PageZero: Mitigating Speculative Execution Attacks by Clearing Page Tables

MSc Thesis *(Afstudeerscriptie)*

written by

Floris Westerman

under the supervision of **dr. Klaus von Gleissenthall** and **dr. Balder ten Cate**, and submitted to the Examinations Board in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:
November 5th, 2024

INSTITUTE FOR LOGIC, LANGUAGE AND COMPUTATION

Abstract

The discovery and subsequent disclosure of the Spectre and Meltdown vulnerabilities have kickstarted an era of speculative execution vulnerabilities that exploit hidden microarchitectural CPU features to exfiltrate secret data. So far, awaiting hardware-level fixes in new products, the majority of mitigations for existing platforms have been 'spot' mitigations. These fix specific vulnerabilities in specific vulnerable and high-risk code paths, unleashing a game of 'Whac-A-Mole' to keep computers secure. Even worse, these mitigations impose significant runtime overhead, making them unattractive.

We introduce *PageZero*, a novel approach for a more comprehensive solution to speculative execution vulnerabilities that combines powerful features from previous work by both VUSec and Microsoft, in the form of Quarantine and Secret-Free Hypervisor. In this hybrid approach, we separate virtual machine workloads between physical cores, while also clearing the hypervisor kernel page tables to prevent leaks via the hypervisor. We will show the effectiveness of this approach against Spectre v1 by introducing a generic and flexible speculative semantics that can be used to model various speculative execution attacks.

Contents

Chapter 1

Introduction

Back in 2018, the discovery of Spectre [1] and Meltdown [2] ushered in a new era of computer security research. Code that was previously thought to be secure turned out to be vulnerable to speculative execution exploits, allowing attackers to exfiltrate all system memory, even from Javascript code in a web browser [1]. After its disclosure, a flood of new and similar vulnerabilities was unleashed [3–9].

Due to the widespread issues and severe security impacts of these vulnerabilities, vendors and developers raced to implement mitigations for the code paths that were most affected. Unfortunately, since at its core these vulnerabilities are hardware issues, it is not always possible or feasible to mitigate the issues in software alone [10–12].

Since then, new vulnerabilities and subsequent mitigations have followed each other in rapid succession. While some issues are fixed in newer hardware revisions, lowering the software mitigation overhead [11], new vulnerabilities continue to be found. Most machines are left only partially protected because of the high performance impact of some mitigations [10], while developers keep struggling to find more and more vulnerable code paths [13, 14].

Clearly, a more comprehensive approach is highly desired, and such attempts have been made [15–20]. Of particular interest are the approaches by Microsoft and VUSec: Secret-Free Hypervisor [16] and Quarantine [15]. Both operate in a hypervisor-virtual machine context, and try to rigorously prevent speculative execution attacks by either making the hypervisor an uninteresting target, or by physically moving it away from guest code, to another processor core.

We will present *PageZero*, a new comprehensive approach aimed at consolidating ideas from both Secret-Free Hypervisor and Quarantine, without the implementation complexities of either. In this hybrid approach, for each virtual machine, the hypervisor kernel has a separate set of page tables, thus making the kernel an uninteresting target for speculative execution attacks, while simultaneously continuing to function normally for benign workloads. Additionally, different VMs are physically separated across different cores, to prevent microarchitectural leaks between VMs.

The research work is divided among two MSc theses; one for formally showing effectiveness of the approach, and one for a prototype implementation and evaluation of the design [21]. In this thesis, we will do the former: we will present a formal model used for reasoning about speculative execution, and use it to formally prove the effectiveness of our approach.

The model we will present is based on previous work [22] and features an x86-inspired three-stage processor design with support for out-of-order and speculative execution. We extend the model to integrate page tables, speculative memory access, and store forwarding. Whereas the original model was used to prove specific code fragments secure against speculative execution, we will now use the model to show overall *system* integrity. Further details can be found in Chapter 5.

To show the effectiveness of our approach, we will first present both sequential and speculative semantics for our model in Section 5.2 and Section 5.3. We will establish confluence for the speculative semantics (Theorem 1), which we will then use to show consistency between the two semantics (Theorem 2). Lastly, we will prove a 'transfer' result (Lemma 6.3.6) that enables us to transfer specific properties about programs under sequential semantics to the speculative realm. In turn, this allows us to show the effectiveness of clearing page tables against Spectre v1 attacks.

The formal model used in this thesis is not limited to Spectre v1, and can easily be modified and extended to implement other vulnerabilities like Spectre v2 [1], Meltdown [2], RIDL [4], LVI [5], Fallout [7], and ZombieLoad [3]. The transfer result obtained in this thesis is more widely applicable than only memory safety, and can be used to port various memory properties to the speculative execution context.

Chapter 2

Background - Computer Architecture

Before we can start analysing the work in this thesis, it is important to have a solid understanding of the inner workings of present-day computers and their operating systems. This chapter is devoted to exactly that: providing an overview of concepts such as the *memory hierarchy*, *address spaces*, *page tables*, *protection rings*, and *virtualisation*. We will introduce these concepts from the bottom up: starting with low-level hardware details as building blocks for high-level software implementations. The information in this chapter largely appears in the other thesis as well, but the content has been tailored to the specific thesis. For more information on these concepts, refer to the excellent book by Tanenbaum [23].

2.1 Memory Hierarchy

A core concept in computer architecture that impacts all aspects of software design is the so-called *memory hierarchy*. Many people might be familiar with the basic distinction between RAM (Random-Access Memory) and hard drive storage, with RAM being the faster, smaller, and volatile 'working memory', and a hard drive providing slower, larger, and long-term data storage.

In reality, there is an entire hierarchy of memory techniques, as shown in Figure 2.1.1. Of note is the logarithmic scale in this illustration: every next 'tier' is at least an order of magnitude slower in terms of latency, but also typically has a significantly larger storage size. Concretely: if a CPU needs to load a value from a register, it can do so immediately; while a value from cache will already take tens of clock cycles. Conversely, a modern CPU only has 16 general-purpose registers (each 64 bits wide), while modern caches reach sizes of multiple megabytes.

Of note is the gigantic gap in latency between RAM and solid state drives, of three orders of magnitude. This gap is why (volatile) 'memory' is treated completely differently from 'storage' in computer architectures; although in recent years, technological developments are starting to close the gap [24].

For the purposes of this thesis, we are only interested in the top three tiers of volatile memory: registers, cache, and RAM. In particular, the gap between the caches and RAM is important.

CPU Caches

Typical x86-based CPUs from Intel and AMD contain a layered cache structure of three to sometimes four layers, simply called 'L1', 'L2', 'L3', and 'L4' cache [25]. The higher the number, the slower and bigger the cache. The various cache levels employ different strategies in terms of

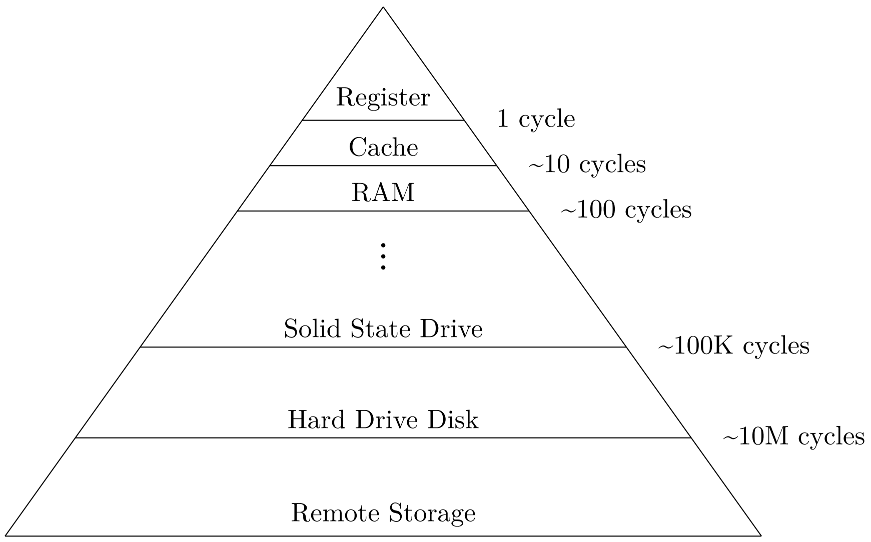

Figure 2.1.1: Memory hierarchy in computers. The faster the memory, the lower the capacity and the higher the price, and vice versa. The top three tiers are typically volatile memory, while the rest is typically non-volatile, long-term storage. The amount of cycles refers to the latency of a read operation in clock cycles. Modern processors achieve around 4 GHz clock frequency.

data structures and eviction rules. Typically, every core in a CPU will have its own L1 and L2 caches, while L3 and L4 are shared between cores.

We should note that these cache layers are built to be transparent implementation details from the perspective of code running on the computer. Code written in Assembly, which amounts to direct instructions to the CPU, deals with registers and memory addresses only. When moving up in abstraction to programming languages like C, even registers are no longer a concern for the user.

Behind the scenes, the CPU will manage the cache for us. For example, if we issue a `LOAD` instruction for a particular address a, the CPU will first look for the value in all cache layers. In case the address is not found (a *cache miss*), the RAM will be queried (which will take ~100 cycles). After a successful query from RAM, the value will be stored in the cache, typically in L1 cache. If we then issue another `LOAD` instruction for the same address, the cache can be used (a *cache hit*), and thus performance is greatly improved.

2.2 Virtual Memory

Early computer designs allowed programs full control over system memory: only a single program was running at a time, and it could thus be offered full control over the entire system, including the memory. This quickly became too limiting: multiple users started to want to use a system simultaneously, or a single user wanted to run multiple programs at the same time.

Nowadays, of course, a computer runs hundreds or thousands of programs at the same time, without them interfering. This is achieved by isolating the various processes, in particular through the use of virtual memory. Concretely, and most relevant for our work, each process running on a computer gets its own *virtual memory address space*. The CPU has a so-called *memory management unit* (MMU) that will translate accessed *virtual memory addresses* into physical addresses using the *page tables*. This is illustrated in Figure 2.2.1.

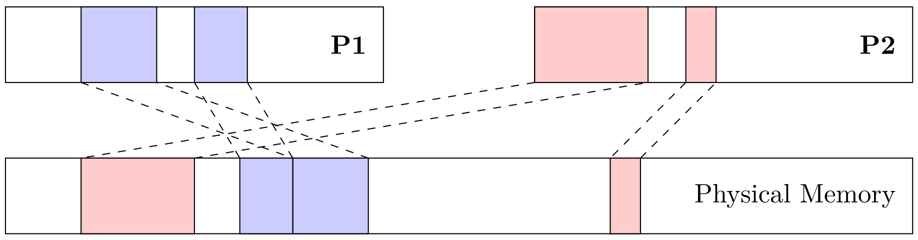

Figure 2.2.1: Illustration of how two virtual address spaces **P1** and **P2** map to physical memory. Coloured sections indicate memory contents, the rest of the address space is empty and unused.

2.2.1 Address Spaces

Modern machines are typically '64-bit', which means that all memory addresses (both virtual and physical) are 64 bits (or 8 bytes) long. In theory, that would mean that each virtual address space would have $2^{64} \approx 18.4\,\text{EB}$ of memory addressable. In practice, hardware limitations cause 'only' 48 bits to be usable, yielding 256 TB of addressable memory per address space.

Note that we only consider *addressable* memory here - the number of memory addresses that are well-formed and 'valid' for the CPU. In practice, there does not (yet) exist any machine that actually has that much physical memory. Operating systems need to manage the physical memory carefully, and thus 'interleave' the sections of virtual memory that are in use to the physical memory space, as seen in Figure 2.2.1. There is not a single best way of arranging data into physical memory, and there is a large design space for so-called memory allocation algorithms with the objective to reduce fragmentation and optimise utilisation.

2.2.2 Page Tables

As mentioned, when using virtual memory, every process has its own virtual memory space, making it unaware of the actual physical memory layout in the system. The program can just read and write to virtual addresses as if they were physical ones, and the CPU (specifically, the MMU) translates them properly. This procedure can be modelled by a simple map function taking virtual addresses to physical ones.Of course, the mapping itself would also need to be stored in memory, which poses a problem: with a domain of 2^{64} entries, such a map is far too large to store.

To combat this problem, computers divide the memory space into *pages* of 4096 bytes, corresponding to the last 12 (least significant) addressing bits. A page is then the smallest unit of memory managed by a machine, and so the virtual to physical memory mapping only needs to map pages and not individual addresses. This leaves us with a domain of 2^{52} entries, still too large to be of any practical use.

To further reduce the size of the mapping, CPUs make use of a data structure called *page tables*. These are essentially a sparse prefix tree, encoded as simple lookup tables. As mentioned before, current architectures typically use 48 out of 64 available bits. Removing the 12 bits used to identify addresses *within* a page, this leaves us with 36 bits to identify a page, which are split up in 4 levels of page tables, each resolving 9 address bits. Because of their sparse nature, only memory that is actually in use needs to be backed by page tables - saving a lot of memory space.

As seen in Figure 2.2.2, the PML4 ('Page Map Level 4') table is the root of the lookup process. This table contains $2^9 = 512$ entries, one for each possible value of bits 39..48 in the memory address[1]. Each entry (when present) contains the (physical) address of the appropriate PDPT

[1]Each table entry is 64 bits and thus 8 bytes long, therefore each table is exactly $8 \cdot 512 = 4096$ bytes long - exactly a page!

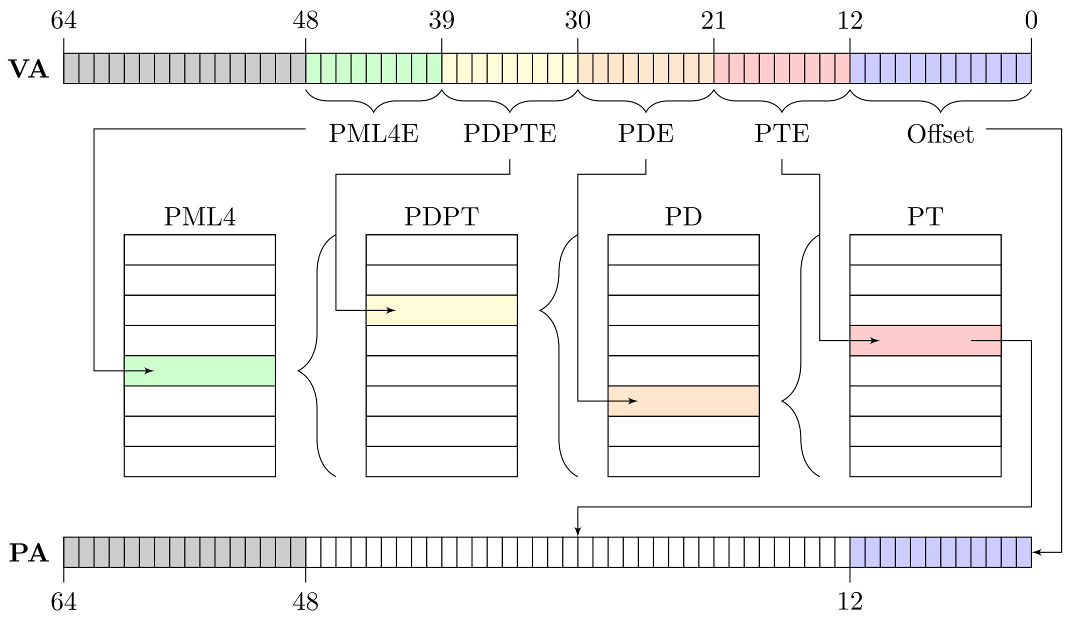

Figure 2.2.2: Illustration of structure and implementation of page tables. The 64-bit virtual memory address at the top is decomposed in 6 sections: the final 12 bits are the offset within a page, the next 4 blocks of 9 bits serve as offsets in the respective layer of page tables, and the first 16 bits are currently ignored. An entry in any of the layers contains a physical address pointing to the next layer of page tables, or to the actual backing page. The final physical address is reconstructed as the page address stored in the last-level page table entry, combined with the offset within the page from the virtual address.

('Page Directory Pointer Table')[2], which will resolve the next 9 bits. After the PDPT follows the PD ('Page Directory'), and finally the PT ('Page Table'). At PT level, an entry contains the physical address of the memory page containing the data for the requested virtual address.

Note that the use of virtual memory adds a lot of complexity (and latency) to the processor runtime. In the worst case, where nothing is cached, a single lookup of a virtual address can trigger 4 additional round trips to RAM to load all the page tables! Even when the page tables are cached to some extent, the CPU still needs 4 round trips to the cache. Therefore, the CPU employs yet another cache: the Translation Lookaside Buffer (TLB), which stores virtual to physical address mappings.

2.3 Protection Rings

As we saw, Virtual Memory is one of the tools used to isolate processes, but it is certainly not the only one. Another core concept in modern computer architectures are the so-called *protection rings*, which dictate the 'privileges' of a process. On x86, there are 4 protection rings numbered 0 to 3, with ring 0 being the most privileged and ring 3 being the least privileged.

In practice, most modern operating systems only use 2 of these rings: numbers 0 and 3. Ring 0 is used for the kernel of the operating system, where it can manage all hardware resources, and in particular the lifecycle of virtual address spaces and page tables. Ring 3 is then used for 'user code'; any program a user will interact with.

[2]As mentioned, the entries are 64 bits, and the tables are the size of an entire page. Therefore, only $48 - 12 = 36$ bits in the entry are relevant for finding the next level table. The remaining bits are used for metadata of the page, such as read/write/execute permissions.

Communication between rings, and switching control between them, can be done in various ways. Roughly speaking, the architectural design assumes a system to be running in ring 3 (user code) as the default. Either a special '`syscall`' instruction or an interrupt will trigger the CPU to enter ring 0, requesting the kernel to handle the event. Afterwards, the kernel will return control back to the user.

2.3.1 Interrupts and Page Faults

Interrupts, as the name suggests, cause the CPU to halt execution of the current code and switch to a so-called *interrupt handler*. Interrupts can be triggered by both hardware and software. Hardware interrupts include clock signals, disk read operations being ready, or simply a keyboard key being pressed. Software interrupts can be triggered by a special instruction, or through various exceptions such as dividing by zero.

One interrupt in particular is important for the work in this thesis: the *page fault*. A page fault is triggered when user code attempts to access a virtual address that is not accessible, for example because the virtual address cannot be resolved to a physical address using the page tables as described above.

When a page fault is triggered, it is the responsibility of the kernel to resolve the situation. In case the memory access is indeed incorrect or forbidden, the kernel will likely terminate the offending program, resulting in the infamous *Segmentation Fault*. In other cases, the memory might not have been available yet, because most kernels will only allocate memory *on demand*, as it is being used.

This last part is a crucial optimisation in kernel design: when a program requests the allocation of a block of memory, odds are that not every single byte of that allocation will be used. Therefore, kernels will internally record the reserved allocation, but will not actually 'hand out' the physical memory yet, and will thus not yet make the memory available in the process' page tables. Only after the user code tries to access this section of memory, when a page fault is triggered, the kernel will perform the allocation, update the page tables, and return control back to the user.

When page faults (or other exception-based software interrupts) are handled by the kernel, this is not visible to the user code. From the perspective of the user, the 'failed' memory access just succeeded directly without any issues. This allows us to abstract away the implementation of page tables and demand paging from the user.

2.3.2 Memory Protection

Switching between protection rings does not switch the virtual address space. Without any further measures, that would mean that a user process could manipulate all kernel memory, which of course is not desirable. To solve this, each page table entry comes with a set of metadata, indicating whether the memory for that address should be readable, writeable and/or executable, and whether it is only accessible by the kernel (ring 0), or any process. The kernel will then mark all its own memory as 'protected', while allowing the user access to its own memory. In case a user does try to access a protected address, this will trigger a page fault which the kernel will then handle by killing the offending program.

2.3.3 Kernel Memory Layout

Typically, a kernel will employ a design where the upper half of the virtual address space is reserved for kernel use, with all its memory addresses marked to only allow kernel access. This upper half is then shared between all processes - every virtual address space created will have the same upper half. This can be easily achieved by copying the last 256 entries of the PML4 table, letting them point to the same underlying page table hierarchies.

Furthermore, a kernel typically contains a so-called *direct map*, a memory region where all the physical memory is made available by linearly mapping a physical address `x` to a virtual address `x + a`, with `a` some constant. In case of Linux, this direct map is made available from `0xFFFF888000000000` onwards [26].

2.4 Virtualisation

Finally, we will shortly discuss *virtualisation.* Virtualisation allows us to run an entire operating system on our machine as if it was a normal user process. The *host* will typically run a normal operating system with potentially other programs running, among which is a special one (the *hypervisor*) running the virtual machine[3]. Within the VM, also called the *guest*, the computer behaves like normal: the act of virtualising a system is fully transparent.

One might wonder how this is achieved: after all, how can the guest run code in ring 0 while not conflicting with the host kernel that also runs in ring 0? This is achieved through hardware features enabling the 'switching' between entire different operating systems. These features are sometimes colloquially called 'ring -1', as intuitively they are 'more privileged' than normal kernel code in ring 0.

The full implementation of virtualisation is very complex, but to us one aspect will be relevant: the handling of page tables.

Nested Paging

Modern CPUs support a feature called *nested paging* or *second level address translation (SLAT)*, that helps with implementing virtual memory inside a virtual machine [27].

As discussed before, in a normal machine, user code deals with virtual addresses, which are translated to physical address through the page tables. Each process has its own virtual address space, and the kernel manages the single physical address space. When adding virtualisation on top of this, we need a way to allow the guest to use virtual memory like normal, while still only using a single physical address space in the host.

Nested paging enables this by allowing virtual machines to use two sets of page tables: inside the guest, the operating system will 'see' a single (virtualised) physical memory address space, and will set up paging like normal. So, user code inside the guest will access a *guest* virtual address, which will be translated by the CPU to a *guest* physical address. In reality, though, the 'physical memory' seen by the guest is just a virtual address space in the host. The *guest* physical address (i.e. the *host* virtual address) is then *again* translated by the CPU to a *host* physical address.

This is a very clean approach: to the virtual machine guest, this process is completely transparent, while for the host, the virtual machine can be treated as any other process for memory management.

Unfortunately, a big issue with this approach is performance. For normal paging, in the worst case without caching, 5 round trips to RAM are needed to perform a single address lookup. However, for nested paging the situation is worse, since the guest page tables are located inside the guest physical memory, so in the host virtual memory! Therefore, each of the 5 guest physical memory lookups might trigger 5 more host RAM round trips - yielding 25 round trips in total.

Luckily, in practice, these major overheads are barely noticeable, and virtual machines typically are nearly as performant as the host operating system. Next, we will see how processors achieve this, and how this affects security.

[3]There exist special operating systems where the hypervisor is the only thing that runs as part of the kernel. In all cases, the code responsible for managing the VM is what we call the hypervisor.

Chapter 3

Background - Speculative Execution Attacks

With a decent understanding of the relevant aspects of computer architecture, we can turn towards the topic of *speculative execution attacks*. Speculative execution is just one of a range of features and 'tricks' employed by modern computers to improve performance. Unfortunately, it is one that brought with it a host of unintended vulnerabilities that have shaped the system security landscape since the initial discovery of the Meltdown and Spectre vulnerabilities [1, 2].

By diving into speculative execution, we cross the boundary between *architectural* and *microarchitectural* behaviour. Architectural behaviour and features are those that are described in official 'Instruction Set Architecture' (ISA) specifications, such as AMD64 [27]. This is what we would typically call the 'behaviour of a system'. For example, registers, page tables, and virtual address spaces are all included in the ISA, while the various levels of caching are left out. The microarchitecture of a system, then, refers to the *implementation* of a particular ISA. It describes what truly occurs in the hardware - not just what 'we', the programmers, can observe in our code. So in our example, the existence and behaviour of the caches are part of the microarchitecture.

This chapter will discuss microarchitectural behaviour that is present in most modern chips, but in particular our explanations will apply to recent Intel and AMD CPUs, from the last 10-15 years. ARM and RISC-V CPUs will have similar behaviour, but with subtle differences and often different names, and will be out of scope. As before, the information in this chapter is largely shared with the other thesis, but has been tailored to fit this thesis.

3.1 Out-of-Order Execution

The classic architectural view of code execution is that the CPU will execute the program instructions one-by-one, in order, and as atomic units. This representation allows developers to reason about their code more easily, as a sequence of instructions. However, it could not be further from the truth: in reality, CPUs will break up instructions in various *micro-operations*, will change the order of execution at will, and will rerun instructions regularly - all in an effort to improve performance. Luckily, this is all hidden from the developer: the ISA requires the CPU to only ever present a coherent and valid state to the user, pretending as if the CPU is executing instructions neatly in order.

The process of executing a single instruction can be decomposed into many smaller steps, which are executed as 'stages' in a pipeline. The exact implementation will differ between CPU vendors and models, but modern Intel CPUs employ as many as 19 stages [28]. For our purposes, we can roughly identify the following steps:

Fetching the next instruction to be executed, possibly reading from main memory.

Decoding this instruction to determine what the instruction is, and what its arguments are.

Executing the instruction, for example requesting a load from RAM, or performing a floating point operation.

Retiring the executed instruction, possibly writing the result to the appropriate registers or memory locations.

As we saw when looking at the memory hierarchy in Section 2.1, loading a memory address can take up to 100 clock cycles. When adding nested paging to the mix, these numbers grow rapidly. If the CPU were to execute all instructions one-by-one and in-order, the entire CPU would halt for 100 cycles while waiting for data, which would be very inefficient. Therefore, modern CPUs are strongly parallelised and pipelined: every stage will be doing something at every clock cycle. Furthermore, CPUs execute instructions out of order: while one instruction might be waiting for a data load from RAM, the next instruction might be a simple floating point operation that is independent of the previous load. The CPU will then opportunistically execute the floating point operation while waiting for the data from memory.

Note that only the execute stage will be out-of-order; fetching instructions is still in-order because the code is written in-order, and importantly, retiring instructions is also in-order, to maintain data consistency and causality guarantees. To allow for sufficient parallelisation and out-of-order execution, CPU pipelines are very wide - up to 600 instructions can be in flight simultaneously [29].

Out-of-order execution combined with a strongly pipelined architecture improves CPU performance significantly, at the cost of a significant increase of implementation complexity. Many of the stages in the CPU pipeline are dedicated to safely reordering instructions, virtually aliasing registers, and ensuring (architectural) consistency. Of particular interest for us are the following microarchitectural features:

The *reorder buffer* contains all instructions that have been fetched and decoded, and are ready for execution.

The *store buffer* contains store operations that have been executed and committed - so the values are final - but not yet retired. The store buffer is used to *forward* these pending stores to future instructions without having to wait for the memory write and load operations.

3.2 Speculative Execution

Even with a massively parallel microarchitecture with very large reorder buffers, a CPU will still hit choke points during execution: branches. No matter how well a CPU is parallelised, if the program contains a conditional jump (if/else) that is dependent on a memory load, your program will inevitably halt until the load has been resolved.

For the next leap in performance, processors jump to *speculative execution*: for each branch point, hardware-level heuristics will speculate and predict an outcome for the pending instructions, in part based on execution history. The CPU will contain a dedicated *branch predictor* to assist in this process.

When the CPU has made a prediction, it will speculatively and opportunistically start fetching, decoding, and executing the selected code path - all in parallel and out of order, of course. Once the branch point has finally been resolved, either the prediction is confirmed and execution can continue, or a *misprediction* has occurred, triggering a complete *rollback* of the processor state to the branch point. Architecturally, then, these mispredictions are invisible and execution remains consistent and predictable.

Let us clarify some terminology: the act of predicting a branch point and opportunistically continuing execution is called *speculative execution*. The time between a misprediction and the eventual rollback is called a *transient window*, while the work done by the CPU in that window is referred to as *transient execution*.

3.3 Side Channels

As previously established, there is a difference between the architecture and microarchitecture of a processor. In previous sections, we often mentioned that *architecturally*, any microarchitectural features and optimisations (such as the L1-3 caches, speculative execution, etc.) are invisible. However, it turns out that by employing *side channels*, we are able to observe and analyse 'hidden' details of the microarchitecture.

Side channels often appear in computing science - they are 'channels' to exfiltrate information from a system that is not intended to be exposed, but without breaking a system. In most cases, side channels are an unintended consequence of a particular *implementation* of a system or protocol, rather than the design of that system or protocol itself.

A good example of a simple side channel would be an audio recording of someone typing on their keyboard. On most keyboards, every key will make a slightly different sound when pressed, and thus an audio recording can be used to deduce the message that was typed, such as a password. There are numerous online demonstrations of this technique [30], which the reader is encouraged to try out.

For speculative execution attacks, we will focus on a combination of two specific side channels: *timing attacks* and *cache attacks*. Our objective is to deduce (secret) memory data based on the time it takes to read specific memory addresses.

3.3.1 Timing Attacks

In a timing attack, (microarchitectural) implementation details that influence latencies in the system are exploited to deduce information about system internals that should be hidden from the user. For example, consider a login system that will compare the hash of an entered password against the 'correct' password hash, where each hash has a length of 128 bytes. If the comparison implementation aborts as soon as an incorrect byte has been found, an attacker could measure the response time of the system for many possible entered passwords, and deduce how many of the input characters are correct: the longer the response time, the more characters have been checked.

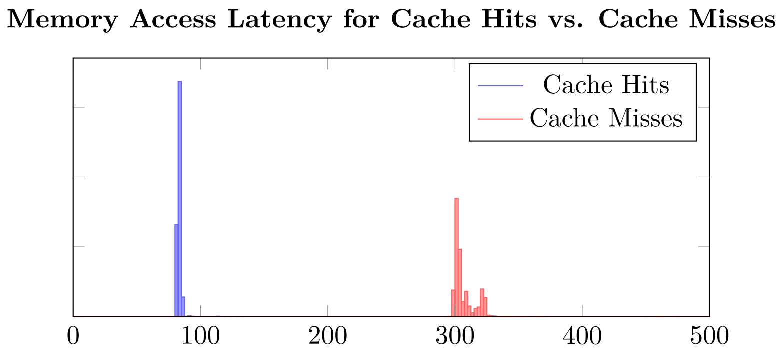

Figure 3.3.1: Memory access latency measurements for both cache hits and cache misses on an Intel Xeon 6150 CPU. Memory locations were flushed before cache miss tests. A clear separation in latencies (clock ticks) can be seen.

The timing attack we will consider is one involving the microarchitectural aspects discussed above. As we have seen in Section 2.1, reading from cache is significantly quicker than reading from RAM. And indeed, in practice, it is very easy to distinguish cache 'hits' from cache 'misses', as seen from the experimental data in Figure 3.3.1. This allows us to 'break' the abstraction and actually analyse what is going on in the caches.

3.3.2 Cache Attacks

In a cache attack, information gathered about the cache is used to deduce other information about the target system. In our case, a cache attack works as follows:

- We set up a memory region of at least 256 bytes[1].
- We initialise this region with some dummy data, thus ensuring all addresses have been accessed once and all parts of the region have the same microarchitectural state.
- We request the CPU to flush the region from all caches, which can be done as normal user code - no special privileges are needed.
- We 'trick' some other program, our target, to read data from our memory region. In particular, we would like the target to read a single address. This will cause the address to be populated in the various CPU caches.
- Using a timing attack, we can then deduce which address was read, since only the target address is in cache.

Depending on the characteristics of your target program, the address that was loaded will tell you something about the internals of your target. For example, if your target program will read at an offset `i`, this attack can deduce the value of `i`, which might be secret.

3.4 Speculative Execution Attacks

We now turn to the main topic of this thesis: speculative execution attacks. The Spectre vulnerability [1] can be seen as the archetypical speculative execution attack, and since its discovery in 2018 a plethora of similar attacks have been described [3–9].

All such attacks depend on microarchitectural features or details that are not properly rolled back upon a misprediction. For example, when speculatively executing a `LOAD` instruction, the loaded address will remain in cache even after the rollback. This makes sense, since the cache itself is a microarchitectural implementation detail, so not performing the rollback properly for the cache does not affect the architectural state.

Speculative execution attacks need two more ingredients: they need to trigger a (sufficiently large) transient window, and they need a side channel to exfiltrate information about the microarchitectural feature that is not properly rolled back.

3.4.1 Spectre

Spectre combines the fact that CPU caches do not properly get rolled back with the aforementioned cache side channel. Together with an appropriate transient window, this typically allows Spectre attacks to read any arbitrary memory address in the current virtual address space.

Over the years, many Spectre variants have been discovered - typically only with different transient window triggers, but with the same objective to use the cache side channel. We will discuss some of these variants, starting with the original 'v1', the simplest and earliest Spectre attack.

[1] In practice, due to CPU memory prefetching, 256 bytes is not sufficient, and a much larger memory region of about 256 pages (so 1 MB) is needed.

(a)

```c
1  int find_item(uint key, int *items) {
2      if(key >= max_key) {
3          return -1; // Out-of-bounds
4      }
5
6      int index = map[key];
7      return items[index];
8  }
```

(b)

```c
1  int spectre_gadget(uint x, int *buffer) {
2      if(x >= max) {
3          return -1; // Out-of-bounds
4      }
5
6      int y = arr[x];
7      return buffer[y];
8  }
```

Code Snippet 3.4.1: Spectre v1 proof of concept. Snippet **(a)** is made more readable by giving some more useful variable names; **(b)** is a default Spectre template. Under normal circumstances, this is perfectly valid and safe code, but under speculative execution, it is possible to bypass the bounds check and perform a memory read of any arbitrary `key` (or `x`), thus leaking the value of any memory address.

Spectre v1

Consider the code in Code Snippet 3.4.1a. At face value, this looks like perfectly normal code: it provides some kind of lookup behaviour inside the provided `items` array, based on some integer value `key`. To prevent an out-of-bounds read on `map`, an appropriate guard is inserted. Architecturally, this is perfectly safe. Unfortunately, when adding speculative execution into the mix, this code can be exploited to read any arbitrary memory address. In fact, any code pattern with some kind of bounds check followed by a double pointer dereference can be vulnerable. In Code Snippet 3.4.1b we have renamed all variables to more clearly show the Spectre 'template'. The exploit works as follows:

1) The attacker will prepare a special buffer to be used in a cache attack; most importantly, the buffer contents are flushed from any caches.

2) The attacker will call `spectre_gadget` repeatedly using some value `x < max`, in a perfectly valid way. This way, the branch predictor is 'mistrained': it is more likely to assume that the next call will again not satisfy the condition on line 2.

3) Now, the attacker will call `spectre_gadget` using any invalid value `x >= max`, and the prepared `buffer`.

4) The bounds check is a branching point that will open a transient window. For example, the check could depend on loading the value `max` from main memory, so after the 'mistraining' from before, the CPU might make the incorrect guess that the bounds check is not violated and will continue executing the next instructions.

5) Within the transient window, `arr[x]` is loaded - which is out of bounds for the array! Therefore, this could point to any arbitrary memory location in the current virtual address space. The CPU will load this value like normal, since of course a process has access to all its own memory, and no memory permission check has been violated.

6) Next, within the transient window, the loaded value for `arr[x]` (`y`) is used to index into our special `buffer`. The CPU will load the `y`th entry of the buffer, thus bringing that value into cache.

7) At some later point, the CPU will finally complete the evaluation of `x >= max`, and determine that the bounds check has failed. Now, the transient execution will be rolled back, so that architecturally the system behaves as if nothing has happened. Crucially, however, the cache loads will not be rolled back.

8) Now, the function will return and the attacker can perform a timing attack on the special `buffer` to determine which entry was loaded - and thus reveal the value of `y`!

```c
int spectre_gadget(uint x, int *buffer) {
    int y = arr[x];
    return buffer[y];
}
int no_op(uint x, int *buffer) {}

// Method takes a callback function accepting two parameters,
// both with the same signature as the two functions above.
int target(uint x, int *buffer, int (*callback)(uint, int *)) {
    (*callback)(x, buffer);
}
```

Code Snippet 3.4.2: Proof of concept for Spectre v2. Again, without speculative execution, this is perfectly fine code that will not cause any issues. Under speculative execution, the indirect jump inside `target()` will be mispredicted by the CPU due to earlier 'training', thus causing the `spectre_gadget()` to be run, which will speculatively load an arbitrary memory address.

This attack is also called 'Bounds Check Bypass', after the fact that the bounds check in the code is bypassed. This allows the attacker to read any arbitrary memory address by encoding its value in the buffer cache. Code snippets of the same format (bounds check + double dereference) are called *Spectre gadgets*.

Spectre v3

Spectre variant 3 is also known as Meltdown or 'Rogue Data Cache Load' [2], and is a special case of variant 1, with the addition of a severe hardware bug in older Intel CPUs. In short, the hardware bug entailed that for a `LOAD` instruction, memory access permissions would only be checked in the retire stage, rather than during execution itself. Concretely, this allowed a user process to transiently read kernel memory in addition to only user memory as in Spectre v1. Since most kernels have access to all physical memory through the direct map, this would allow an arbitrary user process to leak all system memory.

Spectre v2

The last variant of Spectre we will discuss is also called 'Branch Target Injection' [1], and differs from the other two variants by the method through which a transient window is opened. Instead of a direct (mispredicted) conditional statement, variant 2 tricks the CPU into mispredicting an indirect jump.

The sample code in Code Snippet 3.4.2 shows the various components needed for this attack. We again need some kind of 'gadget' code that performs a double pointer dereference to our liking. Now, we also need some kind of no-op function, and our Spectre target that will perform an indirect jump - i.e. a jump or function call that is not fixed in code, but dependent on some memory value. The attack is performed as follows:

1) The attacker will again prepare a special buffer suitable for a cache attack, which is flushed from cache.

2) The attacker will now repeatedly call `target()` with a valid array index `x` - so a value that is within bounds and will not cause any execution error. The callback `&spectre_gadget` is provided, which will 'train' the branch predictor that the indirect jump inside `target()` will point to `spectre_gadget`.

3) Now, the attacker will call `target()` again, but with an invalid array index `x` and callback `&no_op`.

4) At the branching point inside `target`, the CPU will mispredict that again `spectre_gadget()` is the jump destination, and will start speculative execution of that function with parameters `x` and `buffer`.

5) In this transient window, the double dereference inside `spectre_gadget` is performed again, but now with an invalid `x` that could point anywhere in memory.

6) Lastly, the CPU will resolve the misprediction and will rollback the state. However, as before, the transiently-loaded value is already encoded in the buffer.

3.4.2 Consequences

At first sight, it might be unclear why variants 1 and 2 are exactly an issue. If an attacker can allocate and prepare a buffer, call the gadget function, and then do a timing attack, why wouldn't the attacker just try to dereference the target pointer directly? Of course, in cases where an attacker controls the code being compiled, Spectre is a non-issue. There are, however, plenty of cases where Spectre does pose a significant risk, mainly when an attacker can control the flow of *other* code paths:

- A direct pointer dereference will trigger a page fault when the address is not mapped, which will crash your program with a segmentation fault. However, inside a transient window, a page fault is not yet triggered - the speculative load will fail and only when it is being retired, a page fault would be triggered. In the case of Spectre, that will never happen, since the operation is rolled back. This way, Spectre allows for (architecturally) undetected memory probing.
- Many modern applications run in sandboxes: Java code runs in the Java Virtual Machine, browser Javascript code is executed in isolated environments, etc. However, they all typically reside in the same process, and thus share a virtual address space. While the sandboxed code itself is not able to perform a direct memory accesses, it is possible to write code reproducing Spectre conditions, therefore being able to read the memory for all tabs in a browser, for example [1].
- Even outside sandboxes that provide a shared virtual address space, native Linux shared memory features pose a similar risk. Consider a third-party library (say, for cryptography) that can be communicated with to perform basic crypto tasks through a shared memory space. If this library contains an appropriate Spectre gadget, an attacker could leak all memory of this library - including potential encryption keys.
- Lastly, the kernel itself is a prime target for Spectre attacks, as a process shares its entire memory with the kernel. If we find appropriate Spectre gadgets in kernel code, then we could potentially leak all system memory, since the kernel has access to all physical memory. As it turns out, there are thousands of such gadgets in Linux already [13, 14].

3.5 Further Attacks

As mentioned before, Spectre was only the tip of the iceberg, and ever since its discovery there has been a continuous trickle of new speculative execution attacks [8]. While they are not the main subject of this thesis, we will still quickly discuss some of them to give an impression of the enormous scope of the problems with speculative execution, and the size of the attack surface that exists in modern CPUs.

Rogue In-Flight Data Load allows an attacker to leak memory across different virtual address spaces after a CPU will mistakenly speculatively use data values from other threads through the so-called *Line Fill Buffer* [4].

Fallout allows an attacker to leak arbitrary memory after a CPU will mistakenly *forward* a store in the store buffer, without properly checking whether the addresses match [7].

Load Value Injection leaks arbitrary internal CPU data caused by the fact that a failed `LOAD` (e.g. because of a page fault) will still transiently resolve with stale data from earlier operations [5].

Foreshadow allows an attacker to read data from the Intel ‘Software Guard eXtensions’, a separate ‘trusted execution environment’ on Intel CPUs [6].

ZombieLoad is another attack where a failing `LOAD` instruction allows the attacker to extract stale data from internal CPU buffers [3].

CrossTalk is one of the few cross-core attacks, allowing an attacker to leak data from other cores [9].

As is clear from this short overview, these attacks target nearly all microarchitectural details and features of CPUs as a way to access ‘forbidden’ data, which is then exfiltrated using the typical cache side channel, similar to what we saw before.

Chapter 4

Mitigating Speculative Execution Attacks

Now that we have identified the major issues surrounding speculative execution that plague modern CPUs, we should seek to solve them. How are these vulnerabilities typically resolved? Can they even be fixed fully? We will discuss common mitigation approaches and where they fall short, and then we discuss new, more comprehensive approaches. We will also introduce the contribution of this thesis: a new comprehensive approach to mitigating speculative execution vulnerabilities.

4.1 State of the Art

Unfortunately, there is no one-size-fits-all solution that will mitigate and fix all speculative execution vulnerabilities. So far, every newly found vulnerability is approached with a combination of microcode updates, spot mitigations, and hope that the vulnerability is not too severe.

After all, typically, speculative execution vulnerabilities are hardware-level bugs in the silicon of CPUs. Therefore, a proper fix would require updates to the hardware design, which can only be introduced on new products, leaving existing devices vulnerable. For this last category, then, we are left to mitigate the effects of the issue, rather than solving the issue itself.

4.1.1 Common Mitigations

Sometimes, partial mitigations can be applied in microcode. Microcode is yet another microarchitectural implementation detail of CPUs. Not all instructions in the x86 ISA are actually implemented directly in hardware: some are coded in microcode, which decomposes a single x86 instruction into multiple low-level so-called *micro-operations*. For some vulnerabilities, microcode updates can add additional cache flushes or buffer clears, to mitigate or solve a speculative execution vulnerability.

In most cases, however, developers resort to *spot mitigations*: fine-grained code injections all over codebases to dismantle or prevent the abuse of gadgets. For example, to mitigate the Spectre variants we discussed, we could insert a `FENCE` instruction in between both pointer dereferences. Such an instruction will halt all speculation and force all instructions currently in the reorder buffer to be resolved and retired before continuing execution. This will fix the vulnerability, since it closes the transient window, at the cost of some performance.

As is probably already clear: such spot mitigations are very *ad hoc* in nature, and it is extremely hard to comprehensively find and fix all imaginable gadgets [13]. Furthermore, their performance hit is significant: each mitigation approach can quickly add 30% but even up to 1,000% runtime overhead for common benchmarks [8, 10–12]. While newer hardware revisions seem to reduce the performance hit [11], the cumulative nature of these mitigations still pose major issues for overall system performance.

Consequently, the status quo is that existing systems are left vulnerable with only some high-risk code paths being ‘patched’ to mitigate underlying vulnerabilities, while taking a significant hit to performance. Since new vulnerabilities continue to be found, simply waiting for new hardware to be released is not a viable long-term strategy either.

4.1.2 Comprehensive Approaches

Due to the issues with the common route of spot mitigations, several attempts have been made at a more comprehensive approach to fix or mitigate *all* aspects of a particular vulnerability [15, 16, 19, 22]. Specifically, we will focus on virtual machine workloads, in particular in a cloud setting where we are the ‘service provider’.

In such a setting, we first note that we are not interested in preventing exploits *within* applications or virtual machines, as that is not our responsibility as service provider. We are interested in keeping our platform and customer data secure from attackers, so we care about attacks on the infrastructure, the hypervisor, or other virtual machines running on the platform.

Secret-Free Hypervisor

An early comprehensive approach was investigated and implemented by Microsoft as part of its *Secret-Free Hypervisor* [16]. The underlying approach sounds very simple: we make sure the hypervisor does not contain any secret data, and therefore no secret data can be leaked, making the hypervisor an uninteresting target.

At its core, this idea has some practical issues: a hypervisor will inevitably need to access some secret (guest machine) data, for example to handle interrupts and manage VM state. The approach in the SF hypervisor is to only temporarily make this data available when absolutely needed, by adding an *ephemeral map* to the page tables that is removed as soon as possible.

The paper reports a minimal overhead of at most around 10% compared to a baseline without traditional spot mitigations, but this result was only achieved after heavy optimisations in the implementation that complicate the design. Unfortunately, the implementations have not been open sourced, so it is hard to replicate the results. Of note is that according to the paper, the approach has been put into production use on Azure.

Quarantine

A completely different approach was previously taken by VUSec in the form of *Quarantine* [15]. This approach tries to eliminate speculative execution vulnerabilities by physically separating the hypervisor runtime from the virtual machine guests: the hypervisor kernel will run on an isolated, dedicated ‘kernel core’, separate from the guests.

The paper shows how most speculative execution vulnerabilities depend on same-core leaks: either the data is leaked from a core-internal buffer or cache, or the current virtual address space is used to leak an arbitrary address. Furthermore, cache side channels also depend on running on the same core. Consequently, by isolating all hypervisor work on a dedicated core, and having the VM guests run elsewhere with a minimal ‘kernel layer’ that only handles relaying data to and from the hypervisor core, we can eliminate the necessary conditions for an attack for nearly all known vulnerabilities.

The implementation is complicated by constantly switching cores, and scheduling issues limit the scalability of the approach. Overhead is reported to be in the range of 10-30%, but with strong core count dependency.

In some sense, this approach is orthogonal to the Secret-Free Hypervisor: whereas Quarantine provides *physical* isolation between the hypervisor and guests, SF Hypervisor provides *temporal* isolation by only mapping memory ephemerally.

4.2 *PageZero*

In this MSc thesis project, we propose *PageZero* as a hybrid approach combining the strong features of both Quarantine and SF Hypervisor, while cutting out the troublesome parts. In this approach, virtual machines are still physically isolated across cores, like in Quarantine. This will eliminate any cross-VM attacks that depend on core-local microarchitectural features. However, the kernel itself will not be physically isolated - it will instead be made 'practically secret-free': it will only contain secrets of the current virtual machine, but not of any others. This way, any speculative execution attack targeting the hypervisor from a specific virtual machine will only be able to reach uninteresting memory, or memory from itself, but crucially no memory from other virtual machines and other customers.

The trick is in how we make the kernel 'practically secret-free': we achieve this by emptying the page tables of a particular virtual memory space before starting the VM. This way, as soon as we (inevitably) hit a page fault, we know that the memory access was not speculative (since page faults are not triggered speculatively), allowing us to safely re-map the virtual memory address in the page tables. From now on, this address can be speculatively accessed, since we know it is 'safe' for the current VM to access.

4.2.1 Contributions

For this thesis, we will show a formal model and proof of the claim that emptying the page tables is sufficient to prevent most speculative execution attacks. We limit our scope to just Spectre v1, but the approach should easily extend to include other attacks as well. In particular, we will make the following contributions:

- We will present a formal model used for reasoning about (speculative) execution. This model contains a translation from a meta programming language into micro-operations, with a three-stage pipeline inspired by x86 design.
- We will present both sequential and speculative semantics for the formal model, where we make sure that page faults are not handled speculatively.
- We will show confluence for the speculative semantics.
- We will show consistency between both semantics, ensuring that the two semantics produce architecturally equivalent results.
- Finally, we will show how emptying page tables allows us to 'transfer' any property of the observed sequential memory accesses behaviour into the speculative realm.

This last result allows us to immediately conclude that a sequentially 'secure' program (for any definition of security) is also speculatively secure when emptying the page tables up front. This transfer approach is more general, allowing not only security claims to be ported into the speculative realm, but any other property of the set of accessed addresses (e.g. cache-friendliness) as well.

4.3 Attacker Model

For us to be able to properly analyse our approach, we should define an attacker model. This defines the capabilities of our attacker, and what we consider to be in-scope and out-of-scope for the project.

As mentioned above, we assume a hypervisor-guest model where the hypervisor kernel is the target of a speculative execution attack from within a VM. We assume the attacker can run arbitrary code within the VM, and is able to perform any speculative execution attack type. We assume the host to be fully up-to-date, such that the attacker is unable to use any software-level exploits. Lastly, we assume the attacker to not employ any unrelated hardware issues or vulnerabilities - in particular, we assume the CPU and RAM to not display any Byzantine behaviour under any circumstances.

Specifically for the proof work, we can assume an even stronger attacker: one with full control over all caches, and who can observe all memory activity - but not any memory data. We assume the attacker to have full control over the order of operations of the CPU, and the predictions made by the CPU - thus generalising any attack depending on prediction behaviour.

Chapter 5

Formal Model for Speculative Execution

In this chapter, we will introduce the formal machinery used in the rest of this thesis. This will consist of a mini meta programming language that we will use as our program input, and then a model and semantics for both sequential and speculative execution. These will be designed to closely follow real CPU behaviour, and to capture speculative execution attack vulnerabilities.

The foundations of this setup are inspired by the paper introducing *Blade*, a tool that is used to automatically eliminate speculative execution risks from code [22]. The work in this chapter shares the ideas on the three processor stages and the 'Just-in-Time' conversion of programming language code into low-level instructions, but we simplify the meta programming language, overhaul the low-level instructions to more closely match x86 micro-operations, and improve notation. On top of that, we add a virtual memory system, we model page tables and page faults, and therefore we rework nearly all the reduction rules. A comprehensive listing of all syntax can be found in Appendix A.

5.1 Meta Programming Language

We will consider a very small 'programming language' as input code in our system. The language supports variables, pointers, arrays, if-statements, and while-loops. While minimal, it is inspired by a language like C, and thus it is able to easily and comprehensively capture the behaviour we need for our work.

First, some basic variable and data type definitions. In here, for example, the set of variables is called `Var`, and if we come across an x in an expression later on, we know that we refer to some element of `Var`. We store arrays as a tuple of starting memory address and length, instead of the array containing all values directly. Keeping track of the length explicitly is not necessary, but simplifies some expressions down the line.

$$\begin{aligned}
\text{Variables }(\texttt{Var})\text{: } & x \\
\text{Integers }(\texttt{Num})\text{: } & n \in \mathbb{N} \\
\text{Booleans }(\texttt{Bool})\text{: } & b \in \{\texttt{true}, \texttt{false}\} \\
\text{Virtual Memory Addresses }(\texttt{Addr})\text{: } & a \in \mathbb{N} \\
\text{Arrays }(\texttt{Array})\text{: } & xs ::= \langle a, n \rangle \text{ (base address and length)} \\
\text{Values }(\texttt{Val})\text{: } & v ::= n \mid b \mid a \mid xs
\end{aligned}$$

Next, we introduce the language itself. A program is treated as a sequence of *commands*, called a *command stack*.

$$\begin{aligned}
\text{Expressions } (\texttt{Expr})\text{:}\ & e ::= v \mid x \mid e + e \mid e < e \mid e \otimes e \mid e \,?\, e : e \mid \mathsf{len}(e) \mid \mathsf{base}(e) \\
\text{Right-hand Sides:}\ & r ::= e \mid {*}\,e \mid xs\,[\,e\,] \\
\text{Commands } (\texttt{Comm})\text{:}\ & c ::= \texttt{skip} \mid x \mathrel{\texttt{:=}} r \mid \texttt{*}\,e \mathrel{\texttt{:=}} e \mid xs\texttt{[}e\texttt{]} \mathrel{\texttt{:=}} e \\
& \quad \mid \texttt{if}\ e\ \texttt{then}\ cs\ \texttt{else}\ cs \mid \texttt{while}\ e\ \texttt{do}\ cs \\
\text{Command Stacks } (\texttt{[Comm]})\text{:}\ & cs ::= c : cs \mid []
\end{aligned}$$

For expressions, we support simple sums, comparisons, bitwise XOR, a simple (atomic) ternary value, and array metadata accessors for the length and base address. Right-hand sides can be expressions, pointer dereferences, or array accesses. In commands, it is possible to assign to variables, pointer locations, or array elements. Commands are always written using monospaced fonts.

Notation 5.1.1 *Types for sequences of another type are written as* `[Foo]`*, and elements of such a type have 'plural' variable names (e.g. xs), similar to notation in Haskell. Constructing such sequences is done in a similar fashion as well, where $[]$ denotes the empty sequence, $x : xs$ adds a single element x to the front of the sequence, and $xs_1 \mathbin{+\!\!+} xs_2$ concatenates two sequences. We allow writing array literals like $[x, y]$.* ⌟

In this notation, we can construct a Spectre v1 gadget similar to Code Snippet 3.4.1 as follows:

$$\texttt{if}\ i > \mathsf{len}(xs)\ \texttt{then}\ [\texttt{skip}]\ \texttt{else}\ [x \mathrel{\texttt{:=}} xs\texttt{[}i\texttt{]}, y \mathrel{\texttt{:=}} ys\texttt{[}x\texttt{]}], \tag{5.1.1}$$

where we renamed some variables to be in line with the notation conventions.

5.2 Sequential Semantics

First, we will construct a sequential semantics, modelling the traditional in-order non-pipelined execution model. We will construct a notion of system state, and a reduction relation between such states based on the command stack.

5.2.1 System State

For our system state, we will consider a processor with a number of registers (corresponding to variables), a physical memory store, and a set of page tables to map between virtual and physical addresses.

$$\begin{aligned}
\text{Register Maps } (\texttt{Reg})\text{:}\ & \rho \in \texttt{Var} \to \texttt{Val} \\
\text{Physical Memory Addresses } (\texttt{PAddr})\text{:}\ & pa \in \mathbb{N} \\
\text{Page Tables:}\ & \tau \in \texttt{Addr} \to (\texttt{PAddr} \cup \bot) \\
\text{True Page Mapping:}\ & M \in \texttt{Addr} \to \texttt{PAddr} \\
\text{Physical Memory Stores:}\ & \mu \in \texttt{PAddr} \to \texttt{Val} \\
\text{Sequential Program State } (\texttt{SeqState})\text{:}\ & S ::= \langle cs, \rho, \tau, \mu \rangle, \ \text{where } \tau \subseteq M
\end{aligned}$$

In case a virtual address a is not mapped, $\tau(a) = \bot$. We distinguish between the set of page tables τ, that we will use and adapt during execution, and a 'ground truth' mapping M. The latter is used as an abstraction of some kernel page fault handler that will allocate and manage physical memory for us. In our semantics, the page fault handler will be abstracted away by using entries from M to update τ. For the system state to be valid, we require τ to conform to this ground truth mapping M.

5.2.2 Expression Evaluation

Before we can define the reduction relation, we should establish how expressions of type `Expr` can be evaluated. Since expressions can use variables, but cannot use pointers, we only need the register map ρ to evaluate expressions. We define the evaluation function $[\![\cdot]\!]^{\cdot} \in (\texttt{Expr} \times \texttt{Reg}) \to \texttt{Val}$ case by case on `Expr`:

$$\begin{aligned}
[\![v]\!]^{\rho} &= v \\
[\![x]\!]^{\rho} &= \rho(x) \\
[\![e_1 + e_2]\!]^{\rho} &= [\![e_1]\!]^{\rho} + [\![e_2]\!]^{\rho} \\
[\![e_1 < e_2]\!]^{\rho} &= [\![e_1]\!]^{\rho} < [\![e_2]\!]^{\rho} \\
[\![e_1 \otimes e_2]\!]^{\rho} &= [\![e_1]\!]^{\rho} \otimes [\![e_2]\!]^{\rho} \\
[\![e_b \mathbin{?} e_1 : e_2]\!]^{\rho} &= \begin{cases} [\![e_1]\!]^{\rho} & \text{if } [\![e_b]\!]^{\rho} = \texttt{true} \\ [\![e_2]\!]^{\rho} & \text{if } [\![e_b]\!]^{\rho} = \texttt{false} \end{cases} \\
[\![\mathsf{base}(e)]\!]^{\rho} &= \{ a \quad \text{if } [\![e]\!]^{\rho} = \langle a, \_ \rangle \\
[\![\mathsf{len}(e)]\!]^{\rho} &= \{ n \quad \text{if } [\![e]\!]^{\rho} = \langle \_, n \rangle
\end{aligned}$$

For missing cases in the array metadata, so when base or len are used on non-array variables, the function is undefined.

5.2.3 Reduction Relation

We implement the sequential semantics as a reduction relation on system states, where each action can possibly emit some *observation* for the attacker. For now, we only consider a memory access observation - for the speculative semantics, we will add more.

$$\begin{aligned}
\text{Observations: } & o ::= \textbf{touch}(pa) \\
\text{Observation Trace: } & O ::= o : O \mid []
\end{aligned}$$

The (small-step) semantics itself is quite straightforward as a reduction relation on `SeqState`, denoted with a double arrow: $S \Longrightarrow_O S'$. We will drop the observation trace subscript when no observations are emitted, to simplify notation. Since sequential execution behaves like normal computation, we are not concerned with micro-architectural CPU implementation details.

$$\textsc{Seq-Skip}\quad \frac{}{\langle \texttt{skip} : cs, \rho, \tau, \mu \rangle \Longrightarrow \langle cs, \rho, \tau, \mu \rangle}$$

$$\textsc{Seq-Assign}\quad \frac{c = x \mathbin{\texttt{:=}} e \qquad v = [\![e]\!]^{\rho} \qquad \rho' = \rho[x \mapsto v]}{\langle c : cs, \rho, \tau, \mu \rangle \Longrightarrow \langle cs, \rho', \tau, \mu \rangle}$$

The rule for skip is very straightforward, and for an assignment we simply evaluate the expression and assign the result to the correct variable. In case an expression is invalid (say, using base on a non-array variable), execution will get stuck, since the antecedent requiring evaluation of the expression cannot be satisfied.

$$\textsc{Seq-If-Else}\quad \frac{c = \texttt{if}\ e\ \texttt{then}\ cs_{\texttt{true}}\ \texttt{else}\ cs_{\texttt{false}} \qquad b = [\![e]\!]^{\rho}}{\langle c : cs, \rho, \tau, \mu \rangle \Longrightarrow \langle cs_b \mathbin{+\!\!+} cs, \rho, \tau, \mu \rangle}$$

$$\textsc{Seq-While}\quad \frac{c = \texttt{while}\ e\ \texttt{do}\ cs_l \qquad b = [\![e]\!]^{\rho} \qquad cs_{\texttt{true}} = cs_l \mathbin{+\!\!+} [c] \qquad cs_{\texttt{false}} = []}{\langle c : cs, \rho, \tau, \mu \rangle \Longrightarrow \langle cs_b \mathbin{+\!\!+} cs, \rho, \tau, \mu \rangle}$$

For both the `if` and `while` commands, we evaluate the guard expression to a boolean b. Based on its value, we replace the command by the correct branch.

$$\textsc{Seq-Pointer-Read-Present}\quad \frac{\begin{array}{c} c = x \mathbin{\texttt{:=}} \texttt{*}\,e \qquad a = [\![e]\!]^{\rho} \qquad \tau(a) \neq \bot \qquad pa = \tau(a) \\ v = \mu(pa) \qquad \rho' = \rho[x \mapsto v] \end{array}}{\langle c : cs, \rho, \tau, \mu\rangle \Longrightarrow_{[\mathbf{touch}(pa)]} \langle cs, \rho', \tau, \mu\rangle}$$

$$\textsc{Seq-Pointer-Read-Miss}\quad \frac{\begin{array}{c} c = x \mathbin{\texttt{:=}} \texttt{*}\,e \qquad a = [\![e]\!]^{\rho} \qquad \tau(a) = \bot \qquad pa = M(a) \\ v = \mu(pa) \qquad \rho' = \rho[x \mapsto v] \qquad \tau' = \tau[a \mapsto pa] \end{array}}{\langle c : cs, \rho, \tau, \mu\rangle \Longrightarrow_{[\mathbf{touch}(pa)]} \langle cs, \rho', \tau', \mu\rangle}$$

For a pointer read, we consider two cases: either the address a is mapped in the page tables, or it is not. If it is mapped, we can simply resolve the physical address $pa = \tau(a)$, and read the memory $\mu(pa)$, which we store in the right variable. For a miss, we will use the 'ground truth' page map M to resolve the correct address pa and update the page tables in addition to loading the right value. This mimics the behaviour of a real page fault, which is architecturally transparent to the program. In both cases we emit a $\mathbf{touch}(pa)$ observation, since the attacker can observe our memory access.

$$\textsc{Seq-Pointer-Write-Present}\quad \frac{\begin{array}{c} c = \texttt{*}\,e_a \mathbin{\texttt{:=}} e_v \qquad a = [\![e_a]\!]^{\rho} \qquad \tau(a) \neq \bot \qquad pa = \tau(a) \\ v = [\![e_v]\!]^{\rho} \qquad \mu' = \mu[pa \mapsto v] \end{array}}{\langle c : cs, \rho, \tau, \mu\rangle \Longrightarrow_{[\mathbf{touch}(pa)]} \langle cs, \rho, \tau, \mu'\rangle}$$

$$\textsc{Seq-Pointer-Write-Miss}\quad \frac{\begin{array}{c} c = \texttt{*}\,e_a \mathbin{\texttt{:=}} e_v \qquad a = [\![e_a]\!]^{\rho} \qquad \tau(a) = \bot \qquad pa = M(a) \\ v = [\![e_v]\!]^{\rho} \qquad \mu' = \mu[pa \mapsto v] \qquad \tau' = \tau[a \mapsto pa] \end{array}}{\langle c : cs, \rho, \tau, \mu\rangle \Longrightarrow_{[\mathbf{touch}(pa)]} \langle cs, \rho, \tau', \mu'\rangle}$$

For writes, we have equivalent cases as for pointer reads: in case of a miss we transparently resolve the page fault by updating the page tables, and in both cases we update the physical memory and emit the appropriate observation.

Next, we repeat the same four rules but now for arrays instead of pointers. The behaviour is identical, except for using an additional expression to resolve the memory address for the array value. We could have defined these rules differently, where they merely 'translate' the array operation to an equivalent pointer operation, but then we lose the one-to-one link between reduction steps and command execution, which we will need later during our proofs.

$$\textsc{Seq-Array-Read-Present}\quad \frac{\begin{array}{c} c = x \mathbin{\texttt{:=}} xs\texttt{[}e\texttt{]} \qquad e_i = \mathsf{base}(xs) + e \qquad a = [\![e_i]\!]^{\rho} \\ \tau(a) \neq \bot \qquad pa = \tau(a) \qquad v = \mu(pa) \qquad \rho' = \rho[x \mapsto v] \end{array}}{\langle c : cs, \rho, \tau, \mu\rangle \Longrightarrow_{[\mathbf{touch}(pa)]} \langle cs, \rho', \tau, \mu\rangle}$$

$$\textsc{Seq-Array-Read-Miss}\quad \frac{\begin{array}{c} c = x \mathbin{\texttt{:=}} xs\texttt{[}e\texttt{]} \qquad e_i = \mathsf{base}(xs) + e \qquad a = [\![e_i]\!]^{\rho} \\ \tau(a) = \bot \qquad pa = M(a) \qquad v = \mu(pa) \qquad \rho' = \rho[x \mapsto v] \qquad \tau' = \tau[a \mapsto pa] \end{array}}{\langle c : cs, \rho, \tau, \mu\rangle \Longrightarrow_{[\mathbf{touch}(pa)]} \langle cs, \rho', \tau', \mu\rangle}$$

$$\frac{\begin{array}{c}c = xs\texttt{[}e\texttt{] :=} e_v \qquad e_i = \mathsf{base}(xs) + e \qquad a = [\![e_i]\!]^\rho \qquad \tau(a) \neq \bot \\ pa = \tau(a) \qquad v = [\![e_v]\!]^\rho \qquad \mu' = \mu[pa \mapsto v]\end{array}}{\langle c : cs, \rho, \tau, \mu\rangle \Longrightarrow_{[\mathbf{touch}(pa)]} \langle cs, \rho, \tau, \mu'\rangle}\ \textsc{Seq-Array-Write-Present}$$

$$\frac{\begin{array}{c}c = xs\texttt{[}e\texttt{] :=} e_v \qquad e_i = \mathsf{base}(xs) + e \qquad a = [\![e_i]\!]^\rho \qquad \tau(a) = \bot \\ pa = M(a) \qquad v = [\![e_v]\!]^\rho \qquad \mu' = \mu[pa \mapsto v] \qquad \tau' = \tau[a \mapsto pa]\end{array}}{\langle c : cs, \rho, \tau, \mu\rangle \Longrightarrow_{[\mathbf{touch}(pa)]} \langle cs, \rho, \tau', \mu'\rangle}\ \textsc{Seq-Array-Write-Miss}$$

5.2.4 Big-Step Semantics

To simplify execution analysis, we also introduce a big-step semantics. For the purposes of this work, the big-step semantics is a simple transitive reflexive closure of the small-step reduction relation. In contrast to more typical big-step semantics, we do not require termination - which we will define more formally later. The relation is $S \Downarrow_O S'$, with the following inductive definition:

$$\frac{}{\langle cs, \rho, \tau, \mu\rangle \Downarrow_{[]} \langle cs, \rho, \tau, \mu\rangle}\ \textsc{Seq-Done}$$

$$\frac{\langle cs, \rho, \tau, \mu\rangle \Longrightarrow_{O_1} \langle cs', \rho', \tau', \mu'\rangle \qquad \langle cs', \rho', \tau', \mu'\rangle \Downarrow_{O_2} \langle cs'', \rho'', \tau'', \mu''\rangle}{\langle cs, \rho, \tau, \mu\rangle \Downarrow_{O_1 +\!\!+ O_2} \langle cs'', \rho'', \tau'', \mu''\rangle}\ \textsc{Seq-Step}$$

5.3 Speculative Semantics

With our sequential definition in mind, we can look at how we extend and adapt the model to support speculative execution. Our final objective is, of course, to keep both semantics consistent: execution results should be comparable between the two. We will formally prove this in Section 6.2.

5.3.1 Staged Pipeline Model

As mentioned before, we will model three processor 'stages', mimicking real behaviour: fetching, execution, and retiring. The fetch stage will convert a command to an equivalent low-level instruction - analogous to micro-operations in a real CPU. These instructions are stored in a queue, taking the role of the reorder buffer. The execute stage will perform all computation: resolving expressions and reading from memory, converting an instruction into a 'final' form that is then ready to be retired. Like in a real CPU, the retire step will make the effect of an instruction final and architectural, which means updating registers and writing to memory.

$$\begin{array}{rl}\text{Instruction Labels:} & \ell \\ \text{Instruction Trace:} & \ell s ::= \ell : \ell s \mid [] \\ \text{Instructions } (\texttt{Instr})\text{:} & i ::= \mathsf{NOP} \mid \mathsf{MOV}\langle x, e\rangle \mid \mathsf{LOAD}\langle x, e, pv, cs\rangle \mid \mathsf{STORE}\langle e_a, e_v, cs\rangle \\ & \quad \mid \mathsf{ASSERT}\langle e, b, cs\rangle \mid \mathsf{FAULT}\langle a, i, cs\rangle \\ \text{Reorder Buffers } (\texttt{[Instr]})\text{:} & is ::= i_\ell : is \mid []\end{array}$$

We have six different instructions, corresponding to no-op (NOP), simple variable stores (MOV), memory read (LOAD) and write (STORE), and finally an assertion (ASSERT) and a page fault (FAULT).

Inside a sequence of instructions, modelling the reorder buffer, we label all instructions. This allows us to track the ‘flow’ of instructions through their various forms from fetch to retire.

The `LOAD` instruction carries the variable to store the result into, an expression to compute the (virtual) memory address, a predicted value that is populated upon a speculative load or store forward, and a command stack to rollback to in case the prediction was incorrect or when the address was not mapped.

Similarly, the `STORE` carries expressions for the address to store to, and the value to store, as well as a command stack as rollback point for a page fault.

The assertion is used at a branch point in code, such as an `if` or `while`. In our speculative semantics, we will make a prediction for this expression, opening a transient window. If at some later point, after evaluating e, we find that the prediction was incorrect, we will rollback and reset the command stack to the carried cs.

The page fault is used to implement page faults: when a memory load or store fails, we substitute the faulting instruction for a `FAULT`, which carries the faulted address (to be mapped) and instruction, as well as the command stack to reset to during the rollback.

5.3.2 Out-of-Order Execution

Speculative execution in our model is out-of-order, so we need to a way to determine when we will fetch, retire, or execute. We do so using *directives*, which we define for each possible ‘action’ in the model. Later on, every reduction rule will only be valid for a specific directive. In our attacker model, the attacker has full control over these directives. Sequences of directives are called *schedules*.

$$\begin{aligned} \text{Directives: } d &::= \textbf{fetch} \mid \textbf{fetch}\, b \mid \textbf{exec}\, n \mid \textbf{retire} \mid \textbf{map} \\ \text{Schedule: } D &::= d : D \mid [] \end{aligned}$$

Since we support out-of-order execution, the **exec** directive carries an argument n deciding which instruction index in the reorder buffer should be executed. Furthermore, we have two types of fetches: either with or without a speculative prediction for branch points.

When executing an instruction n, we will (try to) include the effect of all earlier instructions in the reorder buffer is into the evaluation of expressions. Just like in real CPUs, if instructions have been resolved already, but have not yet been retired, their result is already used in subsequent computations.

$$\begin{aligned} \text{Predicted Values } (\texttt{PVal})\text{: } pv &::= v \mid \bot \\ \text{Predicted Register Maps } (\texttt{PReg})\text{: } \tilde{\rho} &\in \texttt{Var} \to \texttt{PVal} \end{aligned}$$

We introduce two new predictive types: predicted values and predicted register maps. When the value is $\bot$, the result is unavailable. For some instruction at index n in the reorder buffer, then, we need to ‘propagate’ the effect of previous instructions into the register map, which results in the predicted register map. Any expression evaluation on this state will then produce a predicted value.

To compute the predicted register map, we introduce a function $\varphi \in (\texttt{PReg} \times \texttt{[Instr]}) \to \texttt{PReg}$, which will ‘simulate’ the effect of the instructions in the given reorder buffer is. This can only be determined for instructions that have been computed already - if a final result is not known, we clear the value for that register. We define φ case by case on `Instr`:

$$\begin{aligned}
&\varphi(\tilde{\rho}, []) &&= \tilde{\rho}\\
&\varphi(\tilde{\rho}, \mathsf{MOV}\langle x, v\rangle : is) &&= \varphi(\tilde{\rho}[x \mapsto v], is)\\
&\varphi(\tilde{\rho}, \mathsf{MOV}\langle x, e\rangle : is) &&= \varphi(\tilde{\rho}[x \mapsto \bot], is)\\
&\varphi(\tilde{\rho}, \mathsf{LOAD}\langle x, e, pv, cs\rangle : is) &&= \varphi(\tilde{\rho}[x \mapsto pv], is)\\
&\varphi(\tilde{\rho}, i : is) &&= \varphi(\tilde{\rho}, is)
\end{aligned}$$

In addition, we need to update our evaluation function $[\![\cdot]\!]^{\cdot}$ to deal with predicted values. We update our function to be in $(\texttt{Expr} \times \texttt{PReg}) \to \texttt{PVal}$ now, which is backwards compatible with our previous definition: any cases where $\bot$ is now produced would cause computation to get stuck in the sequential model, and we will only consider programs where this does not happen. These are the updated rules:

$$[\![e_b \,?\, e_1 : e_2]\!]^{\tilde{\rho}} = \begin{cases} [\![e_1]\!]^{\tilde{\rho}} & \text{if } [\![e_b]\!]^{\tilde{\rho}} = \texttt{true}\\ [\![e_2]\!]^{\tilde{\rho}} & \text{if } [\![e_b]\!]^{\tilde{\rho}} = \texttt{false}\\ \bot & \text{otherwise} \end{cases}$$

$$[\![\mathsf{base}(e)]\!]^{\tilde{\rho}} = \begin{cases} a & \text{if } [\![e]\!]^{\tilde{\rho}} = \langle a, \_\rangle\\ \bot & \text{otherwise} \end{cases}$$

$$[\![\mathsf{len}(e)]\!]^{\tilde{\rho}} = \begin{cases} n & \text{if } [\![e]\!]^{\tilde{\rho}} = \langle \_, n\rangle\\ \bot & \text{otherwise} \end{cases}$$

Definition 5.3.1 *An expression e is* calculable *under predicted register map $\tilde{\rho}$ when for all variables x in e, $\tilde{\rho}(x) \neq \bot$.* ⌟

5.3.3 Configurations

To make the difference between sequential and speculative execution clearer, we will call the speculative counterpart of the state a *configuration*. A speculative configuration will carry a reorder buffer, in addition to the command stack, register map, memory map, and page tables of the sequential state.

$$\text{Observations: } o ::= \textbf{touch}(pa) \mid \textbf{rollback}(\ell s)$$
$$\text{Configurations } (\texttt{Conf})\text{: } C ::= \langle is, cs, \rho, \tau, \mu\rangle, \text{ where } \tau \subseteq M$$

We introduce a new observation, the **rollback**. This observation will be emitted upon a rollback, for example after a misprediction or a page fault. It carries the trace of instruction labels that were rolled back. Again, we require the page tables in a configuration to be consistent with the ground truth memory map.

For speculative execution, the semantics will always have a single arrow, for both the small-step and big-step semantics. The small-step semantics is defined in terms of a reduction relation on `Conf`: $C \xrightarrow{d}_O C'$. In words: the system in configuration C transitions to configuration C' upon directive d, emitting observation trace O. Again, to simplify notation, empty observation traces are dropped.

5.3.4 Fetch Directive Rules

All reduction rules for the **fetch** directive convert the first command on the stack into an equivalent instruction at the end of the reorder buffer, adorning it with a fresh label ℓ. By design, fetching is done in-order, just like on real CPUs where only the execute stage is out-of-order.

Fetch-Skip

$$\frac{\text{fresh}(\ell)}{\langle is, \texttt{skip} : cs, \rho, \tau, \mu\rangle \xrightarrow{\textbf{fetch}} \langle is \mathbin{+\!\!+} [\mathsf{NOP}_\ell], cs, \rho, \tau, \mu\rangle}$$

Fetch-Assign

$$\frac{\text{fresh}(\ell)}{\langle is, x \mathrel{\texttt{:=}} e : cs, \rho, \tau, \mu\rangle \xrightarrow{\textbf{fetch}} \langle is \mathbin{+\!\!+} [\mathsf{MOV}\langle x, e\rangle_\ell], cs, \rho, \tau, \mu\rangle}$$

Fetch-Pointer-Load

$$\frac{c = x \mathrel{\texttt{:=}} \texttt{*}e \qquad i = \mathsf{LOAD}\langle x, e, \bot, cs\rangle \qquad \text{fresh}(\ell)}{\langle is, c : cs, \rho, \tau, \mu\rangle \xrightarrow{\textbf{fetch}} \langle is \mathbin{+\!\!+} [i_\ell], cs, \rho, \tau, \mu\rangle}$$

The predicted value in the LOAD instruction is only populated later in one of the **exec** rules, and so for now the loaded value is unknown: $\bot$. Our rollback point is the command stack excluding the current command c, since that command is now translated to instruction ℓ which is not thrown away in a rollback later.

Fetch-Pointer-Store

$$\frac{c = \texttt{*}e_a \mathrel{\texttt{:=}} e_v \qquad i = \mathsf{STORE}\langle e_a, e_v, cs\rangle \qquad \text{fresh}(\ell)}{\langle is, c : cs, \rho, \tau, \mu\rangle \xrightarrow{\textbf{fetch}} \langle is \mathbin{+\!\!+} [i_\ell], cs, \rho, \tau, \mu\rangle}$$

For both pointer operations, unlike in the sequential case, we do not care about whether the address is mapped in the page tables. In fact, we do not even know the address yet. The address is only computed later, as part of the **exec** rules, and only then a page fault might be triggered.

Fetch-Array-Load

$$\frac{c = x \mathrel{\texttt{:=}} xs\,\texttt{[}\,e\,\texttt{]} \qquad e_i = \mathsf{base}(xs) + e \qquad i = \mathsf{LOAD}\langle x, e_i, \bot, cs\rangle \qquad \text{fresh}(\ell)}{\langle is, c : cs, \rho, \tau, \mu\rangle \xrightarrow{\textbf{fetch}} \langle is \mathbin{+\!\!+} [i_\ell], cs, \rho, \tau, \mu\rangle}$$

Fetch-Array-Store

$$\frac{c = xs\texttt{[}e\texttt{]} \mathrel{\texttt{:=}} e_v \qquad e_i = \mathsf{base}(xs) + e \qquad i = \mathsf{STORE}\langle e_i, e_v, cs\rangle \qquad \text{fresh}(\ell)}{\langle is, c : cs, \rho, \tau, \mu\rangle \xrightarrow{\textbf{fetch}} \langle is \mathbin{+\!\!+} [i_\ell], cs, \rho, \tau, \mu\rangle}$$

The array operations are similar to the pointer operations again, with a specific expression for the memory address. Again, we rely on the **exec** rules to compute the actual addresses.

Fetch-If-Predict-True

$$\frac{c = \texttt{if}\ e\ \texttt{then}\ cs_1\ \texttt{else}\ cs_2 \qquad i = \mathsf{ASSERT}\langle e, \texttt{true}, cs_2 \mathbin{+\!\!+} cs\rangle \qquad \text{fresh}(\ell)}{\langle is, c : cs, \rho, \tau, \mu\rangle \xrightarrow{\textbf{fetch}\ \texttt{true}} \langle is \mathbin{+\!\!+} [i_\ell], cs_1 \mathbin{+\!\!+} cs, \rho, \tau, \mu\rangle}$$

Fetch-If-Predict-False

$$\frac{c = \texttt{if}\ e\ \texttt{then}\ cs_1\ \texttt{else}\ cs_2 \qquad i = \mathsf{ASSERT}\langle e, \texttt{false}, cs_1 \mathbin{+\!\!+} cs\rangle \qquad \text{fresh}(\ell)}{\langle is, c : cs, \rho, \tau, \mu\rangle \xrightarrow{\textbf{fetch}\ \texttt{false}} \langle is \mathbin{+\!\!+} [i_\ell], cs_2 \mathbin{+\!\!+} cs, \rho, \tau, \mu\rangle}$$

$$\textsc{Fetch-While-Predict-True}\quad \frac{c = \texttt{while}\ e\ \texttt{do}\ cs_l \qquad i = \mathsf{ASSERT}\langle e, \texttt{true}, cs\rangle \qquad \text{fresh}(\ell)}{\langle is, c : cs, \rho, \tau, \mu\rangle \xrightarrow{\textbf{fetch}\ \texttt{true}} \langle is \mathbin{+\!\!+} [i_\ell], cs_l \mathbin{+\!\!+} c : cs, \rho, \tau, \mu\rangle}$$

$$\textsc{Fetch-While-Predict-False}\quad \frac{c = \texttt{while}\ e\ \texttt{do}\ cs_l \qquad i = \mathsf{ASSERT}\langle e, \texttt{false}, cs_l \mathbin{+\!\!+} c : cs\rangle \qquad \text{fresh}(\ell)}{\langle is, c : cs, \rho, \tau, \mu\rangle \xrightarrow{\textbf{fetch}\ \texttt{false}} \langle is \mathbin{+\!\!+} [i_\ell], cs, \rho, \tau, \mu\rangle}$$

For the `if` and `while` constructs, we require the **fetch** to come with a branch prediction. While normally the CPU has dedicated branch prediction circuitry, in our model the attacker can make the predictions to their liking. This way, we capture any possible branch predictor behaviour, including any possible attacker manipulation of such behaviour. During the fetch, we assume the prediction to be correct, and modify the command stack accordingly. In case the prediction is incorrect, the ASSERT instruction will carry the 'alternative' command stack that is to be restored upon a rollback.

5.3.5 Execute Directive Rules

Since the **exec** directive takes an argument n, we will have to split the reorder buffer accordingly for every reduction rule, to make sure we can work with this nth instruction. To ease notation, we will introduce an auxiliary relation $\xrightarrow{(\tilde{\rho},\tau,\mu)}\!\!\!\!\twoheadrightarrow_O$ that accepts a tuple $\langle is_1, i, is_2, cs\rangle$ where the instruction-to-execute is already isolated, and will return a new state tuple $\langle is', cs'\rangle$ with the full new reorder buffer and command stacks. This relation is passed some read-only data in the form of the predicted register map $\tilde{\rho}$, the current page tables τ, and the memory map μ, and will emit an observation trace O. Again, when no observation is emitted, the subscript is dropped for brevity.

$$\textsc{Execute-}n\textsc{th-Instruction}\quad \frac{|is_1| = n - 1 \qquad \tilde{\rho} = \varphi(\rho, is_1) \qquad \langle is_1, i_\ell, is_2, cs\rangle \xrightarrow{(\tilde{\rho},\tau,\mu)}\!\!\!\!\twoheadrightarrow_O \langle is', cs'\rangle}{\langle is_1 \mathbin{+\!\!+} [i_\ell] \mathbin{+\!\!+} is_2, cs, \rho, \tau, \mu\rangle \xrightarrow{\textbf{exec}\ n}_O \langle is', cs', \rho, \tau, \mu\rangle}$$

We use the previously-defined 'simulation' function φ to propagate the effect of computed-but-not-yet-retired instructions in is_1. Furthermore, this rule shows that none of the execute rules will change the architectural state (ρ, τ, or μ), since the execute stage is a microarchitectural implementation detail.

$$\textsc{Execute-Assign}\quad \frac{i = \mathsf{MOV}\langle x, e\rangle \qquad v = [\![e]\!]^{\tilde{\rho}} \qquad i' = \mathsf{MOV}\langle x, v\rangle}{\langle is_1, i_\ell, is_2, cs\rangle \xrightarrow{(\tilde{\rho},\tau,\mu)}\!\!\!\!\twoheadrightarrow \langle is_1 \mathbin{+\!\!+} [i'_\ell] \mathbin{+\!\!+} is_2, cs\rangle}$$

Executing a MOV only requires us to compute the expression, but we do not store this result yet - that is the job of the retire stage, which will make our work architectural.

Branch Prediction

$$\textsc{Execute-Assert-Success}\quad \frac{i = \mathsf{ASSERT}\langle e, b, cs'\rangle \qquad [\![e]\!]^{\tilde{\rho}} = b}{\langle is_1, i_\ell, is_2, cs\rangle \xrightarrow{(\tilde{\rho},\tau,\mu)}\!\!\!\!\twoheadrightarrow \langle is_1 \mathbin{+\!\!+} [\mathsf{NOP}_\ell] \mathbin{+\!\!+} is_2, cs\rangle}$$

The inserted NOP might seem redundant here, but it is used to still have an instruction to retire later. This prevents instructions from magically 'disappearing'. It allows us to maintain a one-to-one correspondence between instructions fetched and instructions retired, which we will need for our proofs later.

$$\textsc{Execute-Assert-Fail}\quad \frac{i = \mathsf{ASSERT}\langle e, b, cs' \rangle \qquad [\![e]\!]^{\tilde{\rho}} = b' \qquad b \neq b'}{\langle is_1, i_\ell, is_2, cs \rangle \xrightarrow{(\tilde{\rho},\tau,\mu)}\!\!\!\!\rightarrow_{[\mathbf{rollback}(\ell s_2)]} \langle is_1 + [\mathsf{NOP}_\ell], cs' \rangle}$$

We again insert the NOP instruction for the previously mentioned reasons. Furthermore, we emit a **rollback** observation since our prediction turned out to be incorrect. It carries the remainder of the reorder buffer (that is now being discarded), and the command stack is reset to the point just after fetching the assertion (but now with the correct branch).

Memory Loads

We now introduce a number of rules dealing with LOAD instructions. There are two rules that will populate the predicted value, either from a store forward, or by speculatively loading a memory address. Then, there are three rules for verifying this speculative operation: one for triggering a page fault in case the address is not in the page tables, and two for transforming the load into a MOV instruction - thus finalising the memory load. Together, these rules mimic the real-life behaviour of CPUs that is relevant for this thesis.

$$\textsc{Execute-Load-Predict}\quad \frac{\begin{array}{c} i = \mathsf{LOAD}\langle x, e, \bot, cs' \rangle \qquad a = [\![e]\!]^{\tilde{\rho}} \qquad \mathsf{STORE}\langle a, \_, \_ \rangle \notin is_1 \\ pa = \tau(a) \qquad pa \neq \bot \qquad i' = \mathsf{LOAD}\langle x, a, \mu(pa), cs' \rangle \end{array}}{\langle is_1, i_\ell, is_2, cs \rangle \xrightarrow{(\tilde{\rho},\tau,\mu)}\!\!\!\!\rightarrow_{[\mathbf{touch}(pa)]} \langle is_1 + [i'_\ell] + is_2, cs \rangle}$$

Since we allow out-of-order execution, our LOAD instruction might be preceded by a STORE to the same address. This rule will speculatively load a value, based on a 'best effort' determination of whether there is a preceding store to the same address. This rule can only be used when there is no known store to the same address - however, there might be store instructions that have uncomputed address expressions that will point to the same address, or store instructions that write to a different virtual address that maps to the same physical address. Therefore, this is a speculative load.

This speculative value can then (speculatively) be used in later instructions through the propagation function φ, as introduced before. Later on, we will confirm our prediction: if it was incorrect, we perform a rollback.

Note that we only perform a speculative load on addresses that are mapped in the page tables τ, as otherwise there is nothing to load. This rule could be modified to load Byzantine data in case of unmapped addresses, which would model vulnerabilities like RIDL [4], LVI [5], and ZombieLoad [3].

$$\textsc{Execute-Load-Forward}\quad \frac{i = \mathsf{LOAD}\langle x, e, \bot, cs' \rangle \qquad a = [\![e]\!]^{\tilde{\rho}} \qquad \mathsf{STORE}\langle a, v, \_ \rangle \in is_1 \qquad i' = \mathsf{LOAD}\langle x, a, v, cs' \rangle}{\langle is_1, i_\ell, is_2, cs \rangle \xrightarrow{(\tilde{\rho},\tau,\mu)}\!\!\!\!\rightarrow \langle is_1 + [i'_\ell] + is_2, cs \rangle}$$

When an already-computed store for the same address is pending before the current load, we can speculatively forward this store as a 'predicted value' into the current LOAD instruction. This value will then get propagated by φ for use in later instructions.

This 'pending store' models the CPU behaviour of a store buffer, where committed-but-not-yet-retired stores are held before writing back upon retiring. Note that these pending stores *might* not be the final value (there might be other unknown stores to this address, or the computed value might itself depend on speculative values), so the forwarded value is still a prediction. Later, we will check whether the prediction is correct.

We could modify this rule by dropping the match between store target address and load address, which would model Fallout [7].

$$\textsc{Execute-Load-Present-Correct}\quad \frac{\begin{array}{c} i = \mathsf{LOAD}\langle x, a, v, cs'\rangle \qquad \mathsf{STORE}\langle\_,\_,\_\rangle \notin is_1 \\ \tau(a) \neq \bot \qquad v' = \mu(\tau(a)) \qquad v = v' \qquad i' = \mathsf{MOV}\langle x, v\rangle \end{array}}{\langle is_1, i_\ell, is_2, cs\rangle \overset{(\tilde{\rho},\tau,\mu)}{\twoheadrightarrow} \langle is_1 +\!\!+ [i'_\ell] +\!\!+ is_2, cs\rangle}$$

This is the first of three verification rules that will check the predicted values. These verification rules can only take place if there are no preceding stores pending[1], making sure that any value is truly final.

Note that we require the LOAD to carry a non-$\bot$ predicted value, thus forcing either a store forward or speculative load to have taken place before. This way, this rule can be purely a verification rule that does not have to produce a second **touch** observation. We convert the instruction into a MOV, marking the load 'final'.

In practice, a speculative load will not be checked this way - instead, the CPU will detect writes to a previously-speculatively-loaded address and will *then* mark the speculation to be incorrect. Our approach is taken because it is simpler to model, and acts more 'locally' in the reorder buffer, drastically simplifying later analysis. In terms of (attacker-)observable behaviour, the two approaches are equivalent.

$$\textsc{Execute-Load-Present-Mispredict}\quad \frac{\begin{array}{c} i = \mathsf{LOAD}\langle x, a, v, cs'\rangle \qquad \mathsf{STORE}\langle\_,\_,\_\rangle \notin is_1 \\ \tau(a) \neq \bot \qquad v' = \mu(\tau(a)) \qquad v \neq v' \qquad i' = \mathsf{MOV}\langle x, v'\rangle \end{array}}{\langle is_1, i_\ell, is_2, cs\rangle \overset{(\tilde{\rho},\tau,\mu)}{\twoheadrightarrow}_{[\mathbf{touch}(\tau(a)),\mathbf{rollback}(\ell s_2)]} \langle is_1 +\!\!+ [i'_\ell], cs'\rangle}$$

Again, we require a non-$\bot$ predicted value. This rule is for the case where the predicted value is incorrect. We perform a rollback of any speculation after this mispredicted load, while rereading from memory to find the correct value, thus producing two observations. After the rollback, the load can be committed directly since we just read the correct value (there are no pending stores).

$$\textsc{Execute-Load-Miss}\quad \frac{\begin{array}{c} i = \mathsf{LOAD}\langle x, e, pv, cs'\rangle \qquad a = [\![e]\!]^{\tilde{\rho}} \qquad \tau(a) = \bot \\ i' = \mathsf{FAULT}\langle a, \mathsf{LOAD}\langle x, a, pv, cs'\rangle, cs'\rangle \qquad i'' = \mathsf{MOV}\langle x, 0\rangle \qquad \text{fresh}(\ell'') \end{array}}{\langle is_1, i_\ell, is_2, cs\rangle \overset{(\tilde{\rho},\tau,\mu)}{\twoheadrightarrow} \langle is_1, +\!\!+ [i'_\ell, i''_{\ell''}] +\!\!+ is_2, cs\rangle}$$

In case the address to load is not in the page tables, we will trigger a page fault at the current instruction and with the given recovery command stack.

This rule will also introduce a new instruction with fresh label ℓ'', thus messing with the layout of the reorder buffer and potentially breaking our correspondence between fetches and retires of instructions. Luckily, this is safe: the FAULT instruction itself can never be retired, and the page

[1]This rule could be made a bit more flexible, by allowing preceding stores but requiring them all to have their address be computed and different from a, as long as we ensure M to be injective - preventing multiple virtual addresses from mapping to the same physical address.

fault handling rule later will always perform a rollback - thus clearing this extra MOV from the reorder buffer.

The extra MOV loads the value 0 only in an effort to not halt the CPU and allow further speculation, as happens in real CPUs as well. We could modify this to load Byzantine data, which would again model other vulnerabilities [3–5].

Memory Stores

The situation for memory stores is a lot simpler: we only have rules for a store to an address present in the page tables, or for a miss. There are no new predictions being made here, so no rollbacks are needed either. While the computed values might depend on predicted values, these predictions will be rolled back at the point of introduction.

Like in a real CPU, the execute rules will determine the address and value, without actually writing to memory. This corresponds to 'committing' a store on a real CPU, which puts the store result in the *store buffer*, without committing the data to cache or RAM. The store buffer is used as a source for store forwards to later LOAD instructions, as we saw above. Both the store and the load might be in a transient window, and could be reverted in case of an earlier misprediction.

$$\textsc{Execute-Store-Present}\quad \frac{i = \mathsf{STORE}\langle e_a, e_v, cs'\rangle \qquad a = [\![e_a]\!]^{\tilde{\rho}} \qquad v = [\![e_v]\!]^{\tilde{\rho}} \qquad \tau(a) \neq \bot \qquad i' = \mathsf{STORE}\langle a, v, cs'\rangle}{\langle is_1, i_\ell, is_2, cs\rangle \xrightarrow{(\tilde{\rho},\tau,\mu)}\!\!\!\!\rightarrow \langle is_1 +\!\!+ [i'_\ell] +\!\!+ is_2, cs\rangle}$$

$$\textsc{Execute-Store-Miss}\quad \frac{\begin{array}{c} i = \mathsf{STORE}\langle e_a, e_v, cs'\rangle \qquad a = [\![e_a]\!]^{\tilde{\rho}} \qquad v = [\![e_v]\!]^{\tilde{\rho}} \qquad \tau(a) = \bot \\ i' = \mathsf{FAULT}\langle a, \mathsf{STORE}\langle a, v, cs'\rangle, cs'\rangle \qquad i'' = \mathsf{STORE}\langle a, v, cs'\rangle \qquad \text{fresh}(\ell'') \end{array}}{\langle is_1, i_\ell, is_2, cs\rangle \xrightarrow{(\tilde{\rho},\tau,\mu)}\!\!\!\!\rightarrow \langle is_1 +\!\!+ [i'_\ell, i''_{\ell''}] +\!\!+ is_2, cs\rangle}$$

Similar to the case for a page fault during a memory load, we introduce an additional instruction i'' to the reorder buffer. This is used to allow store forwarding of the computed value to later instructions, so speculation is not halted. Again, all these instructions will be rolled back when the FAULT gets resolved, so they have no impact on the architectural state.

5.3.6 Retire Directive Rules

The retire step is where any results and computations become architectural - this is the place where we write to memory and update any registers. Retiring needs to happen in-order, and can only happen for the first instruction in the reorder buffer - enforcing the ordering *and* ensuring that the instruction can not be rolled back. The rules themselves are quite straightforward.

$$\textsc{Retire-Nop}\quad \frac{}{\langle \mathsf{NOP}_\ell : is, cs, \rho, \tau, \mu\rangle \xrightarrow{\textbf{retire}} \langle is, cs, \rho, \tau, \mu\rangle}$$

$$\textsc{Retire-Mov}\quad \frac{}{\langle \mathsf{MOV}\langle x, v\rangle_\ell : is, cs, \rho, \tau, \mu\rangle \xrightarrow{\textbf{retire}} \langle is, cs, \rho[x \mapsto v], \tau, \mu\rangle}$$

$$\textsc{Retire-Store}\quad \frac{i = \mathsf{STORE}\langle a, v, cs'\rangle \qquad pa = \tau(a) \qquad pa \neq \bot \qquad \mu' = \mu[pa \mapsto v]}{\langle i_\ell : is, cs, \rho, \tau, \mu\rangle \xrightarrow{\textbf{retire}}_{[\textbf{touch}(pa)]} \langle is, cs, \rho, \tau, \mu'\rangle}$$

Only upon retiring a write to memory, a **touch** observation is finally emitted, since only then the write is actually performed.

5.3.7 Handling Page Faults

Finally, we need to consider how to deal with page faults. Rules [Execute-Load-Miss] and [Execute-Store-Miss] introduce a `FAULT` instruction, which we need to resolve for execution to proceed. As explained in Section 2.3.1, a page fault will normally trigger an interrupt in the CPU, handing control to the kernel to evaluate the faulted memory operation. The kernel will add the address to the page tables in case it is a valid operation.

The exact implementation of the page fault handler is not of our concern, and not important for the scope of this thesis. Therefore, we abstract away the entire page fault handling as being represented by the 'ground truth' memory map M, describing how each virtual address should be mapped to a physical one. For this model, we do not consider memory permissions or memory that has not been allocated, and thus we do not consider segmentation faults and similar issues.

In our model, we handle page faults through a dedicated **map** directive. Page faults are only handled when they are the first instruction in the reorder buffer - this ensures that any earlier instructions have been fully resolved and retired. This way, page faults are only handled when speculation has stopped. Similar to real CPUs, we perform an effective rollback: the reorder buffer is cleared and the command stack is reset to the value carried by the `FAULT`.

$$\textsc{Map-Address}\quad \frac{i = \mathsf{FAULT}\langle a, i', cs'\rangle \qquad \tau' = \tau[a \mapsto M(a)]}{\langle i_\ell : is, cs, \rho, \tau, \mu\rangle \xrightarrow{\textbf{map}}_{\textbf{rollback}(\ell s)} \langle [i'_\ell], cs', \rho, \tau', \mu\rangle}$$

5.3.8 Big-Step Semantics

Again, we will introduce a big-step semantics as a simple transitive reflexive closure of the small-step semantics. For the speculative version, the relation carries a directive *schedules* D, emitting an observation trace O. We again write a single arrow $C \downarrow^D_O C'$.

$$\textsc{Done}\quad \frac{}{\langle is, cs, \rho, \tau, \mu\rangle \downarrow^{[]}_{[]} \langle is, cs, \rho, \tau, \mu\rangle}$$

$$\textsc{Step}\quad \frac{\langle is, cs, \rho, \tau, \mu\rangle \xrightarrow{d}_{O_1} \langle is', cs', \rho', \tau', \mu'\rangle \qquad \langle is', cs', \rho', \tau', \mu'\rangle \downarrow^{D}_{O_2} \langle is'', cs'', \rho'', \tau'', \mu''\rangle}{\langle is, cs, \rho, \tau, \mu\rangle \downarrow^{d:D}_{O_1 + O_2} \langle is'', cs'', \rho'', \tau'', \mu''\rangle}$$

5.4 Example

To illustrate the behaviour of both the sequential and speculative semantics, we will show how our Spectre gadget from Equation 5.1.1 can be executed, and how the speculative semantics captures the speculative behaviour enabling the Spectre attack.

For our system state, we define $\mu(5) = 42$, and $\mu(a) = 0$ otherwise. Furthermore, we have $\rho(i) = 3$, $\rho(xs) = \langle 2, 2\rangle$, and $\rho(ys) = \langle 100, 256\rangle$. For the page tables, we have $M(a) = a$ and $\tau(a) = a$. In words: we have a secret value 42 at memory address 5 that we do not want to leak, we have an index i that will read out-of-bounds of array xs, and an array ys that acts as our attacker-controlled buffer. The page tables are a simple lienar map from virtual to physical address.

5.4.1 Sequential Execution

In the case of sequential execution, nothing interesting happens:

$$\langle[\texttt{if}\ i > \mathsf{len}(xs)\ \texttt{then}\ [\texttt{skip}]\ \texttt{else}\ [x := xs\texttt{[}i\texttt{]}, y := ys\texttt{[}x\texttt{]}]], \rho, \tau, \mu\rangle$$

$$\Longrightarrow \qquad [\![i > \mathsf{len}(xs)]\!]^\rho = [\![i]\!]^\rho > [\![\mathsf{len}(xs)]\!]^\rho = 3 > 2 = \texttt{true}, \ [\textsc{Seq-If-Else}]$$

$$\langle[\texttt{skip}], \rho, \tau, \mu\rangle$$

$$\Longrightarrow \qquad [\textsc{Seq-Skip}]$$

$$\langle[], \rho, \tau, \mu\rangle.$$

5.4.2 Speculative Execution

For the speculative case, however, the attacker can evoke more interesting behaviour and leak the secret value. For example, the directive schedule $D = [\textbf{fetch}\,\texttt{false}, \textbf{fetch}, \textbf{fetch}, \textbf{exec}\,2, \textbf{exec}\,3, \textbf{exec}\,1, \textbf{retire}, \textbf{fetch}, \textbf{retire}]$ will behave as follows:

$$\langle[], [\texttt{if}\ i > \mathsf{len}(xs)\ \texttt{then}\ [\texttt{skip}]\ \texttt{else}\ [x := xs\texttt{[}i\texttt{]}, y := ys\texttt{[}x\texttt{]}]], \rho, \tau, \mu\rangle$$

$$\xrightarrow{\textbf{fetch}\,\texttt{false}} \qquad [\textsc{Fetch-If-Predict-False}]$$

$$\langle[\mathsf{ASSERT}\langle i > \mathsf{len}(xs), \texttt{false}, [\texttt{skip}]\rangle_{\ell_1}], [x := xs\texttt{[}i\texttt{]}, y := ys\texttt{[}x\texttt{]}], \rho, \tau, \mu\rangle$$

$$\xrightarrow{\textbf{fetch}} \qquad [\textsc{Fetch-Array-Load}]$$

$$\langle[\mathsf{ASSERT}\langle ...\rangle_{\ell_1}, \mathsf{LOAD}\langle x, \mathsf{base}(xs) + i, \bot, [y := ys\texttt{[}x\texttt{]}]\rangle_{\ell_2}], [y := ys\texttt{[}x\texttt{]}], \rho, \tau, \mu\rangle$$

$$\xrightarrow{\textbf{fetch}} \qquad [\textsc{Fetch-Array-Load}]$$

$$\langle[\mathsf{ASSERT}\langle ...\rangle_{\ell_1}, \mathsf{LOAD}\langle x, \mathsf{base}(xs) + i, \bot, [...]\rangle_{\ell_2}, \mathsf{LOAD}\langle y, \mathsf{base}(ys) + x, \bot, []\rangle_{\ell_3}], [], \rho, \tau, \mu\rangle$$

$$\xrightarrow{\textbf{exec}\,2}_{[\textbf{touch}(5)]} \qquad [\![\mathsf{base}(xs) + i]\!]^\rho = 5,\ \tau(5) = 5,\ \mu(5) = 42, \ [\textsc{Execute-Load-Predict}]$$

$$\langle[\mathsf{ASSERT}\langle ...\rangle_{\ell_1}, \mathsf{LOAD}\langle x, 5, 42, [...]\rangle_{\ell_2}, \mathsf{LOAD}\langle y, \mathsf{base}(ys) + x, \bot, []\rangle_{\ell_3}], [], \rho, \tau, \mu\rangle$$

$$\xrightarrow{\textbf{exec}\,3}_{[\textbf{touch}(142)]} \qquad \varphi(\rho, is_1) = \rho[x \mapsto 42],\ [\![\mathsf{base}(ys) + x]\!]^{\tilde{\rho}} = 142, \ [\textsc{Execute-Load-Predict}]$$

$$\langle[\mathsf{ASSERT}\langle ...\rangle_{\ell_1}, \mathsf{LOAD}\langle x, 5, 42, [...]\rangle_{\ell_2}, \mathsf{LOAD}\langle y, 142, 0, []\rangle_{\ell_3}], [], \rho, \tau, \mu\rangle$$

$$\xrightarrow{\textbf{exec}\,1}_{[\textbf{rollback}([\ell_2, \ell_3])]} \qquad [\textsc{Execute-Assert-Fail}]$$

$$\langle[\mathsf{NOP}_{\ell_1}], [\texttt{skip}], \rho, \tau, \mu\rangle$$

$$\xrightarrow{\textbf{retire}} \qquad [\textsc{Retire-Nop}]$$

$$\langle[], [\texttt{skip}], \rho, \tau, \mu\rangle$$

$$\xrightarrow{\textbf{fetch}} \qquad [\textsc{Fetch-Skip}]$$

$$\langle[\mathsf{NOP}_{\ell_4}], [], \rho, \tau, \mu\rangle$$

$$\xrightarrow{\textbf{retire}} \qquad [\textsc{Retire-Nop}]$$

$$\langle[], [], \rho, \tau, \mu\rangle.$$

While the architectural result at the end of this speculative execution is consistent with the sequential execution due to the eventual rollback, the observation trace $O = [\textbf{touch}(5), \textbf{touch}(142), \textbf{rollback}([...])]$ shows that the attacker was able to deduce our secret value 42 through the memory access pattern - similar to how a real Spectre attack can use a cache side channel to obtain the same observation and data leak.

As a short illustration of how our proposed mitigation - clearing the page tables before execution - would help: upon the **exec** 2 directive, the [Execute-Load-Predict] rule cannot be applied since $\tau(5) = \bot$ in this case. Instead, rule [Execute-Load-Miss] will be applied, transforming the LOAD instruction into a FAULT. This blocks further speculative execution: the second load needs a value for x which cannot be determined by φ now, and the FAULT cannot be resolved before the ASSERT is retired, which will trigger a rollback instead. Consequently, the Spectre attack is stopped in its tracks.

Chapter 6

Proving Effectiveness of ***PageZero*** against Spectre v1

This chapter is devoted to proving the effectiveness of *PageZero* against Spectre attacks. As shown in Section 5.4, emptying the page tables worked in that particular example, but in this chapter we will make our intuition more formal, and rigorously prove the result.

We will only focus on Spectre v1 attacks, where we can mispredict branches - as we modelled in the semantics. While *PageZero* will also protect against other variants of Spectre, these are out of the scope for this work - but they can be easily added to the model.

For proving effectiveness against Spectre v1, our final objective is to prove a 'transfer result' for the speculative semantics: for any program (given as a command stack cs), if the sequential, non-speculative execution exhibits specific memory access behaviour, then under speculative semantics, the code behaves the same as long as the page tables are cleared. We can then apply this to any notion of 'security': if the sequential execution is considered 'secure' in terms of memory accesses, then a speculative execution will be as well.

We will achieve this result in three parts: first, we will show confluence of our speculative semantics: execution is 'path-independent' and the results will always converge to the same values. Then, we will show that our speculative semantics is in fact consistent with the sequential semantics: any speculative execution will produce the same architectural results as sequential execution. Lastly, we will show the transfer result.

6.1 Confluence

Before we start, it is important to make an observation about one of the major differences between our speculative and sequential semantics. As is intuitive, sequential execution is deterministic and always follows the same 'path': for every state there is only a single reduction rule that can apply. This can be easily seen from an inspection of the different reduction rules.

In contrast, speculative execution can follow exponentially many paths: at nearly every step along the execution path, one can choose between executing/retiring one of (potentially many) 'current' instructions, or fetching a new command. There is no imposed 'order of operations', so things can get very complicated very quickly.

For us to bring some order to this chaos, in this section, we will establish the confluence of our speculative semantics. This implies that there is no 'path dependence' in the model: from any two configurations on different paths with a common 'origin', it is always possible to converge to some joint future configuration. This is illustrated in Figure 6.1.1, and formally stated below:

Theorem 1 (Confluence)
For any given configuration C, and two configurations C_1, C_2 such that there exist directive schedules D_1, D_2 and observation traces O_1, O_2 such that $C \Downarrow_{O_1}^{D_1} C_1$ and $C \Downarrow_{O_2}^{D_2} C_2$, there exists a configuration C' such that there exist directive schedules D_1', D_2' and observation traces O_1', O_2' such that $C_1 \Downarrow_{O_1'}^{D_1'} C'$ and $C_2 \Downarrow_{O_2'}^{D_2'} C'$. This shows that the big-step speculative semantics $\Downarrow_O^D$ is confluent.⌟

Intuitively, this can be seen from the existence of a notion of 'progress' in the model: the number of commands that are fully processed. No matter the choice of directives, at some point every command that is fetched will be retired (except for when the program gets stuck in an infinite loop). Given some 'root configuration' C then, no matter the paths taken towards C_1 and C_2, there is always some future shared configuration where both paths have retired exactly as many commands. Of course, this also hinges upon the fact that all computations are path-independent, and so the registers, page tables, and memory should be deterministic.

6.1.1 Progress

Before we can make this intuition more formal, we first need to expand our vocabulary for discussing directives and configurations:

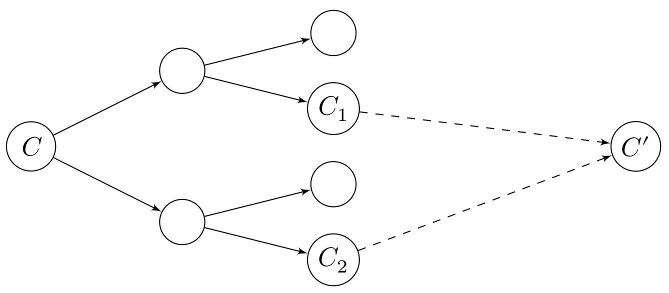

Figure 6.1.1: Graph representation of the confluence property, stating that for every pair of nodes C_1, C_2 originating from a common node C, it is possible to find some node C' that is reachable by both.

Definition 6.1.1 (Architectural Configuration) *A configuration C is said to be* architectural *when it takes the form $C = \langle cs, [], \rho, \tau, \mu \rangle$. In such a configuration, the reorder buffer is empty and thus no speculation is taking place. We define an equivalence on architectural configurations and sequential states as $C = \langle cs, [], \rho, \tau, \mu \rangle \equiv \langle cs, \rho, \tau, \mu \rangle = S$.* ⌟

Definition 6.1.2 (Valid Directive) *A given directive d is called* valid *for a configuration C iff there exists some next configuration C' and some observation trace O such that $C \xrightarrow{d}_O C'$. In other words, valid directives are exactly those directives that can be applied to a configuration.* ⌟

Definition 6.1.3 (Valid Schedule) *A given directive schedule D is called* valid *for a configuration C iff there exists some configuration C' and observation trace O such that $C \downarrow_O^D C'$. In other words, valid schedules consist of only valid directives.* ⌟

Definition 6.1.4 (Final Configuration) *A configuration C is said to be* final *when it takes the form $C = \langle [], [], \rho, \tau, \mu \rangle$. In other words, there are no more instructions or commands to be executed. Similarly, a sequential state S is said to be* final *when it takes the form $S = \langle [], \rho, \tau, \mu \rangle$.* ⌟

Alternatively, we could have defined finality to hold for a configuration that has no valid directives. We choose the current definition because it matches the intuitive notion of the computation being 'finished': there are no commands or instructions left. In fact, it turns out that both definitions are equivalent.

The first direction in this equivalence can be seen from our definitions directly: a final configuration clearly does not have any valid directives, as every reduction rule requires at least one instruction or command to be present in the configuration. The other direction requires a bit more work, and we will show the contrapositive:

Proposition 6.1.5 (Progress) *For any configuration C that is not final, there exists some valid directive d.*

PROOF
This can be seen from analysis of the form of $C = \langle is, cs, \rho, \tau, \mu \rangle$. From non-finality, we know that at least $is \neq []$ or $cs \neq []$. For all forms of is and cs, we will choose at least one valid directive as follows:

- In case $cs = c : cs'$, we distinguish the following forms of c:

  $c =$ `if` e `then` c_1 `else` c_2: In this case, we choose $d = \textbf{fetch}\, b$ where b can be randomly chosen between `true` and `false`. We thus apply either [FETCH-IF-PREDICT-TRUE] or [FETCH-IF-PREDICT-FALSE].

  $c =$ `while` e `do` cs_l: In this case, we choose $d = \textbf{fetch}\, b$ where b can be randomly chosen between `true` and `false`. We thus apply either [FETCH-WHILE-PREDICT-TRUE] or [FETCH-WHILE-PREDICT-FALSE].

  Otherwise: We choose $d = \textbf{fetch}$ and apply the relevant fetch rule corresponding to the command.

- In case $is = i : is'$, we have the following analysis of the form of i:

  $i = \mathsf{NOP}$: We choose $d = \textbf{retire}$ and use rule [RETIRE-NOP].

  $i = \mathsf{MOV}\langle x, v \rangle$: We choose $d = \textbf{retire}$ and use rule [RETIRE-MOV].

  $i = \mathsf{MOV}\langle x, e \rangle$: We choose $d = \textbf{exec}\, 1$ and use rule [EXECUTE-ASSIGN].

  $i = \mathsf{LOAD}\langle x, e, pv, cs \rangle$: We choose $d = \textbf{exec}\, 1$, but which rule applies depends on the values of e and pv:

$pv = \bot$: Either [EXECUTE-LOAD-PREDICT] or [EXECUTE-LOAD-MISS] applies, depending on the evaluation of pa in these rules. In any case, [EXECUTE-LOAD-FORWARD] does not apply, since $is_1 = []$.

$e = a \wedge pv = v$: Either [EXECUTE-LOAD-PRESENT-CORRECT] or [EXECUTE-LOAD-PRESENT-MISPREDICT] applies, depending on the evaluation of $\mu(\tau(a))$ in these rules.

Otherwise: [EXECUTE-LOAD-MISS] applies.

$i = \mathsf{STORE}\langle a, v, cs\rangle$: We choose $d = \textbf{retire}$ and rule [RETIRE-STORE] applies.

$i = \mathsf{STORE}\langle e_a, e_v, cs\rangle$: We choose $d = \textbf{exec}\, 1$, and either [EXECUTE-STORE-PRESENT] or [EXECUTE-STORE-MISS] applies, depending on the evaluation of $\tau(a)$ in these rules.

$i = \mathsf{ASSERT}\langle e, b, cs\rangle$: We choose $d = \textbf{exec}\, 1$, and either [EXECUTE-ASSERT-SUCCESS] or [EXECUTE-ASSERT-FAIL] applies, depending on the evaluation of $[\![e]\!]^{\tilde{\rho}} = b$ in these rules.

$i = \mathsf{FAULT}\langle a, i, cs\rangle$: We choose $d = \textbf{map}$ and rule [MAP-ADDRESS] applies. □

Note that Progress only shows that an execution cannot get stuck, i.e. it is always possible to continue execution until termination. Crucially, it does not show that execution will always terminate: a program could potentially end up in an infinite loop where there is always a next instruction to execute, but that will never lead to a final configuration.

6.1.2 Linking fetches and retires

Next, we direct our attention towards analysing the structure of a directive schedule D. We seek to formalise the notion that 'every fetched command will be retired', regardless of the path taken. In particular, we want to show how we can link **fetch**es and **retire**s, and that after all speculation has been resolved, the layout of the command stack is deterministic and path-independent.

Definition 6.1.6 *For a directive schedule D, $\#_d(D)$ denotes the number of occurrences of d in D. In particular, $\#_{\textbf{exec}}(D)$ denotes the number of occurrences of* **exec** n *for any n in D, and $\#_{\textbf{fetch}}(D)$ denotes the number of occurrences of both* **fetch** *and* **fetch** b *for any b in D.* ⌟

Definition 6.1.7 *Given any arbitrary configuration C and a valid directive schedule D with associated observation trace O, a directive* **fetch** *or* **fetch** b *in D is called* ignored *when the instruction i with label ℓ it introduces in the reorder buffer is erased as part of a later rollback, so when there exists a* **rollback**(ℓs) *in O for which $\ell \in \ell s$. A fetch directive that is not ignored is called* relevant. ⌟

Proposition 6.1.8 *For any given configurations C and C', where C' is architectural, with directive schedule D and associated observation trace O such that $C \Downarrow_O^D C'$, it is possible to construct a bijective (partial) map from* **retire** *directives in D to relevant* **fetch** *directives in D. When C is architectural as well, this map is a bijective function.*

PROOF
Let us first focus on the simpler case where C is also architectural. We observe that each **retire** directive only retires a single instruction at a time, each **fetch** directive only introduces a single instruction at a time, and in most cases **exec** rules will only modify instructions in-place - thus allowing a direct matching between **fetch** and **retire** directives based on instruction labels ℓ. The procedure is illustrated in Figure 6.1.2.

At its core, we match a **retire** directive retiring an instruction labelled with ℓ to the **fetch** directive producing the instruction with that same label ℓ. First, we will consider 'normal circumstances': the situation in which no page faults or mispredictions, and thus no rollbacks take place. In this case, execute rules never change the size of the reorder buffer and do not change

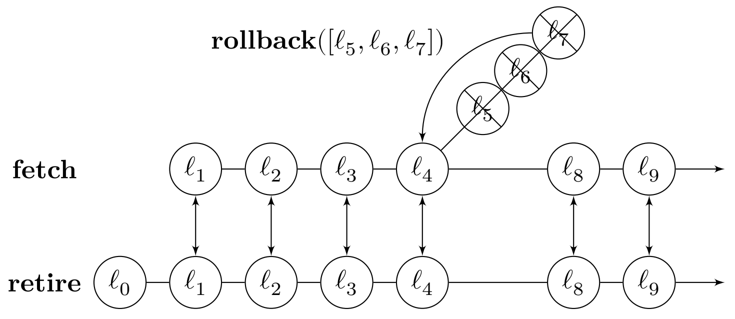

Figure 6.1.2: Graphical representation of the matching of **retire** and **fetch** directives. The bijection is constructed based on the labels for the instructions fetched and retired. In this case, fetch number 4 is a faulting instruction or a misprediction, causing later fetches to be rolled back and thus ignored. In the constructed bijection, only fetches performed after recovering from the fault or misprediction will be included.

instruction labels. Therefore, since both C and C' are architectural, we obtain that necessarily, $\#_{\textbf{fetch}}(D) = \#_{\textbf{retire}}(D)$. Consequently, our label-based mapping is a bijective function, as desired.

Let us now deal with the exceptions to the 'normal circumstances' from above:

- In case of a *page fault*, either rule [Execute-Load-Miss] or [Execute-Store-Miss] is activated. These rules both increase the size of the reorder buffer by 1 and introduce a new instruction label ℓ''; but we note that the extra instructions (beyond i') can never be retired: i' will cause the [Map-Address] rule to clear the rest of the reorder buffer completely. Specifically, the map will cause the execution to be reset to the configuration just after fetching the faulted `LOAD` or `STORE`: cs is restored to the version carried by the `FAULT`, which originates from the fetch rules, and is is reset to only contain the current `LOAD` or `STORE` instruction. Since now the page tables have been updated, this instruction will not fault again and the program flows like no fault has ever happened. The label of the faulted instruction is reused, so that a future **retire** will still be able to be linked to the original **fetch** directive.

  Intuitively, this causes all **fetch**es executed after the faulting fetch to be 'undone', and so they are not *relevant*, as can be seen in Figure 6.1.2. From the map rule we can see that all remaining instructions are recorded in the **rollback** observation, part of O, thus rendering them ignored following Definition 6.1.7.

  Since **retire** only affects the first element in the reorder buffer, none of the instructions fetched after our faulting instruction can be retired. Crucially, this means that these extra ignored fetches do not 'link' to any **retire** directive, allowing us to maintain the bijection between **retire** and *relevant* **fetch** directives.

- In case of a *misprediction*, either rule [Execute-Load-Present-Mispredict] or [Execute-Assert-Fail] is used. These rules both cause the remainder of the reorder buffer to be cleared - thus all **fetch** directives after the misprediction are now ignored. In addition to clearing the reorder buffer, these rules also reset the command stack to the state just after fetching the mispredicted instruction - but now with the correct branch. Similar to page faults, this information is carried in the `ASSERT` or `LOAD` instruction and put there by the appropriate fetch rules. After the rollback, the instruction will not be a misprediction again, since the 'other branch' is now used - the program flows like the correct prediction had been made in the first place.

  This again causes all **fetch**es executed after the mispredicted fetch to be recorded in a **rollback** observation, and so they are not *relevant*, as can be seen in Figure 6.1.2. Since **retire** only affects the first element in the reorder buffer, none of the instructions fetched after our misprediction can be retired. Crucially, this means that these extra ignored fetches do not 'link' to any **retire** directive, allowing us to maintain the bijection.

We can adapt the construction of the bijection to the more general situation where C is not architectural. In this case, the reorder buffer already contains some number of instructions that, when retired, cannot be linked to an associated fetch in D. As illustrated in Figure 6.1.2, we do not map these extra **retire** directives, as there is no **fetch** for the associated label ℓ. □

Proposition 6.1.9 *For any given configuration C, and two architectural configurations $C_1 = \langle [], cs_1, \rho_1, \tau_1, \mu_1 \rangle$, $C_2 = \langle [], cs_2, \rho_2, \tau_2, \mu_2 \rangle$ such that there exist directive schedules D_1, D_2 with associated observation traces O_1, O_2 such that $C \Downarrow_{O_1}^{D_1} C_1$ and $C \Downarrow_{O_2}^{D_2} C_2$; if $\#_{\mathbf{retire}}(D_1) = \#_{\mathbf{retire}}(D_2)$, then $cs_1 = cs_2$. In other words: for architectural configurations, the command stack is only dependent on the number of retire directives.*

Proof
From Proposition 6.1.8, we know that for both D_1 and D_2, there exist bijections between their **retire**s and the associated relevant **fetch**es. Since the number of retires in both schedules is equal, we must have that the number of relevant fetches in both schedules is equal as well (but not necessarily equal to the number of retires): to end up in an architectural configuration, both schedules need to clear the existing reorder buffer in C, in addition to retiring all the fetches done in the schedule itself. We define k to be the number of **retire** directives that do not have an associated **fetch**; these are the **retire**s needed to clear the existing reorder buffer in C.

Now we are left to prove that after $(\#_{\mathbf{retire}}(D_1) - k)$-many relevant fetches on command stack cs (one fetch per **retire**), the command stacks in both C_1 and C_2 are the same. In other words: we need to show that fetches always yield the same result. For this, we observe that all non-predictive fetches are clearly deterministic: they always pop a single command off the command stack. This leaves us with the four predictive fetch rules that have a more substantial impact on the command stack.

Luckily, thanks to the rollback mechanics, for the predictive rules, the choice between **fetch** `true` and **fetch** `false` is irrelevant. In case the correct prediction is made, the `ASSERT` instruction is neatly executed into a `NOP` using [Execute-Assert-Success] which can be retired, leaving some remaining command stack cs'. In case the wrong prediction is made, the `ASSERT` instruction carries the 'alternative command stack' cs', which will be reinstated in the configuration upon hitting the rollback in rule [Execute-Assert-Fail]. This way, no matter the prediction, upon executing the `ASSERT` (and retiring the produced `NOP`), the command stacks are the same. □

6.1.3 Eager Directives

Now that we have established a notion of 'progress' in our model, we need a way to 'direct' our progress: at the end of the day we would like the two execution paths from C_1 and C_2 to converge. For this, we introduce the concept of *eager* directives, which always prioritise clearing is, and postpone any new fetches for as long as possible. Intuitively, this allows us to mimic sequential execution in the speculative context: fetching, executing and retiring commands one by one, in order.

Definition 6.1.10 (Eager Directive) *For any non-final configuration $C = \langle is, cs, \rho, \tau, \mu \rangle$, out of the set of valid directives, a unique* eager *directive d is chosen according to the following rules:*

(1) If **retire** *is valid for C, then $d =$* **retire**.

(2) If **map** *is valid for C, then $d =$* **map**.

(3) If **exec** 1 *is valid for C, then $d =$* **exec** 1.

(4) If **fetch** *is valid for C, then $d =$* **fetch**.

(5) If **fetch** b *is valid for* C*, then* $d =$ **fetch** b*, where* b *is the 'correct prediction'.* ⌟

This definition is constructive, and it shows us how the eager directive is chosen. It will be helpful to show some properties of these eager directives:

Lemma 6.1.11 (Properties of Eager Directives) *For any non-final configuration* $C = \langle is, cs, \rho, \tau, \mu\rangle$*, the following claims hold:*

(1) There exists an eager directive d*,*

(2) When C *is not an architectural configuration, the eager directive is not* **fetch** *or* **fetch** b*, and vice versa,*

(3) When only ever running the eager directive, execution will eventually reach an architectural configuration C'*.*

Proof

(1) This follows from the proof of Proposition 6.1.5. The proof chooses a directive for each possible non-final configuration, so one of these directives will always be in the set of valid directives for C. It turns out that all these directives are possible candidates for the eager directive, and so it is always possible to choose an eager directive.

(2) When C is not architectural, this means that $is \neq []$. Again from the proof of Proposition 6.1.5, we see that in this case, always either **retire**, **exec** 1, or **map** can be chosen. Since these all have higher priority in the definition of the eager directive, that means that **fetch** or **fetch** b will never be chosen for an architectural configuration.

The other way around is easy to see: when C is architectural, necessarily $is = []$, and so only **fetch**es are possible. Since C is non-final, $cs \neq []$ so following the proof of Proposition 6.1.5, such a directive does exist.

(3) If we are already in an architectural configuration, we are done. Otherwise, from part (2) we know that the eager directive will never be **fetch** or **fetch** b. Since none of the execute, map, or retire rules increase the size of the instruction buffer[1], it only remains to show that every instruction is eventually retired by only running the eager directive. We show this using a case inspection on the first instruction in the buffer $is = i : is'$:

$i =$ NOP: This is retired immediately by [Retire-Nop].

$i =$ MOV$\langle x, v\rangle$: This is retired immediately by [Retire-Mov].

$i =$ MOV$\langle x, e\rangle$: After an **exec**, converting this instruction to MOV$\langle x, v\rangle$ using [Execute-Assign], we can retire as above.

$i =$ LOAD$\langle x, a, v, cs\rangle$: Using either [Execute-Load-Present-Correct] or [Execute-Load-Present-Mispredict], the instruction is converted to MOV$\langle x, v\rangle$, which is retired following the above.

$i =$ LOAD$\langle x, e, \perp, cs\rangle$ and $pa \neq \perp$: Using [Execute-Load-Predict], the instruction is converted to the form LOAD$\langle x, a, v, cs\rangle$, which is retired following the above.

$i =$ LOAD$\langle x, e, pv, cs\rangle$: In this last remaining case for LOAD, **exec** will resolve with [Execute-Load-Miss], and the instruction is converted to a FAULT$\langle _, \text{LOAD}, _\rangle$. Using **map** and [Map-Address], the fault is converted back to the same original load, but now $\tau(a) \neq \perp$, so the load can be retired following the previous case.

$i =$ STORE$\langle a, v, cs\rangle$: This can be retired immediately by [Retire-Store].

[1]The only exceptions are [Execute-Load-Miss] and [Execute-Store-Miss], but those cause the first instruction in the reorder buffer to be FAULT (since $is_1 = []$ because the eager directive always has $n = 1$), and this fault will immediately be resolved using **map**. The **exec** 1 and **map** together, then, cause the reorder buffer size to change from ≥ 1 to 1.

$i = \mathsf{STORE}\langle e_a, e_v, cs\rangle$ and $\tau(a) \neq \bot$: Using [Execute-Store-Present], the instruction is converted to the form $\mathsf{STORE}\langle a, v, cs\rangle$, which is retired following the previous case.

$i = \mathsf{STORE}\langle e_a, e_v, cs\rangle$ and $\tau(a) = \bot$: Now, [Execute-Store-Miss] applies, and the instruction is converted to a $\mathsf{FAULT}\langle \_, \mathsf{STORE}, \_\rangle$. Using **map** and [Map-Address], the fault is converted back to this same store, but now $\tau(a) \neq \bot$, so the store can be retired following the previous case.

$i = \mathsf{ASSERT}\langle e, b, cs\rangle$: Depending on the evaluation of $[\![e]\!]^{\tilde{\rho}} = b$, either [Execute-Assert-Success] or [Execute-Assert-Fail] applies. In both cases the instruction is converted to a NOP, which is retired following the first case.

$i = \mathsf{FAULT}\langle a, i, cs\rangle$: After **map** using [Map-Address], the instruction is replaced by the cached instruction i. This instruction will be retired following all previous rule - in particular, a nested FAULT can never exist as it is not introduced by any rule.

In this inspection, care is taken to only ever refer to 'earlier' rules, so that no cyclic dependency can occur. □

6.1.4 Confluence, at Last

Theorem 1 (Confluence)
For any given configuration C, and two configurations C_1, C_2 such that there exist directive schedules D_1, D_2 and observation traces O_1, O_2 such that $C \Downarrow_{O_1}^{D_1} C_1$ and $C \Downarrow_{O_2}^{D_2} C_2$, there exists a configuration C' such that there exist directive schedules D_1', D_2' and observation traces O_1', O_2' such that $C_1 \Downarrow_{O_1'}^{D_1'} C'$ and $C_2 \Downarrow_{O_2'}^{D_2'} C'$. This shows that the big-step speculative semantics $\Downarrow_O^D$ is confluent.

Proof
We will constructively show the existence of a configuration C' that is reachable from both C_1 and C_2. This can be done in two steps: first we will reach an architectural configuration, and then we make both configurations 'find' each other. The situation is illustrated in Figure 6.1.3 that shows the structure of speculative execution. For any two configurations in this graph, we will first obtain the 'nearest' architectural configuration by clearing the reorder buffer without fetching new commands. Then, we will traverse along the architectural configurations until the paths converge.

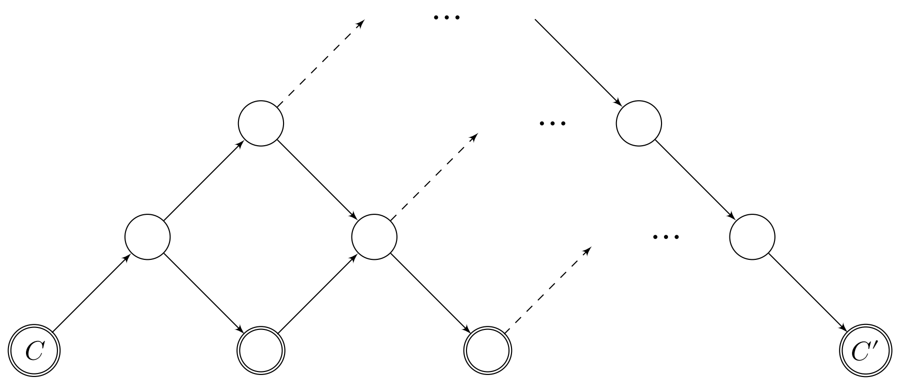

Figure 6.1.3: Graph illustrating the structure of speculative execution. Double-edged nodes represent architectural configurations: configurations with an empty reorder buffer *is*. In this model, a transition 'up' represents a **fetch**, while a transition 'down' represents a **retire**. In other words, the vertical 'distance' from the baseline is a measure of the amount of speculation taking place.

Finding Common Configuration

For the first step, we define architectural configurations C_1^* and C_2^* that we obtain by running the eager directive on C_1 and C_2 respectively, until both have hit an architectural configuration. Lemma 6.1.11 shows that such a configuration exists. We define D_1^* and D_2^* to be the original schedules augmented with the eager directives used to reach the architectural configurations, such that $C \downarrow^{D_1^*} C_1^*$ and $C \downarrow^{D_2^*} C_2^*$.

At this point, we define $n_1 = \#_{\mathbf{retire}}(D_1^*)$ and $n_2 = \#_{\mathbf{retire}}(D_2^*)$ to quantify the 'progress' in execution for both configurations. Without loss of generality, we will assume that $n_1 < n_2$. We then define architectural configuration C_1^{**} that we obtain by continuing to run the eager directive on C_1^* until it holds that $\#_{\mathbf{retire}}(D_1^{**}) = n_2$. It will turn out that $C_1^{**} = C_2^*$, thus being our desired C'.

Such a configuration C_1^{**} exists: Lemma 6.1.11 shows that since C_1^* is architectural, the next eager directive is a **fetch**. After that, from the same lemma, we know that all further eager directives are not **fetch** until the next architectural state is hit. Since there is a bijection between retires and fetches (Proposition 6.1.8), the **fetch** will have a single associated **retire** and thus $\#_{\mathbf{retire}}$ will increase by 1 for each sequence of directives between two architectural configurations, as is illustrated in Figure 6.1.3. Repeating this procedure allows us to traverse across exactly $n_2 - n_1$ architectural configurations, thus yielding the desired architectural configuration C_1^{**}.

At this point, we have two architectural configurations C_1^{**} and C_2^* with directive schedules D_1^{**} and D_2^*, such that $C \downarrow^{D_1^{**}} C_1^{**}$ and $C \downarrow^{D_2^*} C_2^*$. Additionally, we know that $\#_{\mathbf{retire}}(D_1^{**}) = \#_{\mathbf{retire}}(D_2^*)$. Therefore, from Proposition 6.1.9, we have that $cs_1^{**} = cs_2^*$.

Equality of ρ, τ, and μ

It now remains to show that not only the command stacks and (empty) reorder buffers are the same between the configurations, but in fact also the register map, page tables, and memory map are the same. From inspection of the rules, we see that these only change during a **retire** or **map** directive, which only operate on the front of the reorder buffer: they consume a single instruction from the buffer at a time. From Proposition 6.1.8 we know that for each **retire** directive, we can associate a unique **fetch** and thus a unique command c. Since the command stacks for both D_1^{**} and D_2^* are the same, each nth **retire** in both schedules is associated to the exact same command.

Since the effect of **retire** and **map** rules is fully and only dependent on the instructions that are at the front of the reorder buffer, we can simplify our analysis: we consider all possible 'paths' from an instruction generated by **fetch** to the final version of that instruction being retired by **retire**. Then, we show for each of these paths that their final effect on ρ, τ, and μ is independent of the details of the directive schedule (predictions made, order of operations) - as long as it is a valid schedule.

Note that these paths do not contain rollbacks, because in Proposition 6.1.8 we only consider *relevant* **fetch** directives. As we already showed in that proposition, a rollback - either due to misprediction or due to a page fault - will merely reset the command stack and reorder buffer, and as such will not impact ρ or μ. However, a page fault (and thus **map**) will impact τ, which is a case we will discuss separately.

As mentioned before, the effect of **retire** rules on ρ, and μ is fully dependent on the instructions that are consumed - in particular on the values of a, v, and x in the various rules. Tracing back the source of these values, we find that x is determined fully by the **fetch** instructions, and is therefore independent of the schedule. Furthermore, a and v are typically the result of evaluating some expression $[\![e]\!]^{\tilde{\rho}}$.

Schedule-Independence of $\tilde{\rho}$

Before analysing the various paths, we will argue that given an expression e and predicted register map $\tilde{\rho} = \varphi(is_1, \rho)$, the result of $[\![e]\!]^{\tilde{\rho}}$ is independent of the specific directive schedule. Here, is_1 is as in [EXECUTE-nTH-INSTRUCTION]; the first part of the reorder buffer until the nth instruction that is being executed.

We can see this independence by first noting that in all cases, e must be calculable under $\tilde{\rho}$, as otherwise the rule using its result could not have been invoked. In addition, since we consider paths between **fetch** and **retire** directives, we know that the instruction i_ℓ will not be rolled back, so no misspeculation takes place anywhere in is_1. Therefore, in a different directive schedule, all instructions in is_1 could have just as well been executed and retired before executing i, yielding some different ρ' with $\tilde{\rho}' = \varphi([], \rho') = \rho'$. We claim that $[\![e]\!]^{\tilde{\rho}} = [\![e]\!]^{\tilde{\rho}'}$.

At its core, the argument is that φ 'mimics' the behaviour of the [RETIRE-MOV] rule. Let us consider every possible i in is_1:

$\mathsf{MOV}\langle x, v\rangle$ This instruction can be retired, causing $\rho \mapsto \rho[x \mapsto v]$. This effect is the same between [RETIRE-MOV] and φ, and therefore the expression evaluation will remain consistent.

$\mathsf{MOV}\langle x, e\rangle$ The implementation of φ will cause x to be unmapped. Since we assume calculability, either x does not occur in our expression, or a later instruction will override the effect of this instruction. In both cases, this instruction does not affect the expression evaluation.

$\mathsf{LOAD}\langle x, e, pv, cs\rangle$ Since we assume calculability, we must have $pv \neq \bot$ in case x occurs in our expression, due to the implementation of φ. Since we also assume no rollbacks, the instruction must be of the form $\mathsf{LOAD}\langle x, a, v, cs\rangle$, so after an **exec** invoking [EXECUTE-LOAD-PRESENT-CORRECT], this instruction is converted into a MOV that can be retired. Its effect is $\rho \mapsto \rho[x \mapsto v]$, which is the same as through φ.

For any other instruction that is not rolled back, it does not have an effect on ρ, and therefore does not affect the evaluation of our expression.

Path Analysis of Retired Instructions

Now, we will analyse all 'paths' between **fetch**es and **retire**s: we write $\langle i \to i'\rangle$ to denote the bijection pair where instruction i' is retired, which was originally generated by a **fetch** producing i. As before, this matching is based on the label ℓ. For each of the paths, we will show that their effect on ρ, τ, and μ is independent of the directive schedule.

$\langle\mathsf{NOP} \to \mathsf{NOP}\rangle$ In this path, no **exec** directive can be applied, since the produced instruction i can immediately be retired. Therefore, this path is completely unaffected by the rest of the program state.

$\langle\mathsf{ASSERT}\langle e, b, cs\rangle \to \mathsf{NOP}\rangle$ In this path, one of the `if` or `while` fetch rules produced an ASSERT, which was converted to NOP by either [EXECUTE-ASSERT-SUCCESS] or [EXECUTE-ASSERT-FAIL]. In all cases, given a directive schedule applying **exec** to the ASSERT, the result will be a NOP instruction. While the contents of the reorder buffer or command stack might theoretically vary based on $\tilde{\rho}$, we already established that these are the same in both C_1^{**} and C_2^{*}, so we can ignore them here.

$\langle\mathsf{MOV}\langle x, e\rangle \to \mathsf{MOV}\langle x, v\rangle\rangle$ This is another simple case, where there is a single invocation of [EXECUTE-ASSIGN] converting between i and i'. The value of v is dependent on e (determined by the command stack, so fixed) and $\tilde{\rho}$. This predictive map is, unfortunately, dependent on the rest of the configuration; dependent on ρ and is_1 to be specific. However, as seen above, this is not a big issue: regardless of whether **exec** is executed when i is at the front or at the back of the reorder buffer, the evaluation of e under $\tilde{\rho}$ is the same.

$\langle \mathsf{LOAD}\langle x, e, \bot, cs\rangle \rightarrow \mathsf{MOV}\langle x, v\rangle\rangle$ This is arguably the most complex case, as there are many routes to be analysed. In general, each LOAD has two execute steps: first, speculatively a value is populated through either [Execute-Load-Predict] or [Execute-Load-Forward], yielding an instruction of the form $\mathsf{LOAD}\langle x, a, pv, cs\rangle$. This value is not guaranteed to be correct: there might be unresolved stores in the reorder buffer that will affect the outcome later, or the memory address might not be mapped. The case for unmapped memory will be discussed later.

Next, the predicted value is verified through either [Execute-Load-Present-Correct] or [Execute-Load-Present-Mispredict], which will finally replace the LOAD with the correct MOV instruction, preparing it to be retired.

Let us now analyse the flow of dependencies: the value of v in the final MOV instruction is determined by either of the two 'verification' rules; and in both cases it is equal to $\mu(\tau(a))$ as these rules verify the predicted value against the real memory map. Since both rules require there to be no stores in is_1, we can be sure that the loaded value from memory is the correct value - and since all stores have the same effect on the memory map (see below), we know that the loaded value is fixed for the value of a.

The value of v is therefore only dependent on the address a as determined by the two population rules, in both cases computed as $a = [\![e]\!]^{\tilde{\rho}}$. As seen in earlier cases, e is fixed since it is decided by the command stack, and we know that $[\![e]\!]^{\tilde{\rho}}$ is then the same as well. Therefore, the value of a is the same for any execution, and we conclude that no matter the prediction made or directive schedule chosen, the fetched LOAD will resolve to the same MOV.

$\langle \mathsf{STORE}\langle e_a, e_v, cs\rangle \rightarrow \mathsf{STORE}\langle a, v, cs\rangle\rangle$ For this last case, we will again consider the page fault case separately. This leaves only a single rule that can be used for this path: [Execute-Store-Present]. The values of a and v in i' are determined fully by e_a, e_v, and $\tilde{\rho}$. As in earlier cases, both expressions are fixed by the command stack, and we know that also $[\![e]\!]^{\tilde{\rho}}$ is fixed. Therefore, i' is independent of the directive schedule.

Path Analysis of Page Faults

We have now established that in addition to $cs_1^{**} = cs_2^{*}$ and $is_1^{**} = is_2^{*}$, we also have $\rho_1^{**} = \rho_2^{*}$ and $\mu_1^{**} = \mu_2^{*}$. Lastly, we need to deal with **map** directives and τ. As we already showed in the proof of Proposition 6.1.8, **map** directives cause the command stack and reorder buffer to be reset to the state immediately after the original **fetch**. Similar to how we considered the 'paths' between **fetch** and **retire** directives above, we can consider 'paths' between **fetch** and **map** directives. Since **map** directives only change the page tables τ, and since no **retire** can occur in the 'path' between **fetch** and **map**, we know that all paths we consider keep ρ and μ fixed - in line with what we established already.

To show that the effect of **map** on τ is independent on the directive schedule, we observe that the effect of **map** is only dependent on the faulted address a (M itself is fixed). Therefore, the value of a should be independent of the directive schedule. Again, we analyse the various 'paths' from **fetch** to **map**, written as $\langle i \rightarrow i'\rangle$ denoting the path from a fetched instruction i to the FAULT instruction i' handled by **map**:

$\langle \mathsf{LOAD}\langle x, e, \bot, cs\rangle \rightarrow \mathsf{FAULT}\langle a, i, cs\rangle\rangle$ The final FAULT instruction can only be generated by rule [Execute-Load-Miss], which can immediately be applied to i. We should be careful to note that a single **exec** on i could also allow rule [Execute-Load-Forward] to be invoked, after which [Execute-Load-Miss] can still be used. However, since a must be unmapped, the forwarded STORE instruction itself must cause a fault (discussed below), which will be 'earlier' in the reorder buffer. Since resolving a fault with **map** will clear the reorder buffer, a FAULT generated after a stored forward will never be resolved during a **map**, so we can ignore this case and focus only on the path where a single **exec** invoked [Execute-Load-Miss].

In this single rule, the value of a is determined as $[\![e]\!]^{\tilde{\rho}}$. As seen before, e is fixed since it is decided by the command stack, and therefore we know that $[\![e]\!]^{\tilde{\rho}}$ is fixed as well. Consequently, the value of a is the same for any directive schedule.

$\langle \mathsf{STORE}\langle e_i, e_v, cs\rangle \rightarrow \mathsf{FAULT}\langle a, i, cs\rangle\rangle$ Similar to the previous case, the final FAULT instruction can only be generated by rule [Execute-Store-Miss]. In this rule, the value of a is determined as $[\![e_a]\!]^{\tilde{\rho}}$. And again, e_a is fixed by the command stack, and $\tilde{\rho}$ is fixed as well. Therefore, the value of a is the same for any directive schedule.

In addition to showing that the effect of **map** directives on a particular faulted load or store is the same, we should show that both D_1^{**} and D_2^{*} fault on the same addresses. This is straightforward: since both schedules start from the same configuration C their starting page tables are identical as well. As we showed above, the addresses a of the various rules are independent of the exact directive schedule. For any accessed address a, if it is not in the page tables, a FAULT instruction is injected, so the directive schedule must then use a **map** directive for each such faulted address. Since we already know that ρ and μ are fixed between both directive schedules, it must be the case that they have accessed the same addresses and triggered the same page faults. Therefore, we establish that also $\tau_1^{**} = \tau_2^{*}$.

To summarize, we have now obtained equality between C_1^{**} and C_2^{*} in terms of their command stacks, reorder buffers, register maps, page tables, *and* memory maps. Therefore, $C_1^{**} = C_2^{*}$, and so it is the C' we are looking for to prove confluence. $\square$

6.2 Consistency

For our second result, we will investigate the relation between our speculative semantics and the traditional sequential semantics. In particular, we will show *consistency* between the two: the result of a computation is the same for both speculative and sequential semantics. This legitimises the use of our speculative semantics.

Theorem 2 (Consistency)
For any program represented by command stack cs and initial state ρ, τ, μ, with a sequential execution $\langle cs, \rho, \tau, \mu \rangle \Downarrow_O \langle [], \rho', \tau', \mu' \rangle$, any complete directive schedule D on $C = \langle [], cs, \rho, \tau, \mu \rangle$ satisfies $C \downarrow^D_{O'} \langle [], [], \rho', \tau', \mu' \rangle$. Therefore, the speculative semantics is consistent with the sequential semantics. ⌟

To show this consistency, we will construct a specific directive schedule for any given sequential execution. Then we can use confluence to show that not just this specific directive schedule is equivalent, but any speculative execution is equivalent to sequential semantics.

Definition 6.2.1 (Complete Schedule) *A valid directive schedule D is called* complete *for a configuration C iff there exists a final configuration C_f and an observation trace O such that $C \downarrow^D_O C_f$. In other words, a schedule is complete iff it is valid and ends in a final configuration.* ⌟

Proposition 6.2.2 (Equivalence of Complete Schedules) *Given a configuration C, for any complete schedules D_1, D_2, final configurations C_1, C_2, and observation traces O_1, O_2 such that $C \downarrow^{D_1}_{O_1} C_1$ and $C \downarrow^{D_2}_{O_2} C_2$, we have that $C_1 = C_2$.*

Proof
The set up here is exactly the same as for confluence, so from Theorem 1 we know that there must exist some configuration C' for which there exist directive schedules D'_1 and D'_2 such that $C_1 \downarrow^{D'_1} C'$ and $C_2 \downarrow^{D'_2} C'$. However, by assumption C_1 and C_2 are final, so D'_1 and D'_2 must be empty and in fact $C_1 = C_2$. □

The specific directive schedule we will construct to be equivalent to sequential execution will be a *sequential* schedule; a schedule consisting entirely out of eager directives. As we have seen in

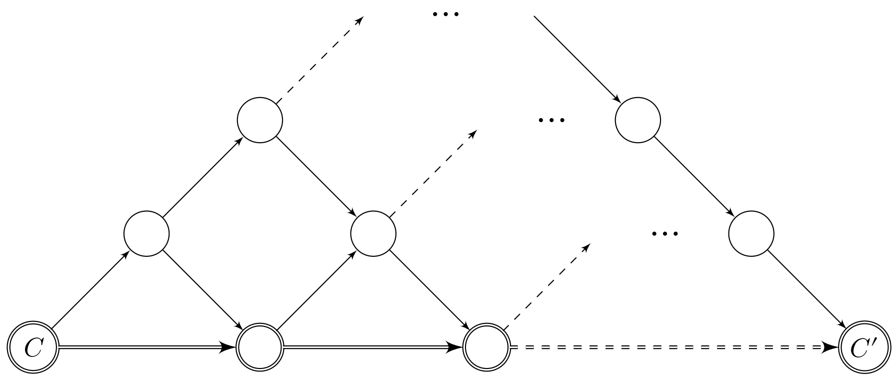

Figure 6.2.1: Graph illustrating the relation between sequential and speculative execution. This is the same as Figure 6.1.3, but now the bottom set of double arrows have been added to represent sequential execution. Again, a transition 'upwards' represents a **fetch**, while a transition 'down' represents a **retire**.

the proof for confluence, executing eager directives will allow us to process commands from the command stack one by one, in line with sequential semantics. This is illustrated in Figure 6.2.1, where the bottom set of double arrows indicate the sequential execution.

Definition 6.2.3 (Sequential Schedule) *A complete directive schedule D is called* sequential *for a configuration C iff the schedule consists entirely out of eager directives.* ⌟

Proposition 6.2.4 (Existence of Sequential Schedules) *For any state $S = \langle cs, \rho, \tau, \mu\rangle$ and sequential execution $S \Downarrow_O \langle [], \rho', \tau', \mu'\rangle$, there exists an equivalent sequential schedule D such that for configuration $C = \langle [], cs, \rho, \tau, \mu\rangle$, we have $C \downarrow^{D}_{O'} \langle [], [], \rho', \tau', \mu'\rangle$.*

Proof

We will prove this statement by induction on the big-step reduction $\Downarrow$. For each of the sequential execution rules for $\Longrightarrow$, we will find a sequence of eager directives that has the same effect. Concretely, given a state $S_i = \langle cs_i, \rho_i, \tau_i, \mu_i\rangle$, if $S_i \Longrightarrow_O \langle cs_{i+1}, \rho_{i+1}, \tau_{i+1}, \mu_{i+1}\rangle$, we will find a set of eager directives D_i such that for the architectural configuration $C_i = \langle [], cs_i, \rho_i, \tau_i, \mu_i\rangle$ we have $C_i \downarrow^{D_i}_{O} \langle [], cs_{i+1}, \rho_{i+1}, \tau_{i+1}, \mu_{i+1}\rangle$.

For simplicity of writing, we will find the set D_i for any arbitrary i and drop the labels:

[Seq-Skip] In this case, $\langle \texttt{skip} : cs, \rho, \tau, \mu\rangle \Longrightarrow \langle cs, \rho, \tau, \mu\rangle$. We use the schedule $D = [\textbf{fetch}, \textbf{retire}]$, which will transform as follows:

$$\langle [], \texttt{skip} : cs, \rho, \tau, \mu\rangle$$
$$\xrightarrow{\textbf{fetch}} \qquad \text{[Fetch-Skip]}$$
$$\langle [\mathsf{NOP}_\ell], cs, \rho, \tau, \mu\rangle$$
$$\xrightarrow{\textbf{retire}} \qquad \text{[Retire-Nop]}$$
$$\langle [], cs, \rho, \tau, \mu\rangle$$

[Seq-Assign] We now have $\langle x := e : cs, \rho, \tau, \mu\rangle \Longrightarrow \langle cs, \rho[x \mapsto v], \tau, \mu\rangle$ where $v = [\![e]\!]^{\rho}$. We use the schedule $D = [\textbf{fetch}, \textbf{exec}\, 1, \textbf{retire}]$ as follows:

$$\langle [], x := e : cs, \rho, \tau, \mu\rangle$$
$$\xrightarrow{\textbf{fetch}} \qquad \text{[Fetch-Assign]}$$
$$\langle [\mathsf{MOV}\langle x, e\rangle_\ell], cs, \rho, \tau, \mu\rangle$$
$$\xrightarrow{\textbf{exec}\,1} \qquad v = [\![e]\!]^{\tilde{\rho}} \text{ and } \tilde{\rho} = \varphi([], \rho) = \rho \text{ - [Execute-Assign]}$$
$$\langle [\mathsf{MOV}\langle x, v\rangle_\ell], cs, \rho, \tau, \mu\rangle$$
$$\xrightarrow{\textbf{retire}} \qquad \text{[Retire-Mov]}$$
$$\langle [], cs, \rho[x \mapsto v], \tau, \mu\rangle$$

[Seq-Pointer-Read-Present] We now have a more complex scenario with a number of additional variables: $\langle x := *e : cs, \rho, \tau, \mu\rangle \Longrightarrow_{[\textbf{touch}(pa)]} \langle cs, \rho[x \mapsto v], \tau, \mu\rangle$. Here, $a = [\![e]\!]^{\rho}$, $pa = \tau(a) \neq$

$\perp$ and $v = \mu(pa)$. We use the schedule $D = [\mathbf{fetch}, \mathbf{exec}\,1, \mathbf{exec}\,1, \mathbf{retire}]$ as follows:

$$\langle [], x \mathrel{:=} {*}\,e : cs, \rho, \tau, \mu \rangle$$

$$\xrightarrow{\mathbf{fetch}} \qquad [\textsc{Fetch-Pointer-Load}]$$

$$\langle [\mathsf{LOAD}\langle x, e, \perp, cs \rangle_\ell], cs, \rho, \tau, \mu \rangle$$

$$\xrightarrow{\mathbf{exec}\,1}_{[\mathbf{touch}(pa)]} \qquad a = [\![e]\!]^\rho,\ pa = \tau(a),\ v = \mu(pa) \text{ - } [\textsc{Execute-Load-Predict}]$$

$$\langle [\mathsf{LOAD}\langle x, a, v, cs \rangle_\ell], cs, \rho, \tau, \mu \rangle$$

$$\xrightarrow{\mathbf{exec}\,1} \qquad [\textsc{Execute-Load-Present-Correct}]$$

$$\langle [\mathsf{MOV}\langle x, v \rangle_\ell], \rho, \tau, \mu \rangle$$

$$\xrightarrow{\mathbf{retire}} \qquad [\textsc{Retire-Mov}]$$

$$\langle [], cs, \rho[x \mapsto v], \tau, \mu \rangle$$

[\textsc{Seq-Pointer-Read-Miss}] This time, we also have to deal with a page fault: $\langle x \mathrel{:=} {*}\,e : cs, \rho, \tau, \mu \rangle \Longrightarrow_{[\mathbf{touch}(pa)]} \langle cs, \rho[x \mapsto v], \tau[a \mapsto pa], \mu \rangle$, where all variables are the same as before, but $pa = M(a)$. Now, we use the schedule $D = [\mathbf{fetch}, \mathbf{exec}\,1, \mathbf{map}, \mathbf{exec}\,1, \mathbf{exec}\,1, \mathbf{retire}]$ as follows:

$$\langle [], x \mathrel{:=} {*}\,e : cs, \rho, \tau, \mu \rangle$$

$$\xrightarrow{\mathbf{fetch}} \qquad [\textsc{Fetch-Pointer-Load}]$$

$$\langle [\mathsf{LOAD}\langle x, e, \perp, cs \rangle_\ell], cs, \rho, \tau, \mu \rangle$$

$$\xrightarrow{\mathbf{exec}\,1} \qquad [\textsc{Execute-Load-Miss}]$$

$$\langle [\mathsf{FAULT}\langle a, \mathsf{LOAD}\langle x, a, \perp, cs \rangle, cs \rangle_\ell, \mathsf{MOV}\langle x, 0 \rangle_{\ell'}], cs, \rho, \tau, \mu \rangle$$

$$\xrightarrow{\mathbf{map}} \qquad pa = M(a) \text{ - } [\textsc{Map-Address}]$$

$$\langle [\mathsf{LOAD}\langle x, a, \perp, cs \rangle_\ell], cs, \rho, \tau[a \mapsto pa], \mu \rangle$$

$$\xrightarrow{\mathbf{exec}\,1}_{[\mathbf{touch}(pa)]} \qquad [\textsc{Execute-Load-Predict}]$$

$$\langle [\mathsf{LOAD}\langle x, a, v, cs \rangle_\ell], cs, \rho, \tau[a \mapsto pa], \mu \rangle$$

$$\xrightarrow{\mathbf{exec}\,1} \qquad [\textsc{Execute-Load-Present-Correct}]$$

$$\langle [\mathsf{MOV}\langle x, v \rangle_\ell], cs, \rho, \tau[a \mapsto pa], \mu \rangle$$

$$\xrightarrow{\mathbf{retire}} \qquad [\textsc{Retire-Mov}]$$

$$\langle [], cs, \rho[x \mapsto v], \tau[a \mapsto pa], \mu \rangle$$

[\textsc{Seq-Pointer-Write-Present}] For writes, the impact is on μ instead of ρ: $\langle {*}\,e_a \mathrel{:=} e_v : cs, \rho, \tau, \mu \rangle \Longrightarrow_{[\mathbf{touch}(pa)]} \langle cs, \rho, \tau, \mu[pa \mapsto v] \rangle$. Here, $a = [\![e_a]\!]^\rho$ and $v = [\![e_v]\!]^\rho$. For the speculative route, we use the straightforward schedule $D = [\mathbf{fetch}, \mathbf{exec}\,1, \mathbf{retire}]$:

$$\langle [], {*}\,e_a \mathrel{:=} e_v : cs, \rho, \tau, \mu \rangle$$

$$\xrightarrow{\mathbf{fetch}} \qquad [\textsc{Fetch-Pointer-Store}]$$

$$\langle [\mathsf{STORE}\langle e_a, e_v, cs \rangle_\ell], cs, \rho, \tau, \mu \rangle$$

$$\xrightarrow{\mathbf{exec}\,1} \qquad [\textsc{Execute-Store-Present}]$$

$$\langle [\mathsf{STORE}\langle a, v, cs \rangle_\ell], cs, \rho, \tau, \mu \rangle$$

$$\xrightarrow{\mathbf{retire}}_{[\mathbf{touch}(pa)]} \qquad [\textsc{Retire-Store}]$$

$$\langle [], cs, \rho, \tau, \mu[pa \mapsto v] \rangle$$

[SEQ-POINTER-WRITE-MISS] For write misses, we once again have to deal with a page fault: $\langle \texttt{*}\, e_a \texttt{ := } e_v : cs, \rho, \tau, \mu\rangle \Longrightarrow_{[\mathbf{touch}(pa)]} \langle cs, \rho, \tau[a \mapsto pa], \mu[pa \mapsto v]\rangle$, with all the variables the same as before. We use the schedule $D = [\mathbf{fetch}, \mathbf{exec}\,1, \mathbf{map}, \mathbf{retire}]$:

$$\langle [], \texttt{*}\, e_a \texttt{ := } e_v : cs, \rho, \tau, \mu\rangle$$
$$\xrightarrow{\mathbf{fetch}} \qquad [\text{FETCH-POINTER-STORE}]$$
$$\langle [\mathsf{STORE}\langle e_a, e_v, cs\rangle_\ell], cs, \rho, \tau, \mu\rangle$$
$$\xrightarrow{\mathbf{exec}\,1} \qquad [\text{EXECUTE-STORE-MISS}]$$
$$\langle [\mathsf{FAULT}\langle a, \mathsf{STORE}\langle a, v, cs\rangle, cs\rangle_\ell, \mathsf{STORE}\langle a, v, cs\rangle_{\ell'}], cs, \rho, \tau, \mu\rangle$$
$$\xrightarrow{\mathbf{map}} \qquad [\text{MAP-ADDRESS}]$$
$$\langle [\mathsf{STORE}\langle a, v, cs\rangle_\ell], cs, \rho, \tau[a \mapsto pa], \mu\rangle$$
$$\xrightarrow{\mathbf{retire}} \qquad [\text{RETIRE-STORE}]$$
$$\langle [], cs, \rho, \tau[a \mapsto pa], \mu[pa \mapsto v]\rangle$$

[SEQ-ARRAY-READ-PRESENT] This rule is essentially the same as [SEQ-POINTER-READ-PRESENT], except that we are not given an index expression e, but instead we compute $e_i = \mathsf{base}(xs) + e$. Consequently, we can use the exact same directive schedule as for that rule, $D = [\mathbf{fetch}, \mathbf{exec}\,1, \mathbf{exec}\,1, \mathbf{retire}]$. To shorten analysis, we will only show that the configuration after **fetch** is identical to the above:

$$\langle [], x \texttt{ := } xs\texttt{[}e\texttt{]} : cs, \rho, \tau, \mu\rangle$$
$$\xrightarrow{\mathbf{fetch}} \qquad [\text{FETCH-ARRAY-LOAD}]$$
$$\langle [\mathsf{LOAD}\langle x, e_i, \bot, cs\rangle_\ell], cs, \rho, \tau, \mu\rangle$$

[SEQ-ARRAY-READ-MISS] As above, again this is the same as [SEQ-POINTER-READ-MISS] but with $e_i = \mathsf{base}(xs) + e$ instead of e itself. We use again $D = [\mathbf{fetch}, \mathbf{exec}\,1, \mathbf{map}, \mathbf{exec}\,1, \mathbf{exec}\,1, \mathbf{retire}]$, and from the derivation for the 'present' case, we see that the first **fetch** does indeed give the same configuration.

[SEQ-ARRAY-WRITE-PRESENT] Again, we compare to [SEQ-POINTER-WRITE-PRESENT] but with $e_i = \mathsf{base}(xs) + e$ again instead of e as target memory address. Therefore, we again use $D = [\mathbf{fetch}, \mathbf{exec}\,1, \mathbf{retire}]$ and we will show equivalence of the configuration after the first **fetch**:

$$\langle [], xs\texttt{[}e\texttt{]} \texttt{ := } e_v : cs, \rho, \tau, \mu\rangle$$
$$\xrightarrow{\mathbf{fetch}} \qquad [\text{FETCH-ARRAY-STORE}]$$
$$\langle [\mathsf{STORE}\langle e_i, e_v, cs\rangle_\ell], cs, \rho, \tau, \mu\rangle$$

[SEQ-ARRAY-WRITE-MISS] And again, we compare to [SEQ-POINTER-WRITE-MISS]. So, again $D = [\mathbf{fetch}, \mathbf{exec}, \mathbf{map}, \mathbf{retire}]$ can be used, and the first **fetch** is the same for this and the 'present' case.

[SEQ-IF-ELSE] For if-else statements, we deal with $\langle \texttt{if } e \texttt{ then } cs_{\texttt{true}} \texttt{ else } cs_{\texttt{false}} : cs, \rho, \tau, \mu\rangle \Longrightarrow \langle cs_b +\!\!+ cs, \rho, \tau, \mu\rangle$, where cs_b is the command stack chosen based on $b = [\![e]\!]^\rho$. We can choose directive schedule $D = [\mathbf{fetch}\,b, \mathbf{exec}\,1, \mathbf{retire}]$, where the predicted value b in $\mathbf{fetch}\,b$ is always correct by construction of the eager directive.

In case we correctly predict $b = \texttt{true}$:

$$\begin{aligned}
&\langle [], \texttt{if}\ e\ \texttt{then}\ cs_{\texttt{true}}\ \texttt{else}\ cs_{\texttt{false}} : cs, \rho, \tau, \mu\rangle \\
\xrightarrow{\textbf{fetch}\,\texttt{true}}\ & \qquad\qquad [\textsc{Fetch-If-Predict-True}] \\
&\langle [\mathsf{ASSERT}\langle e, \texttt{true}, cs_{\texttt{false}} +\!\!+\, cs\rangle_\ell], cs_{\texttt{true}} +\!\!+\, cs, \rho, \tau, \mu\rangle \\
\xrightarrow{\textbf{exec}\,1}\ & \qquad\qquad [\textsc{Execute-Assert-Success}] \\
&\langle [\mathsf{NOP}_\ell], cs_{\texttt{true}} +\!\!+\, cs, \rho, \tau, \mu\rangle \\
\xrightarrow{\textbf{retire}}\ & \qquad\qquad [\textsc{Retire-Nop}] \\
&\langle [], cs_{\texttt{true}} +\!\!+\, cs, \rho, \tau, \mu\rangle
\end{aligned}$$

And in case $b = \texttt{false}$:

$$\begin{aligned}
&\langle [], \texttt{if}\ e\ \texttt{then}\ cs_{\texttt{true}}\ \texttt{else}\ cs_{\texttt{false}} : cs, \rho, \tau, \mu\rangle \\
\xrightarrow{\textbf{fetch}\,\texttt{false}}\ & \qquad\qquad [\textsc{Fetch-If-Predict-False}] \\
&\langle [\mathsf{ASSERT}\langle e, \texttt{false}, cs_{\texttt{true}} +\!\!+\, cs\rangle_\ell], cs_{\texttt{false}} +\!\!+\, cs, \rho, \tau, \mu\rangle \\
\xrightarrow{\textbf{exec}\,1}\ & \qquad\qquad [\textsc{Execute-Assert-Success}] \\
&\langle [\mathsf{NOP}_\ell], cs_{\texttt{false}} +\!\!+\, cs, \rho, \tau, \mu\rangle \\
\xrightarrow{\textbf{retire}}\ & \qquad\qquad [\textsc{Retire-Nop}] \\
&\langle [], cs_{\texttt{false}} +\!\!+\, cs, \rho, \tau, \mu\rangle
\end{aligned}$$

[Seq-While] Lastly, we consider while statements, where we have to deal with $\langle \texttt{while}\ e\ \texttt{do}\ cs_l : cs, \rho, \tau, \mu\rangle \implies \langle cs_b +\!\!+\, cs, \rho, \tau, \mu\rangle$ where again cs_b is the command stack chosen based on $b = [\![e]\!]^\rho$, and $cs_{\texttt{true}} = cs_l +\!\!+\, [c]$ and $cs_{\texttt{false}} = []$. We can choose directive schedule $D = [\textbf{fetch}\,b, \textbf{exec}\,1, \textbf{retire}]$, where the prediction b is again always correct due to eagerness.

In case $b = \texttt{true}$:

$$\begin{aligned}
&\langle [], \texttt{while}\ e\ \texttt{do}\ cs_l : cs, \rho, \tau, \mu\rangle \\
\xrightarrow{\textbf{fetch}\,\texttt{true}}\ & \qquad\qquad [\textsc{Fetch-While-Predict-True}] \\
&\langle [\mathsf{ASSERT}\langle e, \texttt{true}, cs\rangle_\ell], cs_l +\!\!+\, c : cs, \rho, \tau, \mu\rangle \\
\xrightarrow{\textbf{exec}\,1}\ & \qquad\qquad [\textsc{Execute-Assert-Success}] \\
&\langle [\mathsf{NOP}_\ell], cs_l +\!\!+\, c : cs, \rho, \tau, \mu\rangle \\
\xrightarrow{\textbf{retire}}\ & \qquad\qquad [\textsc{Retire-Nop}] \\
&\langle [], cs_l +\!\!+\, c : cs, \rho, \tau, \mu\rangle
\end{aligned}$$

And, finally, in case $b = \texttt{false}$:

$$\begin{aligned}
&\langle [], \texttt{while}\ e\ \texttt{do}\ cs_l : cs, \rho, \tau, \mu\rangle \\
\xrightarrow{\textbf{fetch}\,\texttt{false}}\ & \qquad\qquad [\textsc{Fetch-While-Predict-False}] \\
&\langle [\mathsf{ASSERT}\langle e, \texttt{false}, cs_l +\!\!+\, c : cs\rangle_\ell], cs, \rho, \tau, \mu\rangle \\
\xrightarrow{\textbf{exec}\,1}\ & \qquad\qquad [\textsc{Execute-Assert-Success}] \\
&\langle [\mathsf{NOP}_\ell], cs, \rho, \tau, \mu\rangle \\
\xrightarrow{\textbf{retire}}\ & \qquad\qquad [\textsc{Retire-Nop}] \\
&\langle [], cs, \rho, \tau, \mu\rangle
\end{aligned}$$

□

We can now proceed to use the existence of sequential schedules to prove consistency between sequential and speculative semantics.

Theorem 2 (Consistency)
For any program represented by command stack cs and initial state ρ, τ, μ, with a sequential execution $\langle cs, \rho, \tau, \mu \rangle \Downarrow_O \langle [], \rho', \tau', \mu' \rangle$, any complete directive schedule D on $C = \langle [], cs, \rho, \tau, \mu \rangle$ satisfies $C \downarrow^{D}_{O'} \langle [], [], \rho', \tau', \mu' \rangle$. Therefore, the speculative semantics is consistent with the sequential semantics.

Proof
This proof is a simple combination of the previous proposition and confluence. From Proposition 6.2.4, we know that there exists a specific complete sequential schedule D satisfying $C \downarrow^{D}_{O'} \langle [], [], \rho', \tau', \mu' \rangle$. Then, since D is complete, from Proposition 6.2.2, we have that any other complete schedule D' on C also satisfies $C \downarrow^{D'}_{O'} \langle [], [], \rho', \tau', \mu' \rangle$. We can conclude that the speculative semantics is consistent with traditional sequential semantics. $\square$

6.3 Security

The previous sections have established confluence and consistency of our speculative semantics, thus providing a degree of legitimacy to its design. In Section 5.4, we have already seen how our semantics allow us to capture the risks of speculative execution, as we model both store forwarding and speculative loads. What remains now is to show the core of *PageZero*: to establish that, as suspected, clearing page tables allows us to mitigate speculative execution attacks.

We approach this issue by proving a 'transfer' result that allows us to transfer any property of the memory access behaviour of a program under sequential semantics into the speculative realm - as long as we clear the page tables. Thus, if a program is deemed 'secure' under sequential semantics (for any notion of 'security'), then it will remain 'secure' under speculative semantics when the page tables are cleared.

Our proof will not be specific to a particular notion of security, instead, we will prove the more general and stronger property that *the set of accessed addresses* is consistent between sequential and speculative semantics. Consequently, any property that describes a set of memory addresses, including 'security', will be ported from the sequential into the speculative realm.

This property might seem counter-intuitive at first, due to the nature of speculative execution. The crucial 'trick' is to impose that in the speculative setting, the page tables are cleared before execution. This way, we don't stop speculation altogether, but we stop it just enough to prevent undesirable accesses. The intuition is that the first time a memory address is accessed, the access is 'validated' through the page fault mechanism. Since page faults do not happen speculatively, this stops invalid accesses. Once it is determined that an address is accessible, it will become available for speculation in the future.

Notation 6.3.1 *By* $\texttt{PAddr}(O)$ *we denote the set of physical addresses emitted by all* **touch** *observations in* O. ⌟

Definition 6.3.2 *We define a* memory property $P \subseteq \mathcal{P}(\texttt{PAddr})$ *to* hold *for a set of physical addresses* $X \in \mathcal{P}(\texttt{PAddr})$ *when* $X \in P$. ⌟

Theorem 3 (Memory Access Equivalence)
Consider any program represented by command stack cs and initial state ρ, μ *with a sequential execution* $\langle cs, \rho, \tau, \mu \rangle \Downarrow_O \langle [], \rho', \tau', \mu' \rangle$ *with arbitrary* τ. *Then, any complete directive schedule D on* $C = \langle [], cs, \rho, \emptyset, \mu \rangle$ *with* $C \downarrow^D_{O'} \langle [], [], \rho', \tau_\emptyset, \mu' \rangle$ *satisfies the property that* $\texttt{PAddr}(O) = \texttt{PAddr}(O')$. *In other words, the set of touched physical addresses for sequential execution is the same as for speculative execution.* ⌟

This result will immediately give us the 'transfer property' we need to show *PageZero* effective.

Corollary (Memory Property Transfer) *Consider any program represented by command stack cs and initial state* ρ, μ, *with a sequential execution* $\langle cs, \rho, \tau, \mu \rangle \Downarrow_O \langle [], \rho', \tau', \mu' \rangle$ *with arbitrary* τ, *where some memory property P holds for* $\texttt{PAddr}(O)$. *Then, for any complete directive schedule D on* $C = \langle [], cs, \rho, \emptyset, \mu \rangle$ *with* $C \downarrow^D_{O'} \langle [], [], \rho', \tau_\emptyset, \mu' \rangle$, *P also holds for* $\texttt{PAddr}(O')$. *Therefore, by clearing the initial page tables, any memory property can be 'transferred' into the speculative realm.* ⌟

6.3.1 Integrity of Speculative Execution

We will now direct our attention to the integrity of speculative execution. In other words, we seek to prove that speculative execution can only access memory available according to the page tables τ.

Proposition 6.3.3 (Integrity) *For any configuration $C = \langle is, cs, \rho, \tau, \mu \rangle$, there does not exist a valid directive d on C with an observation trace O containing* **touch**(pa) *where $pa \notin \mathrm{Im}(\tau)$ (image of τ). In other words, under speculative semantics it is not possible to access unmapped memory.*

Proof
To prove this property, we analyse all rules in the speculative semantics that emit a **touch** observation. These rules are [Execute-Load-Predict], [Execute-Load-Present-Mispredict], and [Retire-Store]. We see that in each of these rules, the pa that is emitted in **touch** is determined as $pa = \tau(a)$. This immediately means that, in fact, $pa \in \mathrm{Im}(\tau)$. $\square$

6.3.2 Empty Page Tables

Next, we will analyse the behaviour of programs when clearing the page tables before execution. Clearing the page tables will be the 'trick' that allows us to safely run programs under speculative semantics. We will show a parallel between speculative and sequential semantics.

Proposition 6.3.4 *For any configuration $C = \langle is, cs, \rho, \emptyset, \mu \rangle$ where the page tables are empty (i.e. the domain of τ is empty), and for any architectural configuration $C' = \langle [], cs', \rho', \tau', \mu' \rangle$ for which there exists a directive schedule D and observation trace O such that $C \Downarrow_O^D C'$, it holds that* `PAddr`$(O) = \mathrm{Im}(\tau')$. *In other words, the set of accessed memory addresses is exactly the set of addresses mapped in the page tables after execution.*

Proof
To show `PAddr`$(O) = \mathrm{Im}(\tau')$, we will show inclusion both ways:

`PAddr`$(O) \subseteq \mathrm{Im}(\tau')$ During execution, our page tables expand from $\emptyset$ to τ'. Since only [Map-Address] can change τ and since this rule will only expand the map, we can clearly see that $\emptyset \subseteq \tau_1 \subseteq \cdots \subseteq \tau_i \subseteq \cdots \subseteq \tau'$ (in terms of their images). Here, τ_i corresponds to the configuration after executing the ith directive from D. Note that while [Map-Address] might in theory *change* an entry in τ, in practice `FAULT` instructions are only generated when a is not yet mapped in τ (by rules [Execute-Load-Miss] and [Execute-Store-Miss]). Furthermore, the action of [Map-Address] cannot be rolled back - this can be seen both from the fact that the rule only operates on the first item in the reorder buffer, and because all other rollback rules ([Execute-Assert-Fail] and [Execute-Load-Present-Mispredict]) do not touch the page tables.

Since our directive schedule D is valid, from Proposition 6.3.3, we know that O does not contain any **touch**(pa) where $pa \notin \mathrm{Im}(\tau_i) \subseteq \mathrm{Im}(\tau')$ where τ_i corresponds to the intermediate configuration associated with the emission of the **touch**. We conclude that `PAddr`$(O) \subseteq \mathrm{Im}(\tau')$.

$\mathrm{Im}(\tau') \subseteq$ `PAddr`(O) For the other direction, we again observe that our page tables expend from $\emptyset$ to τ', and that we have the 'hierarchy' of τ_i. We see that every expansion of τ by [Map-Address] puts an instruction back in the reorder buffer. Since we consider an architectural C', we know that this instruction must be executed and retired – it can't be erased as part of a **rollback** since it will be the first element in the reorder buffer.

The instruction that is put back in the reorder buffer is determined by [Execute-Load-Miss] and [Execute-Store-Miss]. In the latter case, the `STORE` instruction is for the exact same address as the fault, so the address that is mapped will also be accessed by [Retire-Store] afterwards. Similarly, in the load case, the `LOAD` instruction is for the exact same address as the fault as well. When executing the `LOAD`, this address is accessed by either [Execute-Load-Predict] or [Execute-Load-Present-Mispredict].

We conclude that every map, and thus every expansion of the page tables, will be followed by an access to that same address, before reaching C'. Therefore, since we start with empty page tables, every address in the page tables is also being accessed: $\mathrm{Im}(\tau') \subseteq \texttt{PAddr}(O)$. □

Note that in particular, Proposition 6.3.4 holds for complete schedules D and final configurations C_f as a special case of the valid schedules and architectural configurations. For this case, the sequential semantics turns out to have the same behaviour:

Proposition 6.3.5 *For any sequential state $S = \langle cs, \rho, \emptyset, \mu \rangle$ for which there exists a final state $S_f = \langle [], \rho', \tau', \mu' \rangle$ and observation trace O such that $S \Downarrow_O S_f$, it holds that* $\texttt{PAddr}(O) = \mathrm{Im}(\tau')$.

Proof
We again show equivalence by showing inclusion both ways:

$\texttt{PAddr}(O) \subseteq \mathrm{Im}(\tau')$ Similar to the speculative case, we will consider the sequential rules that produce a **touch** observation. These are the pointer read and write, and array read and write rules, each for the present and miss case – 8 rules in total. For all the 'present' rules, we see that the emitted pa is determined as $pa = \tau(a)$, and so we are sure that $pa \in \mathrm{Im}(\tau_i)$ for the intermediate state S_i. Similarly, for all the 'miss' rules, pa is determined as $pa = M(a)$, after which τ is updated to $\tau[a \mapsto pa]$ - so again we are sure that $pa \in \mathrm{Im}(\tau_i)$.

As for the speculative case, we can build a 'hierarchy' in page tables during execution, where $\emptyset \subseteq \tau_1 \subseteq \cdots \subseteq \tau_i \subseteq \cdots \subseteq \tau'$ in terms of images. Therefore, every $pa \in \mathrm{Im}(\tau')$, so we conclude that $\texttt{PAddr}(O) \subseteq \mathrm{Im}(\tau')$.

$\mathrm{Im}(\tau') \subseteq \texttt{PAddr}(O)$ For the other direction, we consider all rules that update the page tables: the four 'miss' variants of the pointer and array reads and writes. An update to the page tables is only ever additive: in each of these rules, the update is $\tau[a \mapsto pa]$, where it was established that $\tau(a) = \perp$. Next, we observe that each of these rules will also emit a **touch** for the pa that was just mapped.

Since we start execution with empty page tables, and since every addition is also being accessed during execution, we must conclude that $\mathrm{Im}(\tau') \subseteq \texttt{PAddr}(O)$. □

By now, we have shown that *given empty page tables*, both speculative and sequential semantics access the same set of addresses: exactly the set that is being mapped into the page tables. This is reassuring and further highlights the consistency between sequential and speculative execution.

Note that we could not have reached the same result by simply invoking our earlier consistency result: Theorem 2 does not consider equivalence of, or any other relation between the observation traces O for both sequential and speculative semantics.

6.3.3 Arbitrary Initial Page Tables

Next, we show that sequential execution does not need to start with empty page tables to limit its set of accessed addresses; the set of accessed addresses is independent of the starting τ.

Lemma 6.3.6 *Consider any sequential state $S = \langle cs, \rho, \emptyset, \mu \rangle$ for which there exists a final state $S_f = \langle [], \rho', \tau', \mu' \rangle$ and observation trace O such that $S \Downarrow_O S_f$. For any τ, the state $S_\tau = \langle cs, \rho, \tau, \mu \rangle$ satisfies $S_\tau \Downarrow_O \langle [], \rho, \tau'_\tau, \mu' \rangle$ for the same observation trace O. In addition, it holds that $\tau'_\tau = \tau \cup \tau'$.*

Proof
This property can be seen easily by inspecting the rules for sequential execution. For each pointer or array read or write, two rules can apply: a 'present' rule or a 'miss' rule. The latter will update the page tables to include the new address, but apart from that, the behaviour of both rules is identical: the same modifications to ρ and μ take place, and the same observation **touch**(pa) is emitted. Therefore, O is the same for all these executions and starting states S_τ.

In Proposition 6.3.5, the argument for $\texttt{PAddr}(O) \subseteq \text{Im}(\tau')$ was independent of the assumption that computation started with empty page tables. Therefore, $\texttt{PAddr}(O) \subseteq \text{Im}(\tau'_\tau)$ as well. In fact, from Proposition 6.3.5 we also know that $\texttt{PAddr}(O) = \text{Im}(\tau')$ since S has empty page tables: $\text{Im}(\tau') \subseteq \text{Im}(\tau'_\tau)$.

Furthermore, as observed before, all 'miss' rules are only additive: existing entries in the page tables are not changed. Therefore, also $\text{Im}(\tau) \subseteq \text{Im}(\tau'_\tau)$. Since page table entries are only added one-by-one, and since all additions during execution of S_τ are in τ' as well, it must be the case that $\tau'_\tau = \tau \cup \tau'$. □

Note that this property can clearly not apply in the speculative realm: there are exponentially many complete directive schedules that would perform the execution from C to C_f. Each of these directives can come with completely different observation traces: consider rollbacks and speculative memory accesses that turn out to be incorrect, which is the entire problem we're trying to address.

6.3.4 Equivalence

We will now achieve our objective: to show that under speculative semantics, when starting with empty page tables, the same set of memory addresses is accessed as under sequential execution with arbitrary page tables.

Theorem 3 (Memory Access Equivalence)
Consider any program represented by command stack cs and initial state ρ, μ with a sequential execution $\langle cs, \rho, \tau, \mu\rangle \Downarrow_O \langle [], \rho', \tau', \mu'\rangle$ with arbitrary τ. Then, any complete directive schedule D on $C = \langle [], cs, \rho, \emptyset, \mu\rangle$ with $C \downarrow^D_{O'} \langle [], [], \rho', \tau_\emptyset, \mu'\rangle$ satisfies the property that $\texttt{PAddr}(O) = \texttt{PAddr}(O')$. *In other words, the set of touched physical addresses for sequential execution is the same as for speculative execution.*

Proof
With the work done in previous sections, we can simply chain our arguments. As seen in Lemma 6.3.6, the particular sequential execution $\langle cs, \rho, \emptyset, \mu\rangle \Downarrow_O \langle [], \rho', \tau_\emptyset, \mu'\rangle$ will have an identical observation trace O, and therefore identical $\texttt{PAddr}(O)$.

By Theorem 2, we know that any complete directive schedule on $C = \langle [], cs, \rho, \emptyset, \mu\rangle$ will yield the same $\langle [], [], \rho', \tau_\emptyset, \mu'\rangle$, but with potentially different O'.

Lastly, from Proposition 6.3.5, we know that $\texttt{PAddr}(O) = \text{Im}(\tau_\emptyset)$. In addition, from Proposition 6.3.4, we also know that $\texttt{PAddr}(O') = \text{Im}(\tau_\emptyset)$. We conclude that $\texttt{PAddr}(O) = \texttt{PAddr}(O')$, as desired. □

Chapter 7

Conclusions and Discussions

In this MSc thesis, we have formally shown the effectiveness of *PageZero* against Spectre v1. We have introduced a formal model for reasoning about speculative execution, and used it to prove that when clearing page tables, any speculative execution of a program accesses the exact same set of memory addresses as a traditional sequential execution. We showed this for a very strong attacker model, with deep insight into and control over CPU behaviour.

Model for Speculative Execution

While the current model only demonstrates effectiveness of *PageZero* against Spectre v1, other speculative execution vulnerabilities are easy to incorporate. In particular, Spectre v2 should be easy to implement by adding an indirect jump to the meta programming language and introducing appropriate reduction rules.

Vulnerabilities like RIDL [4], LVI [5], ZombieLoad [3], and Fallout [7] are also easy to model, but need further thought in terms of *what* the Byzantine data could be, and how the 'security' claim needs to be adapted.

The current model also lacks any notion of a process, a thread, or a core. While the current approach suffices to model and protect a kernel, the model is not immediately fit for user-space processes in a multi-process and multi-user environment. While such additions would make the model more powerful, it will also get increasingly more complicated as the surface for interactions between parts of the model (kernel and user space, multiple processes) grows quickly. It might be more feasible to make dedicated models for different use cases, focusing on the important aspects for each use case.

PageZero

While we have only proved effectiveness against Spectre v1, in principle, *PageZero* also protects against most other speculative execution vulnerabilities: the security of *PageZero* hinges on the fact that even speculative execution cannot access unmapped memory, and that cross-core attacks are not possible. Vulnerabilities that break this condition, like CrossTalk [9] or Meltdown [2], are not mitigated by this approach.

In the other MSc thesis, we show a prototype implementation of *PageZero* in the Linux kernel [21]. Unfortunately, fully clearing the page tables is inherently not possible in practice, as we quickly hit a chicken-and-egg problem: the page fault handler itself also has to be mapped in the page tables. Therefore, specific regions of memory have to be made available on an allow-list basis, after which the page fault handler can handle the rest of the address space.

Fundamentally, any implementation will face this trade-off between security and implementation complexity or performance. Fortunately, a significant portion of the practical attack surface can already be eliminated by clearing only the direct map region of the kernel page tables. This is also the approach we take for our prototype implementation. Future research might make the implementation more fine-grained, by clearing more regions of the kernel page tables.

Nevertheless, we manage to show that a 'lightweight' version of *PageZero*, achieving a great degree of protection against speculative execution attacks, can be implemented in practice. The implementation requires minimal code changes in the KVM subsystem of Linux, and has relatively low runtime overhead compared to enabling all relevant mitigations. This shows that *PageZero* is a promising, more comprehensive approach to mitigating speculative execution attacks.

Appendix A

Complete Syntax

This is a full and complete listing of all syntax used in the thesis.

Variables (`Var`):	x
Integers (`Num`):	$n \in \mathbb{N}$
Booleans (`Bool`):	$b \in \{\texttt{true}, \texttt{false}\}$
Arrays (`Array`):	$xs ::= \langle a, n \rangle$ (base address and length)
Virtual Memory Addresses (`Addr`):	$a \in \mathbb{N}$
Physical Memory Addresses (`PAddr`):	$pa \in \mathbb{N}$
Values (`Val`):	$v ::= n \mid b \mid a \mid xs$
Predicted Values (`PVal`):	$pv ::= v \mid \bot$
Expressions (`Expr`):	$e ::= v \mid x \mid e + e \mid e < e \mid e \otimes e \mid e \, ? \, e : e \mid \mathsf{len}(e) \mid \mathsf{base}(e)$
Right-hand Sides:	$r ::= e \mid * \, e \mid xs \, [\, e \,]$
Commands (`Comm`):	$c ::= \texttt{skip} \mid x \texttt{ := } r \mid \texttt{*}\, e \texttt{ := } e \mid xs\texttt{[}e\texttt{] := } e$
	$\mid \texttt{if } e \texttt{ then } cs \texttt{ else } cs \mid \texttt{while } e \texttt{ do } cs$
Command Stacks (`[Comm]`):	$cs ::= c : cs \mid []$
Instruction Labels:	ℓ
Instructions (`Instr`):	$i ::= \mathsf{NOP} \mid \mathsf{MOV}\langle x, e \rangle \mid \mathsf{LOAD}\langle x, e, pv, cs \rangle \mid \mathsf{STORE}\langle e_a, e_v, cs \rangle$
	$\mid \mathsf{ASSERT}\langle e, b, cs \rangle \mid \mathsf{FAULT}\langle a, i, cs \rangle$
Reorder Buffers (`[Instr]`):	$is ::= i_\ell : is \mid []$
Directives:	$d ::= \textbf{fetch} \mid \textbf{fetch}\, b \mid \textbf{exec}\, n \mid \textbf{retire} \mid \textbf{map}$
Schedule:	$D ::= d : D \mid []$
Observations:	$o ::= \textbf{touch}(pa) \mid \textbf{rollback}(is)$
Observation Trace:	$O ::= o : O \mid []$
Register Maps (`Reg`):	$\rho \in \texttt{Var} \to \texttt{Val}$
Predicted Register Maps (`PReg`):	$\tilde{\rho} \in \texttt{Var} \to \texttt{PVal}$
Page Tables:	$\tau \in \texttt{Addr} \to (\texttt{PAddr} \cup \bot)$
True Page Mapping:	$M \in \texttt{Addr} \to \texttt{PAddr}$
Physical Memory Stores:	$\mu \in \texttt{PAddr} \to \texttt{Val}$
Configurations (`Conf`):	$C ::= \langle is, cs, \rho, \tau, \mu \rangle$
Sequential Program State (`SeqState`):	$S ::= \langle cs, \rho, \tau, \mu \rangle$

Appendix B

Bibliography

[1] Paul Kocher et al. 'Spectre Attacks: Exploiting Speculative Execution'. In: *Communications of the ACM* 63.7 (18th June 2020), pp. 93–101. ISSN: 0001-0782. DOI: 10.1145/3399742. URL: https://dl.acm.org/doi/10.1145/3399742 (visited on 31/03/2024).

[2] Moritz Lipp et al. 'Meltdown: Reading Kernel Memory from User Space'. In: *Communications of the ACM* 63.6 (21st May 2020), pp. 46–56. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3357033. URL: https://dl.acm.org/doi/10.1145/3357033 (visited on 31/03/2024).

[3] Michael Schwarz et al. 'ZombieLoad: Cross-Privilege-Boundary Data Sampling'. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19: 2019 ACM SIGSAC Conference on Computer and Communications Security. London United Kingdom: ACM, 6th Nov. 2019, pp. 753–768. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3354252. URL: https://dl.acm.org/doi/10.1145/3319535.3354252 (visited on 24/09/2024).

[4] Stephan Van Schaik et al. 'RIDL: Rogue In-Flight Data Load'. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE, May 2019, pp. 88–105. ISBN: 978-1-5386-6660-9. DOI: 10.1109/SP.2019.00087. URL: https://ieeexplore.ieee.org/document/8835281/ (visited on 24/09/2024).

[5] Jo Van Bulck et al. 'LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection'. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE, May 2020, pp. 54–72. ISBN: 978-1-72813-497-0. DOI: 10.1109/SP40000.2020.00089. URL: https://ieeexplore.ieee.org/document/9152763/ (visited on 24/09/2024).

[6] Jo Van Bulck et al. 'FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution'. In: ().

[7] Claudio Canella et al. 'Fallout: Leaking Data on Meltdown-resistant CPUs'. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19: 2019 ACM SIGSAC Conference on Computer and Communications Security. London United Kingdom: ACM, 6th Nov. 2019, pp. 769–784. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3363219. URL: https://dl.acm.org/doi/10.1145/3319535.3363219 (visited on 24/09/2024).

[8] Claudio Canella, Michael Schwarz and Moritz Lipp. 'A Systematic Evaluation of Transient Execution Attacks and Defenses'. In: ().

[9] Hany Ragab et al. ‘CrossTalk: Speculative Data Leaks Across Cores Are Real’. In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE, May 2021, pp. 1852–1867. ISBN: 978-1-72818-934-5. DOI: `10.1109/SP40001.2021.00020`. URL: `https://ieeexplore.ieee.org/document/9519489/` (visited on 29/09/2024).

[10] Michael Larabel. *The Brutal Performance Impact From Mitigating The LVI Vulnerability*. Phoronix. 12th Mar. 2020. URL: `https://www.phoronix.com/review/lvi-attack-perf` (visited on 24/09/2024).

[11] Jonathan Behrens, Adam Belay and M. Frans Kaashoek. ‘Performance Evolution of Mitigating Transient Execution Attacks’. In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys ’22: Seventeenth European Conference on Computer Systems. Rennes France: ACM, 28th Mar. 2022, pp. 251–265. ISBN: 978-1-4503-9162-7. DOI: `10.1145/3492321.3519559`. URL: `https://dl.acm.org/doi/10.1145/3492321.3519559` (visited on 24/09/2024).

[12] Michael Larabel. *In Light Of Spectre BHI, The Performance Impact For Retpolines On Modern Intel CPUs*. Phoronix. 10th Mar. 2022. URL: `https://www.phoronix.com/review/spectre-bhi-retpoline` (visited on 24/09/2024).

[13] Brian Johannesmeyer et al. ‘Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel’. In: *Proceedings 2022 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA, USA: Internet Society, 2022. ISBN: 978-1-891562-74-7. DOI: `10.14722/ndss.2022.24221`. URL: `https://www.ndss-symposium.org/wp-content/uploads/2022-221-paper.pdf` (visited on 24/09/2024).

[14] Sander Wiebing et al. ‘InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre V2’. In: ().

[15] Mathé Hertogh et al. ‘Quarantine: Mitigating Transient Execution Attacks with Physical Domain Isolation’. In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID 2023: The 26th International Symposium on Research in Attacks, Intrusions and Defenses. Hong Kong China: ACM, 16th Oct. 2023, pp. 207–221. ISBN: 9798400707650. DOI: `10.1145/3607199.3607248`. URL: `https://dl.acm.org/doi/10.1145/3607199.3607248` (visited on 22/01/2024).

[16] Hongyan Xia et al. ‘A Secret-Free Hypervisor: Rethinking Isolation in the Age of Speculative Vulnerabilities’. In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022 IEEE Symposium on Security and Privacy (SP). May 2022, pp. 370–385. DOI: `10.1109/SP46214.2022.9833726`. URL: `https://ieeexplore.ieee.org/document/9833726` (visited on 22/01/2024).

[17] AMD. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. White Paper. Jan. 2020.

[18] Jonathan Behrens et al. ‘Efficiently Mitigating Transient Execution Attacks Using the Unmapped Speculation Contract’. In: ().

[19] Ofir Weisse et al. ‘NDA: Preventing Speculative Execution Attacks at Their Source’. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’52: The 52nd Annual IEEE/ACM International Symposium on Microarchitecture. Columbus OH USA: ACM, 12th Oct. 2019, pp. 572–586. ISBN: 978-1-4503-6938-1. DOI: `10.1145/3352460.3358306`. URL: `https://dl.acm.org/doi/10.1145/3352460.3358306` (visited on 25/09/2024).

[20] Qian Ge et al. ‘Time Protection: The Missing OS Abstraction’. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys ’19: Fourteenth EuroSys Conference 2019.

Dresden Germany: ACM, 25th Mar. 2019, pp. 1–17. ISBN: 978-1-4503-6281-8. DOI: 10.1145/3302424.3303976. URL: https://dl.acm.org/doi/10.1145/3302424.3303976 (visited on 18/07/2024).

[21] Floris Westerman. ‘PageZero: Mitigating Speculative Execution Attacks by Clearing Page Tables’. MSc Thesis. Rijksuniversiteit Groningen, forthcoming.

[22] Marco Vassena et al. ‘Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade’. In: *Proceedings of the ACM on Programming Languages* 5 (POPL 4th Jan. 2021), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3434330. URL: https://dl.acm.org/doi/10.1145/3434330 (visited on 13/02/2024).

[23] Andrew S. Tanenbaum and Todd Austin. *Structured Computer Organization*. 6th ed. Boston: Pearson, 2013. 775 pp. ISBN: 978-0-13-291652-3.

[24] Vladimir Mironov et al. ‘Performance Evaluation of the Intel Optane DC Memory With Scientific Benchmarks’. In: *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC). Nov. 2019, pp. 1–6. DOI: 10.1109/MCHPC49590.2019.00008. URL: https://ieeexplore.ieee.org/document/8946136 (visited on 17/09/2024).

[25] Bevin Brett. *Memory Performance in a Nutshell*. Intel. 6th June 2016. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html (visited on 17/09/2024).

[26] Kernel Development Community. *Memory Management*. The Linux Kernel Documentation. URL: https://www.kernel.org/doc/html/v6.6/arch/x86/x86_64/mm.html (visited on 24/09/2024).

[27] AMD. *AMD64 Architecture Programmer’s Manual, Volume 2: System Programming, 24593*. June 2023. URL: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf.

[28] Anand Lal Shimpi. *Intel’s Haswell Architecture Analyzed: Building a New PC and a New Intel*. AnandTech. URL: https://www.anandtech.com/show/6355/intels-haswell-architecture (visited on 18/09/2024).

[29] Anton Ertl. *Reorder Buffer Size of Various CPUs*. URL: http://www.complang.tuwien.ac.at/anton/robsize/ (visited on 18/09/2024).

[30] Georgi Gerganov. *Keytap3: Acoustic Keyboard Eavesdropping*. C++ and stuff. 25th Apr. 2022. URL: https://keytap3.ggerganov.com/ (visited on 19/09/2024).