform a write to memory, it puts information about the write into the store buffer such that execution can continue and the pipeline does not stall. However, a load might be issued which depends on an earlier store. If the store has not yet been handled, it still resides in the store buffer, and the CPU performs *store-to-load forwarding* [20, 9, 90]. This means that the store from the store buffer must be forwarded to the load because the load cannot load from memory since it will read an old value. Memory dependence speculation predicts if this is the case, *i.e.*, it predicts if a load depends on an earlier store [90] and, therefore, indirectly allows the CPU to predict whether the data has to be searched in the store buffer or not.

Figure 2.9 gives an overview of the two types of execution which might happen in a CPU. *Architectural execution* means that the results were *committed* and are architecturally visible. *Transient execution* means that the CPU executed instructions it should not have, for example, because of a branch misprediction, and the results are *rolled back*. Transiently executed instructions do not leave architectural traces, but microarchitectural traces, for example, changes in the CPU's cache state. It has been shown that this can be a security issue since an adversary can use for example a cache covert channel to make those traces architecturally visible [12, 50, 44].

The *speculation window* (or *instruction window*) describes the time frame in which the CPU executes instructions speculatively. The size of this window is limited, and this thesis focuses on analyzing this limit in different situations. Attackers benefit from a larger speculation window because more instructions can be executed before the rollback

```
1  if(x < len)
2    maccess(oracle[array1[x] * 4096]);
```

Listing 2.5: Spectre-PHT gadget

happens, leaving more microarchitectural traces.

2.6 Transient Execution Attacks

Transient execution attacks exploit the observation that transiently executed instructions leave microarchitectural traces [12]. These traces are made architecturally visible by attackers using - in most cases - a cache covert channel, as already described in the previous sections.

The fundamental works in this direction were published in January 2018. There has been Meltdown et al. [50], exploiting out-of-order execution to read kernel memory from userspace. Spectre attacks [44] leak information through speculative execution in combination with branch prediction.

Since then, there has been done a lot of research in this direction. New attack variants of both Spectre and Meltdown are discovered continuously [43, 34, 54, 86, 84, 8, 77, 12].

We briefly describe Meltdown-like attacks since they are relevant for parts of this thesis. We also show how to improve Spectre attacks, so we are describing them in more detail.

2.6.1 Spectre Attacks

Spectre attacks target transient execution caused by speculative execution. Several Spectre variants have been discovered, what they all have in common is that they exploit the CPU's branch prediction mechanisms by mistraining the respective prediction unit. Canella et al. [12] propose a categorization of those attacks based on the type of mistraining and the prediction unit. Spectre-PHT (Variant 1) uses the Pattern History Table (PHT) [44], Spectre-BTB (Variant 2) uses the Branch Target Buffer (BTB) [44], Spectre-STL (Variant 4) exploits store-to-load forwarding, and Spectre-RSB exploits mispredictions of the Return Stack Buffer (RSB) [54, 45]. Besides that, there have been various attacks based on the original Spectre attacks [13, 64, 73, 11].

Spectre-PHT (Spectre Variant 1) exploits conditional branch misprediction. It requires the existence of a Spectre gadget as suggested by Listing 2.5, and can be located in any arbitrary function. `array1` is an array of `len` bytes, and `oracle` is an array of `256 * 4096` bytes. First, a bounds check is done which prevents an out-of-bounds access to `array1`. The function `maccess` is used to access the memory on a given address. The goal of the attacker is to leak a secret located in the victim's process. Before the attack

begins, the attacker ensures that all entries of `oracle` are not cached, for example by using the `clflush` instruction. The actual Spectre attack consists of multiple phases. In the first phase, the branch-mistraining is done. There are four different strategies for branch mistraining [12]. The attacker can either mistrain within the same address space as the victim or in another, attacker-controlled address space. Furthermore, the attacker can use a branch at the same address as the victim (in-place) or a branch at a congruent address (out-of-place). The goal of the mistraining phase is to influence the branch predictor in a way that it will recommend to take the branch the first time after the mistraining phase. In the second phase, the gadget is executed with an invalid `x`, *i.e.*, `x >= len`. The branch predictor recommends taking the branch because it was trained like that before. `maccess` is called with an invalid `x` and speculatively loads `oracle[array1[x]*4096]` into the cache. However, the CPU later realizes that the branch was mistakenly taken and actually should not, so it does a rollback. In the last phase of the attack, the attacker leaks the value of the secret byte, `array1[x]`, via a cache covert channel by finding out which location in `oracle` was cached. He will get cache misses for all indices of `oracle` which were not cached, but one cache hit for index `array1[x]`. The attacker effectively leaked `array1[x]`.

What we see in Spectre-PHT attacks is a race condition since it is not guaranteed that the speculative execution happens before the condition is evaluated. Depending on various factors, for example, interrupts or the workload on execution ports, the evaluation might be finished before the speculation can happen. In this thesis, we try to make this situation more unlikely by extending the speculation window. Spectre-PHT attacks have shown to be feasible on Intel, ARM, AMD, and IBM Power9 processors. Recently, a proof-of-concept was published for RISC-V platforms [22]. It was also shown that not only transient loads but also transient stores to memory are possible in Spectre attacks [43]. The gadget would change to `oracle[array1[x]*4096] = value;`.

Spectre-BTB (Spectre Variant 2) targets indirect branch misprediction, more specifically the prediction of branch targets. The attacker poisons the BTB with addresses he wants the victim to jump to [12, 44]. A prerequisite for a successful attack is again the existence of a gadget which transfers the secret data to the attacker over a covert channel. An attacker who wants to mistrain the branch target predictor must be aware of the virtual address of the gadget in the victim's address space. The attacker can then start a co-located process on the same physical core as the victim and execute indirect branches to the victim's gadget address. The BTB is shared among logical cores, and therefore the BTB will be poisoned with the gadget address.

Spectre-STL (Spectre Variant 4) exploits store-to-load forwarding [12, 35] and misspeculating memory dependencies. In Spectre-STL, the store buffer is bypassed because of a wrong prediction and old values are read from memory while the CPU searches the store buffer for the correct value.

Spectre-RSB exploits the Return Stack Buffer (RSB) [12, 45]. The RSB is a stack containing return addresses of previous call instructions. When the CPU executes a return instruction, it does not use the BTB but the RSB to predict the target of the

```
1  int v = *secret;
2  maccess(array2[v * 4096]);
```

Listing 2.6: Transient execution sequence in Foreshadow-SGX

unconditional jump. Due to the limited size of the RSB, incorrect speculations can easily occur because old entries get overwritten by new ones if too many return instructions are issued. The same thing can happen vice versa: if there are more return instructions than entries in the RSB, the CPU uses the BTB instead, and one can run a Spectre-BTB attack [45].

Since the release of Spectre attacks, several attacks based on the original Spectre attacks have been discovered. SGXPectre [13] demonstrates how to disclose secrets from Intel SGX enclaves using Spectre-BTB while SGXSpectre [64] tries the same but using Spectre-PHT. NetSpectre [73] shows how to leak secrets using Spectre over the network. SMoTherSpectre [11] combines port contention and Spectre to disclose secret information.

2.6.2 Meltdown Attacks

Meltdown attacks exploit transient out-of-order execution [50, 12], more specifically exploit the observation that exceptions are only raised when instructions retire. Whatever is executed out-of-order before the exception appears leaves microarchitectural traces and can be read out by an attacker using a covert channel. Canella et al. [12] propose a categorization of Meltdown attacks based on the type of exception. Meltdown-US (simply called Meltdown) uses a page fault raised because of an access to kernel memory from userspace [50]. Foreshadow exploits a page fault occurring because of a zeroed present bit [84, 86]. There exist various other non-PF-based Meltdown variants, for example, Meltdown-GP which exploits a general protection fault [8].

Meltdown-US can be used to read kernel memory from userspace. In modern operating systems, page tables are equipped with the supervisor flag, which indicates whether a page belongs to the user space or the kernel space. Whenever a userspace process tries to access a kernel page, the supervisor flag is checked, and a page fault is thrown due to insufficient permissions. The attack consists of three phases. In the first phase, the attacker chooses a kernel memory location which he wants to leak and loads it into a register by dereferencing it. Dereferencing a memory location triggers a page table walk, and a page fault is raised because a kernel space access is detected from user space by checking the supervisor flag.

In the second phase, which happens before the exception is raised, the attacker transiently executes an instruction which accesses a cache line depending on the secret value in the register. Note that meanwhile, the exception is raised and must be handled, other-

wise the program aborts. There are several possibilities, including an exception handler, a TSX transaction or speculative execution. The last phase is the same as in Spectre attacks: the attacker finally leaks the secret by launching a cache covert channel.

Foreshadow [84, 86] is a variant of Meltdown and comes in three different types. Intel named Foreshadow-type attacks *L1 Terminal Fault* (L1TF). The initial variant, Foreshadow-SGX, was published as an attack against Intel-SGX. Intel SGX is a secure enclave, a private region of memory residing inside the virtual address space of a process, allowing secure computations, for example for cryptographic purposes. An attacker who wants to exploit Foreshadow-SGX has the goal to read a secret value which is stored in the enclave from outside. The attack comprises three phases. In the first phase, the attacker makes sure that the enclave secret resides in the L1 cache. This can be done by calling an enclave function which accesses the secret. In the next phase, the attacker makes the enclave page non-present, for example by calling `mprotect`. In the fourth phase, the attacker executes code as shown in Listing 2.6. It starts with dereferencing `secret`, which causes a page fault due to the cleared present bit of the previous phase, skipping SGX's abort page handler, which is called whenever an enclave address is accessed from the outside. However, the instruction of line 2 is executed transiently, loading the contents into the cache. The last phase of the attack involves a standard cache covert channel to leak the secret value.

Two more Foreshadow variants, Foreshadow-OS and Foreshadow-VMM were published [86], both generalizing Foreshadow-SGX to a Meltdown variant targeting a zeroed present bit. While Foreshadow-OS exploits terminal faults to read kernel memory from user space, Foreshadow-VMM allows an attacker, who controls a virtual machine, to read any chosen physical memory address in L1 from the host.

2.7 Port Contention

PortSmash, an attack exploiting SMT and execution ports, was published in 2018 [4], followed by SMoTherSpectre [11] in 2019. Both attacks exploit port contention on Intel platforms. Port contention occurs when too many μops requiring the same execution port have to be executed in a small amount of time. The scheduler is responsible for managing which μops will be sent to which execution ports and do this assignment as efficient as possible, e.g., by applying load balancing. Each execution port fulfills a specific task. Therefore, μops can only be sent to the ports which can actually execute them. For example, the *popcnt* instruction can solely be scheduled to port 0 on an Intel Skylake CPU, while *shld* can be issued to port 0 or port 6 [21].

One can build a covert channel out of this, given the prerequisite that sender and receiver are co-located to each other. Co-located means that they run on the same physical core, but on a different logical core. Sender and receiver agree on two instructions, one for sending a 0 and one for sending a 1. For example, they can choose *popcnt* for sending a 0 and *fadd* for sending a 1. If the sender wants to transmit a 1, it starts executing

a sequence of *popcnt* instructions. Concurrently, the receiver executes first a sequence of *popcnt* and secondly a sequence of *fadd* instructions and measures the execution times of both sequences. Either port 0 (in case both execute *popcnt*) or port 5 (in case both execute *fadd*) will be contented, yielding more time consumption which allows the receiver to know if either 0 or 1 was transmitted.

As proposed by Aldaya et al. [4], this covert channel can be transformed into a side channel which they call PortSmash. The prerequisites are - same as for the covert channel - the co-location of victim and attacker. The main idea is that the attacker contends one or more ports and can draw conclusions about what the victim executes by measuring the time consumption of the issued instruction sequence. If the instructions executed by the victim depend on a secret, the attacker can learn about this secret.

Chapter 3

Attack Analysis Setup and Framework

In this chapter, we first describe the automated framework we use to analyze Spectre-PHT attacks. We describe the structure of the framework and the interaction of the framework's components. Additionally, we give details on how tests are executed fully automatically and which configurations can be made by the user. In the second part of the chapter, we outline a test scenario in our framework. We describe which categorization was applied to the test scenarios.

3.1 Automated Framework

We provide an automated framework for testing scenarios involving Spectre-PHT attacks. This framework was ported to all our test platforms, *i.e.*, it can be used on Intel, AMD, ARM and IBM Power. The test devices used for our experiments are described in Table 3.1. All of them except the ARM Cortex-A57 have simultaneous multithreading enabled.

The primary goal of our analysis is to give insights on how different conditions increase or decrease the performance of Spectre-PHT attacks on different platforms. The design of our experiment is sketched in Listing 3.2. By performance, we mean the likelihood that an attacker can leak data. Respectively, increased performance means the attack works better, and the leakage rate is higher. Generally, one can configure four parameters: `test_id`, `repetitions`, `process_repetitions` and `cache_miss` which we describe in the following. Note that `test_id` is the only non-optional parameter. `repetitions`, `process_repetitions` and `cache_miss` can be chosen by the user, but are provided by the framework by default.

Table 3.1: Overview of test devices used in the experiments

Motherboard/SoC	CPU	SMT (#logical/#physical)	Compiler
Z170-WS	i7-6700K	✓	gcc 7.3.0
	4.00GHz	(4/2)	
AMD PRIME X399-A	AMD Ryzen Threadripper 1920X	✓	gcc 7.2.0
	3.50GHz	(24/12)	
Nvidia Tegra TX1	ARM Cortex-A57	×	gcc 5.4.0
	2.00GHz	(4/4)	
Raptor Talos II	IBM Power9	✓	gcc 7.3.0
	2.154GHz	(16/4)	

3.1.1 Attack Accuracy (TPR) and Throughput

The goal of our analysis is to provide an exact measurement which allows us to compare attacks using different conditions. For this purpose, we use the true positive rate (TPR) and throughput in bytes per second achieved by a specific attack scenario.

As suggested by Listing 3.2, in one test run we leak one byte and check the outcome of the experiment. For this purpose, we define b, a predefined byte, (`H` in our example), and a, a byte which is not equal to b. The outcome of the experiment can either be true positive, true negative, false positive or false negative:

1. The result is true positive in case we tried to leak b and actually leak b.
2. The result is false negative in case we tried to leak b but leaked any byte a.
3. The result is false positive in case we did not try to leak b but leaked b.
4. The result is true negative in case we did not try to leak b but actually leaked any byte a.

Our analysis focuses on the condition positive scenario, *i.e.*, we are only interested in true positive and false negative results. In a more practical setting, this means that in one test run we try to leak b and check if b was effectively leaked (increment the counter for true positives) or not (increment the counter for false negatives). From the number of true positives and false negatives, we compute the true positive rate (TPR), also known as recall [18].

Not all tests leak bytes equally fast. For example, a test which accesses a chain of flushed variables might achieve a higher TPR than a test which involves a multiplication chain, but due to repeatedly flushing the cache it might be slower in leaking than the multiplication test. We capture this by stating the throughput in bytes per second.

```
1 char access_array(int x)
2 {
3   int res = 0;
4   asm volatile(
5     "mov $0, %%r11\n"
6     "imul %%r11d, %%r11d\n"
7     "imul %%r11d, %%r11d\n"
8     "imul %%r11d, %%r11d\n"
9     "add %1, %%r11d\n"
10    "mov %%r11d, %0" : "=r"(res) : "r"(x) : "r11");
11  if(res < len)
12    maccess(oracle + data[x] * 4096);
13 }
```

Listing 3.1: Example of a dependency chain for x86 with $n = 3, i =$`imul`

3.1.2 The `test_id` Parameter

`test_id` selects which test should be executed, *i.e.*, which condition should be used in the Spectre-PHT attack gadget. Since each platform has its own instruction set architecture, the instructions, as well as the conditions, will be slightly different for each. Examples are the basic condition, `x < len`, single integer ALU operations, AVX instructions and referencing uncached memory. A detailed overview is given in Section 3.2. In Chapter 4, when we discuss analysis results, we apply an intuitive naming scheme to the test cases. This naming scheme is also explained in Section 3.2.

A large part of our tests works with *dependency chains*. A dependency chain consists of n instructions i_s, where $s \in \{1, ..., n\}$ and it holds that i_s *depends* on i_{s-1}, *i.e.*, i_s cannot be computed until i_{s-1} is finished. Additionally, the prefix and postfix operations of the dependency chain (initializations, ...) are dependent on the instructions. Listing 3.1 gives an example of a dependency chain constructed of the *imul* instruction. In line 5, register `r11` is zeroed, then it is multiplied three times. Note that the second *imul* instruction depends on the first one and the third one depends on the second. In the end, the dependency chain's result is added to the index `x` and stored in a variable `res` to establish a data dependency here as well. In our tests, we investigate the influence of n and i on the performance of the attack. However, what is crucial in these tests is that there is a strict data dependency between instructions because then the CPU is forced to execute all instructions before the branch instruction. This is time-consuming, and therefore the speculative memory access has a higher chance to succeed. If this dependency did not exist, the CPU would start to reorder them, and the speculation window would not be extended.

```
#define test_id 5
#define repetitions 1000
#define cache_miss 170

char* data "data|H";
char* oracle;

char access_array(int x)
{
  size_t len = 5;
  #if test_id == 0
    if(x < len) {
  #elif test_id == 1
  ...
  #endif
    maccess(oracle + data[x] * 4096);
  }
}

int main()
{
  //Cache miss threshold for the cache covert channel
  if(chache_miss == 0)
    cache_miss = auto_detect_cache_miss_threshold();

  unsigned int TP = 0;
  unsigned int FN = 0;

  // Allocate memory for oracle
  //...

  while(cnt < repetitions)
  {
    //Start throughput measurement
    t1 = rdtsc();

    //Flush page
    flush(oracle + 'H' * 4096);

    // Mistrain
    for(int i = 0; i < 20; i++)
      access_array(0);

    // Out-of-bounds access
    access_array(5);

    // Leak via cache covert channel if access was out-of-bounds
    if(is_cache_hit(oracle + 'H' * 4096))
      TP++;
    else
      FN++;

    //End throughput measurement
    t2 = rdtsc();
    print("Throughput:", t2-t1);
  }
  print("TPR:" TP / (TP+FN));
}
```

Listing 3.2: General structure of an experiment testing the performance of Spectre-PHT with different conditions

3.1.3 The `process_repetitions` and `repetitions` Parameters

In general, Spectre attacks do not always perform the same because of the race condition which allows transient execution. Various other factors can influence the performance, including the overall system load and the behavior of other processes on the same machine. Therefore, we must ensure that our test results are statistically relevant.

We define the term *experiment* as trying to leak a byte, each experiment is repeated `repetitions` times in one process. To minimize the impact of noise coming from different sources, e.g., ASLR, we repeat this test `process_repetitions` times in different processes. The user can optionally choose both parameters, but we provide meaningful values for each platform by conducting a preliminary study. In this preliminary study, we investigate the TPR, averaged over `process_repetitions`, which is reached for a specific number `repetitions`. The TPR will converge and stabilize after sufficiently many tries and will fix `repetitions` and `process_repetitions`.

3.1.4 The `data` Parameter

We use the same data in all our tests to make the test results comparable. As shown in line 5 of Listing 3.2, this data consists of two parts, an accessible part, `data|` and a secret part, `H`. The first part is used to train the branch predictor whereas the second part is used to demonstrate how secret data can be leaked using an out-of-bounds access. In a real attack environment, the attacker can, of course, choose the address from which location he wants to leak the data.

3.1.5 The `cache_miss` Parameter

This parameter differs from CPU to CPU and defines the cache miss threshold. Although this parameter is chosen automatically by default, the user has the option to set it manually.

3.1.6 Size of the Speculation Window

Another indicator of good or bad attack performance is the size of the speculation window. Our framework provides an automated way to find the optimal size of the speculation window for each of the conditions.

The size of the speculation window is measured using *add* instructions, as it is sketched by Listing 3.3. First, the respective register is zeroed, and then we execute *add*-instruction pairs, adding 1 and -1 alternately. We store the result into a register which is finally used in the memory access. Theoretically, any instruction can be used for measurement, but since *add* is available on all platforms tested, it was the most convenient to use.

The measurement instructions must be data-dependent. If there is no data dependency

```
1  char access_array(int x)
2  {
3    //...
4    if(/*condition to test*/)
5    {
6      int res2;
7      asm volatile(
8        "mov $0, %%rdx\n"
9        "add $1, %%rdx\n" "add $-1, %%rdx\n"
10       "add $1, %%rdx\n" "add $-1, %%rdx\n"
11       "add $1, %%rdx\n" "add $-1, %%rdx\n"
12       //...
13       "mov %%rdx, %0" : "=r"(res2) : : "rdx");
14       );
15     maccess(mem + data[x+res2] * 4096);
16   }
17 }
```

Listing 3.3: Example of how the size of the speculation window can be measured on an Intel CPU

between them, the CPU could reorder the instructions and the memory access is executed at the very beginning, before the *add* instructions. When the rollback happens, the memory access might already be executed, but not all the *add* instructions, and we cannot measure the size of the speculation window. In our implementation, the CPU must execute the *add*-pairs before the memory access can happen and by the number of *add*-pairs, we can measure the size of the speculation window.

We apply a binary-search-like algorithm in order to find the size of the speculation window for each condition. We define the lower bound for the search as 0 and the upper bound at half of the size of the reorder buffer. During this process, we test a specific number of *add*-pairs. If the average TPR is higher than a pre-defined threshold, we continue trying an increased number of *add*-pairs. Otherwise, we try a lower number of *add*-pairs until the average TPR is sufficiently large and we found our window size.

3.1.7 Framework Structure

Figure 3.1 shows the four most essential parts of the framework. The outermost script, *start_experiment_multiple.sh*, defines a list of tests to be executed. For each of these tests, *start_experiment.sh* is invoked, taking the test number as an input parameter. Actually the test number (`test_id`) is the only thing the user has to give to the framework but there are three other optional configuration parameters, `repetitions`, `process_repetitions` and `cache_miss`. Next, *prepare_main.py* is called which creates *main.c*, the main file, taking over the user configurations. Coming back to *start_experiment.sh*, we distinguish between tests which involve chains and tests which do not. The difference

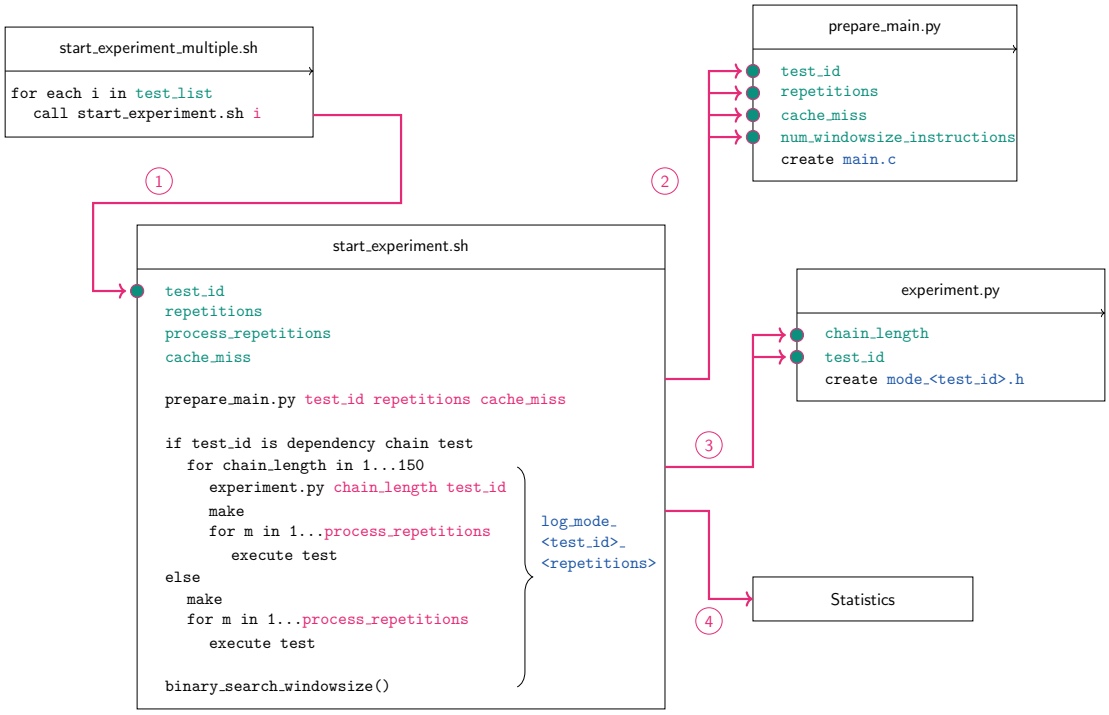


Figure 3.1: The automated test framework used in this work. In *start_experiment_multiple.sh* one specifies which scenarios to test. The actual test is executed by *start_experiment.sh*, which calls *prepare_main.py* to create *main.c*. In the next step, the test is started. The results are saved in a log file which statistically analyzed in the last step.

between them is that normal tests are executed `process_repetitions`× `repetitions` times and chain tests are executed `process_repetitions`× `repetitions` times for each chain length. Chain tests also call *experiment.py* which takes the length of the dependency chain and `test_id`. In *start_experiment.sh*, we invoke `make` to build our binary and execute it. The result of a test run is a log file, which is finally handed over to a script extracting statistical measurements including TPR and throughput as well as the speculation window size.

3.2 Test Scenarios

We define a general naming scheme for these tests to avoid confusion. The name of a test is composed of three parts: <`datatype`>_<`mode`>_<`operation`>_<`operand_modes (optional)`>. `datatype` can either be `INT`, `FLOAT` or `AVX` and defines if the test uses integer, floating point or AVX operation. `mode` can either be `C` or `S` and defines if we are testing the instruction in a dependency chain (`C`) or only one single time (`S`) in the condition. `operation` indicates which specific operation was used in the condition. It

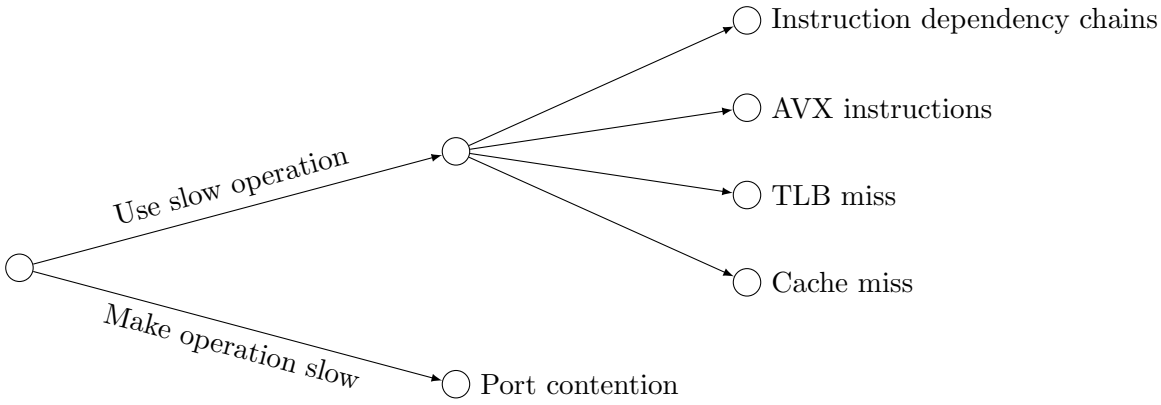


Figure 3.2: The tests we execute can be classified into either instructions which are slow by themselves or fast instructions which are made slow. Slow instructions include loads from addresses which are not in the cache or which TLB entry is missing and AVX instructions.

can either be the name of an instruction (e.g. `shl`), or the sign of an arithmetic operation (e.g. "+"). Some tests chain up n blocks, each block consisting of m instructions. This is denoted by the instructions which comprise a block, wrapped in round brackets. For example, `(not-not)` means one block consists of two subsequent `not` instructions. In case blocks are used, n refers to the number of blocks, not the number of instructions. The last part, `operand_modes`, is optional and only present for tests involving memory interaction (loads and stores). `operand_modes` consists of three letters, `d o1 o2`, telling what the destination of the operation is (`d`) and where the operands come from (`o1` and `o2`). For example, `rrm` means that the result of the operation is stored into a register, the first operand is loaded from a register and the second operand is loaded from memory.

For each processor, we create test scenarios which can be divided into different categories:

1. **Basic**: The basic condition, `if(x < len)`, is used as a reference value in all our tests. Its abbreviation is simply `BASIC`.
2. **Single floating point operations:** This category comprises all test scenarios using exactly one floating point operation in the condition, for example `if((x + (f_y/f_z)) < len)`. The name prefix is `FLOAT_S`.
3. **Integer instruction chains:** This category comprises all test scenarios using integer dependency chains. The name prefix is `INT_C`. We test all values for n between 1 and 150.
4. **Floating point instruction chains:** This category comprises all test scenarios using floating point dependency chains. The name prefix is `FLOAT_C`. Just like in integer instruction chains, we test all values for n from 1 to 150.
5. **Memory interaction:** All the experiments described in the categories before solely operate on registers, but in this category, tests involve loads and stores

from and to memory. These experiments are organized in chain tests. The name postfixes of those are `rrm`, `mmr` or `mmi`. `rrm` means that the chain instruction writes to a register and reads from a register and memory. `mmr` means that the chain instruction writes to memory and reads from a register and memory. `mmi` means that the chain instruction writes to memory and reads from a register and memory.

6. **Missing TLB entry:** The Translation Lookaside Buffer (TLB) caches page table entries. Whenever a DRAM access happens, the TLB is searched for the respective page table entry. If the entry is present, the physical address can directly be retrieved from it. If the entry is missing, a page table walk happens, which is quite costly. Pointer chasing means that an application dereferences a pointer to a pointer to a pointer and so on [87]. We assume that the attack success rate can be increased by chasing pointers which do not have a present TLB entry. These tests have the postfix `_TLBFLUSH`.

7. **Missing cache entry:** Similar to missing TLB entries, missing cache entries also introduce latency. These tests have the postfix `_FLUSH`.

8. **AVX spin up:** SIMD (single instruction multiple data) instructions perform an instruction on multiple data values in parallel and are offered by various modern CPUs. Intel and AMD provide AVX (Advanced Vector Extensions) instructions as one implementation of SIMD. AVX insructions are executed by a saparate unit and for power saving purposes, parts of this unit are powered down when it is not used [59, 19]. We assume that the attack success rate can be increased if the condition uses AVX instructions in a powered down AVX unit. We execute tests on Intel and AMD CPUs. The name prefix is `AVX`.

9. **Port contention:** Since for now, port contention attacks were only proven fully working on Intel CPUs and AMD CPUs [4, 63]. Theoretically, all SMT CPUs are vulnerable but there is no public proof-of-concept for IBM Power9. We execute our tests only on Intel CPUs and leave the test on AMD CPUs and IBM Power9 as future work. The general idea is that if an instruction is executed on a contended port, the overall latency of the instruction is increased, which increases the attack success rate.

Chapter 4

Real-world Analysis Results

In this chapter, we present the results of the analysis which we conduct according to Chapter 3. As already mentioned, tests are executed on four different processors, Intel, AMD, ARM, and IBM Power. In total, around 84 test scenarios are constructed for the Intel and AMD CPUs and around 64 for IBM Power and ARM. The reason for the difference is that AMD and Intel are CISC architectures whereas IBM Power and ARM are RISC and therefore do not have such a rich instruction set.

The following sections discuss the results of our analysis in detail in terms of TPR and throughput as well as the size of the speculation window. At the end of this chapter, we compare the four different platforms based on their test results.

4.1 Intel

In this section, we present our analysis results for the Intel i7-6700K (Skylake). We work on an isolated CPU core and pin the execution of our test processes to this core.

4.1.1 Preliminary Study

In order to find out an appropriate number of repetitions (values for `repetitions` and `process_repetitions`) for our experiments, we conduct a preliminary study.

Figure 4.1 shows the average TPR for `process_repetitions` = 20 achieved for a specific number of `repetitions`. As we can see, the average TPR stabilizes at around 200000 repetitions. Therefore, we set `process_repetitions` = 20 and `repetitions` = 200000 for all the tests executed on the Intel CPUs.

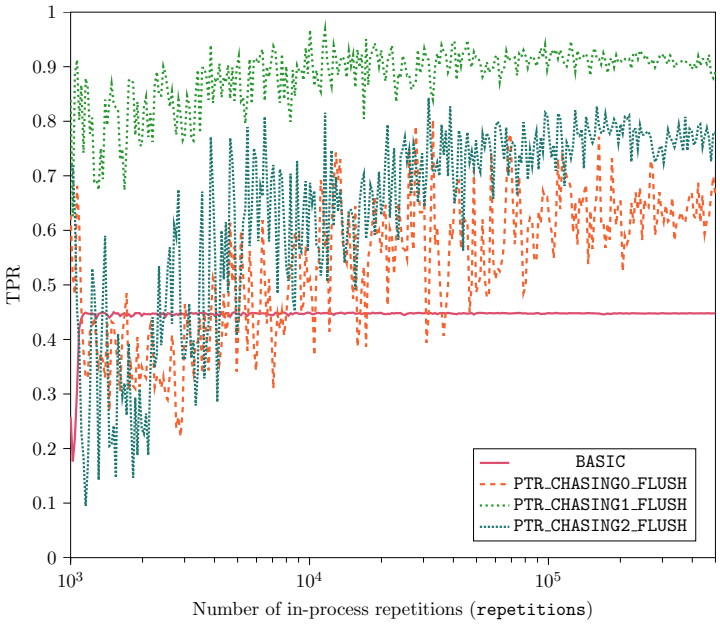


Figure 4.1: Average TPR achieved on Intel for a specific `repetitions` for `process_repetitions` = 10. Convergence was achieved at around `repetitions` = 200000.

4.1.2 Results

Basic As shown in Table 4.1, the basic condition achieves a TPR of 25%. The throughput of about 567000 bytes per second is also quite high, and the speculation window is large at about 146 instructions.

Single floating point operations We construct 13 different scenarios for testing single floating point operations in the condition. As shown by Table 4.1, single floating point operations improve the TPR compared to `BASIC` , and the throughput is not significantly lower, in some cases even higher. Also, the window size is equally large. Tests combining two different floating point operations in the condition partially give worse results. The reason for that lies in the structure of the execution units (Figure 2.1). For example, consider the condition of `FLOAT_S_(+,-)`, which is
`if ((x < len) & ((ftest1 + ftest2) >= 0.0f) & ((ftest3 - ftest4) >= 0.0f))`.
Port 0 is occupied by the floating point addition and port 1 by the floating point subtraction because there are no other execution ports for floating point operations. The logical ANDs can be sent to port 5 and port 6. Now we still need three ports for comparison, and one port to execute the branch, but we do not have them. The pipeline stalls and speculative execution is aborted very early.

Integer instruction dependency chains We construct 24 different scenarios testing integer dependency chains. The results are listed in Table 4.2. `INT_C_imul`, a multiplication chain, seems to work best with a maximum average TPR of 96% at a chain length of 25, which is also rather short. The throughput is high, and the window size of 150 instructions is large. The `INT_C_imul` (imm.) test uses multiplications with immediate values. As we can see, the TPR is also higher in this case compared to other immediate tests, like `INT_C_add`(imm.). This might be because there is only one execution port handling multiplications, as shown in Figure 2.1. If the instruction stream, *i.e.*, the multiplication instructions, reaches the scheduler, the instructions can only be scheduled to one single port. Therefore, the evaluation of the condition lasts much longer, giving the CPU more time to speculate. Additionally, Figure 4.2 shows that `INT_C_imul` can achieve a higher TPR if the chain length is larger than 60. The improvement is about 15%.

Other tests involving immediates, like `INT_C_sub`(imm.), `INT_C_and`(imm.), `INT_C_(xor,xor)`(imm.), or `INT_C_or`(imm.) do in most cases not perform as good as chains involving two register operands. A possible explanation is that the CPU has one less dependency check to make because one of the operands is a constant and therefore they execute faster. Although Fog [21] proposed that the latency of addition, subtraction, and logical ANDs does not differ between situations where the second operand is a register and situations where the second operand is an immediate, we can clearly see that there must be a difference which is why we get varying values for the TPR.

As shown by the test results of tests `INT_C_(add-not-not)` and `INT_C_(add-imul)`, mixing operations does not significantly increase the TPR. However, we can see in Figure 4.2 that both tests show a quite stable TPR for increasing numbers of the chain length which is above 70%. In both cases, the speculation window size is smaller than in other tests.

As shown by Figure 4.2, some tests show that the TPR drops for a chain length which is larger than 100. This happens in tests like `INT_C_add` or `INT_C_or` , and we explain this by the limited size of the reorder buffer. If it is too full by the dependency chain's instructions, it redues the remaining window size for the attacker too much. Exceptions are `INT_C_crc32`, `INT_C_popcnt` and `INT_C_bsf`.

Floating point instruction dependency chains We test multiplication, division, addition, and subtraction of floating point operations in chains. `AVX_C_mulss` gave the highest score at the shortest dependency chain length. The TPR graphs of these tests show that there are three phases. In the first phase, the TPR is low, before reaching a peak in the second phase and dropping again in the third phase. All four tests show this behavior, even though in `AVX_C_divss` it is less visible. Also the size of the speculation window is quite high in all those tests. However, this comes at the cost of lower throughput, as shown in Table 4.1.

Memory interaction We build 18 scenarios testing dependency chains with writes to memory or loads from memory. In Table 4.2 we list all of the tests.

Tests where we only load from memory (`rrm` tests) show a high slope of the TPR in the diagram at the beginning for chain lengths below 50. Then, a peak can be seen before the TPR drops slightly again. The reason for this might be that there are not enough loads to increase the evaluation time of the condition for low chain lengths and, therefore, the TPR is low in the beginning. However, after the peak there might be too many loads which can be handled by the CPU and the TPR drops again slightly. `FLOAT_C_mulss_rrm` has a similar graph as `FLOAT_C_mulss`, however, the peak is more extreme. The speculation window sizes are large (around 150 instructions) in these tests, and also the maximum reached TPR is high (around 76%). The throughput is lower than for normal integer dependency chain tests because loads from memory or the cache are significantly slower than loads from registers.

Tests where we also store to memory (`mmr` and `mm` tests) show a strict decline of the TPR in some test cases, for example, `INT_sub_mmr_a`nd `INT_C_xor_mmr`. We can explain this by the limited size of the store buffer. On Skylake, the store buffer has 56 entries [52]. Once the store buffer is full, the CPU has to wait for free slots in the buffer before it can continue issuing stores. The TPR curve starts to decline strictly at a chain length of about 60 and reaches zero at about 100. The average TPR for such tests is around 70%, and the throughput is also rather low due to the loads and stores from and to memory.

Tests where we do loads and stores but do not involve registers (`mmi` tests) show a rather constant TPR curve. The average TPR is rather low in these tests (around 40%), except in the `INT_C_or_mmi` test where it reaches 77%. The values throughput and speculation window size are similar to other tests.

Missing TLB entry `PTR_CHASING1_TLBFLUSH` and `PTR_CHASING2_TLBFLUSH` achieve the highest TPR among all tests which do not involve dependency chains. However, the throughput of both is low. We used the `invlpg` instruction to flush the TLB. This instruction is a privileged instruction which means we need a kernel module in order to execute it. The communication with this kernel module takes a very long time. The values we provide for the throughput do not include the time it takes to call the kernel module. Of course, an attacker can also flush the TLB without a kernel module, but this requires knowledge about the indexing function of the TLB on the respective platform. We leave this open for future work. Additionally, the sizes of the speculation windows are only 72 and 80 in these scenarios.

Missing cache entry `PTR_CHASING1_FLUSH` shows that a high TPR of 94% at a chain length of 94. Additionally, we see that the graph of the TPR increases with the length of the chain. The average throughput which is at about 335313 byte per second which is about 20000 byte per second lower than throughput other chain tests give.

AVX spin up We test 8 instructions involving AVX operations. These instructions operate on integer vectors. Table 4.1 shows that the TPR is at around 45% and the throughput is at around 500000 bytes per second. The window size is between 138 and 150.

Port contention In this experiment, we show whether port contention can be used to extend the size of the speculation window. In the following, we first sketch the main idea and then present a short practical evaluation.

We assume that if an instruction is executed on a contended port, the overall latency of the instruction is increased. We want to construct a scenario involving two hardware threads, HT_1 and HT_2, running on the same physical core. HT_1 is executing the `access_array` function as sketched in Listing 3.2, using an integer instruction dependency chain of instruction i. In our proof of concept implementation, HT_1 is also establishing the cache covert channel to leak data. The goal of HT_2 is to cause contention on port 1. To do that, HT_2 issues m instructions i to port 1. Note that the choice of m is extremely important since the size of the reorder buffer and scheduler is limited. On Skylake, the reorder buffer has 224 entries; the scheduler has 97 entries [52]. If we choose m too large, the scheduler and reorder buffer become full and HT_1 cannot execute anything until HT_2's instructions are worked through. If m is too small, there is no contention.

Synchronization between HT_1 and HT_2 is essential for our proof of concept implementation, as already argued by Bhattacharyya et al. [11]. We want HT_1, and HT_2 start at two points in time which are closest as possible. Ideally, HT_2 starts sending the instructions to the scheduler first and then HT_1 starts to evaluate the condition, for which it needs to execute instructions as well. We make use of shared variables and a plain busy wait mechanism in order to steer both processes to stick to our proposed timing schedule.

For n, the instruction chain length of HT_1, we test three different values, 16, 32 and 64. For m, the number of instructions of HT_2, we test five different values, 2, 4, 8, 16 and 32. This yields 15 different test scenarios. We want to compare these results to test scenarios without contention. In total, there are three non-contention scenarios, one for each n. In order to make sure that we genuinely keep execution ports free in these tests, we let HT_2 execute `nop` instructions.

Since the port contention scenario serves only as a very basic proof of concept, we test it on a limited number of instructions and leave a detailed analysis for future work. Considering the test cases for integer instruction dependency chains as listed in Table 4.2, we can group the instructions used there into four groups by the execution ports they use [21]:

- Port 1: `bsf`, `crc32`, `popcnt`, `imul`
- Ports 0, 1, 5, 6: `inc`, `dec`, `neg`, `xor`, `not`, `and`, `or`, `sub`, `add`

- Ports 1, 5: `andn`
- Ports 0, 6: `ror`, `shr`

We will select `popcnt`, `inc`, `andn` and `shr` from this list for our experiments. We assume that instructions scheduled to the same ports give the same results, for example, `popcnt` can be exchanged by `bsf` , and we will still get the same results.

The results of our tests are summarized in Table 4.3. In the `popcnt` scenario for $n = 16$, we see clearly that the TPR is higher in the contention tests than in the tests without contention. For $n = 32$ we do not see a big difference, except for the test where $m = 16$. The TPR is even lower in the contention scenario for $m = 32$. This might already be due to too many instructions in the reorder buffer. The window size is between 70 and 90 instructions for all the tests. The contended tests do not show a higher window size.

In the `inc` scenario, we do not see an improvement. The reason for this might be that the `inc` instruction can be scheduled to one out of four ports. This means that the load is distributed to four ports and is therefore only one-fourth of the load we have on the execution unit, for example, in the `popcnt` test. Therefore, we also experimented with 64 instructions and could see a slight improvement. 64 instructions would mean 16 instructions per port. The window size for all the tests is equal.

In the `ror` scenario for $n = 16$ we can see a a difference of 2% in the TPR for all contention tests except $m = 32$. In this situation, we have a TPR which is equal to the no contention test. For $n = 32$ we only see a slight improvement (1%) in the $m = 32$ test. In the tests without contention, we reach a window size of 60 and 64 instructions. The tests with contention show a window size between 74 and 90.

The `andn` scenario shows a higher TPR for contention tests with $n = 16$. In the test without contention, we get a TPR of 16%, in the test with contention we get values between 18% and 19%. The tests where $n = 32$ do not show a higher TPR. The window size is almost the same for all scenarios.

In summary, we can see that although the exact correlation between m and n is not entirely clear, we see that port contention has a positive influence on the performance of Spectre attacks.

Table 4.1: Test for single operations (Intel Skylake). The table gives the average TPR and throughput including standard deviation (SD) and standard error of the mean (SEM) as well as the window size (number of instructions).

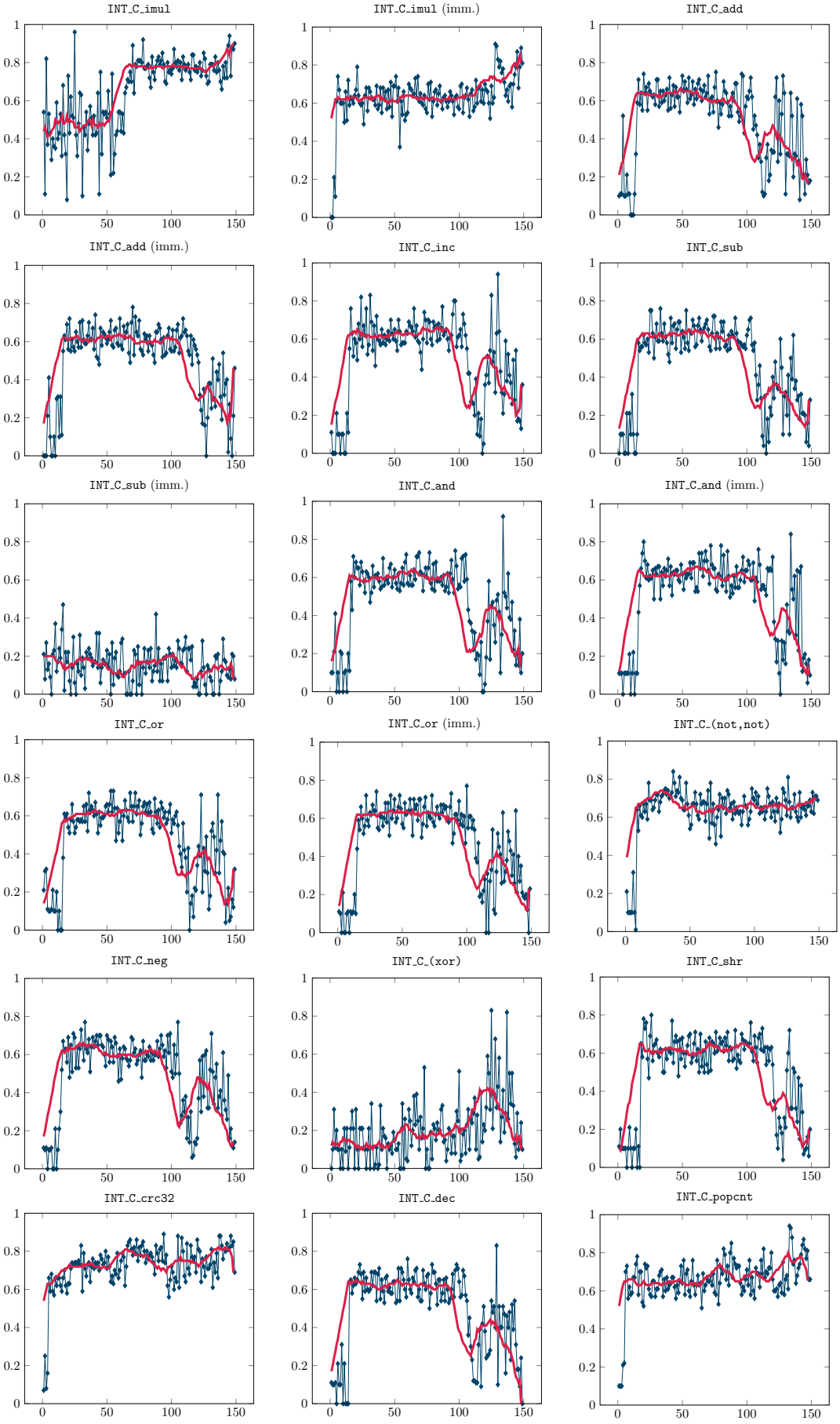
Mode	TPR	TPR SD	TPR SEM	Throughput	Throughput SD	Throughput SEM	Window size
BASIC	25%	0.39	0.12	566817.08	66501.02	9312.01	146
PTR_CHASING0_FLUSH	43%	0.16	0.05	462566.71	69558.43	15553.74	148
PTR_CHASING1_TLBFLUSH	89%	0.29	0.06	20409.48	19.57	4.38	72
PTR_CHASING2_TLBFLUSH	91%	0.20	0.04	11058.27	31.98	7.15	80
FLOAT_S_*	48%	0.48	0.15	558550.20	61368.02	6167.72	80
FLOAT_S_+	66%	0.42	0.13	561788.11	63446.96	6376.66	150
FLOAT_S_-	56%	0.45	0.14	565366.79	63736.95	6405.80	150
FLOAT_S_/	34v	0.44	0.14	561635.76	63950.30	6427.25	148
FLOAT_S_(+,+)	43%	0.43	0.14	562207.94	65262.09	6559.09	126
FLOAT_S_(+,-)	32%	0.42	0.13	562853.40	63169.88	6348.81	140
FLOAT_S_(+,*)	43%	0.42	0.13	566365.20	62179.16	6249.24	144
FLOAT_S_(+,/)	10%	0.30	0.09	565696.95	61380.97	6169.02	146
FLOAT_S_(-,-)	19%	0.37	0.12	572774.10	58892.21	5918.89	126
FLOAT_S_(-,*)	34%	0.42	0.13	568111.45	59312.84	5961.16	50
FLOAT_S_(-,/)	21%	0.34	0.11	549572.17	65907.42	6623.94	142
FLOAT_S_(*,*)	32%	0.42	0.13	559229.32	65085.35	6541.32	140
FLOAT_S_(*,/)	43%	0.43	0.14	557378.42	62918.64	6323.56	120
FLOAT_S_(/,/)	32%	0.42	0.13	557170.10	63708.56	6402.95	60
AVX_S_vaddsd	43%	0.43	0.14	474254.02	188741.95	59685.45	148
AVX_S_vmulsd	38%	0.40	0.13	451306.66	172940.41	54688.56	150
AVX_S_vsubsd	27%	0.37	0.12	506704.32	148574.47	46983.37	130
AVX_S_vsqrtsd	46%	0.39	0.12	502394.87	122510.76	38741.30	146
AVX_S_vandpd	55%	0.45	0.14	575855.29	62807.02	19861.32	138
AVX_S_vorpd	42%	0.43	0.14	561279.12	74951.32	23701.69	134
AVX_S_vandnpd	55%	0.36	0.11	546638.92	156192.27	49392.33	150
AVX_S_vxorpd	44%	0.44	0.14	617937.62	79591.46	25169.03	150

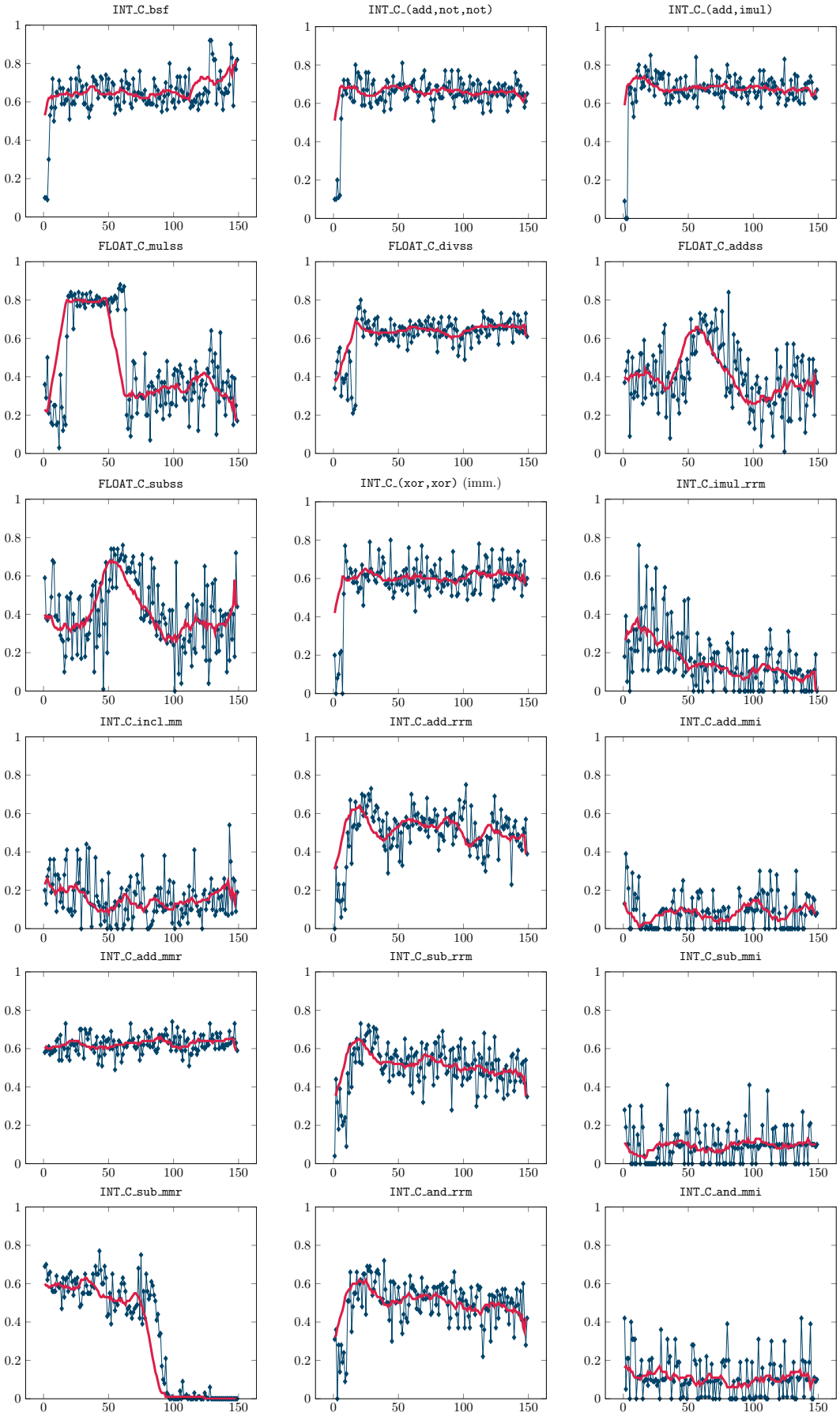
Table 4.2: Test for instruction chains (Intel Skylake). First, the chain length for which the maximum TPR was achieved is shown. Second, the chain length for which the maximum throughput (bytes/second) was achieved is given and third the chain length for the maximum window size is stated.

Mode	n	Max TPR	n	Max Throughp.	TPR SD	TPR SEM	Through-put SD	Through-put SEM	n	Win. size
INT_C_imul	25	96%	58	592455.11	0.36	0.01	50384.67	4127.67	95	150
INT_C_imul (imm.)	128	91%	17	488407.59	0.20	0.01	23596.48	1933.10	53	150
INT_C_add	76	75%	141	573281.46	0.33	0.01	23148.63	1896.41	103	150
INT_C_add (imm.)	70	78%	147	573117.60	0.30	0.01	29758.46	2437.91	49	136
INT_C_inc	130	94%	119	567484.01	0.33	0.01	24078.24	1972.57	84	138
INT_C_sub	33	76%	134	571968.19	0.33	0.01	23648.45	1937.36	10	128
INT_C_sub (imm.)	16	47%	77	586112.05	0.27	0.01	21705.06	1778.15	39	150
INT_C_and	134	92%	120	565631.22	0.32	0.01	22959.34	1880.90	70	150
INT_C_and (imm.)	134	84%	147	567140.04	0.31	0.01	27290.74	2235.74	38	150
INT_C_or	53	73%	142	567811.98	0.34	0.01	22878.04	1874.24	75	136
INT_C_or (imm.)	100	77%	148	580193.59	0.35	0.01	24410.31	1999.77	28	136
INT_C_(not,not)	37	84%	141	492135.76	0.22	0.01	19018.40	1558.05	71	142
INT_C_neg	33	77%	119	562087.28	0.33	0.01	22481.73	1841.78	15	136
INT_C_xor	125	83%	139	574174.97	0.35	0.01	23919.93	1959.60	100	136
INT_C_(xor,xor) (imm.)	44	80%	141	469067.77	0.22	0.01	18342.14	1502.65	27	138
INT_C_shr	26	80%	141	572103.19	0.32	0.01	30784.53	2521.97	60	124
INT_C_ror	140	87%	146	579498.45	0.31	0.01	31625.62	2590.87	31	136
INT_C_crc32	94	89%	126	457346.55	0.18	0.00	19777.89	1620.27	23	150
INT_C_andn	33	75%	149	582252.48	0.24	0.01	30752.76	2519.36	85	150
INT_C_dec	129	83%	149	579665.40	0.33	0.01	24771.71	2029.38	45	136
INT_C_popcnt	133	94%	13	480389.38	0.21	0.01	22595.83	1851.12	76	152
INT_C_bsf	128	92%	26	491092.11	0.20	0.01	22363.21	1832.07	47	150
INT_C_(add,not,not)	53	81%	141	398920.89	0.19	0.01	11595.68	949.96	135	136
INT_C_(add,imul)	21	85%	18	426806.88	0.18	0.00	26380.97	2161.21	98	128
FLOAT_C_mulss	59	88%	12	482746.79	0.40	0.01	57855.46	4739.70	45	148
FLOAT_C_divss	21	80%	20	485786.72	0.21	0.01	95985.78	7863.46	76	152
FLOAT_C_addss	81	84%	39	507553.82	0.40	0.01	50764.34	4158.78	53	152
FLOAT_C_subss	61	76%	46	504643.91	0.39	0.01	50216.77	4113.92	41	152
INT_C_imul_rrm	12	76%	10	438914.57	0.35	0.01	25864.65	2118.91	32	138
INT_C_add_rrm	102	75%	60	500962.26	0.26	0.01	20090.10	1645.84	4	132
INT_C_add_mmi	2	39%	5	475310.36	0.25	0.01	58681.76	4807.40	89	150
INT_C_add_mmr	99	74%	6	470694.29	0.13	0.00	63179.54	5175.87	23	150
INT_C_incl_mm	143	54%	3	509505.59	0.31	0.01	56285.45	4611.08	99	150
INT_C_sub_rrm	21	73%	65	513094.35	0.25	0.01	20805.15	1704.42	18	122
INT_C_sub_mmi	97	41%	3	472171.37	0.26	0.01	57782.86	4733.76	150	122
INT_C_sub_mmr	43	77%	11	462597.91	0.32	0.01	62350.39	5107.94	1	154
INT_C_and_rrm	39	72%	72	500553.90	0.26	0.01	19794.81	1621.65	6	122
INT_C_and_mmi	1	42%	11	571601.86	0.29	0.01	75287.33	6167.78	53	138
INT_C_or_rrm	25	79%	61	508055.31	0.26	0.01	18718.57	1533.48	10	122
INT_C_or_mmi	3	77%	3	483637.96	0.13	0.00	65595.73	5373.81	38	150
INT_C_notl_mm	118	43%	0	508232.40	0.30	0.01	81184.63	6650.90	17	136
INT_C_(neg,neg)_mm	81	52%	3	419692.44	0.31	0.01	67705.07	5546.62	51	150
INT_C_xor_rrm	26	74%	50	521710.18	0.26	0.01	22275.81	1824.91	9	128
INT_C_xor_mmi	117	40%	4	448320.30	0.29	0.01	80368.27	6584.03	21	128
INT_C_xor_mmr	6	61%	4	475758.46	0.27	0.01	63981.99	5241.61	35	138
FLOAT_C_mulss_rrm	53	99%	27	447892.26	0.46	0.01	80035.96	6556.80	27	150
INT_C_PTR_CHASING	54	97%	39	592993.24	0.33	0.01	75640.05	6196.67	1	150
INT_C_PTR_CHASING_FLUSH	94	94%	1	335313.14	0.15	0.00	63137.94	5172.46	17	153

Table 4.3: Results of tests investigating the influence of port contention on the size of the speculation window - HT_1 executes n instructions while HT_2 executes m instructions simultaneously.

$n\,(HT_1)$	$m\,(HT_2)$	**TPR**	**TPR SD**	**TPR SEM**	**Throughput**	**Throughput SD**	**Throughput SEM**	**Window size**
					`popcnt` (p1)			
16	-	40%	0.22	0.00	2612040.19	10206140.66	32297.91	88
16	2	45%	0.19	0.00	2669514.15	9435763.34	29860.01	86
16	4	42%	0.19	0.00	2620854.14	9676752.87	30622.64	70
16	8	43%	0.19	0.00	2613388.39	10163167.58	32161.92	76
16	16	43%	0.20	0.00	2590891.72	10182289.88	32222.44	70
16	32	44%	0.20	0.00	2614379.08	9782716.05	30957.96	70
32	-	26%	0.37	0.01	2614994.38	9339949.45	29556.80	86
32	2	26%	0.37	0.01	2632982.05	9779741.70	30948.55	86
32	4	25%	0.36	0.01	2646342.75	9364532.88	29634.60	82
32	8	26%	0.37	0.01	2634803.11	9452916.32	29914.29	88
32	16	27%	0.37	0.01	2599715.33	9454189.22	29918.32	78
32	32	24%	0.36	0.01	2620613.75	9745971.81	30841.68	82
					`inc` (p0156)			
16	-	18%	0.27	0.00	2649690.98	9321546.20	29498.56	88
16	2	17%	0.27	0.00	2636974.33	9687350.41	30656.17	78
16	4	17%	0.27	0.00	2614652.51	9265981.63	29322.73	64
16	8	17%	0.27	0.00	2617030.32	10212847.62	32319.14	86
16	16	18%	0.27	0.00	2597318.27	9444385.87	29887.30	64
16	32	17%	0.27	0.00	2587372.33	9773578.60	30929.05	88
32	-	25%	0.37	0.01	2606933.14	9489377.98	30029.68	86
32	2	25%	0.36	0.01	2606899.16	9891806.05	31303.18	84
32	4	26%	0.37	0.01	2631440.45	9681269.67	30636.93	86
32	8	26%	0.37	0.01	2634803.11	9452916.32	29914.29	88
32	16	26%	0.37	0.01	2620047.29	10513431.22	33270.35	86
32	32	25%	0.36	0.01	2604488.84	9690287.14	30665.47	88
64	-	10%	0.23	0.00	2505700.47	8854601.99	28020.89	84
64	16	11%	0.24	0.00	2507978.51	9132576.09	28900.56	84
64	32	10%	0.24	0.00	2528045.50	8888111.09	28126.93	84
64	64	11%	0.24	0.00	2534645.43	8793627.85	27827.94	84
					`andn` (p15)			
16	-	16%	0.26	0.00	2628103.62	9071974.53	28708.78	60
16	2	18%	0.28	0.00	2582244.48	9924381.19	31406.27	90
16	4	18%	0.27	0.00	2591210.61	10156377.98	32140.44	88
16	8	19%	0.26	0.00	2596728.12	11772716.36	37255.43	88
16	16	19%	0.24	0.00	2590237.40	29201737.18	92410.56	88
16	32	17%	0.27	0.00	2587372.33	9773578.60	30929.05	86
32	-	25%	0.30	0.00	2595616.00	11809896.18	37373.09	64
32	2	25%	0.30	0.00	2584380.01	26629904.03	84271.85	82
32	4	25%	0.30	0.00	2580611.86	37644231.44	119127.31	78
32	8	25%	0.30	0.00	2569835.27	37510362.53	118703.68	74
32	16	24%	0.30	0.00	2576506.13	37526335.71	118754.23	84
32	32	25%	0.30	0.00	2585281.99	26563646.29	84062.17	52
					`ror` (p06)			
16	-	16%	0.27	0.00	2566553.95	9905336.10	31346.00	76
16	2	18%	0.28	0.00	2616311.39	9756128.96	30873.83	58
16	4	18%	0.28	0.00	2550532.42	9646291.54	30526.24	64
16	8	18%	0.28	0.00	2594033.72	9507941.07	30088.42	90
16	16	17%	0.27	0.00	2599580.17	9966377.90	31539.17	66
16	32	16%	0.26	0.00	2628639.02	10007070.01	31667.94	90
32	-	12%	0.24	0.00	2561524.62	9828842.12	31103.93	90
32	2	12%	0.24	0.00	2589734.29	9161837.29	28993.16	86
32	4	12%	0.24	0.00	2615233.74	9337829.27	29550.09	88
32	8	13%	0.25	0.00	2567987.47	9643125.64	30516.22	78
32	16	12%	0.24	0.00	2574797.88	9802237.89	31019.74	74
32	32	13%	0.24	0.00	2577236.56	9325393.52	29510.74	78


INT_C_imul
INT_C_imul (imm.)
INT_C_add
INT_C_add (imm.)
INT_C_inc
INT_C_sub
INT_C_sub (imm.)
INT_C_and
INT_C_and (imm.)
INT_C_or
INT_C_or (imm.)
INT_C_(not,not)
INT_C_neg
INT_C_(xor)
INT_C_shr
INT_C_crc32
INT_C_dec
INT_C_popcnt


INT_C_bsf
INT_C_(add,not,not)
INT_C_(add,imul)
FLOAT_C_mulss
FLOAT_C_divss
FLOAT_C_addss
FLOAT_C_subss
INT_C_(xor,xor) (imm.)
INT_C_imul_rrm
INT_C_incl_mm
INT_C_add_rrm
INT_C_add_mmi
INT_C_add_mmr
INT_C_sub_rrm
INT_C_sub_mmi
INT_C_sub_mmr
INT_C_and_rrm
INT_C_and_mmi

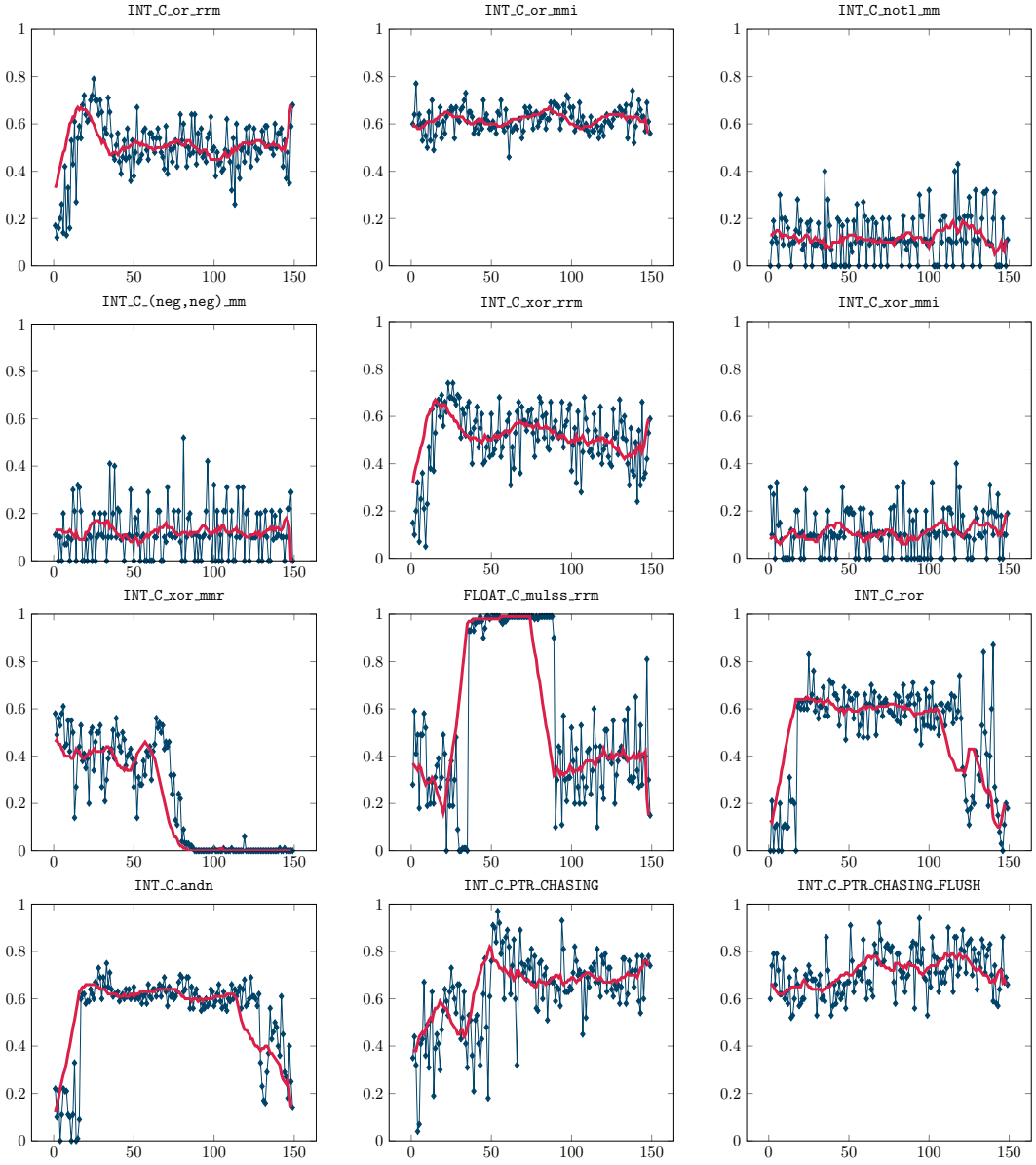


Figure 4.2: Test results for different test scenarios (Intel Skylake). The x-axis illustrates the chain length n, the y-axis illustrates the average TPR. The blue graph (■) shows the TPR achieved for each chain length. The pink graph (■) shows the moving average.

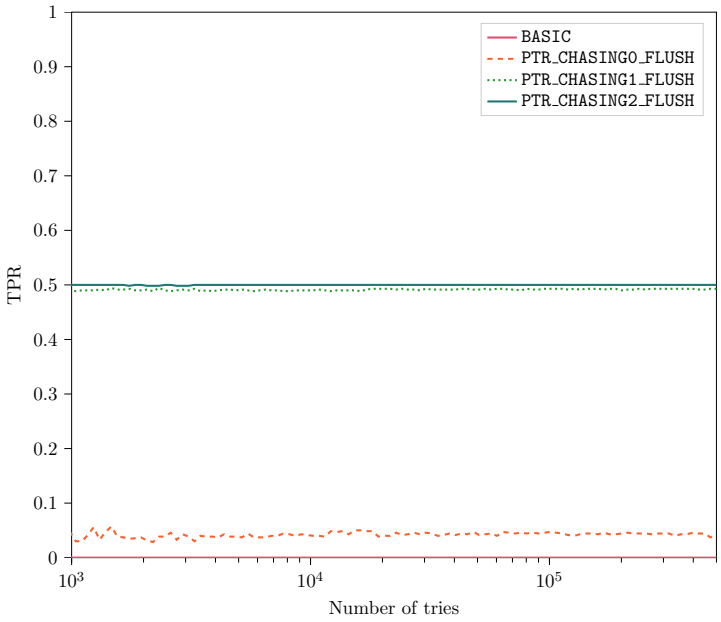


Figure 4.3: Average TPR achieved on AMD for a specific `repetitions` for `process_repetitions` = 50. Convergence was achieved at around `repetitions` = 200000.

4.2 AMD

In this section, we present our analysis results for the AMD Ryzen Threadripper 1920X. Since both Intel and AMD CPUs implement the x86 instruction set architecture, we execute exactly the same tests for AMD.

4.2.1 Preliminary Study

In order to find out an appropriate number of repetitions (values for `repetitions` and `process_repetitions`) for our experiments, we conduct a preliminary study.

Figure 4.3 shows the average TPR for `process_repetitions` = 50 achieved for a specific number of `repetitions`. As we can see, the average TPR is converging very fast. Therefore, we set `process_repetitions` = 50 and `repetitions` = 20000 for all the tests executed on the AMD CPU.

4.2.2 Results

Basic The performance of the attack in the basic version is not very good. A TPR of 0.03 is achieved on average, as one can see in Table 4.4. The throughput is 173000 bytes per second, and the speculation window is 98 instructions.

Integer instruction dependency chains Table 4.5 gives an overview of the tests which are executed. `INT_C_imul` achieves a good maximum TPR (0.46). This is the highest score which was reached by any integer instruction dependency chain test. The throughput, 153000 bytes per second, is lower than the one measured in `BASIC`, which was 173000 bytes per second. The TPR graph in Figure 4.4 shows that the curve stabilizes around 0.35 TPR. `INT_C_imul`(imm.) shows a similar graph. The window size is 132 and 118 instructions.

The test cases in which we test blocks of instructions (`INT_C_(not,not)`, `INT_C_(xor,xor)`, `INT_C_(and,not,not)` and `INT_C_(add,imul)`) we see that the TPR stabilizes at about 0.35. The throughput of those tests is lower than of the other tests. The window size is between 112 and 118 for those tests.

`INT_C_bsf` shows a very interesting TPR graph. The TPR first stabilizes at 0.1 and then suddenly rises to 0.3. We cannot see any other abnormalities regarding this mode, except that the throughput is rather low. The window size is 112 instructions.

All the other test cases follow the same pattern in terms of their TPR graph, as shown in Figure 4.4. For very low chain lengths ($\leq$10) the TPR is increasing. Then it stabilizes until a chain length of about 120 before it drops and forms a trough. We can see this trough in every graph, sometimes clearly visible as in `INT_C_add` or `INT_C_dec`, sometimes more flat as in `INT_C_popcnt`.

Single floating point operations Table 4.4 shows that single floating point operations increase the TPR of the attack. Single floating point operations have a higher throughput than the `BASIC` test case. This can be explained by the structure of the AMD Zen Microarchitecture, as shown in 2.2. An AMD CPU has a separated floating point unit which operates more independent from the integer unit than it is the case in, for example, Intel CPUs. Therefore, if we do a standard integer test, all the μops are scheduled to the integer execution units. If we do floating point tests, part of the execution load is taken over by the floating point unit, yielding higher overall throughput.

Especially when divisions are involved, the TPR is high. For example, regarding `FLOAT_S_*`, `FLOAT_S_+`, `FLOAT_S_-` and `FLOAT_S_/`, the division reaches a TPR of 0.18 while the others reach less than 0.12. Surprisingly, the division also has the highest throughput rate.

Floating point instruction dependency chains We test four different floating point instruction dependency chains. All of them achieve lower TPRs than integer tests do, except for `AVX_C_divss`. The throughput is similar to that of integer dependency chains; it is about 158000 bytes per second.

The TPR graph in Figure 4.4 shows that `AVX_C_mulss` has an overall slightly increasing TPR and `AVX_C_divss` has a jump in the TPR at a chain length of 50. `AVX_C_addss` and `AVX_C_subss` do not show a good TPR at all. It stabilizes at around 0.1.

Memory interaction Tests where we only load from memory (`rrm` tests) all have a maximum average TPR of about 0.15-0.30 and a maximum throughput of 165000 bytes per second. We would assume that loads from memory have a negative impact on the maximum throughput of the test case. In fact, the average throughput is the same as for integer dependency chains, but the standard deviation of the throughput shows higher values than for integer dependency chains. We assume that this might have something to do with the cache. Sometimes the value is loaded from the cache which gives similar throughput rates than for the standard integer dependency chains but sometimes the value is loaded from DRAM which explains the low throughput and the high standard deviation. The TPR graphs in Figure 4.4 all show similar behavior. For example, the graph for `INT_C_xor_rrm` shows a stabilized TPR for low and high chain lengths but a trough for middle chain lengths between 75 and 100.

Tests where we also store to memory (`mmr` and `mm` tests) show an almost linear TPR graph which slightly increases. The TPR is at 38% at a maximum, and we see the same standard deviation of the throughput as for `rrm` tests.

Tests where we do loads and stores but do not involve registers (`mmi` tests) have a lower maximum TPR than all the other memory interaction tests, except for `INT_C_or_mmi`, which has a maximum TPR of 40%. The TPR graphs look similar to those we saw for the `mmr` tests.

Missing TLB entry The TLB flush tests achieve the highest size of the speculation window, 132 instructions and a high TPR of 35% and 41%. The throughput of `PTR_CHASING1_TLBFLUSH` and `PTR_CHASING2_TLBFLUSH` is lower than for other tests, which is 75000 and 56000 bytes per second. This is significantly lower than in other tests. For example, `BASIC_a_c`hieved a twice a throughput which was twice as high. We achieve a window size of 130 to 132 instructions which is the highest among all single tests.

Missing cache entry `PTR_CHASING_FLUSH` also shows an almost linear TPR graph, as shown in Figure 4.4. The maximum TPR is at 49% which is the highest among all chain tests. However, the throughput is similar to that of other tests. It is slightly lower which is due to the use of the `clflush` instruction. The achieved window size is 132.

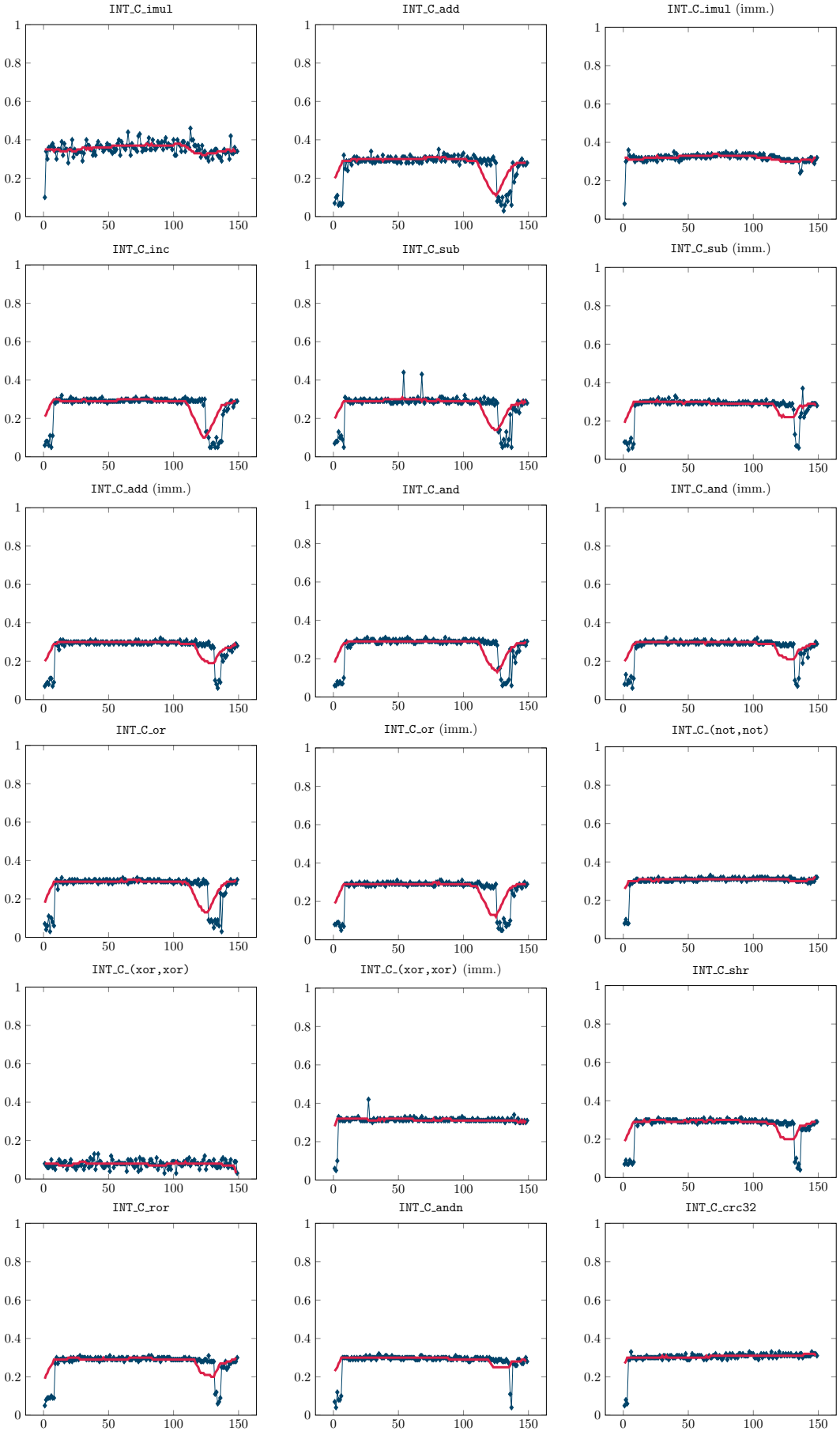
AVX spin up On the AMD CPU, we test 8 different AVX instructions operating on integer vectors. They do not give better values for the TPR than other single tests, but the throughput is lower than in, for example, single floating point tests. However, the window sizes are quite high for these tests (130 instructions).

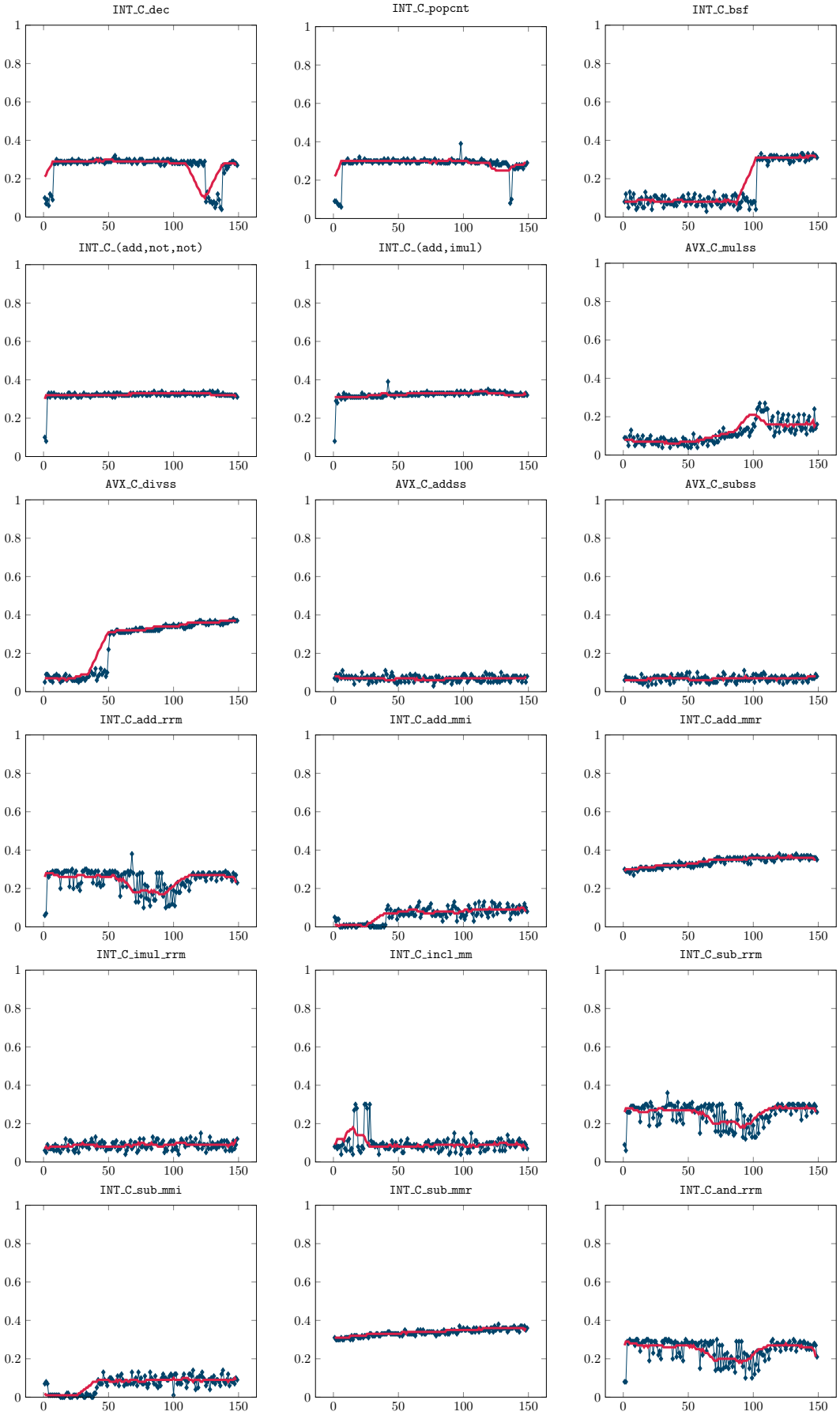
Table 4.4: Test for single operations (AMD Ryzen Threadripper 1920X). The table gives the average TPR and throughput including standard deviation (SD) and standard error of the mean (SEM) as well as the window size (number of instructions).

Mode	TPR	TPR SD	TPR SEM	Throughput	Throughput SD	Throughput SEM	Window size
BASIC	3%	0.10	0.01	172988.61	12005.93	1732.91	98
PTR_CHASING0_FLUSH	5%	0.11	0.02	174272.28	14033.75	2004.82	0
PTR_CHASING1_TLBFLUSH	35%	0.22	0.03	75333.32	12851.10	1835.87	132
PTR_CHASING2_TLBFLUSH	41%	0.19	0.03	56538.53	7813.07	1116.15	130
FLOAT_S_*	12%	0.21	0.03	180544.97	14687.30	2098.19	66
FLOAT_S_+	7%	0.16	0.02	175996.25	11599.76	1657.11	60
FLOAT_S_-	9%	0.18	0.03	177105.31	13447.70	1921.10	102
FLOAT_S_/	18%	0.23	0.03	183058.22	15529.42	2218.49	128
FLOAT_S_(+,+)	10%	0.19	0.03	179431.20	13753.63	1964.80	118
FLOAT_S_(+,-)	16%	0.23	0.03	180483.89	14743.56	2106.22	128
FLOAT_S_(+,*)	10%	0.19	0.03	178698.97	12906.38	1843.77	124
FLOAT_S_(+,/)	14%	0.21	0.03	181162.63	14968.18	2138.31	128
FLOAT_S_(-,-)	10%	0.19	0.03	178939.40	13540.22	1934.32	76
FLOAT_S_(-,*)	11%	0.20	0.03	179564.89	13832.03	1976.00	58
FLOAT_S_(-,/)	15%	0.21	0.03	181324.70	14141.05	2020.15	80
FLOAT_S_(*,*)	8%	0.18	0.03	176640.95	12032.80	1718.97	122
FLOAT_S_(*,/)	12%	0.21	0.03	179652.10	13774.80	1967.83	128
FLOAT_S_(/,/)	15%	0.22	0.03	181665.72	14838.46	2119.78	124
AVX_S_vaddsd	9%	0.19	0.06	178947.11	16480.27	5211.52	12
AVX_S_vmulsd	7%	0.15	0.05	177293.75	15084.33	4770.08	118
AVX_S_vsubsd	7%	0.16	0.05	181309.08	15897.23	5027.15	124
AVX_S_vsqrtsd	7%	0.15	0.05	169555.62	6206.19	1962.57	80
AVX_S_vandpd	1%	0.01	0.00	175457.02	14816.48	4685.38	130
AVX_S_vorpd	3%	0.08	0.02	175523.91	13426.42	4245.81	22
AVX_S_vandnpd	26%	0.17	0.05	171869.34	11338.11	3585.43	122
AVX_S_vxorpd	12%	0.19	0.06	182320.47	15499.92	4901.51	54

Table 4.5: Test for instruction chains (AMD Ryzen Threadripper 1920X). First, the chain length for which the maximum TPR was achieved is shown. Second, the chain length for which the maximum throughput (bytes/second) was achieved is given and third the chain length for the maximum window size is stated.

Mode	n	Max TPR	n	Max Throughp.	TPR SD	TPR SEM	Through-put SD	Throug-put SEM	n	Win. size
INT_C_imul	113	46%	144	152598.46	0.09	0.00	17577.17	5.59	116	132
INT_C_ imul (imm.)	4	36%	102	135876.58	0.07	0.00	14963.31	5.48	22	118
INT_C_add	81	35%	136	166170.42	0.11	0.00	13634.57	5.00	64	106
INT_C_add (imm.)	90	32%	134	158029.53	0.09	0.00	14204.34	5.20	19	118
INT_C_inc	14	32%	146	158544.12	0.10	0.00	13822.31	5.06	119	130
INT_C_sub	54	44%	68	160655.67	0.10	0.00	13997.43	5.13	35	118
INT_C_sub (imm.)	138	37%	138	160668.61	0.09	0.00	14189.19	5.20	32	106
INT_C_and	40	31%	132	158327.47	0.10	0.00	14035.22	5.14	12	106
INT_C_and (imm.)	33	32%	142	157860.39	0.08	0.00	14271.66	5.23	115	118
INT_C_or	61	31%	137	158594.04	0.10	0.00	14040.47	5.14	96	118
INT_C_or (imm.)	99	31%	127	158867.48	0.09	0.00	14036.17	5.14	57	108
INT_C_(not,not)	67	33%	139	149080.32	0.06	0.00	14760.57	5.41	29	112
INT_C_(xor,xor)	42	13%	133	158382.01	0.13	0.00	14463.23	5.30	105	110
INT_C_(xor, xor) (imm.)	27	42%	139	147898.56	0.06	0.00	14766.68	5.41	106	50
INT_C_shr	80	31%	134	158115.05	0.08	0.00	14245.40	5.22	102	130
INT_C_ror	83	31%	135	157647.27	0.08	0.00	14249.51	5.22	51	118
INT_C_andn	35	32%	137	158274.64	0.07	0.00	14511.62	5.32	73	130
INT_C_crc32	114	33%	6	146314.79	0.07	0.00	15453.64	5.66	131	130
INT_C_dec	55	32%	137	158246.01	0.09	0.00	14007.44	5.13	46	130
INT_C_popcnt	98	39%	136	157215.29	0.07	0.00	14502.27	5.31	93	118
INT_C_bsf	138	33%	4	23558.39	0.16	0.00	3232.17	1.18	134	112
INT_C_ (add,not,not)	128	34%	3	139459.66	0.05	0.00	15331.45	5.62	84	118
INT_C_(add,imul)	42	39%	42	143466.60	0.04	0.00	15695.86	5.75	61	118
FLOAT_C_mulss	109	27%	59	159997.57	0.13	0.00	21197.22	7.77	82	130
FLOAT_C_divss	146	38%	36	158440.49	0.14	0.00	26331.73	9.65	38	130
FLOAT_C_addss	40	11%	87	162667.60	0.11	0.00	18164.50	6.65	86	130
FLOAT_C_subss	93	11%	93	162800.62	0.11	0.00	18290.62	6.70	55	118
INT_C_add_rrm	68	38%	118	165255.32	0.11	0.00	19126.57	7.01	1	106
INT_C_add_mmi	93	13%	7	153255.37	0.13	0.00	21270.38	7.79	60	130
INT_C_add_mmr	120	38%	2	155483.52	0.06	0.00	22194.03	8.13	18	132
INT_C_imul_rrm	121	15%	3	152754.10	0.15	0.00	16341.83	5.99	89	130
INT_C_incl_mm	28	30%	2	159139.91	0.15	0.00	22107.44	8.10	8	106
INT_C_sub_rrm	34	36%	112	164216.01	0.11	0.00	17772.55	6.51	67	46
INT_C_sub_mmi	115	14%	0	153443.17	0.13	0.00	19817.64	7.26	141	130
INT_C_sub_mmr	127	38%	0	151735.21	0.05	0.00	21269.80	7.79	1	132
INT_C_and_rrm	32	30%	119	164437.68	0.11	0.00	18943.96	6.94	30	106
INT_C_and_mmi	100	12%	6	169799.54	0.13	0.00	24189.82	8.86	73	114
INT_C_and_mmr	73	2%	2	156685.86	0.04	0.00	22712.19	8.32	1	0
INT_C_or_rrm	31	36%	106	165065.77	0.10	0.00	19262.31	7.06	3	106
INT_C_or_mmi	116	40%	0	155334.00	0.05	0.00	22290.79	8.17	21	130
INT_C_notl_mm	93	16%	0	157652.15	0.15	0.00	26431.08	9.68	4	130
INT_C_(neg, neg)_mm	139	18%	1	148900.14	0.16	0.00	24536.14	8.99	106	112
INT_C_xor_rrm	4	31%	126	164605.03	0.11	0.00	18859.85	6.91	9	112
INT_C_xor_mmr	149	38%	0	155964.54	0.05	0.00	22132.53	8.11	92	132
INT_C_xor_mmi	129	14%	4	153387.14	0.15	0.00	25000.47	9.16	27	106
AVX_C_mulss_rrm	108	25%	48	158264.87	0.12	0.00	21365.20	7.83	22	130
INT_C_PTR_CHASING	149	36%	4	169053.14	0.16	0.00	17839.72	7.81	99	130
INT_C_PTR CHASING_FLUSH	143	49%	0	134977.01	0.03	0.00	22911.95	10.03	5	132


INT_C_imul
INT_C_add
INT_C_imul (imm.)
INT_C_inc
INT_C_sub
INT_C_sub (imm.)
INT_C_add (imm.)
INT_C_and
INT_C_and (imm.)
INT_C_or
INT_C_or (imm.)
INT_C_(not,not)
INT_C_(xor,xor)
INT_C_(xor,xor) (imm.)
INT_C_shr
INT_C_ror
INT_C_andn
INT_C_crc32


INT_C_dec
INT_C_popcnt
INT_C_bsf
INT_C_(add,not,not)
INT_C_(add,imul)
AVX_C_mulss
AVX_C_divss
AVX_C_addss
AVX_C_subss
INT_C_add_rrm
INT_C_add_mmi
INT_C_add_mmr
INT_C_imul_rrm
INT_C_incl_mm
INT_C_sub_rrm
INT_C_sub_mmi
INT_C_sub_mmr
INT_C_and_rrm

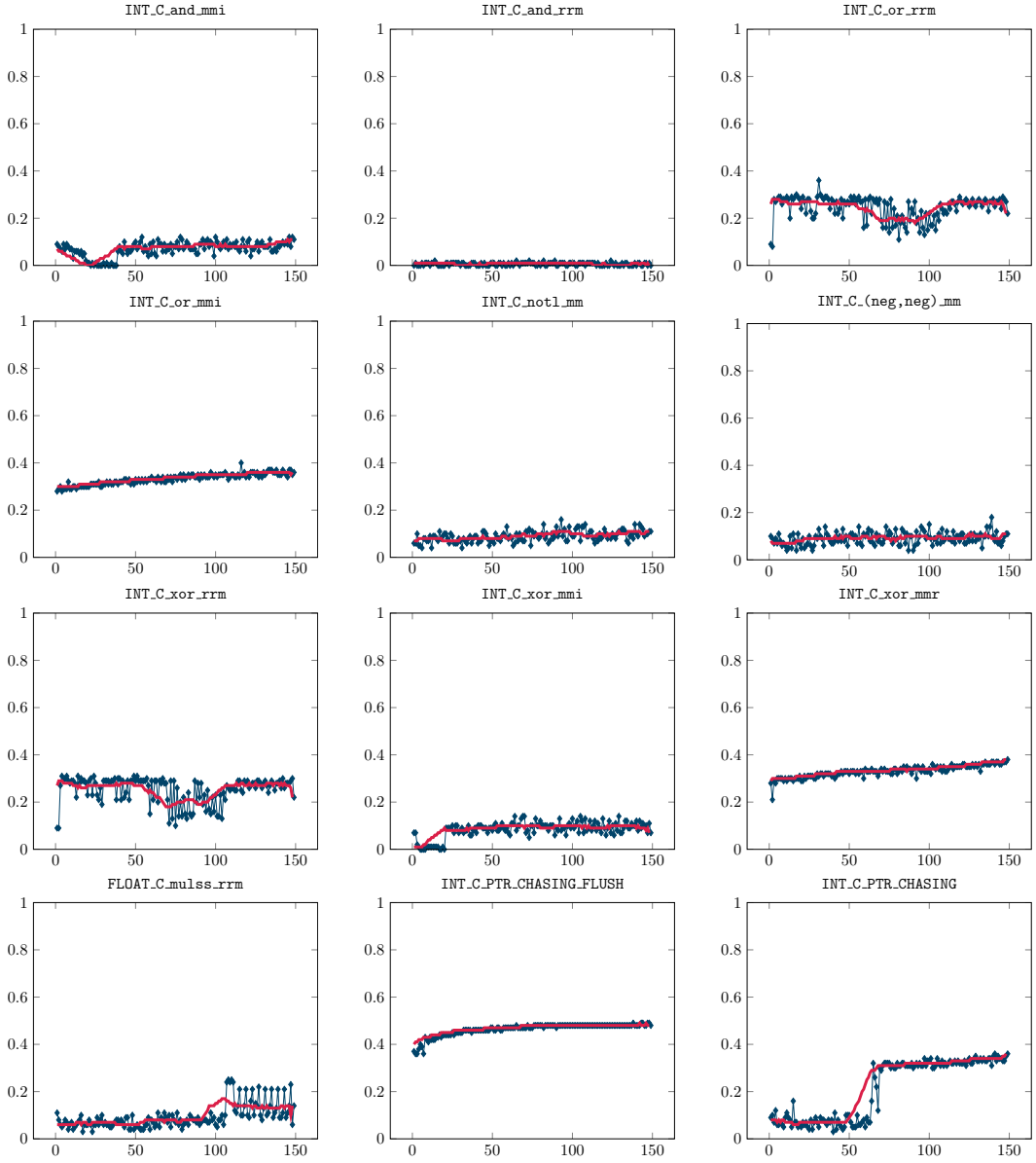


Figure 4.4: Test results for different test scenarios (AMD Ryzen Threadripper 1920X). The x-axis illustrates the chain length n, the y-axis illustrates the average TPR. The blue graph (■) shows the TPR achieved for each chain length. The pink graph (■) shows the moving average.

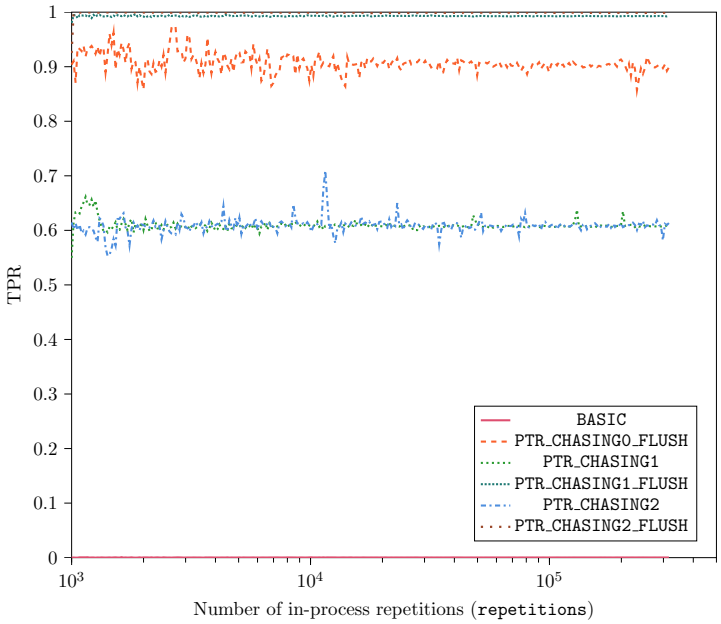


Figure 4.5: Average TPR achieved on ARM Cortex-A57 for a specific `repetitions` for `process_repetitions` = 10. Convergence was achieved at around `repetitions` = 1 50000.

4.3 ARM

In the following, we present the test results for the ARM Cortex-A57 CPU.

4.3.1 Preliminary Study

In order to find out an appropriate number of repetitions (values for `repetitions` and `process_repetitions`) for our experiments, we conduct a preliminary study.

Figure 4.5 shows the average TPR for `process_repetitions` = 20 achieved for a specific number of `repetitions`. As we can see, if we execute the same test cases as we did in the Intel tests, the results are inconclusive since the variation is very small. Therefore, we execute two more tests (`PTR_CHASING2_a_nd PTR_CHASING1_)_a`nd see that we can set `repetitions` = 75000 for all the tests executed on the ARM CPUs.

4.3.2 Results

Basic As shown in Table 4.6, `BASIC` leaks such a low number of TP values that the TPR becomes zero. As the result, also the window size becomes zero.

Single floating point operations We observed that using single floating point operations in the condition do not increase the TPR of the attack and just as the `BASIC` test showed, it is effectively zero.

Integer instruction dependency chains The ARM instruction set is RISC and is, therefore, less rich than the x86 instruction set. In particular, instructions like *inc* or *dec* are missing. Our test suite comprises 17 test scenarios for integer instruction chains. Table 4.7 gives an overview of the results. We observe that all the tests achieve a reasonable TPR and the size of the speculation window is between 42 and 58 instructions.

`INT_C_mul` gives a TPR rate of 100%. Also, `INT_C_(add,mul)`, which also involves the `mul` instruction, gives a score of 100%. Both achieve a maximum speculation window size of 24. As it is the case in the Intel Skylake CPU, there is only one execution port for multiplication, which means that a chain of multiplications needs more execution time. The TPR graph as shown in Figure 4.6 shows that the performance of the attack increases with the chain length before reaching a peak at about 50.

The other test cases show similar results compared to `INT_C_mul`. In the beginning, the TPR is low and then increases steadily before reaching a constant TPR. This can be seen for example in `INT_C_add`, where the TPR stabilizes around 30% or in `INT_C_ror`. What all the test cases have in common is that they all stabilize around 30% and never reach a higher TPR. The window sizes are around 50 instructions. The maximum throughput which is reached by all chain tests is around 100000 bytes per second.

Floating point instruction dependency chains `FLOAT_C_fmul`, `FLOAT_C_fdiv` , and `FLOAT_C_fabs` test the usage of floating point instruction chains. *fabs* computes the absolute value of a floating point value in a register. It is apparent from Table 4.7 that these tests achieve a very high TPR, namely 100%. They maximize the speculation window of 58 instructions. The maximal throughput is similar to those of integer instruction dependency chains. The graphs in Figure 4.6 show a similar behavior to that of `INT_C_mul`. For a chain length smaller than 50, we see a steep descent in the TPR. Then it stabilizes at a TPR of about 95%-100%.

Memory interaction As already mentioned, the ARM Cortex-A57 is a RISC processor. All instructions have the same size: 4 bytes. The ARM instruction set does not provide instructions to add and store at the same time, as it is done by the x86 instruction set. However, we want to test to which extend memory stores and memory loads have an influence on the size of the speculation window.

Tests where we only load from memory (`rrm` tests) either work very good or very bad. `INT_C_add_rrm`, `INT_C_and_rrm`, `INT_C_or_rrm` and `INT_C_eor_rrm` show a slight peak in the beginning for chain lengths smaller than 25 but completely stagnate afterwards. The speculation window size is at about 52 instructions. By contrast, `INT_C_sub_rrm` and `INT_C_mul_rrm` show a constant TPR rate of above 0.9. The speculation window size is

at about 58 instructions. We also see in Table 4.7 that the throughput is lower than it is in tests without memory interaction.

Tests where we also store to memory (`mmr` tests) also show an entirely different behavior. Either the rate is between 30% and 50%, as we can see in `INT_C_add_mmr` or the rate is almost constant zero, for example in `INT_C_sub_mmr` or the rate is almost constant one, for example in `INT_C_neg_mmr`. In most cases, the average throughput is higher than for `rrm` tests.

Tests where we do loads and stores but do not involve registers (`mmi` tests) show a low value for the TPR in the beginning for short chain lengths but a higher TPR in the end.

Missing TLB entry `PTR_CHASING1_TLBFLUSH` and `PTR_CHASING2_TLBFLUSH` achieve the highest TPR among all tests which do not involve dependency chains. However, the throughput is again lower than in other test cases. The speculation window size is 58 instructions.

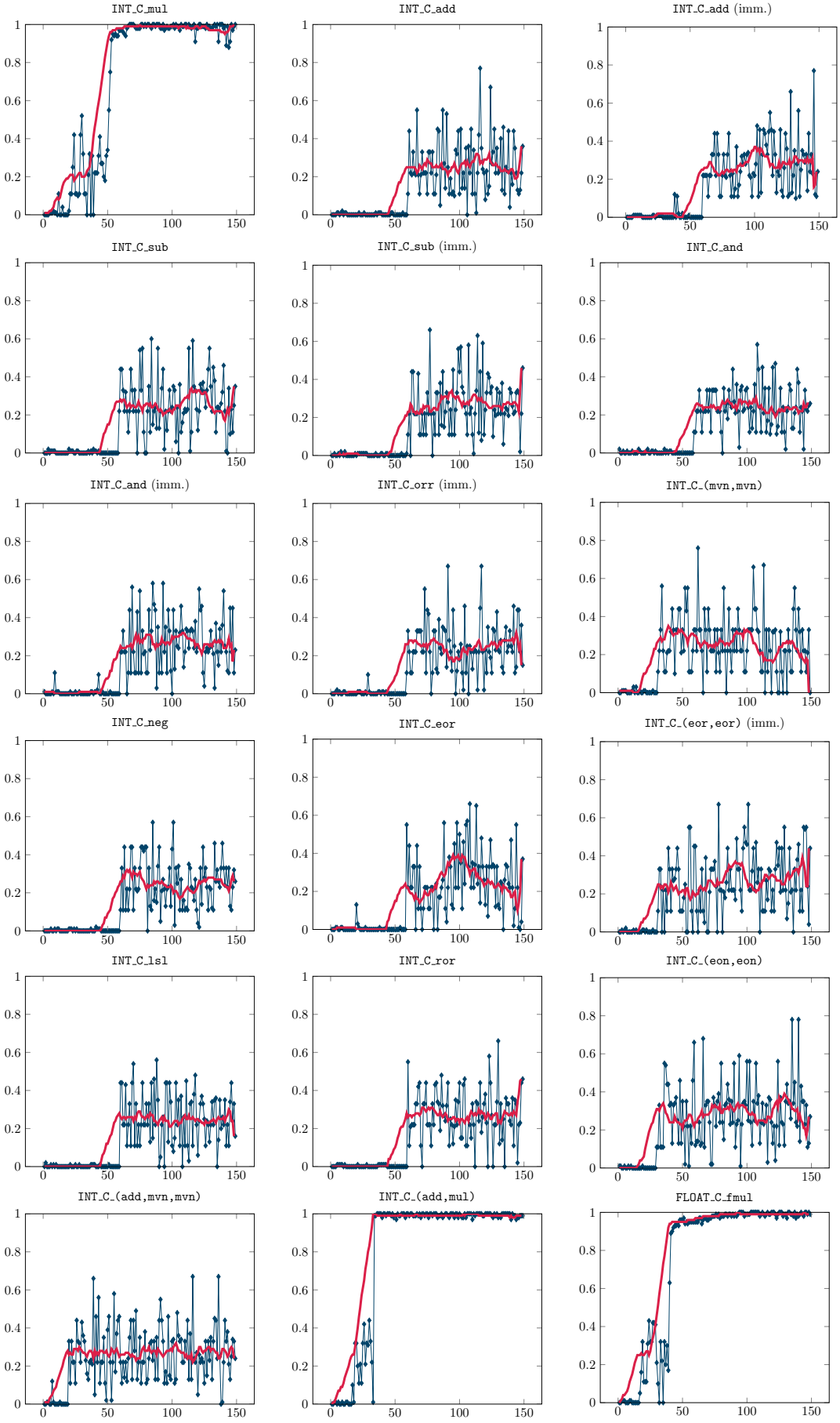
Missing cache entry `PTR_CHASING_FLUSH` shows a very high TPR of 100%. The speculation window size is at 58 instructions, and also the throughput about half of the throughput achieved by other chain tests (56000 bytes per second).

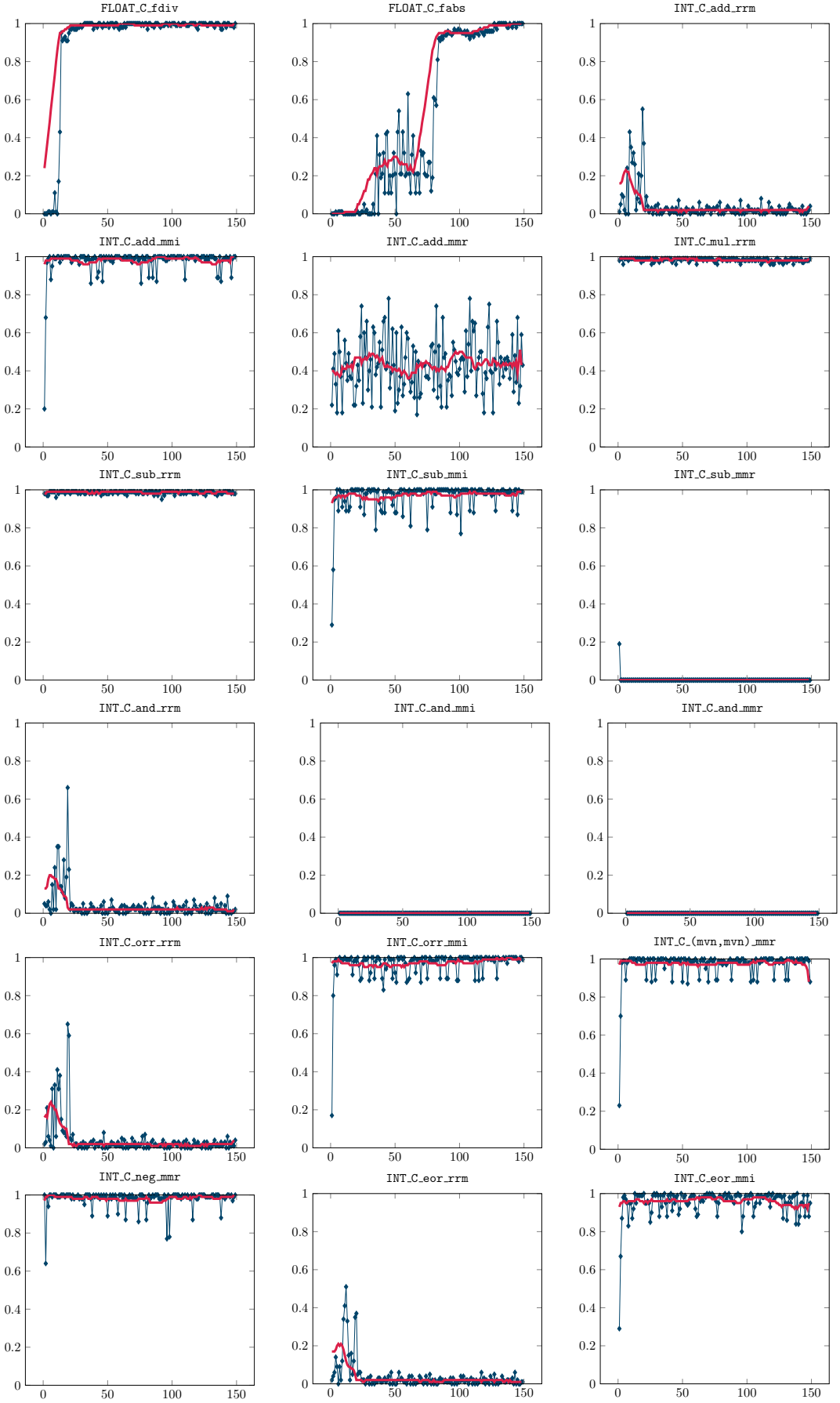
Table 4.6: Test for single operations (ARM Cortex-A57). The table gives the average TPR and throughput including standard deviation (SD) and standard error of the mean (SEM) as well as the window size (number of instructions).

Mode	TPR	TPR SD	TPR SEM	Throughput	Throughput SD	Throughput SEM	Window size
BASIC	0%	0.00	0.00	104278.89	5672.10	1793.67	0
PTR_CHASING0_FLUSH	60%	0.42	0.13	96801.84	3274.93	1035.62	24
PTR_CHASING1_TLBFLUSH	98%	0.03	0.01	36883.97	1804.53	570.64	58
PTR_CHASING2_TLBFLUSH	97%	0.02	0.01	36635.32	2065.55	653.18	58
FLOAT_S_*	0%	0.00	0.00	101831.66	4507.20	1425.30	0
FLOAT_S_+	0%	0.00	0.00	102733.87	5532.64	1749.57	0
FLOAT_S_-	0%	0.00	0.00	105521.83	3413.66	1079.50	0
FLOAT_S_/	0%	0.00	0.00	103150.17	5259.97	1663.35	0
FLOAT_S_(+,+)	0%	0.00	0.00	9919.59	40085.35	12676.10	0
FLOAT_S_(+,-)	0%	0.00	0.00	102323.40	5066.64	1602.21	0
FLOAT_S_(+,*)	0%	0.01	0.00	102797.45	3995.64	1263.53	0
FLOAT_S_(+,/)	0%	0.00	0.00	102679.81	4502.69	1423.88	0
FLOAT_S_(-,-)	0%	0.00	0.00	103848.87	3824.94	1209.55	0
FLOAT_S_(-,*)	0%	0.01	0.00	102898.74	4732.54	1496.56	0
FLOAT_S_(-,/)	0%	0.01	0.00	101571.75	6008.49	1900.05	0
FLOAT_S_(*,*)	0%	0.00	0.00	102956.93	4756.55	1504.15	0
FLOAT_S_(*,/)	0%	0.00	0.00	103238.33	4814.90	1522.60	0
FLOAT_S_(/,/)	0%	0.00	0.00	103284.31	5295.83	1674.69	0

Table 4.7: Test for instruction chains (ARM Cortex-A57). First, the chain length for which the maximum TPR was achieved is shown. Second, the chain length for which the maximum throughput (bytes/second) was achieved is given and third the chain length for the maximum window size is stated.

Mode	n	Max TPR	n	Max throughput	TPR SD	TPR SEM	Throughput SD	Throughput SEM	n	Window size
INT_C_mul	81	100%	17	105050.81	0.44	0.01	10898.97	2.10	22	58
INT_C_add	116	77%	39	104745.76	0.35	0.01	7400.85	1.28	75	58
INT_C_add (imm.)	146	77%	45	105898.54	0.37	0.01	7615.88	1.47	124	52
INT_C_sub	84	60%	58	105092.13	0.36	0.01	7404.48	1.43	61	58
INT_C_sub (imm.)	77	66%	29	105060.74	0.36	0.01	6624.33	1.57	71	58
INT_C_and	108	57%	5	105094.45	0.34	0.01	7249.98	1.25	77	58
INT_C_and (imm.)	93	58%	65	104525.08	0.36	0.01	7467.48	1.44	119	42
INT_C_orr	117	67%	24	105780.46	0.35	0.01	6585.19	1.56	101	50
INT_C_(mvn,mvn)	62	76%	7	104661.19	0.41	0.01	8502.29	1.64	150	58
INT_C_neg	101	57%	35	104272.43	0.35	0.01	7280.26	1.41	107	44
INT_C_eor	108	66%	3	106213.87	0.36	0.01	7419.24	1.43	134	58
INT_C_(eor,eor) (imm.)	78	67%	30	103930.58	0.40	0.01	8758.05	1.69	82	58
INT_C_lsl	88	56%	33	104316.08	0.35	0.01	6777.26	1.61	117	50
INT_C_ror	130	66%	4	105010.40	0.36	0.01	7427.46	1.43	119	58
INT_C_(eon,eon)	140	78%	0	104395.98	0.41	0.01	8847.54	1.71	106	42
INT_C_(add,mvn,mvn)	136	67%	2	104534.65	0.41	0.01	9652.51	1.86	138	50
INT_C_(add,mul)	48	100%	33	104604.67	0.39	0.01	12339.71	2.38	20	58
FLOAT_C_fmul	104	100%	4	105093.13	0.41	0.01	12103.92	2.34	44	58
FLOAT_C_fdiv	118	100%	15	103405.42	0.27	0.01	14467.71	2.79	13	58
FLOAT_C_fabs	146	100%	0	105371.13	0.47	0.01	7734.89	1.49	95	58
INT_C_add_rrm	19	55%	2	102607.50	0.13	0.00	6580.48	1.27	2	42
INT_C_add_mmi	19	100%	0	100236.05	0.12	0.00	13086.30	2.53	67	58
INT_C_add_mmr	45	78%	9	102244.38	0.39	0.01	4833.93	0.93	1	58
INT_C_mul_rrm	6	99%	50	80319.60	0.02	0.00	3054.85	0.59	1	58
INT_C_sub_rrm	115	99%	24	77127.89	0.02	0.00	2992.40	0.58	1	58
INT_C_sub_mmi	32	100%	3	101212.74	0.15	0.00	13026.42	2.52	1	56
INT_C_sub_mmr	1	19%	2	107374.02	0.02	0.00	14233.49	2.75	1	54
INT_C_and_rrm	19	66%	3	101451.11	0.12	0.00	6033.15	1.43	3	42
INT_C_and_mmi	131	0%	0	9107166.82	0.00	0.00	14259.89	2.75	1	0
INT_C_and_mmr	145	0%	3	106259.89	0.00	0.00	40360.46	7.79	1	0
INT_C_orr_rrm	19	65%	0	101578.91	0.13	0.00	11681.89	2.77	11	42
INT_C_orr_mmi	29	100%	0	100101.81	0.15	0.00	13073.98	2.52	26	58
INT_C_(mvn,mvn)_mmr	51	100%	0	101625.48	0.14	0.00	13256.46	2.56	2	54
INT_C_neg_mmr	63	100%	2	96126.49	0.10	0.00	288379.47	7.53	1	58
INT_C_eor_rrm	12	51%	2	104512.22	0.12	0.00	6617.09	1.28	2	52
INT_C_eor_mmi	120	100%	4	101333.17	0.15	0.00	13468.50	2.60	1	54
INT_C_eor_mmr	1	20%	3	105428.89	0.02	0.00	14307.12	2.76	1	54
INT_C_PTR_CHASING	34	100%	14	100709.52	0.09	0.00	9074.80	3.79	1	58
INT_C_PTR_CHASING_FLUSH	9	100%	0	55669.88	0.02	0.00	10663.34	4.46	1	58


INT_C_mul
INT_C_add
INT_C_add (imm.)
INT_C_sub
INT_C_sub (imm.)
INT_C_and
INT_C_and (imm.)
INT_C_orr (imm.)
INT_C_(mvn,mvn)
INT_C_neg
INT_C_eor
INT_C_(eor,eor) (imm.)
INT_C_lsl
INT_C_ror
INT_C_(eon,eon)
INT_C_(add,mvn,mvn)
INT_C_(add,mul)
FLOAT_C_fmul


FLOAT_C_fdiv
FLOAT_C_fabs
INT_C_add_rrm
INT_C_add_mmi
INT_C_add_mmr
INT_C_mul_rrm
INT_C_sub_rrm
INT_C_sub_mmi
INT_C_sub_mmr
INT_C_and_rrm
INT_C_and_mmi
INT_C_and_mmr
INT_C_orr_rrm
INT_C_orr_mmi
INT_C_(mvn,mvn)_mmr
INT_C_neg_mmr
INT_C_eor_rrm
INT_C_eor_mmi

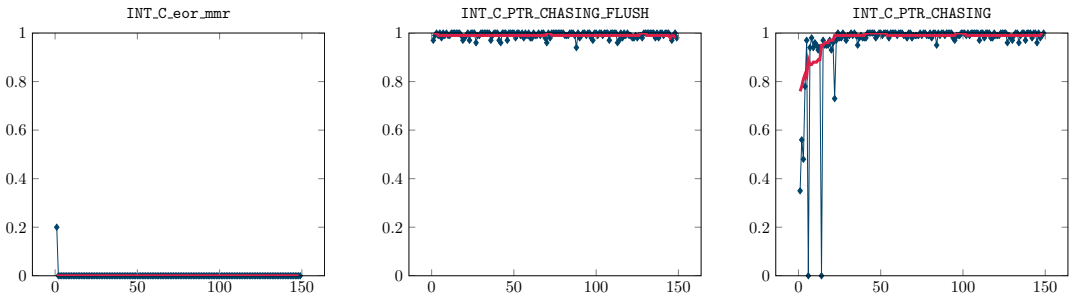


Figure 4.6: Test results for different test scenarios (ARM Cortex-A57). The x-axis illustrates the chain length n, the y-axis illustrates the average TPR. The blue graph (■) shows the TPR achieved for each chain length. The pink graph (■) shows the moving average.

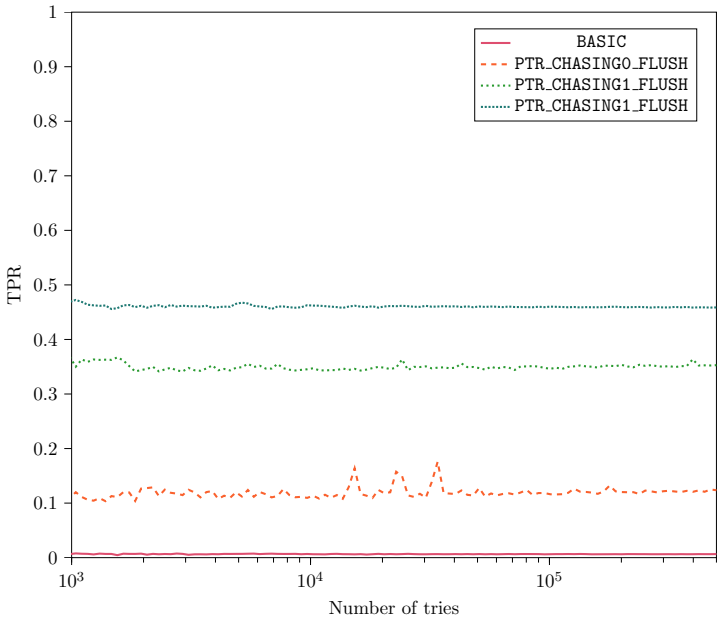


Figure 4.7: Average TPR achieved on IBM Power9 for a specific `repetitions` for `process_repetitions` = 15. Convergence was achieved at around `repetitions` = 75000.

4.4 IBM Power

This section discusses results for the IBM Power9 CPU. A high-precision timing source can be retrieved by the *mfspr* instruction, which moves data from a special purpose register, the TSC, to a general purpose register.

4.4.1 Preliminary Study

In order to find out an appropriate number of repetitions (values for `repetitions` and `process_repetitions`) for our experiments, we conduct a preliminary study.

Figure 4.7 shows the average TPR for `process_repetitions` = 20 achieved for a specific number of `repetitions`. As we can see, if we execute the same test cases as we did in the Intel tests and although the results are already very stable, we see that over 75000 the peaks in the graph of `PTR_CHASING0_FLUSH` vanish.

4.4.2 Results

Basic Our experiments demonstrate that the basic condition does not achieve good scores on IBM Power9, as it can be seen in Table 4.8.

Integer instruction dependency chains Like the ARM Cortex-A57, the IBM Power9 is a RISC processor, and therefore we do not test as many instructions in dependency chains as we do for x86 architectures. In total, 17 instruction chain experiments are evaluated, as presented in Table 4.9.

The highest TPR of 50% is achieved by `INT_C_(not,not)`, `INT_C_(xori,xori)`, `INT_C_(add,not,not)` and `INT_C_(add, mulld)`. All those tests have in common that they use blocks of instructions, not instructions alone. The throughput is higher than in the basic test case, and a speculation window size of around 170 instructions is reached. Figure 4.8 shows that the TPR graph can be divided into three stages for these test cases. In the first stage shows that the TPR is slightly decreasing. For chain lengths bigger than 60, the TPR stabilizes around 10%. For chain lengths bigger than 120, the TPR increases steadily again.

The other integer dependency chain tests achieve a TPR above 40% and a throughput of roughly 70000 bytes per second. The window size is between 94 and 96 instructions. Figure 4.8 shows that the TPR graphs for most tests are high in the beginning but slowly decline in the end. We assume that the reason for this is the limited size of the buffer in the dispatch unit, as shown in Figure 2.4. This buffer might be full with instructions belonging to the dependency chain, and therefore the instructions used for leaking the data might not fit into the buffer anymore. The condition is then evaluated before the speculative memory access can be made.

Single floating point operations We tested the influence of a single floating point operation and, as it is represented in Table 4.8, they increase the TPR of the attack. Especially the conditions which involve a division, `FLOAT_S_/`, `FLOAT_S_(+,/)`, `FLOAT_S_(-,/)` and `FLOAT_S_(/,/)` achieve a very good TPR of 40% at a very high throughput. The window sizes of those tests lie between 28 and 60.

Floating point instruction dependency chains Floating point chains give very good values for TPR in general (see Table 4.9). However, they achieve this TPR of 50% at large values for the chain length. The throughput is at about 70000 bytes per second, and the window size is 150 instructions.

`FLOAT_C_fmul` and `FLOAT_C_fadd` show a similar TPR graph. This graph has a peak at a chain length of 25 and a TPR of about 40%. Then, at a chain length of 50, it drops to a TPR of 30% and increases again to a TPR of 50% before it drops to 10% at a chain length of 90. Then it increases again and finally stays constant with an increasing trend at a chain length greater than 120.

`FLOAT_C_fdiv` behaves differently. The TPR graph converges earlier to a TPR of about 50. The reason for this might be that floating point divisions are treated specially by the CPU as it can be seen in Figure 2.4.

Memory interaction Just like the ARM Cortex-A57, the IBM Power9 ISA does not provide instructions which do both arithmetic operations and memory operations.

Tests where we only load from memory (`rrm` tests) achieve a maximum TPR of 50%, except for `INT_C_add_rrm` which only achieves a TPR of 18%. The speculation window size is between 172 and 220 for all these tests. The TPR graph follows the same pattern for all these tests. It has a through at a chain length of 40 and then converges to a TPR of 50%. The throughput of these tests is higher than of normal chain tests because of the involved memory operations, as shown by Table 4.9.

Tests where we also store to memory (`mmr` tests) work rather good, they achieve a stable TPR rate of roughly 50%. As shown by Figure 4.8, we do not see the through which we see in `rrm` tests. The throughput is the same as it is in `rrm` tests, about 70000 bytes per second. The window size is around 210 instructions.

Tests where we do loads and stores but do not involve registers (`mmi` tests) behave almost identical as `mmr` tests, as it can be seen in Figure 4.8. The TPR rate stabilizes very early at 50%.

Missing TLB entry `PTR_CHASING1_TLBFLUSH` and `PTR_CHASING2_TLBFLUSH` achieve an average TPR rate of 27% and 29%. This is worse than, for example, single floating point tests but better than the `BASIC` case. However, as it is in tests where we have to flush the TLB, the throughput is very low. It is about eight times lower than in any other test which does not involve flushing the TLB. However, the speculation window size is higher than in any other single test case, namely 82 and 90 instructions.

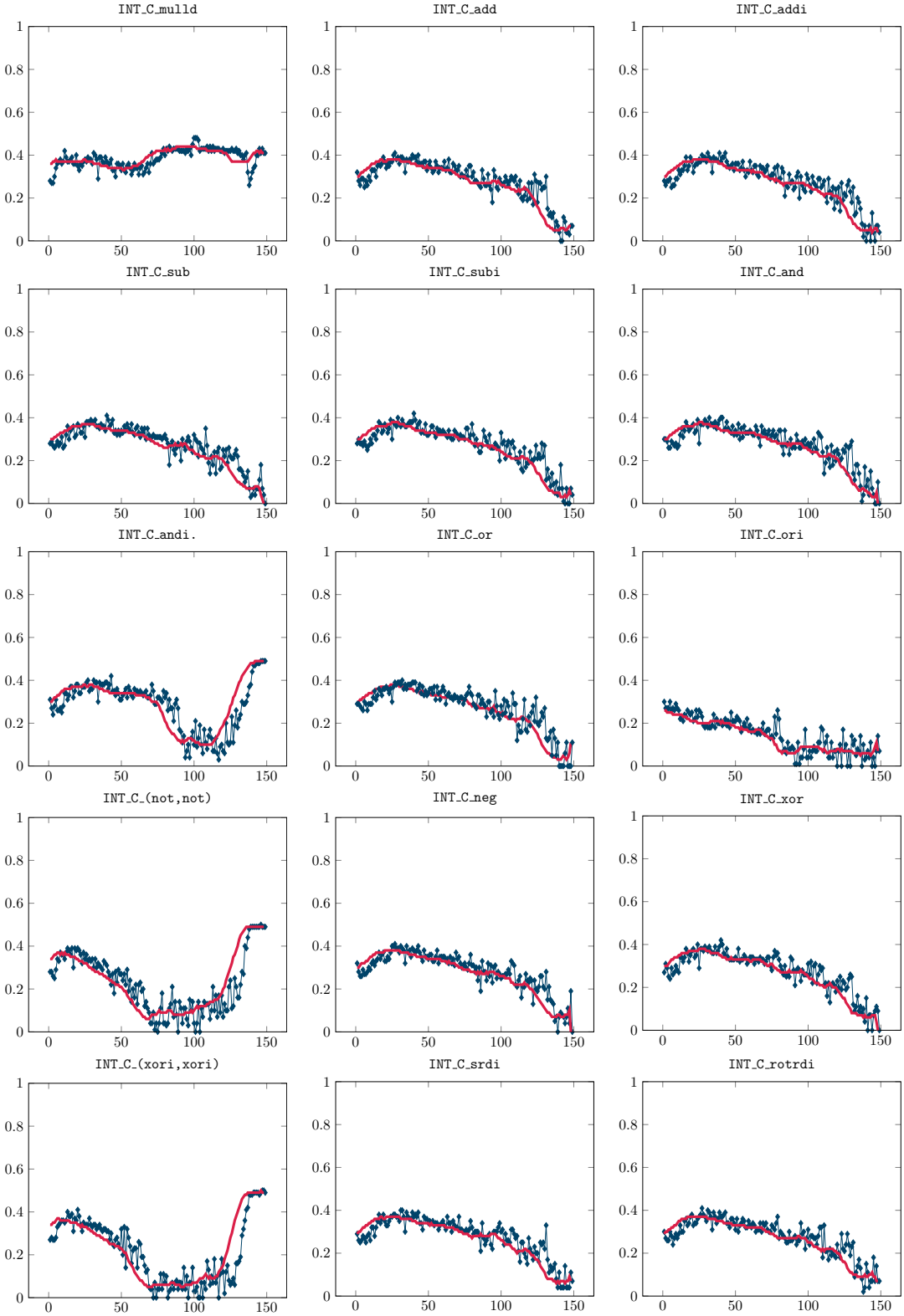
Missing cache entry `PTR_CHASING_TLBFLUSH` achieves a TPR of 50% and a throughput which is similar to that achieved by any other dependency chain test. We reach a window size of 216 instructions which is very good.

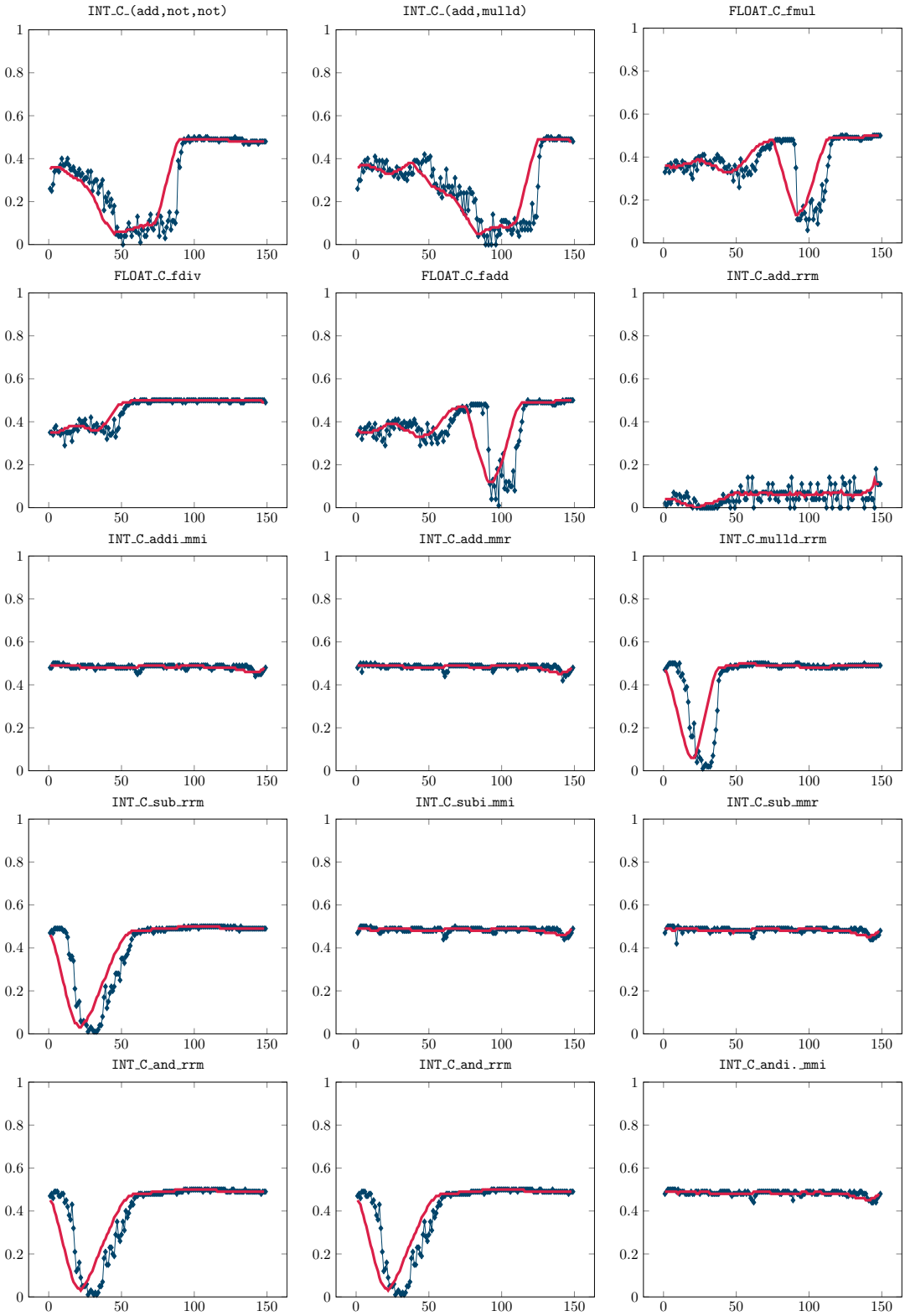
Table 4.8: Test for single operations (IBM Power 9). The table gives the average TPR and throughput including standard deviation (SD) and standard error of the mean (SEM) as well as the window size (number of instructions).

Mode	TPR	TPR SD	TPR SEM	Throughput	Throughput SD	Throughput SEM	Window size
BASIC	1%	0.00	0.00	69097.68	1853.91	586.26	32
PTR_CHASING0_FLUSH	7%	0.02	0.01	70030.23	2082.96	658.69	94
PTR_CHASING1_TLBFLUSH	27%	0.03	0.01	11099.49	124.49	39.37	82
PTR_CHASING2_TLBFLUSH	29%	0.01	0.00	5735.26	43.90	13.88	90
FLOAT_S_*	4%	0.07	0.02	71613.29	2763.41	873.87	62
FLOAT_S_+	4%	0.07	0.02	71754.96	2756.77	871.77	66
FLOAT_S_-	6%	0.09	0.03	70841.88	2586.57	817.95	36
FLOAT_S_/	26%	0.03	0.01	68650.35	1851.02	585.34	60
FLOAT_S_(+,+)	8%	0.03	0.01	71421.79	2970.78	939.44	20
FLOAT_S_(+,-)	14%	0.11	0.04	70923.40	3162.01	999.92	52
FLOAT_S_(+,*)	10%	0.09	0.03	71534.57	2681.40	847.93	70
FLOAT_S_(+,/)	40%	0.03	0.01	68972.21	1775.18	561.36	28
FLOAT_S_(-,-)	9%	0.05	0.02	68445.27	1926.42	609.19	44
FLOAT_S_(-,*)	9%	0.09	0.03	71807.38	1962.28	620.53	32
FLOAT_S_(-,/)	39%	0.02	0.01	69442.12	1585.05	501.24	36
FLOAT_S_(*,*)	7%	0.01	0.00	69002.02	1494.11	472.48	66
FLOAT_S_(*,/)	37%	0.04	0.01	71163.68	3626.50	1146.80	36
FLOAT_S_(/,/)	39%	0.02	0.01	68766.97	1639.92	518.59	36

Table 4.9: Test for instruction chains (IBM Power 9). First, the chain length for which the maximum TPR was achieved is shown. Second, the chain length for which the maximum throughput (bytes/second) was achieved is given and third the chain length for the maximum window size is stated.

Mode	n	Max TPR	n	Max Throughput	TPR SD	TPR SEM	Through-put SD	Throug-put SEM	n	Window size
INT_C_mulld	100	48%	145	70087.16	0.06	0.00	1718.91	0.33	139	96
INT_C_add	27	41%	64	70317.59	0.12	0.00	1710.57	0.33	107	94
INT_C_addi	43	41%	3	70098.86	0.12	0.00	1764.21	0.34	106	94
INT_C_sub	40	41%	137	70323.53	0.12	0.00	1792.48	0.35	88	94
INT_C_subi	40	42%	76	70491.01	0.12	0.00	1738.77	0.34	103	94
INT_C_and	32	40%	130	70544.94	0.12	0.00	1724.91	0.33	100	94
INT_C_andi.	149	49%	81	70230.55	0.15	0.00	1907.28	0.37	112	150
INT_C_or	32	40%	124	70306.53	0.12	0.00	1712.02	0.33	90	94
INT_C_ori	1	30%	136	70432.64	0.14	0.00	1660.63	0.32	102	94
INT_C_(not,not)	146	50%	78	70414.14	0.18	0.00	1998.11	0.39	109	178
INT_C_neg	27	41%	107	70612.24	0.12	0.00	1743.02	0.34	115	94
INT_C_xor	40	42%	143	70300.93	0.12	0.00	1798.45	0.35	92	94
INT_C_(xori,xori)	147	50%	52	70209.02	0.19	0.00	1927.01	0.37	134	164
INT_C_srdi	31	40%	98	70456.56	0.12	0.00	1766.17	0.34	97	94
INT_C_rotrdi	27	41%	123	70477.58	0.12	0.00	1759.59	0.34	105	94
INT_C_(add,not,not)	100	50%	45	70100.20	0.20	0.00	2758.53	0.53	77	212
INT_C_(add,mulld)	134	50%	63	70289.31	0.18	0.00	2066.92	0.40	137	156
FLOAT_C_fmul	146	50%	91	70024.60	0.13	0.00	2325.93	0.45	128	150
FLOAT_C_fdiv	122	50%	2	69556.90	0.07	0.00	9085.35	1.75	36	216
FLOAT_C_fadd	147	50%	72	69929.48	0.14	0.00	2324.71	0.45	105	150
INT_C_add_rrm	146	18%	24	70045.56	0.14	0.00	2156.86	0.42	53	178
INT_C_addi_mmi	7	50%	10	69867.25	0.02	0.00	7169.06	1.38	3	216
INT_C_add_mmr	5	50%	6	69747.62	0.02	0.00	7201.38	1.39	3	216
INT_C_mulld_rrm	7	50%	23	70093.49	0.14	0.00	3130.55	0.60	27	220
INT_C_sub_rrm	100	50%	23	70009.14	0.16	0.00	2234.36	0.43	1	212
INT_C_subi_mmi	7	50%	8	69677.29	0.02	0.00	7165.81	1.38	3	216
INT_C_sub_mmr	6	50%	2	70128.82	0.02	0.00	7201.31	1.39	3	216
INT_C_and_rrm	114	50%	19	70163.44	0.16	0.00	2251.53	0.43	2	208
INT_C_andi._mmi	3	50%	0	69875.77	0.02	0.00	7219.30	1.39	3	216
INT_C_and_mmr	5	50%	8	69710.92	0.02	0.00	7240.11	1.40	5	216
INT_C_or_rrm	110	50%	24	70231.07	0.16	0.00	2206.01	0.43	1	212
INT_C_or_mmi	5	50%	0	69659.58	0.02	0.00	7197.95	1.39	7	216
INT_C_(not,not)_mmr	142	50%	4	69930.43	0.02	0.00	7765.99	1.50	2	214
INT_C_xor_rrm	104	50%	19	70091.91	0.16	0.00	2266.84	0.44	1	210
INT_C_(xori,xori)_mmi	141	50%	0	69083.90	0.03	0.00	7754.94	1.50	3	214
INT_C_xor_mmr	5	50%	12	69857.25	0.02	0.00	7245.67	1.40	135	220
INT_C_PTR_CHASING	106	50%	17	70870.29	0.10	0.00	3183.94	0.61	107	206
INT_C_PTR_CHASING_FLUSH	46	50%	0	68773.81	0.02	0.00	11277.05	2.18	6	216


INT_C_mulld
INT_C_add
INT_C_addi
INT_C_sub
INT_C_subi
INT_C_and
INT_C_andi.
INT_C_or
INT_C_ori
INT_C_(not,not)
INT_C_neg
INT_C_xor
INT_C_(xori,xori)
INT_C_srdi
INT_C_rotrdi


INT_C_(add,not,not)
INT_C_(add,mulld)
FLOAT_C_fmul
FLOAT_C_fdiv
FLOAT_C_fadd
INT_C_add_rrm
INT_C_addi_mmi
INT_C_add_mmr
INT_C_mulld_rrm
INT_C_sub_rrm
INT_C_subi_mmi
INT_C_sub_mmr
INT_C_and_rrm
INT_C_and_rrm
INT_C_andi._mmi

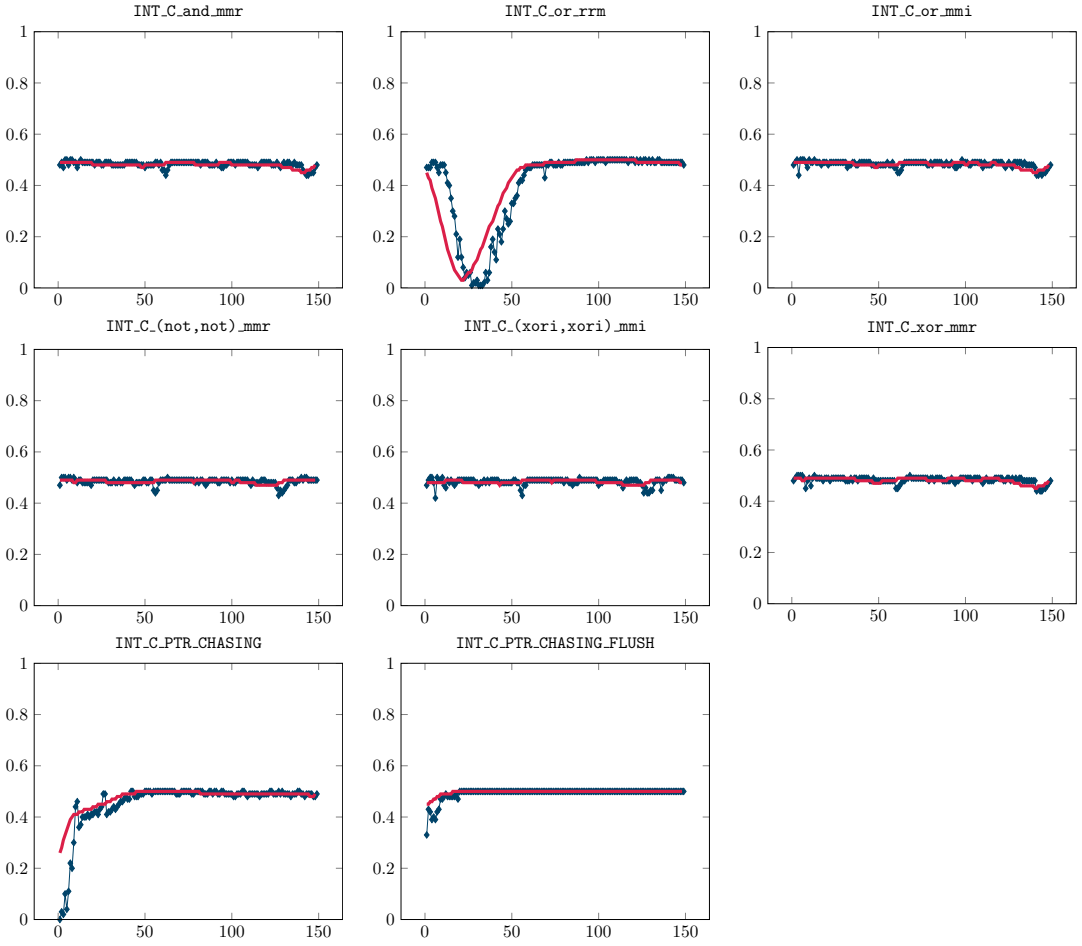


Figure 4.8: Test results for different test scenarios (IBM Power9). The x-axis illustrates the chain length n, the y-axis illustrates the average TPR. The blue graph (■) shows the TPR achieved for each chain length. The pink graph (■) shows the moving average.

4.5 Comparison

This section analyzes the differences and commonalities of our findings on the different platforms. We point out which test scenarios seem to work best or worse on which platforms and highlight possible similarities. The `BASIC` test case gives a TPR of 0.25 on Intel, 0.03 on AMD, 0.00 on ARM and 0.01 on IBM Power 9. Single floating point operations show a higher TPR than `BASIC` on Intel, AMD, and IBM Power9. On AMD and IBM Power9 we see that the TPR is especially high when a floating point division is involved.

On Intel and AMD we also test conditions depending on the result of single AVX instructions. They show a higher TPR than the `BASIC` test case does. Especially `AVX_S_vandnpd`

gives a high TPR on both platforms. The sizes of the speculation windows are between 100 and 150 instructions on both platforms.

Generally, integer instruction dependency chains increase the TPR on all platforms. We observe that `INT_C_imul` gives the best results among all integer instruction dependency chains on all platforms. Instruction chains which do not use one instruction but mix two or more also work very good on all platforms. Examples are `INT_C_(add,not,not)`. Floating point instruction dependency chains result in almost the same TPR as integer instruction dependency chain tests do on Intel and AMD but result in a higher TPR on ARM and IBM Power9. However, on all platforms, they reach a large window size.

The most effective way to increase performance on all four platforms is to implement pointer chasing in a chain and flush the involved pointers. On all platforms, this test maximized the speculation window. On Intel, a TPR of 0.94 is reached, on AMD 0.49, on ARM 1.00 and on IBM Power9 we get 0.50. The reached window sizes are very large. We also see that pointer chasing without flushing the pointers works well.

Dereferencing variables which have a flushed TLB entry also works very good on all platforms. However, the throughput is lower than for other tests.

Chained tests involving loads and stores (`mmr` tests) showed a drop in the TPR for chain lengths above 100 on Intel platform. On IBM, ARM, and AMD the TPR curve develops linearly. Tests involving only loads from memory `rrm` show a very stable TPR after a specific chain length on all platforms.

Chapter 5

Enhancing Real-World Attacks

In this chapter, we apply our findings from Chapter 4. The research we did so far tends to focus on a broad range of analytical tests, rather than how these results could be used in practical settings. In the following we first highlight the most obvious gain an attacker has from a larger speculation window, that is, leaking more data. The last part investigates Foreshadow and emphasizes the role an extended speculation window might play. We show that the reason why Foreshadow attacks do not work with speculation as an exception suppression mechanism is not that the speculation window is too small.

5.1 Leaking more data

An attacker's primary goal is to leak data - the more, the better. Especially when it comes to microarchitectural attacks, the leakage rate is often low. Using an extended speculation window, we show how one can leak twice as much data as with the basic version of the window. The idea is that if the CPU has more time to speculate, it will be able to do more memory accesses.

In order to demonstrate this, we need to modify the body of the condition. We focus on Spectre-PHT attacks based on loads in this experiment. Our goal is to leak up to two bytes per try, *i.e.*, make two memory accesses instead of one. Therefore, we copy the existing resources we use to leak data, *i.e.*, `oracle` and `data`, and establish a second cover channel to leak the second byte in each round. We need to enforce that the instructions in the condition body are not executed out-of-order by making them dependent on each other. Out-of-order execution would distort our results because one could never know how often the second memory access is executed before the first one. The updated condition body is shown in Listing 5.1. When there is a data dependency between them, the CPU is forced to execute always the first memory access before the second. In the last step of the attack, the attacker leaks the data of `data` with a separate cache covert channel, which looks the same as in the basic version (see Listing 3.2).

```
1 char access_array(int x)
2 {
3     int res;
4     if(/* some condition */)
5     {
6       asm volatile("movq (%1), %0\n" : "=r"(res): "c"(mem+data[x]*4096));
7       asm volatile("movq (%0), %%r10\n" : : "b"(mem2+data2[x+res]*4096) : "r10");
8     }
9 }
```

Listing 5.1: Test code to leak two byte in one try. There is a data dependency between the first and second memory access established by `res`, which is written in line 6 and read in line 7. This forces the CPU to in-order execution.

Table 5.1: Test results achieved on different platforms for the scenario where two bytes should be leaked.

Platform	TPR	TPR SD	TPR SEM	TPR	TPR SD	TPR SEM
Intel	0.66	0.34	0.03	0.34	0.34	0.03
AMD	0.67	0.06	0.01	0.66	0.43	0.01
ARM	1.00	0.00	0.00	0.99	0.00	0.00
IBM	1.00	0.00	0.00	0.99	0.00	0.00

We execute concrete tests using the condition `INT_C_PTR_CHASING_FLUSH` with the chain length which gave the maximum TPR according to Tables 4.2, 4.5, 4.7, 4.9 and report the average TPR for each platform. The results are shown in Table 5.1.

This test only demonstrates one very obvious way of how our results can be used. In an attack scenario, where the attacker controls the victim's code or parts of it, this can be very powerful.

5.2 Foreshadow using Speculative Execution

Foreshadow attacks involve an attacker accessing memory in transient execution which should not be accessible due to a zeroed present bit in the page table entry. This invalid access raises a page fault which leads to the termination of the userspace program. However, the attacker still wants to establish the cache side channel to leak the secret bytes. There are two ways to achieve this [50], exception handling and exception suppression. Exception handling uses a userspace exception handler, which is called in case a page fault is raised and hands over control to the attacker, who will start the cache side-channel there. Exception suppression means that instead of handling a raised exception, exceptions are prevented from being raised at all. Currently, there are two methods known for exception suppression, Intel TSX and speculative execution.

```
1 _xbegin();
2 maccess(array1[array2[x]* 4096]);
3 _xend();
4 //Start cache covert channel
5 //...
```

Listing 5.2: Exception surpression unsing TSX

```
1 //1. Mistrain
2 for(int i = 0; i < 20; i++)
3   access_array(0);
4 //2. Possible out-of-bounds access
5 access_array(secret_idx);
6 //3. Start cache covert channel
7 //...
```

Listing 5.3: Exception surpression unsing speculative execution

Transactional Synchronization Extensions (TSX) is implemented as an instruction set extension to the x86 instruction set [71]. Similar to database transactions, which guarantee atomic and consistent operations on databases, TSX offers to mark specific regions of code to be executed transactionally. This means that the instructions in those regions are executed in an all-or-nothing fashion: errors cause the transaction to be rolled back (all operations are undone) and successful execution causes the transactions to be committed architecturally.
In the case of Foreshadow, the attacker executes the invalid memory access in a TSX transaction. No fault is raised, the transaction is rolled back silently, and the attacker can start the cache side channel. Listing 5.2 gives an example.

Speculative execution can be used for exception suppression mechanisms as well, as shown in Listing 5.3. Whenever a branch misprediction happens, the CPU has to discard all operations which were executed speculatively, which can be seen equal to a rollback in TSX. In this case, exceptions are also not raised because the actual execution will continue somewhere else and the CPU wants to act as if the page fault never occurred. As is well known, Meltdown-US works with all both TSX and speculative execution [50]. However, research has failed to explicitly show that exceptions can be suppressed using speculative execution for Foreshadow as well. We suspect that this might be due to the insufficient size of the speculation window. Therefore, we want to evaluate this situation and have a closer look at it. Note that all our experiments are done on an Intel Skylake CPU, supporting TSX.

5.2.1 Experiment design and results

The basis of our experiments is a proof-of-concept Foreshadow-OS attack which supports exception suppression using TSX and speculative execution.

In the first experiment, we investigate if our assumption about the too short speculation window preventing a successful Foreshadow-OS attack using speculative execution is correct. We refer to our results of Chapter 4 in order to test different conditions. According to Table 4.2, `INT_C_PTR_CHASING_FLUSH` maximizes the speculation window for $n = 94$, which is 153 instructions. We also test a second dependency chain, `FLOAT_C_divss`

($n = 21$). Our results show that none of the modified conditions in the scenarios we tested managed to leak a single byte. The initial hypothesis that the reason for this is a too short speculation window is therefore wrong.

We conclude that this shows us that the size of the window in our first experiment is large enough and there must be some other mechanism preventing the Foreshadow attack with speculative execution from being successful.

Chapter 6

Conclusion

In this thesis, we presented a fully-automated framework for the analysis of the success rate of Spectre-PHT attacks. The success rate of these attacks primarily depends on the size of the speculation window because a larger speculation window implies a higher likelihood for the attack to succeed. The window size can be influenced by choice of the condition in Spectre-PHT attacks. The framework tests different conditions and detects the size of the speculation window automatically. We developed methods for robust measurements and evaluated the success rate in terms of the true positive rate (TPR) and the throughput of the attack. The framework was ported to Intel, AMD, ARM, and IBM Power9 CPUs.

We introduced a categorization of conditions in Spectre-PHT attacks which can be used by an attacker to increase the TPR of the attack. The attacker can either use slow instructions in the condition or fast instructions and make them slow. Slow instructions include simple integer or floating point instructions arranged in a dependency chain and accesses to DRAM, which are slow when the data is uncached, or the corresponding TLB entry is missing. Intel and AMD platforms offer the usage of AVX instructions which are slow whenever the AVX unit is not enabled prior to their use. Our analysis shows that dependency chains give the best results on almost all platforms. Dereferencing a chain of pointers which do not have a valid cache or TLB entry also yields high TPRs. Fast instructions made slow mainly include port contention scenarios. The attacker can execute a co-located process on the victim's machine which contends execution ports and makes the execution of instructions slower. We demonstrated that this has a positive effect on the success rate of the attack.

We addressed the practical relevance of our analysis in two different scenarios. First, we demonstrated that an attacker could leak twice as much data using an extended speculation window. Second, we investigated Foreshadow-OS attacks with speculative execution as an exception suppression mechanism. It was an open research question whether this could be possible with sufficiently large speculation windows. Our experiments showed that the size of the speculation window is large enough in this attack and that some other mechanism causes the failure of the attack.

Bibliography

[1] Advanced Micro Devices, Inc. Software Optimization Guide for AMD Family 17h Processors. https://www.amd.com/system/files/TechDocs/56255_OSRR.pdf, 2018.

[2] Advanced Micro Devices, Inc. Software Techniques for managing Speculation on AMD Processors. https://developer.amd.com/wp-content/resources/90343-B_SoftwareTechniquesforManagingSpeculation_WP_7-18Update_FNL.pdf, 2018.

[3] Advanced Micro Devices, Inc. Open-Source Register Reference for AMD Family 17h Processors Models 00h-2Fh, 2019.

[4] Aldaya, A. C., Brumley, B. B., ul Hassan, S., García, C. P., and Tuveri, N. Port Contention for Fun and Profit.

[5] Archibald, J., and Baer, J.-L. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM 4*, 4 (1986), 273–298.

[6] ARM Limited. ARM Cortex-A57 MPCore Processor - Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0488c/DDI0488C_cortex_a57_mpcore_r1p0_trm.pdf, 2016.

[7] ARM Limited. ARM Cortex-A57 Software Optimization Guide. http://infocenter.arm.com/help/topic/com.arm.doc.uan0015b/Cortex_A57_Software_Optimization_Guide_external.pdf, 2016.

[8] ARM Limited. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. https://developer.arm.com/support/security-update, 2018.

[9] Bakhvalov, D. Store forwarding by example. https://easyperf.net/blog/2018/03/09/Store-forwarding, 2013. Retrieved on March 14, 2019.

[10] Bakhvalov, D. What optimizations you can expect from CPU? https://dendibakh.github.io/blog/2018/04/22/What-optimizations-you-can-expect-from-CPU, 2018. Retrieved on March 14, 2019.

[11] Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., and Kurmus, A. SMoTherSpectre: exploiting speculative execution through port contention. (2019).

[12] Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., and Gruss, D. A Systematic Evaluation of Transient Execution Attacks and Defenses.

[13] Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., and Lai, T. H. SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution. (2018).

[14] Cheng, C.-C. The schemes and performances of dynamic branch predictors. (2000).

[15] Cheng, C.-H. Design example of useful memory latency for developing a hazard preventive pipeline high-performance embedded-microprocessor. (2013).

[16] Chevtchenko, S., and Vale, R. A Comparison of RISC and CISC Architectures.

[17] Clark, M. A new x86 core architecture for the next generation of computing. *Hot Chips Symposium* (2016).

[18] Fawcett, T. An introduction to ROC analysis. *Pattern recognition letters* (2006).

[19] Fog, A. Test results for Broadwell and Skylake. https://www.agner.org/optimize/blog/read.php?i=415#415, 2015.

[20] Fog, A. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly pro-*

grammers and compiler makers, 2016.

[21] Fog, A. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, 2018.

[22] Gonzales, A., Younis, E., Korpan, B., and Zhao, J. BOOM Speculative Attacks. `https://github.com/riscv-boom/boom-attacks`, 2019. Retrieved on March 17, 2019.

[23] Gonzalez, A., Latorre, F., and Magklis, G. Processor microarchitecture: An implementation perspective. *Synthesis Lectures on Computer Architecture* (2010).

[24] Goto, H. ARM Cortex-A57 Block Diagram, 2013.

[25] Gras, B., Razavi, K., Bos, H., and Giuffrida, C. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium* (2018).

[26] Gruss, D. Embedded Security - 2. Cache Template Attacks. `https://teaching.iaik.tugraz.at/_media/embsec/2018_lecture2_slides.pdf`, 2018. Retrieved on March 14, 2019.

[27] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA* (2016).

[28] Gruss, D., Spreitzer, R., and Mangard, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium* (2015).

[29] Gullasch, D., Bangerter, E., and Krenn, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *S&P* (2011).

[30] Gupta, S., Xiang, P., Yang, Y., and Zhou, H. Locality principle revisited: A probability-based quantitative approach. *Journal of Parallel and Distributed Computing 73*, 7 (2013), 1011–1027.

[31] Hall, B., Bergner, P., Housfater, A. S., Kandasamy, M., Magno, T., Mericas, A., Munroe, S., Oliveira, M., Schmidt, B., Schmidt, W., et al. *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8*. IBM Redbooks, 2017.

[32] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach (Fifth Edition)*. 2012.

[33] Hill, M. D., and Smith, A. J. Evaluating associativity in CPU caches. *IEEE Transactions on Computers* (1989), 1612–1630.

[34] Horn, J. Speculative Execution, Variant 4: Speculative Store Bypass. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`, 2018.

[35] Horn, Jann. Reading privileged memory with a side-channel. `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`, Jan. 2018.

[36] IBM. Out of Order Execution of Computer Instructions. `https://researcher.watson.ibm.com/researcher/view_page.php?id=6887`, 2019. Retrieved on March 15, 2019.

[37] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual. `https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf`, 2019.

[38] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. `https://software.intel.com/en-us/articles/intel-sdm`, 2019.

[39] Isen, C., John, L. K., and John, E. A tale of two processors: Revisiting the RISC-CISC debate. In *SPEC Benchmark Workshop* (2009), Springer, pp. 57–76.

[40] Kaeli, D., and Yew, P.-C. *Speculative execution in high performance computer architectures*. CRC Press, 2005.

[41] Khasawneh, K. N., Koruyeh, E. M., Song, C., Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. *arXiv:1806.05179* (2018).

[42] Kiriansky, V., Lebedev, I., Amarasinghe, S., Devadas, S., and Emer, J. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. *Cryptology ePrint Archive: Report 2018/418* (May 2018).

[43] Kiriansky, V., and Waldspurger, C. Speculative Buffer Overflows: Attacks and Defenses.

arXiv:1807.03757 (2018).

[44] Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre Attacks: Exploiting Speculative Execution. *arXiv:1801.01203* (2018).

[45] Koruyeh, E. M., Khasawneh, K., Song, C., and Abu-Ghazaleh, N. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT* (2018).

[46] Lee, B., Malishevsky, A., Beck, D., Schmid, A., and Landry, E. Dynamic branch prediction. *Oregon State University*.

[47] Limited, A. Cortex-A65AE. `https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a65ae`, 2018. Retrieved on March 15, 2019.

[48] Lipasti, M. H., and Shen, J. P. Exceeding the dataflow limit via value prediction. In *IEEE international symposium on Microarchitecture* (1996).

[49] Lipp, M. Cache Attacks and Rowhammer on ARM, 2016.

[50] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security* (2018).

[51] LLC, W. POWER9 - Microarchitectures - IBM. `https://en.wikichip.org/wiki/ibm/microarchitectures/power9`, 2019. Retrieved on March 11, 2019.

[52] LLC, W. Skylake (client) - Microarchitectures - Intel. `https://en.wikichip.org/wiki/intel/microarchitectures/skylake`, 2019. Retrieved on March 6, 2019.

[53] LLC, W. Zen - Microarchitectures - AMD. `https://en.wikichip.org/wiki/amd/microarchitectures/zen`, 2019. Retrieved on March 10, 2019.

[54] Maisuradze, G., and Rossow, C. ret2spec: Speculative execution using return stack buffers. In *CCS* (2018).

[55] Mandelblat, J. Intel's next generation microarchitecture code-named Skylake. `http://intelstudios.edgesuite.net/idf/2015/sf/ti/150818_spcs001/index.html`, 2015. Retrieved on March 9, 2019.

[56] Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., and Upton, M. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal 6*, 1 (2002).

[57] Masood, F. Risc and cisc. *arXiv preprint arXiv:1101.5364* (2011).

[58] Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Alberto Boano, C., Mangard, S., and Römer, K. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS* (2017).

[59] McCalpin, J. D. Test results for Intel's Sandy Bridge processor. `http://agner.org/optimize/blog/read.php?i=378#378`, 2015.

[60] Moreno, R., Pinuel, L., Del Pino, S., and Tirado, F. A power perspective of value speculation for superscalar microprocessors. In *Proceedings 2000 International Conference on Computer Design* (2000), IEEE, pp. 147–154.

[61] Moshovos, A., and Sohi, G. S. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture* (1997), IEEE Computer Society, pp. 235–245.

[62] Mutlu, O. Computer Architecture Lecture 19: Caching II, 2011.

[63] NISEC. `https://twitter.com/NISEC_TAU/status/1100460299906359296`, 2019.

[64] O'Keeffe, Dan and Muthukumaran, Divya and Aublin, Pierre-Louis and Kelbert, Florian and Priebe, Christian and Lind, Josh and Zhu, Huanzhou and Pietzuch, Peter. Spectre attack against SGX enclave. `https://github.com/lsds/spectre-attack-sgx`, Jan. 2018.

[65] Osvik, D. A., Shamir, A., and Tromer, E. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA* (2006).

[66] Page, D. *A Practical Introduction to Computer Architecture*. Springer Science & Business Media, 2009.

[67] Pantazi-Mytarelli, I. The history and use of pipelining computer architecture: Mips pipelining implementation. In *2013 IEEE LISAT* (2013), IEEE.

[68] Peckol, J. K. *Embedded systems: a contemporary design tool*. Wiley, 2019.

[69] Pinto Rivero, D. Study and evaluation of several cache replacement policies on a commercial MIPS Processor.

[70] Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C., and Emer, J. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News 35*, 2 (June 2007), 381.

[71] Reinders, J. Transactional Synchronization in Haswell. `https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`, 2012. Retrieved on March 28, 2019.

[72] Reineke, J., Grund, D., Berg, C., and Wilhelm, R. Predictability of cache replacement policies. Tech. rep., 2006.

[73] Schwarz, M., Schwarzl, M., Lipp, M., and Gruss, D. NetSpectre: Read Arbitrary Memory over Network. *arXiv:1807.10535* (2018).

[74] Shen, J. P., and Lipasti, M. H. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[75] Smith, A. J. Design of CPU Cache Memories. Tech. rep., EECS Department, University of California, Berkeley, 1987.

[76] Smotherman, M. IBM Stretch (7030) - Aggressive Uniprocessor Parallelism. `https://people.cs.clemson.edu/~mark/stretch.html`, 2010. Retrieved on March 14, 2019.

[77] Stecklina, J., and Prescher, T. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels, 2018.

[78] Stuecheli, J. POWER8/9. `https://openpowerfoundation.org/wp-content/uploads/2016/11/Jeff-Stuecheli-POWER9-chip-technology.pdf`, 2016. Retrieved on March 11, 2019.

[79] Tanenbaum, A. S. *Structured computer organization*. Pearson Education India, 2016.

[80] Tromer, E., Osvik, D. A., and Shamir, A. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology 23*, 1 (July 2010), 37–71.

[81] Tucker, A. B., Ed. *The Computer Science and Engineering Handbook*. CRC Press, 1997.

[82] Tullsen, D. M., Eggers, S. J., and Levy, H. M. Simultaneous multithreading: Maximizing on-chip parallelism. In *ACM SIGARCH Computer Architecture News* (1995), ACM.

[83] Turner, P. Retpoline: a software construct for preventing branch-target-injection, 2018.

[84] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-order Execution. In *USENIX Security Symposium* (2018).

[85] Wang, Z., and Lee, R. B. Covert and side channels due to processor architecture. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)* (2006), pp. 473–482.

[86] Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T. F., and Yarom, Y. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report* (2018).

[87] Weisz, G., Melber, J., Wang, Y., Fleming, K., Nurvitadhi, E., and Hoe, J. C. A study of pointer-chasing performance on shared-memory processor-fpga systems. ACM.

[88] Wong, H. Intel ivy bridge cache replacement policy. `http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/`, 2013.

[89] Wong, H. Measuring reorder buffer capacity. `http://blog.stuffedcow.net/2013/05/measuring-rob-capacity/`, 2013.

[90] Wong, H. Store-to-Load Forwarding and Memory Disambiguation in x86 Processors. `http://blog.stuffedcow.net/2014/01/x86-memory-disambiguation/`, 2014.

[91] Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C. W., and Torrellas, J. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *MICRO* (2018).

[92] Yarom, Y., and Falkner, K. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014).

[93] Yeh, T.-Y., and Patt, Y. N. Alternative implementations of two-level adaptive branch prediction.

[94] Zhang, P. *Advanced industrial control technology*. 2010.