UNIVERSITÄT BONN

Transient Execution Emulator

Meltdown and Spectre Behind the Scenes

Lab Report

by

Felix Betke
3099892

Lennart Hein
3012079

Melina Hoffmann
2824792

Jan-Niklas Sohn
3121407

submitted to

Rheinische Friedrich-Wilhelms-Universität Bonn

Institut für Informatik IV

Arbeitsgruppe für IT-Sicherheit

in degree course

Computer Science (M.Sc.)

Supervisor: Prof. Dr. Michael Maier
University of Bonn

Sponsor: Dr. Felix Jonathan Boes
University of Bonn

Bonn, 31. March 2022

ABSTRACT

The original transient execution attacks, Meltdown and Spectre, allow attackers to read arbitrary memory addresses and temporarily alter the program flow of victim processes. Since their publication, numerous patches have been implemented in hardware or software. This makes it difficult to experiment with the original variants of the attacks for educational purposes. Additionally, physical CPUs provide almost no mechanisms for introspection. We design and implement a vulnerable CPU emulator that can be attacked using select variants of Meltdown and Spectre, and provides insight into the inner workings of the CPU during such attacks. Further, the emulator includes optional mitigations for either attack and allows its users to implement their own using microprograms. We demonstrate its functionality, supply example programs and are confident our emulator can help users better understand the relevant attacks.

CONTENTS

1 INTRODUCTION

Felix Betke

As an advancement over older processors, modern Intel CPUs implement a number of optimization techniques that increase their efficiency. One of which is the concept of out-of-order execution, which takes advantage of the mutual independence of instructions that would normally be executed sequentially. A second optimization technique is called speculative execution and involves the prediction of whether or not a given branch is taken. With either technique, the CPU might encounter cases where the current CPU state must be rolled back to a previous one to ensure correct execution. For out-of-order execution this happens when an instruction raises an exception (e.g. accessing inaccessible memory). For speculative execution, this happens when a branch is mispredicted. An initiated rollback allows some instructions that are currently being executed (in-flight) to continue execution for a short amount of time.

Even though rollbacks are meant to make sure in-flight instructions do not cause any lasting side effects on the microarchitectural state of the CPU, it was discovered that they can affect the contents of caches and other buffers. The disclosure of both Meltdown [Lip+18] and Spectre [Koc+19] in early 2018 introduced a whole family of vulnerabilities that take advantage of both out-of-order and speculative execution to leak secrets over the processor's caches, or temporarily alter the program flow of other processes. And while the performance losses introduced by software and hardware mitigations are measurable, neither family of vulnerabilities can be exploited as originally presented on a fully patched system. As a result, however, the process of trying to exploit one of the vulnerabilities for the sake of learning how they work in detail can be challenging. Apart from the software, which may be obtained by installing an older version of an operating system that does not implement any mitigations, one must also make sure their CPU is affected by the vulnerabilities and has not yet received any relevant microcode updates from Intel. Often times, this means that a user's personal computer does not meet these requirements.

Building upon what we learned in the Side Channel Attacks (SCA) lecture [Boe21], we design and implement a graphical CPU emulator that supports single-step, out-of-order, and speculative execution and is specifically designed to be vulnerable to select variants of Spectre and Meltdown. While it is a simplification in comparison to real hardware, the emulator provides a noise-free environment that allows its users to gain a better understanding of how exactly the two vulnerabilities work and can be exploited. Furthermore, the user may experiment with (ineffective) mitigations or implement their own microcode-programs that are executed once rollbacks are completed. We supply example programs that can be run by the emulator and serve both as an entry point for the user as well as the basis of our evaluation.

Firstly, chapter 2 briefly gives an overview of relevant components of vulnerable Intel CPUs, presents the concepts of out-of-order and speculative execution in greater detail, and introduces both Meltdown and Spectre to which the resulting emulator is designed to be vulnerable. Secondly, chapter 3

further describes the target audience of the emulator and to which variants of the vulnerabilities the emulator is vulnerable, while chapter 4 documents the implementation of the emulator by describing each main component. Additionally, it contains an overview of the default configuration file and the set of ISA instructions available to the user. Furthermore, chapter 5 explores the graphical user interface by defining the goals its design is supposed to accomplish and documenting design choices and important features. Chapter 6 provides a demonstration of the emulator, which includes example programs of Meltdown and Spectre attacks, and determines how effective the implemented mitigations are. Finally, chapter 7 summarizes the other chapters, and reflects on whether our goals are reached. It also provides some limitations regarding the emulator and ideas for future improvements.

2 Background

Felix Betke

This chapter briefly covers the theoretical background needed to use the presented emulator. The reader is assumed to have an understanding of elementary CPU concepts, such as pipelining and caching. Students attending the Side Channel Attacks (SCA) [Boe21] lecture fall into this category. Firstly, sec. 2.1 introduces the three main components of a CPU, while sections 2.2 and 2.3 explain the optimization techniques out-of-order and speculative execution, respectively. Lastly, sec. 2.4 gives a short overview of the Meltdown and Spectre vulnerabilities relevant for the emulator and presents some mitigations.

2.1 CPU

Felix Betke

A CPU consists of a frontend, an execution engine, and a memory subsystem. As per Intel's Skylake architecture [Wikc], the frontend fetches the instructions, maintains a queue of instructions that are to be executed, and decodes them into less complex microinstructions, which are then communicated to the execution engine. Additionally, it is responsible for the branch prediction (see sec. 2.3). [Gru20, pp. 15-16]

The execution engine consists of multiple sets of execution units, each set being responsible for a specific type of microinstruction, such as loads, stores, or arithmetic. Further, the scheduler allows the execution units to work on independent instructions in parallel, while the reorder buffer makes sure that instructions retire in the correct order. A common data bus (CDB) connects the reorder buffer, scheduler, and execution units. Its purpose is further described in sec. 2.2. [Gru20, pp. 15-16]

Lastly, the memory subsystem handles all memory accesses of the execution units by maintaining caches and ensuring data is fetched from higher level caches or DRAM if needed [Gru20, p. 409]. Most importantly, requests for data can be served faster if this data is present in the cache as opposed to requests for data that must be fetched from memory first. A visualization of the aforementioned components can be found in fig. 1.

2.2 Out-of-Order Execution

Felix Betke

As the name implies, out-of-order execution refers to the idea of executing instructions in a different order than the one in which they are given [Gru20, p. 15]. With multiple execution units that run in parallel (as described in sec. 2.1), CPUs can take advantage of mutually independent instructions and execute them at the same time.

The basic realization of this concept is provided by Tomasulo's algorithm [Tom67]. It introduces two new components, the first of which is the reservation station, which collects the operands of

Figure 1: *Simplified overview an Intel Skylake CPU [Gru20, fig. 2.1]. For the memory subsystem, detailed knowledge of the load and store buffers, as well as the TLBs, is not required. The same applies to the allocation queue of the frontend.*

instructions until they are ready to be executed by the execution units. Crucially, the corresponding execution units do not stall until all operands are present and can instead compute other instructions whose operands are already available. Once all operands are available, an execution unit executes the instruction. The second component of Tomasulo's algorithm is the Common Data Bus (CDB), which connects all reservation stations and execution units. Whenever a result is computed by an execution unit, the result is broadcast onto the CDB and thus made available to all reservation stations that are waiting for it. This important step ensures that results are not needed to be written to registers first, just to be read again by other instructions that need them as operands.

According to the original design by Tomasulo, each set of execution units needed its own reservation station. However, Intel's latest CPUs use a single unified reservation station, called scheduler, that handles all types of instructions, rather than just one [Tom67] [Wikc] [Gru20, p. 228]. This can also be seen in fig. 1.

2.3 SPECULATIVE EXECUTION

Felix Betke

Speculative execution primarily allows a CPU to predict the outcome of branch instructions, but it is also used to predict the address at which execution will be resumed after a return instruction. This prevents stalling while waiting for the instruction that determines which branch is taken to finish. With speculative execution, rollbacks are needed in cases where branches were mispredicted. [Gru20, pp. 17-18, 53]

We limit speculative execution to branches in the context of our emulator. To predict the outcome of branch instructions, CPUs include a branch prediction unit (BPU) [Gru20, pp. 16, 228]. While available in different configurations, many modern CPUs record the most recent outcomes of a branch with a counter [Gru20, p. 228] that is either incremented if the branch is taken, or decremented.

2.4 MELTDOWN AND SPECTRE

Felix Betke

Meltdown [Lip+18] abuses out-of-order execution to leak data from memory addresses that are normally inaccessible to the attacker, while Spectre [Koc+19] temporarily alters the program flow of victim processes. Both are made possible by the fact that rollbacks do not affect caches, which allows attackers to retrieve information from them after the rollbacks.

2.4.1 MELTDOWN

On a high level, Meltdown [Lip+18] works by encoding temporarily available data that was illegally read into the cache to retrieve it once the rollback has completed. What enables Meltdown is a small time window between an invalid memory access and the raising of an exception [Lip+18].

The Meltdown-US-L1 variant of Meltdown [Lip+18], which is almost identical to the one to which the presented emulator is vulnerable, works by accessing a memory address for which the attacker has no permission. Firstly, the attacker allocates an oracle array and ensures none of its entries are present in the cache. Upon loading the contents of the inaccessible memory location into a register,

the attacker uses the value to access the array at a specific offset. When the rollback caused by the access violation has completed, the attacker measures the access time to each of the array entries to determine which one has been accessed. For the attack to work, it is important that the data to be stolen is cached when the attacker performs the illegal read. While there are numerous other variants of Meltdown that differ from the basic variant mainly by how they force an exception and from which microarchitectural buffer they leak data, these types of attacks are out of scope.

2.4.2 SPECTRE

Spectre [Koc+19], on the other hand, relies on the CPU mispredicting a branch and transiently executing an incorrect execution path. This misprediction in a victim process can be induced by maliciously training the BPU. Depending on the instructions that are wrongfully executed by the victim, traces may later be found in the processor's cache. Similarly to Meltdown, different variants of Spectre exist. The presented emulator is vulnerable to the original Spectre v1, which takes advantage of the CPU mispredicting the outcome of branch instructions.

2.4.3 MITIGATIONS

Different mitigations for Meltdown exist in software or hardware, some of which are implemented in the emulator to allow users to better understand how they work, experiment with them, and determine their effectiveness. A first, rather simple, mitigation for Meltdown is to disable out-of-order execution [Lip+18], which would completely prevent an attacker from encoding the inaccessible data into the cache before the time window closes. Later revisions of Intel's architectures introduced further mitigations. Although undocumented by Intel, researchers suspect the processor still performs the illegal read, but zeros out the data that is given to dependent instructions before raising an exception [Can+20]. The emulator implements this mitigation, which may be enabled by the user. Other mitigations deployed by operating systems, such as KPTI (or KAISER) [Lip+18], are highly effective, but out of scope, as there exists no operating system for our emulator.

Unlike Meltdown, which is a bug, Spectre appears to be a design flaw. While one might argue that transiently computing on real values after it is already known an exception will occur is a bug, the behavior abused by Spectre is a direct consequence of speculative execution (sec. 2.3). As a result, an effective yet inefficient mitigation is to simply disable speculative execution [Koc+19]. Alternatively, Intel recommends potential victims of Spectre v1 to use an "lfence" instruction where appropriate, which ensures prior load instructions retire before continuing, thus effectively disabling speculative execution for certain parts of an application [Int18]. This mitigation could either be implemented by a compiler, or the developer. A fence instruction is provided as part of the emulator's instruction set (see sec. 4.4).

An additional mitigation that works against both Meltdown and Spectre is to flush the entire cache after a rollback. While also inefficient, it completely prevents the retrieval of otherwise inaccessible data after the transient execution phase of the attacks. This mitigation can be realized using the emulator by a sequence of instructions (microprogram) that are executed after each rollback (see sec. 4.5).

3 Specification of Our Task

Lenni Hein,
Felix Betke

This chapter briefly specifies our task and defines our target audience. As motivated in the introduction (sec. 1), exploiting Meltdown or Spectre for educational purposes on personal hardware can be challenging. Therefore, our goal is to develop a CPU emulator that is vulnerable to select variants of the two vulnerabilities and runs on most operating systems and architectures. To achieve this, it must support out-of-order and speculative execution.

Our target audience includes anyone that wants to gain a better understanding of these vulnerabilities. We assume the user of our emulator is attending the SCA lecture [Boe21] (or similar) or already knows about some of the theoretical background regarding transient execution attacks.

While numerous variants of Meltdown and Spectre exist, we limit the scope of the emulator to two original attacks. For Meltdown, users can run a slightly modified version of the Meltdown-US-L1 attack. This is the version that was published as the original Meltdown vulnerability [Lip+18] and the one presented to students first in the SCA lecture [Boe21]. Given the default configuration of our emulator, we allow users to steal secrets even if they are currently not in the cache. This modification does not impact the way Meltdown works and should simplify the users' programs. As mentioned in sec. 4.5, the emulator can be configured to more closely model the real Meltdown-US-L1 attack. For Spectre, the emulator is vulnerable to Spectre v1, which is one of the original variants published [Koc+19] and, again, first to be introduced in the SCA lecture [Boe21]. The emulator should implement Intel's microcode mitigation (see sec. 2.4.3) and allow users to write their own microprograms that are executed after rollbacks. The mitigation that flushes the entire cache after a rollback should be supplied as a microprogram.

A graphical user interface is required to visualize the current state of the components of the emulator. Additionally, the UI should allow users to step through the execution one step at a time and also to go back. Users should be able to use the UI to manipulate some parts of the running program, such as writing to memory.

4 CPU Emulator

Melina Hoffmann

In this chapter, we introduce the backend of our emulator program. It contains the elements of our program that emulate actual CPU components. Our emulator is based on information about modern real life CPUs, especially the x86 Intel Skylake architecture [Wikc].

In our program, we model the distinct components of a real life CPU with a modular setup making use of the object oriented functionalities of `Python3`. Breaking up the source code into individual CPU components makes it easier to understand and maintain. Additionally, we have made simplifications and modifications in comparison to a real life CPU, so the emulator is as clear and easy to understand as possible while still implementing a functioning out-of-order execution and providing the necessary functionality for Meltdown and Spectre attacks.

In this chapter we firstly introduce the components of our CPU emulator, how they work and interact and which part of a real life CPU they emulate in sec. 4.1. Then, in sec. 4.2, we explain how our emulator provides out of order execution and how it may differ from the Tomasulo algorithm introduced in sec. 2.2. Subsequently, we show how we implement rollbacks and exception handling, which are necessary for Meltdown and Spectre attacks, in sec. 4.3. Then we give an overview over the available instruction set architecture in sec. 4.4. Lastly we show how our emulator can be adapted for different demonstrations and attacks via a config file without changing the source code in sec. 4.5.

4.1 CPU Components and Our Models

Jan-Niklas Sohn

This section describes the individual components of our CPU emulator and various interactions between them. Each component is modelled after one or multiple components found in typical modern x86 CPUs. The main CPU component described in sec. 4.1.1 initializes all other components and interfaces between them. The parser component described in sec. 4.1.2 parses users' program code into a sequence of instruction objects. Sec. 4.1.3 introduces the data representation used throughout the CPU, in particular with respect to memory accesses. The CPU frontend described in sec. 4.1.4 supplies the execution engine with a stream of instructions. The memory subsystem is detailed in sec. 4.1.5 and manages main memory and the cache. Sec. 4.1.6 describes the execution engine, which is responsible for actually performing computations.

4.1.1 CPU

Felix Betke

The primary purpose of the `CPU` class is to allow all other components to work together. That is, the `CPU` initializes all other components and provides interfaces to their functions. Most importantly, it

provides a `tick` function that models a single cycle and provides the snapshot functionality which allows a user of the emulator to step back to previous cycles.

The `tick` function is called each cycle and does the following: instructions from the instruction queue are fetched from the frontend and forwarded to the execution engine, until either no more slots in the unified reservation station are available, or the instruction queue is empty. It then calls the `tick` function of the execution engine. In case of a rollback, the instruction queue maintained by the frontend is flushed. If configured, a microprogram corresponding to the faulting instruction type is added to the instruction queue. That way, it will be issued before regular execution is resumed. If the rollback was caused by a faulting load instruction, execution is resumed at the next instruction. This prevents users from having to implement signal handlers, thus keeping their programs simple. In case a branch was mispredicted, the frontend is notified and refills the instruction queue accordingly. Lastly, a snapshot of the current state of the CPU is taken. To provide the UI with useful information to display to the users, the `tick` function returns an instance of a `CPUStatus` class that contains a boolean indicating if the program has terminated, whether an exception has raised a fault and, if so, whether a microprogram is to be executed, and, lastly, a list of instructions that have been issued during this tick.

The second main purpose of the `CPU` class is to provide the snapshot functionality, which allows the user of the emulator to step back to previous cycles. The snapshot list is simply a list that grows at each cycle, where ech entry is a deepcopy of the `CPU` instance. To simply be able to deepcopy the `CPU` class, the snapshot list is held separately and is not one of its members. Instead, the `CPU` class, and therefore each snapshot, maintains an index to its own entry in the snapshot list. This index makes traversing the snapshot list one step at a time easier. Due to the low complexity of the programs we expect our users to run, at the moment, no maximum number of available snapshots is configured. To restore snapshots, a static `restore_snapshot` function exists in the `CPU` class. Crucially, this function returns the deepcopy of an entry of the snapshot list relative to the value of `_snapshot_index` of the current instance. This allows users to step back and forth between snapshots. However, manipulating a restored snapshot by calling the `tick` function, for example, discards all more recent snapshots.

Other functions exist to load programs from files and initialize the frontend and execution engine accordingly. Further, references to each component are exposed by getter functions to allow the UI to visualize their current state.

4.1.2 Instructions and Parser

Users of our CPU emulator provide programs as an assembly-like source code. This source code is parsed by the `parser` module into a sequence of `Instruction` objects. Information about individual instructions is provided by the `instructions` module. The parsed instruction sequence is used throughout the rest of our CPU, indexed by the current program counter. This Harvard-style architecture is in contrast to x86 CPUs, where instructions and data share a single address space. However, parsing the program code once and from a textual representation simplifies the design of our CPU emulator, and for instance allows us to completely omit instruction memory from our memory model.

Jan-Niklas Sohn

The general instruction format used by our instruction set is the instruction mnemonic followed by a comma-separated list of instruction operands, as is common in assembly languages. In our instruction set, the instruction mnemonic already determines the exact instruction, including the number and types of its operands. This greatly simplifies parsing. There are three different types of operands in our instruction set:

- *Register* operands specify a register the instruction should operate on. They are introduced by an r followed by the decimal register number.

- *Immediate* operands specify a 16-bit immediate value used by the instruction. They take on the usual decimal (without prefix) or hexadecimal (prefixed with 0x) forms for integer literals. Immediate values can optionally be prefixed by a sign, in which case they take the value of the corresponding two's complement signed integer.

- *Label* operands specify the destination of a branch instruction. The label referenced has to be defined somewhere in the assembly file, using the label name followed by a colon.

Our instruction set, including all concrete instructions and their semantics, are covered in detail in sec. 4.4. We do not support reading or writing instruction memory. Thus, instructions have no defined in-memory representation.

The instructions of our instruction set are further distinguished based on their *instruction category*. The possible instruction categories are *register-register* instructions, *register-immediate* instructions, *branch* instructions, *load* instructions, *store* instructions, *flush* instructions, and three special categories for the individual *rdtsc*, *fence*, and *flushall* instructions. Our implementation uses InstructionKind objects to model the individual instructions of our instruction set. Each such object defines an instruction by its mnemonic, the number and types of its operands, the instruction category it belongs to, as well as some category-specific information. For register-register and register-immediate instructions, this is the concrete computation performed. For branch instructions, this is the branch condition. And for load and store instructions, this is the width of the memory access.

Grouping similar instructions into categories allows the execution engine to handle executed instructions based solely on their instruction category; the execution engine does not need to handle every concrete instruction separately. New instructions that match an existing instruction category can be added easily by users, without having to modify the execution engine. The mechanisms involved in the execution engine are described in detail in sec. 4.1.6.

Our parser is based on an abstract description of the instructions of our instruction set. This description is limited to the instruction's mnemonic and the number and types of its operands. The parser handles all instructions uniformly and has no information about the semantics of any instruction. Operation of the parser is divided into two passes over the input file. The first pass exclusively handles label definitions. The parser maintains an internal directory of labels, associating each label name with the immediately following instruction. The second pass parses the actual program, with one instruction per line. After determining the mnemonic and looking up the corresponding instruction, it parses all operands, subject to the rules for operand types described above. Having identified the instruction and parsed all operands, the parser builds an Instruction object, which models a concrete instruction in program code. Every Instruction object references the InstructionKind object of the instruction it represents, and contains the concrete values of all

operands. During both passes, the parser skips over any comments, which are lines starting with two slashes (`//`). Performing two passes in this way allows labels to be used both before and after their definition.

4.1.3 Data Representation

Jan-Niklas Sohn

The data representation used throughout our CPU is based on 16-bit values, called *words*. All CPU registers store a word. All instructions operate on words, and all immediate operands of instructions are words. A minor exception to this are the *store byte* and *load byte* instructions, that truncate a word to an 8-bit byte and zero-extend an 8-bit byte to a word, respectively. See sec. 4.4 for a detailed description of memory operations. Words are interpreted either as unsigned or two's complement signed integer values. However, the distinction between unsigned and signed values is only relevant for the comparison operations used by branch instructions. See sec. 4.4 for a detailed description of branch instructions.

Since our memory model is based on 8-bit *bytes*, words are separated into two 8-bit values when representing them in memory. The two individual bytes of words are stored in memory in little endian order, i.e. the least-significant byte is stored at the lowest memory address. For a detailed description of the mechanics involved in memory operations, see sec. 4.1.5.

4.1.4 CPU Frontend

Melina Hoffmann

In modern CPUs, the CPU frontend provides an interface between the instructions in the memory and the execution engine. It contains components to fetch and decode the instructions from a cache and to supply them to the CPU in a queue. It is also involved in speculative execution by predicting the result of conditional jumps and supplying instructions to the execution engine accordingly. [Wikc]

In our emulator we simplify the components and procedures. We also separate the branch prediction unit (BPU), that manages and predicts the results of conditional jumps, from the rest of the frontend. This makes our code easier to understand and makes potential future changes or additions to the BPU more convenient.

Branch Prediction Unit (BPU)

The BPU of modern CPUs plays a vital role in enabling speculative execution. It stores information about previously executed conditional branch instructions and predicts the outcome of future branch instructions accordingly. The rest of the CPU can then speculatively execute further instructions at an address based on the predicted outcome of the branch instead of stalling until the branch instruction is processed by the execution engine. If the prediction was incorrect, a rollback is performed on the speculatively executed instruction. If the prediction was correct, the execution is overall faster than without speculative execution. [Boe21]

Detailed information about the BPU of our modern base CPU is not widely available [Wikc]. In general, the Gshare BPU of modern CPUs consists of multiple components. It contains a branch target buffer (BTB) that holds the predicted target addresses for conditional branches [Koc+19]. It

also uses a pattern history table (PHT), that can be implemented as a 2-bit-saturating counter, and a global history register (GHT) to predict whether a conditional jump will be taken or not [EPA15].

In our emulator, we forgo the BTB entirely. Real life CPUs benefit from stored target addresses since they have to expensively decode each branch instruction before they can work with the target address [Wikc]. In our emulator, the parser (sec. 4.1.2) decodes the jump labels from the assembler code and directly provides them to the frontend, so storing them in an additional buffer is unnecessary. We further forgo the GHT because it is not strictly necessary to execute a Meltdown or Spectre attack. Additionally, our emulator and its behavior are easier to understand and predict without it, which is important when implementing microarchitectural attacks for didactic purposes. The BPU of our emulator only consists of a PHT, which is enough for simple Spectre variants as we demonstrate in sec. 6.3.

The default PHT used in our emulator holds an array called `counter` of configurable length 2^n to store several predictions. The instructions are assigned to different prediction slots by the least significant n bits of their index in the instruction list. For each of the slots, the prediction can take the four different values from 0 to 3, where 0 and 1 indicate that the branch will probably not be taken and 2 and 3 indicate that the branch is likely to be taken. The source code for our emulator also contains a simpler BPU with only one slot for all instructions. While the more advanced BPU is used by default, the simple BPU can be chosen in the config file described in sec. 4.5.

When the execution engine executes a branch instruction, the BPU is updated with the actual branch outcome for the instruction and the prediction in the PHT is updated by a 2-bit-saturating counter as introduced in [Boe21]. This means that the counter has values 0 (strongly not taken), 1 (weakly not taken), 2 (weakly taken) and 3 (strongly taken). When the branch is actually taken, the counter is increased by one, unless it is already at 3 and cannot be increased further. Similarly the counter is decreased to as low as 0 when the branch is not actually taken.

Instruction Queue

In a real life CPU, the overall purpose of the frontend is to provide the execution unit with a steady stream of instructions so the backend is utilized as much as possible. In a modern x86 CPU, the frontend has to fetch x86 macro-instructions from a cache and decode, optimize and queue them repeatedly to provide the backend with a queue of microinstructions ready for issuing in the execution engine. [Wikc]

In our emulator, except for the BPU, the functionality of the CPU frontend is bundled in `frontend.py`. It is significantly simplified compared to a real life x86 CPU, especially since we only use one type of instructions instead of distinguishing between macro- and microinstructions, as we discuss in sec. 4.4. They are already provided as a list by the parser (sec. 4.1.2), which renders the decoding and optimization steps in the frontend unnecessary.

The main task of our frontend is to act as an interface between the instruction list provided by the parser and the execution engine, which is described in sec. 4.1.6. It provides and manages the instruction queue, which holds the instructions that the execution engine should issue next. Conditional branches with their respective BPU predictions are taken into account when filling the queue. This enables speculative execution which is needed for Spectre attacks, as we see in sec. 2.4.

The central component of our emulated frontend ist the instruction queue. In our version, it does not only hold the instructions themselves, but also for every instruction in the queue, the respective index in the instruction list is stored. For branch instructions it also holds the respective branch prediction from the BPU at the time that the instruction was added to the queue. This additional information is needed by the execution engine to handle mispredictions and other exceptions.

When adding instructions to the queue, the frontend selects them from the instruction list, adds the additional information for the execution engine and places them into the instruction queue until the queue's maximum capacity is reached. The frontend maintains a program counter `pc` that points to the next instruction in the list that should be added to the queue. When the frontend encounters a branch instruction and the branch is predicted to be taken, the frontend adjusts the `pc` to resume adding instructions at the branch target. If a branch was mispredicted, the frontend provides a special function to reset the `pc` and refill the instruction queue with the correct instructions.

Additionally, the frontend provides a function to add a microprogram to the queue. It consists of a list of instructions separate from the parser instruction list. When adding the microprogram to the queue, the frontend may exceed the maximum queue capacity. This functionality can be used to implement mitigations against microarchitectural attacks, e.g. by adding a microprogram as part of the exception handling after an illegal load as demonstrated in sec. 6.4.

The frontend provides interfaces to both read and take instructions from the queue. It also provides a function that combines taking an instruction from the queue and refilling it. Additionally, the frontend has an interface for flushing the whole queue at once without taking the instructions from the queue.

The frontend provides further basic interfaces, e.g. for reading the size of the instruction queue and reading and setting the `pc`. These are used by the other components during regular execution, e.g. when issuing instructions to the execution engine, but also to reset the queue to a certain point in the program after an exception has occurred, see sec. 4.3. Since our emulator only executes one program at a time, the other components can check via another interface whether the frontend has reached the end of the program.

4.1.5 Memory

Felix Betke

Memory is primarily managed by the Memory Subsystem (MS). As a simplification over an MS as it is assumed to be used by Intel's Skylake architecture [Wikc], our version maintains only a single cache and no load, store, or fill buffers. Further, it maintains the main memory directly. Due to the lack of an operating systems, there are no virtual addresses. These simplifications are possible, since our objective is to allow users to learn about Meltdown-US-L1 and Spectre v1, none of which rely on any of the MS components we removed in our version. Further, since our CPU consists of a single core only, the MS can directly use the main memory.

To represent the main memory, the `MemorySubsystem` class contains a simple array that has $2^{WordWidth}$ entries. Since our emulator does not run an operating system and therefore does not support paging, a different method is needed to model a page fault that allows attackers to enter the transient execution phase of the Meltdown-US-L1 attack (as explained in sec. 2.4). To solve this, we declare the upper half of the address space (`0x8000` to `0xffff`, by default) to be inaccessible. Any reads or

writes to an address within the upper half result in a fault which causes a rollback a couple of cycles later. Naturally, the value written to the inaccessible addresses is `0x42`, while the ones accessible by programs are initialized to `0`.

To handle memory accesses, `read_byte`, `read_word`, `write_byte`, and `write_word` functions are available, which do as their names suggest, optionally without any cache side effects. Each function returns a `MemResult` instance, which contains the data, the number of cycles this operation takes, whether the operation should raise a fault (i.e. memory address is inaccessible), and, if so, in how many cycles this happens. For `write` operations, only the variables regarding faults are of relevance. Notice that one does not actually have to wait for the data. Instead, the caller receives the data immediately with a counter indicating in how many cycles it should be used.

Other functions that allow the UI to visualize the memory contents are provided. More specifically, `is_addr_cached` and `is_illegal_access` return whether an address is currently cached and whether a memory access to a specific address would raise a fault, respectively. Further, the `MemorySubsystem` includes functions that handle the cache management, such as `load_line`, `flush_line`, and `flush_all`.

Meltdown Mitigation

As explained in sec. 2.4.3, researchers suggest one of Intel's mitigations to zero out any data illegally read during the transient execution phase. To model this, both the `read_byte` functions still perform the read operation, but provide `0` as the data in the returned `MemResult`, if the mitigation is enabled. As of now, the read operation still changes the cache, but since only the contents of the inaccessible memory address are cached and not the corresponding oracle entry of the attacker, the mitigation still works. We believe a consequence of the CPU still performing the illegal read operation but zeroing out the result is that there are cache side effects. If desired, this behavior can be changed easily in the `read_byte` function.

Cache

To enable attackers to encode transiently read data, the MS maintains a single cache. The number of sets, ways, and entries per line can be configured via the config file (see sec. 4.5). The base `Cache` class firstly initializes all cache sets as an array with each entry holding an array of instances of the `CacheLine` class. There are functions that allow components using the cache to `read`, `write`, and `flush` data given an address and data, if applicable. The `parse_addr` can be used to obtain the index, tag, and offset of an address. Lastly, the `Cache` class includes functions that allow the UI to visualize its state (`get_num_sets`, `get_num_lines`, `get_line_size`, and `get_cache_dump`). Note that the abstract `Cache` class cannot be used directly, as it does not implement the `_apply_replacement_policy_` function.

The `CacheLine` class holds an integer array for the data, a tag, and its own size. Further, functions to `read`, `write`, and `flush` are provided. Lastly, there are functions that set the tag (`set_tag`) and return whether or not the cache line is currently in use by checking if the tag is set. Contrary to the `Cache` class, the `CacheLine` class can be used directly.

By default, there are three available cache replacement policies that determine which cache line is evicted from a full cache set in case new data should be added. All are implemented by extending

the base `Cache` and `CacheLine` classes accordingly. The first policy is the random replacement policy (RR) whose implementation can be found in the `CacheRR` class. This policy simply picks a cache line at random for eviction. Even though the policy introduces noise into the side channel, users may experiment with it for their cache attacks, if they so choose (see sec. 4.5).

The second cache replacement policy is the least-recently-used policy (LRU). By this policy, the cache line to be evicted from a full cache set should be the one whose most recent access was the longest time ago. To achieve this functionality, a new `CacheLineLRU` class updates a `lru_timestamp` variable each time the `read` or `write` functions are called. This variable is then used by the new `CacheLRU` class in its implementation of the `_apply_replacement_policy` function.

Lastly, the third cache replacement policy is the first-in-first-out (FIFO). In case of a full cache set, this policy picks the cache line that was first populated with data and flushes it. The classes `CacheFIFO` and `CacheLineFIFO` implement the required functions by using a `first_write` variable.

Even though more complex replacement policies exist, the exact way in which they work is often undocumented [VKM19]. However, we believe our chosen policies are sufficient to understand Meltdown, Spectre, and most cache timing attacks, such as Flush+Reload. New cache replacement polices can easily be added by defining the desired behavior in new cache and cache line classes that inherit from `Cache` and `Cache_Line`, respectively. Further, the constructor of the MS needs to be adjusted to account for the existence of the new policy.

4.1.6 Execution Engine

Jan-Niklas Sohn

The execution engine is the central component of a CPU. It is the component responsible for actually performing computations, by executing the stream of instructions provided by the frontend. Just like the execution engine of modern x86 processors, our execution engine executes instructions out-of-order, i.e. not necessarily in the order of the incoming instruction stream [Gru20, p. 15]. In order to preserve the semantics of the program, any data dependencies have to be honored during reordering. For this we use a modified version of Tomasulo's Algorithm, that is described in detail in sec. 4.2.

The execution engine contains the Reservation Station with a fixed number of instruction slots. Each slot is either empty or contains an instruction that is currently being executed. We call such instructions *in-flight*. Our Reservation Station is unified, i.e. each slot can contain any kind of instruction. The same is often found in modern CPUs [Wikc]. The slots of our Reservation Station are also used to model Load Buffers and Store Buffers; the specifics of executing memory accesses are handled by the slots directly instead of separate components. We also have no concept of Execution Units that instructions need to be dispatched to, which means that instructions' ability to execute concurrently is only limited by the number of available slots.

All instructions pass through two phases during execution: In the first phase the instruction is said to be *executing*. It waits for any source operands to become available and computes its result. Once the result is computed, the instruction waits for a specific amount of CPU cycles. This delay is introduced to mimic the latency of real execution units, which may take several CPU cycles to compute a result. After this delay expires, the instruction's result is made available to waiting instructions, and the instruction transitions to the second phase.

In the second phase the instruction is said to be *retiring*. The instruction determines if it causes a *fault*, which in this case means a *microarchitectural* fault. These can be architecturally visible faults like memory protection violations or architecturally invisible faults like branch mispredictions; both are handled the same way in the Execution Engine. Once the instruction finishes retiring its slot becomes available again and may be used to execute a new instruction.

In each clock cycle only a single instruction may finish execution or retirement. This models the contention of the Common Data Bus, which is used to provide information about computation results inside the Execution Engine and to other components and can only transmit information about a single result each clock cycle.

Besides the Reservation Station, the Execution Engine also contains the Register File, with one entry for each register. Each register entry either contains the concrete value of the register or references a slot of the Reservation Station that will produce the register's value. Since instructions are issued in program order, the state of the register file at a single point in time represents the architectural register state at that point in time, with yet-unknown register values present as slot references.

4.2 OUT-OF-ORDER EXECUTION

Melina Hoff-mann

Our emulator implements out-of-order execution. This allows transient execution of instructions before the fault handling of previous instructions is finalized, which is essential for Meltdown type attacks, as discussed in sec. 2.4. Our version of out-of-order execution is based on Tomasulo's algorithm, as introduced in sec. 2.2. Since the goal of our out-of-order execution is to enable and clearly illustrate microarchitectural attacks, not optimal performance, we use a simple version of Tomasulo's algorithm. In our emulator, the components necessary for Tomasulo's algorithm are located in the Execution Engine (sec. 4.1.6). Below, we provide a detailed look at our version of out-of-order execution and the components involved in its implementation.

4.2.1 ISSUING INSTRUCTIONS

Since we implement out-of-order execution according to Tomasulo's algorithm, our execution engine does not try to execute instructions directly when it receives them from the frontend. Instead, it issues them to the reservation station where multiple instructions can wait until all their operands are ready. If all operands of an instruction are ready, the execution engine can execute it. This does not generally happen in the order of instructions as provided by the program, but will always lead to the intended effect since data dependencies are respected.

The instructions are provided by the frontend (sec. 4.1.4) in program order. The execution engine issues them into the reservation station, if it is not yet fully occupied. The reservation station is modelled as a list of `slots` which can each hold an instruction together with additional information about the instruction. It is unified in that each spot in the list can hold slots for all types of instructions. This models the unified reservation stations of modern Intel CPUs, as we see in sec. 2.2.

To issue an instruction, the execution engine creates a `slot` object that fits the type of the instruction and puts it into the reservation station. Besides the instruction itself, it holds additional information,

including a list of the instruction's operands. While immediate operands can be directly converted to a `Word`, register operands have to be resolved during the issuing process.

The registers are modelled by a list in which each entry can either be a `Word` value or the ID of the slot in the reservation station which holds the instruction that will produce the next register value as its result. As we see in sec. 4.1.6, since the instructions are issued in program order, this reflects the expected register state at the point of issuing the current instruction if the program was executed in order. The only difference being, that the results of yet unexecuted instructions are being represented by the respective `SlotID`. To resolve the register operands, the current content of the respective register is added to the operand list, so the operand list can contain both `Word`s and `SlotID`s. As described below, `SlotID`s in the operand list will be replaced by the result of the instruction which produces the value when it finishes executing.

Resolving the register operands this way ensures that data dependencies between instructions that use the same registers are adhered to. To increase performance, real life CPUs practice register renaming in order to further eliminate data dependency hazards [Gru20, p. 226]. Modern CPUs rename registers by assigning ISA level registers to different microarchitectural registers [Boe21]. Since, as described in sec. 4.4, we do not differentiate between the ISA and microarchitectural level, and aim to keep our emulator easy to comprehend, we do not implement register renaming.

Once the slot with the new instruction is placed into the reservation station, if the instruction will produce a result for a target register, the `SlotID` of the instruction is put into this target register. This ensures, that when the next instruction is issued, the register state again represents the expected register state if the instructions where executed in-order. Note that placing the `SlotID` into the target register cannot only overwrite a `Word` value but also a `SlotID`, if the previous instruction that uses this register as its target register is not yet fully executed. This is not a problem, since every instruction, that may need the result of the respective instruction of the previous `SlotID` as an operand, already holds this `SlotID` in its own operand list. It will be notified of the result when it is ready, regardless of whether the `SlotID` is still present in the register or not.

4.2.2 Executing Instructions

According to the basic Tomasulo algorithm, when all operands of an instruction in the reservation station are ready, it is transferred to a free execution unit and executed, as explained in sec. 2.2. In our emulator, execution of the instructions is triggered by the `tick` function of the execution engine, which is executed once per CPU cycle. We do not model a finite number of execution units as separate components. Instead, the `tick` function goes through the occupied slots in the reservation station and tries to execute each instruction by calling the `tick_execute` function of its respective slot. This follows the order of the slots in the reservation station, regardless of when the instruction in each slot was issued, i.e. regardless of their order in the program. If the operands of the current instruction are not ready yet, i.e. there are still `SlotID`s in the operand list, the instruction is skipped.

Once the instruction is executed and produces a result, i.e. all operands are available and the wait time is over, according to Tomasulo's algorithm described in sec. 2.2, this result has to be broadcasted via the CDB to the other slots and the registers. In our emulator, the CDB is modelled by the `_notify_result` function. It goes through all registers and replaces all occurrences of the

`slotID` of the instruction with the result it just produced. It also notifies all occupied slots of the result together with the `slotID` of the instruction which produced it, so they can replace the `slotID` if it is in their operands. If a result is produced like this, the `tick` function returns without executing further slots. This mimics that a real life CDB can only broadcast one result each cycle. It has the side effect that instructions do not necessarily execute in the same number of `ticks`, depending on where they are in the reservation station. The `tick` function also returns before all slots have been executed if the instruction in a slot retires, in order to properly handle potentially faulting instruction as we see in sec. 4.3.

4.2.3 MEMORY HAZARDS

As described above, we handle data dependencies between instructions that use the same registers by using `SlotIDs` as placeholders for as yet uncomputed results. We also need to handle data dependencies between memory accesses. For this, each slot that contains a memory instruction also holds a set of `SlotIDs` of other memory instructions that potentially lead to a memory hazard together with the current instruction. The memory instruction is only executed when all other instructions from its list of potential hazards have retired.

This list is filled when the memory address that the instruction will access is computed, beforehand it is set to the placeholder value `none`. To fill the list, the `_tick_execute` function of the slot goes through its `faulting-preceding` list that i.a. contains all in-flight memory instructions that precede the current instruction in program order, and includes them if they access the same address. If there is a previously issued memory instructions for which the memory address is not yet available, the instruction waits until the hazard list can be completed.

By adding all instructions to the hazard list that access the same memory address, we potentially generate false positives in the case that two load operations read from the same memory address one after the other. Since efficiency is not our priority, we accept this in order to keep our emulator simple. Additionally, to simplify fault handling, store instructions wait until all other possibly faulting instructions have retired.

4.2.4 FENCE INSTRUCTION

The `fence` instruction is a special instruction in that it does not produce a result or a lasting side effect in the other components of the emulator. It creates a fixed point in the execution of the program, effectively suspending the out-of-order execution with regards to the fence instruction, as we describe in further detail in sec. 4.4. It holds a list of all instructions that were already in the reservation station when itself was issued. Similar to the memory instructions, it waits for all the instructions in the list to be retired before executing itself. Additionally, no other instructions can be issued to the reservation station while it contains a `fence` instruction.

4.3 Exception- and Fault-Handling

Jan-Niklas Sohn

Exceptions in general are certain conditions that can occur during execution and require handling before execution can be continued. We distinguish between *architectural* and *microarchitectural* exceptions.

Architectural exceptions are visible to the program being executed and are usually handled by that program or the underlying operating system. In our CPU emulator there is no operating system that could handle architectural exceptions, and requiring the program to handle these would increase the complexity of both our CPU and user programs. For this reason architectural exceptions are handled implicitly, by skipping execution of the instruction that caused the exception. After an exception occurred, execution continues with the following instruction instead. The only architectural exceptions present in our CPU are caused by invalid memory accesses, when a memory operation is performed on an inaccessible address.

Microarchitectural exceptions in contrast are not visible to the program being executed and handled directly by the microarchitecture. In our case the only cause for a microarchitectural exception are mispredicted branches.

Both architectural exceptions and microarchitectural exceptions cause what we call microarchitectural *faults*. All microarchitectural faults are handled in the same way inside the Execution Engine; the difference in behavior between architectural exceptions and microarchitectural exceptions is only introduced in the main CPU component, as mentioned in sec. 4.1.1.

4.3.1 Rollbacks

As discussed in sec. 4.2, our CPU executes instructions out-of-order. Because of this, special care must be taken when handling microarchitectural faults. In particular, the following effects need to be considered:

- Instructions preceding the faulting instruction in program order might have not yet been executed.
- Instructions following the faulting instruction in program order might already have been executed.

The abstraction that, from an architectural point of view, instructions are executed in program order has to be preserved. Thus, we have to restore the architectural state from the time the faulting instruction was issued before we can properly handle the fault. This process of restoring the architectural state is called *rollback*. Instructions that follow the faulting instruction in program order, but are executed before the rollback is performed, are said to occur in *transient execution*; they are executed as usual, but afterwards their operation is rolled back.

In our CPU, rollbacks are local to the Execution Engine. In case of a fault, the Execution Engine performs the rollback, and returns information about the fault to the main CPU component. The main CPU component then performs the remainder of the fault handling.

There are two possible approaches to performing rollbacks. The first approach tracks any changes that executed instructions make to the architectural state. When a fault occurs, these tracked

changes can be performed in reverse in order to recover the target architectural state. The second approach records a snapshot of the architectural state when the faulting instruction is issued. When a fault occurs, this snapshot can be restored in order to recover the target architectural state. It is not publicly documented what approach real x86 CPUs take to performing rollbacks. Our implementation follows the snapshot-based approach, since it is judged to be easier to implement in a software-based emulator.

In our case, the architectural state that needs to be restored includes the register state and the contents of memory. The state of the cache and the BPU are not considered part of the architectural state and are not restored when performing a rollback. If multiple instructions might cause a fault, the fault that comes first in program order must be the one that is handled. We employ different techniques for ensuring that the register state and the contents of memory have the proper state after the rollback, and for making sure the first fault in program order is handled. These are described in the following sections. Our implementation uses the category of *potentially faulting instructions*, which includes branch instructions and memory operations.

Restoring the Register State

When a potentially faulting instruction is issued, a copy of the current register state is stored in the reservation station slot. As described in sec. 4.1.6, the register state might contain references to other slots of the reservation station. When the instruction actually faults, execution continues normally until all of the slots referenced by the captured register state have finished executing. Then the captured register state contains no more slot references, and can be restored.

Restoring the Contents of Memory

Storing a snapshot of the contents of memory every time a store instruction is issued would require a lot of space and prevent the rollback from being local to the Execution Engine. Thus, we instead serialize the execution of store instructions with respect to other potentially faulting instructions. When a store instruction is issued, all slots of the reservation station that contain potentially faulting instructions are recorded. Before performing the store operation, execution halts until all of the recorded slots have retired. This ensures that store operations are only performed when it is known that no preceding instructions cause a fault.

Handling Faults in Program Order

To make sure that faults are handled strictly in program order, we use the same technique as used to avoid having to snapshot and restore the contents of memory, described above. When a potentially faulting instruction is issued, all slots of the reservation station that contain potentially faulting instructions are recorded. When the instruction actually faults, execution continues normally until all of the recorded slots have retired. This ensures that potentially faulting instructions retire strictly in program order.

The techniques described above require waiting on the execution or retirement of preceding instructions. However, except for store instructions waiting for preceding potentially faulting instructions, this is only required in the case when a fault actually occurs. Thus, in the expected case of no faults the out-of-order execution is not unnecessarily restricted.

4.3.2 Transient Execution Attacks

As mentioned above, the state of the cache is not restored during a rollback. Thus, transiently executed instructions can influence the cache in a way that persists beyond the rollback. During normal execution, the state of the cache can then be observed using a timing-based side channel. This allows using the cache as a transmission channel from the transient execution domain to the usual architectural domain. Such a transmission channel is typically used in Meltdown- and Spectre-type attacks, and integral to their success.

The state of the BPU is similarly not restored during a rollback. Transiently executed instructions can influence the BPU by performing branches, and during normal execution the state of the BPU can be observed using the differences in execution time caused by mispredicted branches. Because of this, the BPU could be used as a transmission channel just like the cache.

The result of a faulting load operation is made available to following instructions before the fault is handled. During transient execution, this result can be transmitted to the architectural domain via a cache-based channel. Thus, Meltdown-type attacks are possible in our CPU emulator. This is demonstrated and described in detail in sec. 6.2.

During the transient execution after a mispredicted branch, memory loads can be performed. In the same transient execution, their result can be transmitted to the architectural domain via a cache-based channel. Thus, Spectre-type attacks are possible in our CPU emulator. This is demonstrated and described in detail in sec. 6.3.

4.4 ISA

Melina Hoffmann

Real life Intel x86 CPUs differentiate between two types of instructions or operations. Macro-operations refer to the relatively easily human readable and convenient but complex instructions that are described by the x86 ISA. Their length differs between the instructions. Internally, in the execution units, the CPU works on micro-operations, which are small operations of a fixed length. One macro-operation contains one or multiple micro-operations. The CPU frontend has to decode the macro-operations into micro-operations in an expensive multi step process. [Wikc], [Wika], [Wikb]

Our CPU emulator only uses one type of instructions. They are directly read from our assembler code by the parser (sec. 4.1.2) and passed to the execution engine without further decoding, splitting or replacing, as described in sec. 4.1.4. To show basic Meltdown and Spectre variants as we demonstrate in sec. 6.2 and sec. 6.3, we do not need overly complex instructions, e.g. instructions that contain multiple memory accesses in one or that are used to perform encryption in hardware. Basic arithmetic operations, memory accesses, branches and a few special operations are sufficient for the demonstrated attacks. They are both easy to implement as single instructions and to use in assembler code that should be well understood by the author. Using the same operations throughout the emulator also makes the visualization more clear and easier to follow, e.g. when the same operations appear, one after the other, in the visualization of the assembler code, the instruction queue and the reservation stations.

4.4.1 Default Instruction Set

In order that our CPU emulator can recognize and work with an instruction, it has to be registered with the parser as described in sec. 4.1.2. In our default setting, we register a basic set of instructions with the parser so students can start writing assembler code and using the emulator right away. This basic instruction set is also used in our example programs in sec. 6.

Our relatively small instruction set is based on a subset of the RISC-V ISA [And17, pp. 13-29]. It offers a selection of instructions that is sufficient to implement Meltdown and Spectre attacks as well as other small assembler programs, while still being of a manageable size so students can start to write assembler code quickly without spending much time to get to know our ISA. The syntax of the assembler representation is also based on RISC-V as introduced in the "RISC-V Assembly Programmer's Handbook" chapter of the RISC-V ISA [And17, pp. 137-140]. Also as in the RISC-V ISA, with our default configuration as described in sec. 4.5 we have 32 registers available. If needed, students can add further instructions by registering them with the parser as described in sec. 4.1.2.

In the following subchapters we introduce the instructions of our default ISA. They are grouped according to their respective instruction type in the emulator except for the special instructions which which do not share an instruction type but are still grouped together.

Arithmetic and Logical Instructions Without Immediate

These are basic arithmetic and logical instructions that operate solely on register values, i.e. both source operands and the destination operand reference registers. For simplicity, we write, for example, Reg1 when referring to the value read from or stored in the register referenced by the first register operand.

Each of these default instructions uses the respective `python` standard operator on our `Word` class to compute the result, except for the right shifts. For the logical and the arithmetic right shift, the python standard right shift operator is used on the unsigned and the signed version of the register value respectively. When returning the result as a `Word`, it is truncated to the maximal word length by a modulo operation, if necessary. This means, that any potential carry bits or overflows are effectively ignored.

Arithmetic and Logical Instructions without Immediate		
Instr. Name	Operators	Description
add	Reg1, Reg2, Reg3	Reg1 := Reg2 + Reg3
sub	Reg1, Reg2, Reg3	Reg1 := Reg2 − Reg3
sll	Reg1, Reg2, Reg3	Reg1 := Reg2 << Reg3
srl	Reg1, Reg2, Reg3	Reg1 := Reg2 >> Reg3 logical
sra	Reg1, Reg2, Reg3	Reg1 := Reg2 >> Reg3 arithmetical
xor	Reg1, Reg2, Reg3	Reg1 := Reg2 xor Reg3
or	Reg1, Reg2, Reg3	Reg1 := Reg2 or Reg3
and	Reg1, Reg2, Reg3	Reg1 := Reg2 and Reg3

Arithmetic and Logical Instructions With Immediate

These are analogous to the instructions introduced in sec. 4.4.1. The main difference is, that the second source register is replaced by an immediate operand which is set directly in the assembler code. This immediate is used as the value of a `Word`, so it is truncated by a modulo operation to be in the appropriate range.

Arithmetic and Logical Instructions with Immediate		
Instr. Name	Operators	Description
addi	Reg1, Reg2, Imm	Reg1 := Reg2 + Imm
subi	Reg1, Reg2, Imm	Reg1 := Reg2 − Imm
slli	Reg1, Reg2, Imm	Reg1 := Reg2 << Imm
srli	Reg1, Reg2, Imm	Reg1 := Reg2 >> Imm logical
srai	Reg1, Reg2, Imm	Reg1 := Reg2 >> Imm arithmetical
xori	Reg1, Reg2, Imm	Reg1 := Reg2 xor Imm
ori	Reg1, Reg2, Imm	Reg1 := Reg2 or Imm
andi	Reg1, Reg2, Imm	Reg1 := Reg2 and Imm

Memory Instructions

These instructions provide basic interactions with the emulated memory introduced in sec. 4.1.5. Load and store instructions exist in two versions, one that operates on `Word` length data chunks, for convenience, and one that operates on `Byte` length data chunks, for the fine granular access needed in microarchitectural attacks. The `flush` instruction flushes the cache line for the given address. The `flushall` instruction flushes the whole cache. The address is calculated in the same way for all memory instructions: `addr:=Reg2+Imm`, and `addr:=Reg+Imm` for the `flush` instruction respectively.

Memory Instructions		
Instr. Name	Operators	Description
lw	Reg1, Reg2, Imm	Reg1:=Mem_word[addr]
lb	Reg1, Reg2, Imm	Reg1:=Mem_byte[addr]
sw	Reg1, Reg2, Imm	Mem_word[addr]:=Reg1
sb	Reg1, Reg2, Imm	Mem_byte[addr]:=Reg1
flush	Reg, Imm	flush cache line of addr
flushall	-	flush whole cache

Branch Instructions

All branch instructions compare the values of two source registers. If the comparison evaluates to true, the execution of the program is resumed at the given label in the assembler code. If it evaluates to false, the next instruction in the program is executed. Depending on the instruction, the register values are interpreted as signed (s) or unsigned (u) integers. Labels in the assembler code are automatically resolved by the parser, as we can see in our examples in sec. 6.1.

Branch Instructions		
Instr. Name	Operators	Description
beq	Reg1, Reg2, Label	jump to Label if Reg1==Reg2
bne	Reg1, Reg2, Label	jump to Label if Reg1! =Reg2
bltu	Reg1, Reg2, Label	jump to Label if u(Reg1)<u(Reg2)
bleu	Reg1, Reg2, Label	jump to Label if u(Reg1)<=u(Reg2)
bgtu	Reg1, Reg2, Label	jump to Label if u(Reg1)>u(Reg2)
bgeu	Reg1, Reg2, Label	jump to Label if u(Reg1)>=u(Reg2)
blts	Reg1, Reg2, Label	jump to Label if s(Reg1)<s(Reg2)
bles	Reg1, Reg2, Label	jump to Label if s(Reg1)<=s(Reg2)
bgts	Reg1, Reg2, Label	jump to Label if s(Reg1)>s(Reg2)
bges	Reg1, Reg2, Label	jump to Label if s(Reg1)>=s(Reg2)

Special Instructions

`Rdtsc` acts like a basic timing instruction. It returns the number of `ticks` the execution unit has executed so far into the given register.

The `fence` instruction acts as a fixed point in the out-of-order execution. All instructions that are already issued into the reservation station at the point of issuing the `fence` instruction are executed before the `fence` is executed. No new instructions are issued before the execution of the `fence` instruction is complete. This can be used to model mitigations against microarchitectural attacks, as demonstrated in sec. 6.4.

Special Instructions		
Instr. Name	Operators	Description
rdtsc	Reg	Reg:=cyclecount
fence	-	add execution fixpoint at this code position

4.5 Config File

Felix Betke

To allow users to change certain parameters of the presented emulator or to toggle certain mitigations on or off, a YAML configuration file is available. It can be found in the root folder of the project. Internally, a dictionary containing the entire configuration is passed to each individual component. This section briefly mentions which parts of the demonstrated emulator can be configured, and why which default values have been chosen.

4.5.1 Memory

In the "Memory" section, users may configure how many cycles write operations should take (`num_write_cycles`), and how many cycles should be between a faulting memory operation and the corresponding instruction raising a fault (`num_fault_cycles`). By default, the former is set to 5 cycles, the latter to 8. While the specific values are not as important, the difference between the number of cycles it takes to perform a faulting memory operation (`num_write_cycles`, `cache_hit_cycles`, and `cache_miss_cycles`, see sec. 4.5.2) and the number of cycles it takes to raise the fault (`num_fault_cycles`) should be at least 1. Otherwise, the rollback is initiated before the

faulting instruction makes the data available to subsequent instructions, thus removing the transient execution time window on which Meltdown and Spectre rely. Note that `num_write_cycles` only affects write operations. To configure the number of cycles it takes to read data, see sec. 4.5.2.

4.5.2 Cache

In the "Cache" section, users may configure the size of the cache by setting the number of sets, ways, and entries per cache line. For readability when printing, the default value we chose for all values is 4. However, it is important to note that if one were to perform the full Meltdown/Spectre attacks by measuring access times to each oracle entry, a sufficient cache size that ensures accessing an oracle entry does not evict another is required.

Further, the cache replacement policy can be set to either "RR", for random replacement, "LRU", for least-recently-used, and "FIFO", for first-in-first-out. Each policy is explained in sec. 4.1.5. As RR introduces noise, we have decided against using it as the default. Although widely undocumented, Intel's Skylake architecture is believed to use some kind of specialized version of LRU [VKM19]. For that reason, and since our choice is largely irrelevant as long as users clearly understand how the policy works, LRU is set as the default.

The other two values that can be configured are the number of cycles cache hits and misses take (`cache_hit_cycles` and `cache_miss_cycles`). As already mentioned in sec. 4.5.1, the specific values are not as important as their differences. As explained in sec. 4.5.1, Meltdown is not possible if it takes longer to read than to raise an exception. Due to our decision to allow users to perform the Meltdown-US-L1 attack even if the data to be stolen is not currently cached (see sec. 3), the number of cycles needed for retrieve data from memory after a cache miss (`cache_miss_cycles`) is lower than the number of cycles it takes for a faulting memory operation to raise a fault (`num_fault_cycles`, see sec. 4.5.1). To model the Meltdown-US-L1 attack more accurately, a cache miss would have to take more cycles to retrieve the data than the CPU takes to initiate the rollback. In the more realistic scenario, attackers would have to perform two illegal reads, one to cache the secret, and one to steal it. By default, the first read is not necessary.

4.5.3 Instruction Queue

The config allows users to configure the size of the instruction queue maintained by the frontend. It determines the maximum number of instructions that can be issued per tick. As the transient execution window is largely determined by memory access times and the number of available entries in the unified reservation station, this choice is not as important. Our testing shows the desired attacks are possible with an instruction queue size of 5. As pointed out in sec. 4.1.4, this limit is ignored in case microprograms are to be executed.

4.5.4 BPU

Here, the user may configure whether to use the simple BPU, or the advanced one. The difference is that the simple BPU maintains only a single counter, while the advanced one maintains a counter for each index. Additionally, the number of bits used for the counter as well as its initial value can

be chosen. By default, we use the advanced BPU with 4 index bits and an initial counter of 2, which represents "weakly taken".

4.5.5 Execution Engine

The execution engine allows the configuration of the number of available slots in the reservation station. In part, this value determines the width of the transient execution window. We find that with 8 slots, Meltdown and Spectre attacks are possible. Additionally, the number of registers can be configured. Since our instruction set is based on the RISC-V ISA, we offer the same number of registers by default, which is 32 [And17, p. 9].

4.5.6 UX

The UX can be configured to omit display of certain elements. When selecting to use a large cache, it may be desirable to hide empty cache ways and sets to prevent clutter. By default, we choose to show the empty cache ways so the user can easily can see whether or not a cache set is full. We also choose to show the empty cache sets so there is a visual representation of which addresses correspond to certain sets. Another option is to hide the unused reservation station slots. In this case every entry in the reservation station will be numbered. Showing the unused slots may help the user to keep track of bottlenecks in the execution, and is therefore chosen per default. Finally, the user may choose between capitalized or lowercase letters for the registers. By default, we choose to use lowercase letters.

4.5.7 Microprograms

This section allows users to configure microprograms that are run once a rollback has been completed and before regular program execution is resumed. For each instruction type, the user can provide the path to a file containing the microprogram. If no microprogram should be run after a specific instruction type faults, the user can either set the file path to `None`, or simply not include it in the config. The instruction types must be their corresponding class names as they are given in the `execution.py` file. We expect users to mostly use microprograms for faulting `InstrLoad` and `InstrBranch` instructions. By default, no microprograms are configured, but one that flushes the entire cache can be found in `demo/flushall.tea`.

4.5.8 Mitigations

This section allows users to toggle Intel's Meltdown mitigation that overwrites the illegally read value with 0 (see sec. 2.4.3) on or off. By default, it is disabled.

Further, mitigations that are implemented in microcode, such as flushing the entire cache after a rollback (see sec. 2.4.3), can be enabled by users using microprograms in the Microprograms section of the config file (see sec. 4.5.7).

5 USER INTERFACE AND USAGE

Lennart Hein

This chapter focuses on the usage of the application. The user interface and the design choices are described. An instruction on how to prepare the system and how to run the program is given. Furthermore, the application and it's features are explained in detail. Finally, we showcase the application in action.

5.1 PURPOSE AND INSPIRATION

In order to demonstrate Meltdown and Spectre attacks, this program is designed to visualize the execution of a modern CPU on a microarchitectural level. We implement a simplified model of a CPU to keep focus on the essential parts to understand the demonstrated attacks. At the same time, the CPU needs to show all information required to follow the attack, and offer convenient features to the user.

The emulator follows the rough structure of the GNU Debugger GNU [Fre] and more specifically the extension Pwndbg [Rig]. The assumption is that many members of the target audience are already well familiar with the GNU Debugger and the Pwndbg extension. GDB and Pwndbg can be interacted with via the command line. Different to other similar solutions, GDB can only be interacted with using commands. Most notable is the 'context' screen, which is issued after execution is paused. The context screen prints out the current state of the CPU, including the registers, the stack, a backtrace, and the disassembly.

As with GDB, the emulator can only be interacted with using commands. We further implement auto-completion and auto-suggestion using the `python-prompt-toolkit` [Fis]. This should lower the difficulty of getting started with a new tool. The emulator also implements a Pwndbg-style context-visualization, albeit with different elements shown to adapt to the different goal compared to GDB. This ensures most of the relevant information is on the screen at all times.

5.2 SYSTEM REQUIREMENTS AND INSTALLATION

The following things need to be installed to run the program:

- Python >= 3.8
- Python-Benedict
- Python-Prompt-Toolkit

To install the required packages, one may use the following command:[1]

---

[1] The requirements file is located in the root directory of the project. In some distributions pip may be called pip3.

```
1 │ pip install -r requirements.txt
```

Optionally, the following things can be installed to allow for git fingerprints in the log output:

- Git

It is recommended to use the program on a Linux system, although Windows functionality is mostly implemented. Further, it's recommended to have a terminal with a width of at least 120 characters.

The program is tested on the following system:

- Debian GNU/Linux 11 (bullseye)
- Python 3.9.10
- Python-Prompt-Toolkit 3.0.28
- Python-Benedict 0.25.0
- Git Commit 419678d33a41eefb0bcd775966dae0418ba51245

5.3 Running the Program

The syntax for running the emulator from the root folder of the repository is:

```
1 │ ./main.py <path_to_target_program>
```

If no target program is supplied, the emulator will print system information, along with a help message. The target program may contain instructions as specified in sec. 4.4, separated by line-breaks. Comments can be added to the target program by preceding them with //. Furthermore, the `config.yml` file in the root folder can be modified to configure the emulator, see sec. 4.5.

5.4 The Context Screen

The core of the visualization is the context screen. Here, most of the relevant information is shown. The context screen is printed out every time the execution is paused. Alternatively, pressing enter with an empty input will also print the context screen. Figure 2 shows an example output of the context screen.

The context screen is divided into three sections: first, the registers are being shown, as if the `show regs` command was issued (for details on the commands, refer to the following sec. 5.5). The second section shows the Memory, also analogous to the `show mem` command. The third section shows the whole pipeline, itself being divided into visualization of the Program, the Instruction Queue and finally the Reservation Stations. By default only a part of the Program is printed - all in-flight instructions as well as one further instruction each before and after will be displayed. Further, arrows connect the different stages of the pipeline. These arrows show the location of the instruction that will be issued next into the Reservation Station in both the Program, as well as the Instruction Queue.

FIGURE 2: *Example output of the context screen*

5.5 COMMANDS

Following, we describe the commands available in the emulator. The commands are grouped into the following categories: displaying information, modifying the CPU, commands that revolve around pausing and resuming the execution, breakpoint management, as well as some miscellaneous commands.

5.5.1 DISPLAY INFORMATION

When displaying specific information, the following command serves as a base:

```
1  show
```

The display commands were implemented side-effect free, looking at the internals of the execution without interfering is the core idea of the emulator.

The following subcommands are available:

SHOW MEM

Show the memory of the CPU, visualized as words in hexadecimal.

```
1  show mem <start in hex> <words in hex>
```

The first parameter is the start address of the memory to be shown, the second is the number of words to be shown. The parameters are optional, the emulator defaults to showing memory starting at address 0x0000 until 8 lines of the terminal are printed. Displaying contents from the memory does neither load the memory into the cache if not already present, nor does it change the order of future eviction from the cache. The subcommand is thus side effect free.

SHOW CACHE

Show the cache. The visualization is configurable in the `config.yml` file, see sec. 4.5.

```
1 show cache
```

SHOW REGS

Show the CPU registers. For each register, either the value of the word is shown in hexadecimal, or, in case the register is waiting for the result of a reservation station, a reference to that reservation station is shown. The nomenclature is configurable in the `config.yml` file, see sec. 4.5.

```
1 show regs
```

SHOW QUEUE

Display all instructions currently in the instruction queue. The topmost instruction is the one being issued next, the bottommost is the instruction last loaded into the queue.

```
1 show queue
```

SHOW RS

Displays the reservation stations. Whether or not empty slots are shown is configurable in the `config.yml` file, see sec. 4.5. The information supplied includes the index of the instruction in the source code, the instruction itself, as well as current values for the source operands, either as a word in hexadecimal, or as a reference to another reservation station. This allows the user to quickly determine, which reservation stations are currently waiting for the result of other reservation stations, and are stalled as a result. Furthermore, retiring instructions are marked with a ticked checkbow.

Should the user select to use a microcode in case a fault is encountered (see sec. 4.5), the instructions originating from the injected microcode will be marked with a μ symbol instead of the index of the instruction.

```
1 show rs
```

SHOW PROG

Displays a visualization of the source program. Further, there are icons indicating which instructions are currently in-flight, as well as which instruction is marked as a breakpoint.

```
1 show prog
```

▶ In-Flight

▶ In-Flight and Breakpoint

◉ Breakpoints

⬚ Disabled Breakpoint

SHOW BPU

Displays the value of the 2bit counter for every index in the BPU.

```
1  show bpu
```

5.5.2 MODIFYING THE CPU

When modifying the CPU, the following command serves as a base:

```
1  edit
```

The following subcommands are available:

EDIT WORD

Takes an address and value in hexadecimal, and overwrites the word at that address with the new value.

```
1  edit word <address in hex> <value in hex>
```

Editing the memory through this command does not affect which lines are stored in the cache. It does update the contents of the cache, however, if the address is in the cache, to keep the state of the cache and memory consistent. The user should be aware that the memory is addressed bytewise, but the subcommand is writing a word in little-endian order.

EDIT BYTE

Takes and address and a value in hexadecimal, and overwrites the byte at that address with the new value.

```
1  edit byte <address in hex> <value in hex>
```

This command is analogous to the `edit word` command, but operates on a byte instead of a word.

EDIT FLUSH

Allows to flush the cache, selectively or fully.

```
1  edit flush <address in hex>
```

If no address is supplied, the entire cache is flushed. If an address is supplied, the cacheline containing that address is flushed, if present in the cache.

EDIT REG

Changes the value of a register.

```
1  edit reg <register (0-31)> <value in hex>
```

Any of the 32 registers (0-31) can be modified.

EDIT BPU

Overwrites the 2bit counter for a specific index in the BPU.

```
1  edit bpu <pc in dec> <value (0-3)>
```

The values correspond to the following:

- 0: Strongly not taken
- 1: Weakly not taken
- 2: Weakly taken
- 3: Strongly taken

See further in chapter 4.1.4 for more information.

5.5.3 BREAKPOINT MANAGEMENT

Inspired by GDB and other debuggers, the emulator implements support for Breakpoints. Breakpoints are a mechanism to pause the execution of the emulator. They can be set for a specific instruction, upon issuing an instruction with an active breakpoint set, the execution will be paused, and control given to the user. The execution will thus be paused immediately after an instruction has been loaded into a reservation station slot.

The emulator allows to set, clear, toggle and list breakpoints. The base command is:

```
1  break
```

BREAK ADD

Sets a breakpoint for the instruction at the given index.

```
1  break add <index in decimal>
```

BREAK DELETE

Deletes the breakpoint for the instruction at the given index, if it exists.

```
1  break delete <index in decimal>
```

BREAK TOGGLE

Disables an active breakpoints, and likewise enables a disabled breakpoint.

```
1  break toggle <index in decimal>
```

BREAK LIST

Outputs a list of all breakpoints, and whether they are enabled or disabled.

```
1  break list
```

Note that the breakpoints also can be seen when calling `show prog`. See sec. 5.5.1 for more information.

5.5.4 PAUSE AND RESUME EXECUTION

Since the emulator is thought to be used similar to a debugger, to provide a more convenient way to pause and resume execution, the following commands are available:

CONTINUE

Continues execution of the program.

```
1 | continue
```

The execution will be resumed until the next breakpoint is reached, a fault is encountered, or the program terminates.

STEP

Allows to step through the program one cycle at a time.

```
1 | step <steps>
```

If steps is not supplied, the emulator will execute a single cycle. If steps is supplied, the emulator will execute the specified number of cycles. The execution is also paused when encountering a breakpoint, a fault, or the program terminates.

If a negative number is supplied, the emulator will execute the specified number of cycles in reverse order. This allows the user to conveniently review the events that occur during the execution without having to restart the program. Furthermore, this can be combined with other commands such as `continue`, to see the status during the last cycle before a notable event, such as a fault, caused the execution to be paused.

RETIRE

Continues execution of the program until the next instruction is retired.

```
1 | retire
```

As with `step` and `continue`, the execution is also paused when encountering a breakpoint, a fault, or the program terminates.

RESTART

Resets the program to the state it was in after launching the emulator.

```
1 | restart
```

5.5.5 Miscellaneous Commands

Further, the following commands are available:

QUIT

Quits the program.

```
1  quit
```

Also can be called using:

```
1  q
```

HELP

Displays help.

```
1  help
```

CLEAR

Clears the screen

```
1  clear
```

6 Demonstration and Evaluation

As specified in chapter 3, our goal is to implement a CPU emulator that offers out-of-order and speculative execution in order to demonstrate a Meltdown and a Spectre attack. In this chapter we demonstrate that our emulator allows for the user to execute both a Meltdown and a Spectre attack, with the use of basic example programs. Firstly, we introduce the general functionality and visualization of our emulator with a simple example program that does not yet implement microarchitectural attacks in sec. 6.1. Then, we demonstrate both the Meltdown and the Spectre variant which are possible on our emulator in sec. 6.2 and sec. 6.3 respectively. Lastly, we show different mitigations against these microarchitectural attacks on our emulator, which are based on mitigations against real life microarchitectural attacks, in sec. 6.4.

Melina Hoff-mann

6.1 Example Program

In this section, we introduce different components of the visualization of our emulator and showcase our emulators out-of-order and speculative execution. To this end, we `step` through an example program and introduce the central components of our vizualisation. Introducing all the commands and visualizations our emulator implements with an example program would be out of scope of this section, but a complete list of commands is given in chapter 5.

Melina Hoff-mann

This is our example program. it is included as `demo/demo.tea` in our gitlab repository. To produce the examples for this section, the program is run on the default config settings as discussed in sec. 4.5.

```
1   addi r1, r0, 3
2   slli r2, r1, 8
3   addi r2, r2, 66
4   sw r2, r0, 4
5   lb r4, r0, 5
6   addi r3, r0, 33768
7   lb r3, r3, 1
8   fence
9   loop_label:
10      subi r4, r4, 1
11      beq r4, r0, loop_label
12  rdtsc r0
```

When we start the emulator, it automatically loads the program and shows the first context screen, as described in sec. 5.4. As we see in fig. 3, the registers and the shown memory section are initialised to zero and the instruction queue and reservation station are still empty.
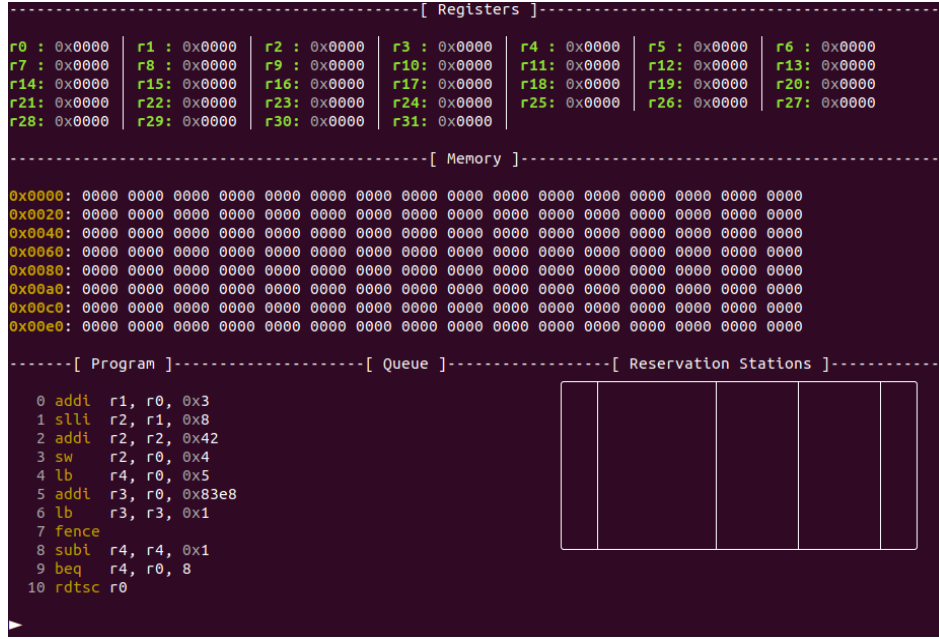
FIGURE 3: *Context screen at the start of the program*

As we can see in fig. 4, within the first two `steps` the instruction queue is filled with the first five instructions, which are subsequently issued into the reservation station as per Tomasulo's algorithm for out-of-order execution as discussed in sec. 4.2. Instruction 0 is immediately executed, since it is in the first slot of the reservation station and all of it operands are ready as soon as it is issued. By the checkmark in the rightmost column in the reservation station, we can see that it is ready to retire. Since its result is broadcasted directly after the instruction finishes executing, register 1 and the first operand of instruction 1 are already set to `0x003`.

As per Tomasulo's algorithm, in fig. 4 we see `SlotIDs` as placeholders for the result of the instruction in the respective reservation station slot, both in the registers and the operand lists of instructions in the reservation station. For example, register 2 is waiting for the result produced by the `addi` instruction in slot 2 of the reservation station and contains the placeholder `RS 002`. Note in particular, that for the `slli` instruction with `SlotID` 1 we see that its target register 2 already contains the `SlotID` of the next instruction. But in the operand list of the `addi` in slot 2 the reservation station we can see the `SlotID` of the `slli` instruction waiting to be replaced with the result value.

In the next step, the fence instruction is issued into the reservation station. Therefore no more instructions are issued into the reservation station until all currently issued instructions have been executed. This can be observed in fig. 5 and fig. 7, in which the empty slots in the reservation station are not refilled with the instructions waiting in the instruction queue.

After the `slli` and the `addi` instructions in slots 1 and 2 of the reservation station, the program contains two memory operations `sw` and `lb`. Since memory instructions take longer to execute, the `addi` instruction in slot 5 of the reservation station is executed out-of-order before the memory instructions retire. Figure 5 shows the reservation station that still contains the `sw` and `lb` instructions but from which the `addi` instruction has already retired.
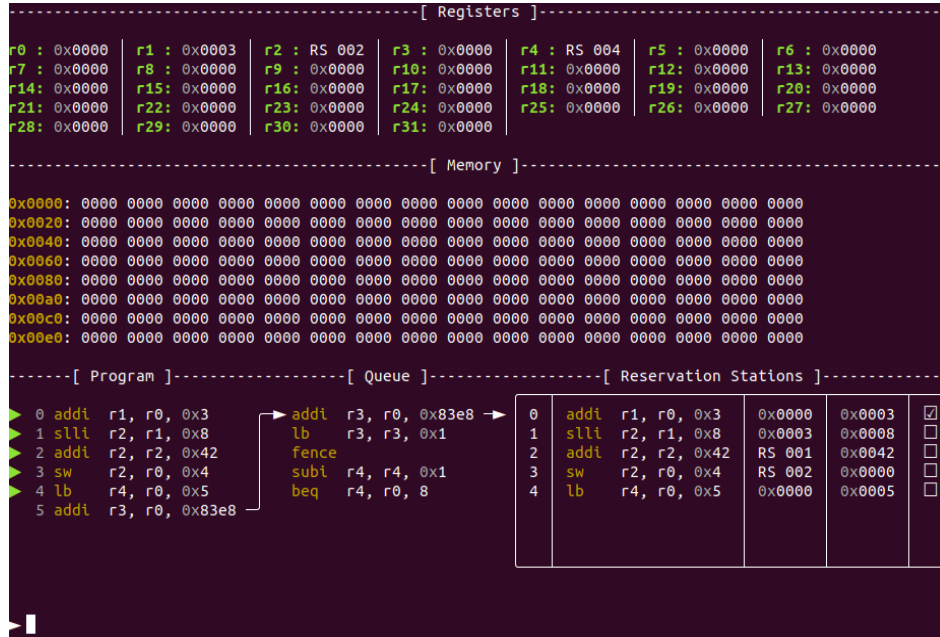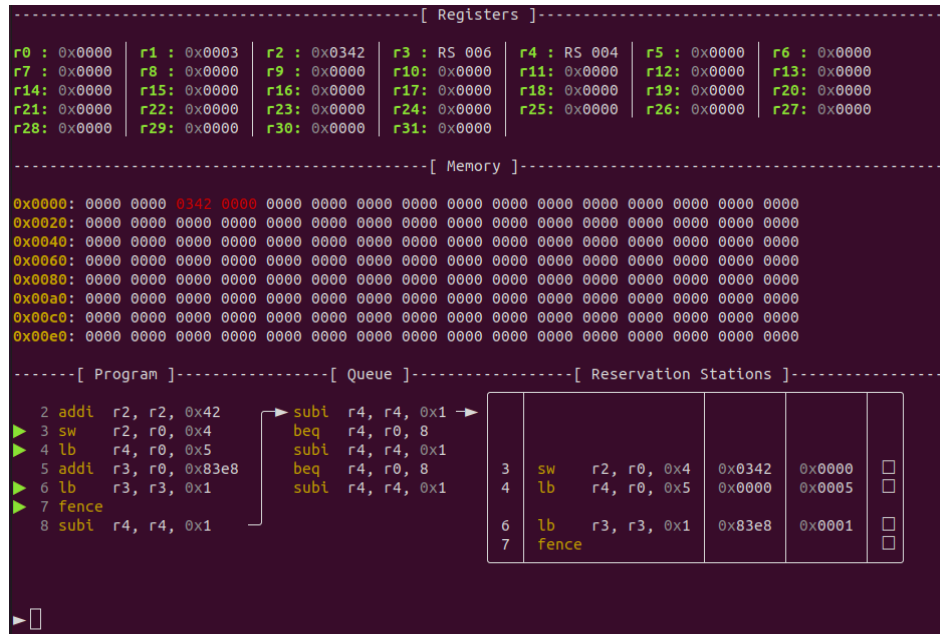
Figure 4: *Context screen after two* `steps`

Figure 5: *Context screen after out-of-order execution of the* `addi` *instruction in slot 5*

Figure 6: *Cached target address of the sw instruction*

During the execution of the `sw` instruction, the value `0x0342` is stored as a `Word` starting at memory address 4. In our example this is highlighted further by the the memory addresses 4-7 changing their color to red. This signifies that they are placed in the cache as a `cacheline` of length four bytes. Since we cannot visualize the whole memory all at once, we also offer a more detailed visualization of the whole cache, as depicted in fig. 6. The result of the store instruction is placed into memory multiple cycles before the instruction retires, both to model real world latencies in memory accesses and to leave enough time to check for faults during which transient execution happens, as we see in sec. 6.2. Therefore, we can already observe the aforementioned changes in the memory visualization in fig. 5, while the `sw` instruction is still being executed and present in the reservation station.

The `lb` instruction in slot 4 only reads one byte from memory address 5. Since the `sw` instruction places its `Word` value into memory in little endian order, the result of reading one byte from memory address 5 is `0x03`. This can be observed as the new value of register 4 in fig. 7, where the `lb` instruction has finished executing and is ready to retire, as shown by the tickmark.

With the `lb` instruction in slot 6 of the reservation station, we attempt to load a value from the inaccessible part of the memory, as described in sec. 4.1.5. During the execution, the value `0x42` from the inaccessible address is present in the target register 3, as we can observe in fig. 7. But before the instruction can retire, the fault is detected and the target register is rolled back to its previous state. Due to the rollback the reservation station is cleared and the subsequent instructions are put back into the instruction queue sec. 4.3. In fig. 8 we can see the previous value of `0x83e8` from the address calculation in register 3, and the rolled back instruction queue and reservation station as well as the fault message.

Now the `fence` instruction can be executed and the subsequent instructions are put into the instruction queue and issued to the reservation station. Since per the default settings all branches are
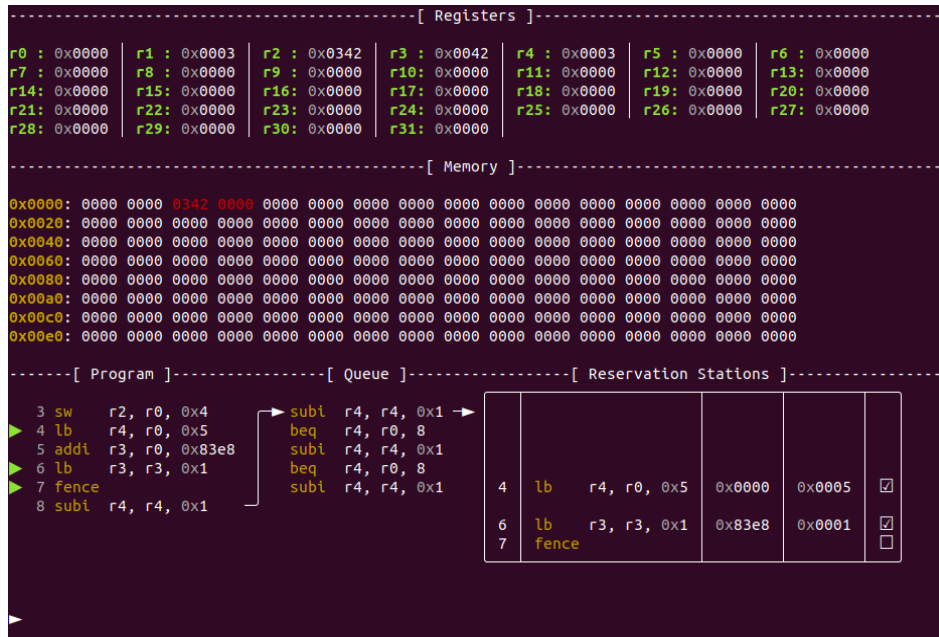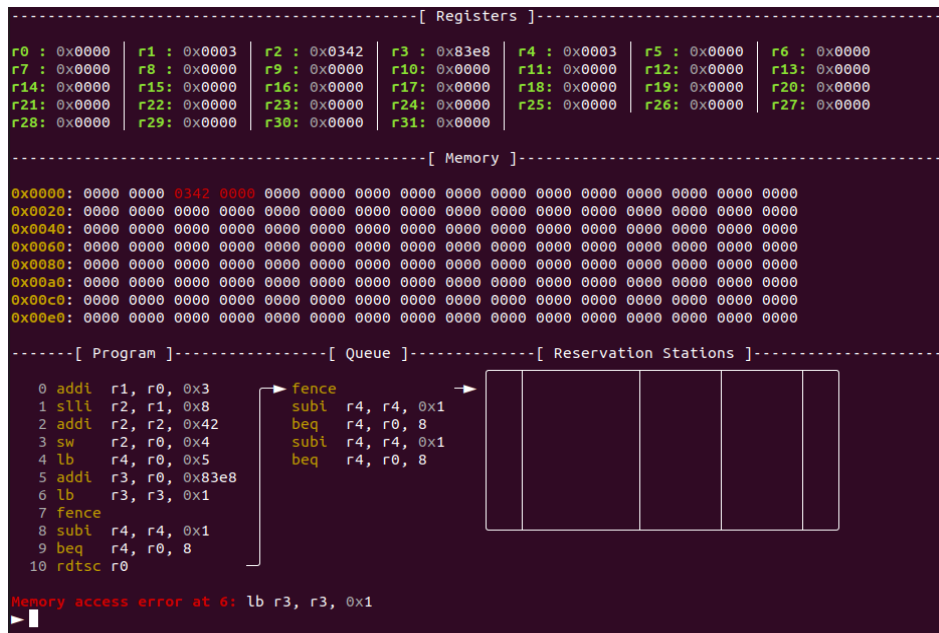
Figure 7: *Context screen with the* lb *instruction in slot 4 ready to retire*

Figure 8: *Rolled back memory fault*

Figure 9: *Loop instructions are speculatively put into the instruction queue and issued into the reservation station*

initially predicted as taken, we speculatively fill the instruction queue and reservation station with as many iterations of the loop as fit into them, as seen in fig. 9. Since the branch condition is already violated in the first iteration of the loop and the branch is not taken, we have a misprediction that results in a fault message and a rollback. During the rollback the reservation station is cleared of the excess instructions from the loop iterations that are not executed after all. Instead the `rtdsc` instruction, which follows the loop in the program, is put into the instruction queue, as we can see in fig. 10.

Lastly the `rdtsc` instruction is executed and shows in register 0 that the program took `0x0026` cycles to execute so far. The end of the program is marked by the end message as well as the empty instruction queue and reservation station, as we can see in fig. 11.

6.2 Demonstration of a Meltdown-Type Attack

Jan-Niklas Sohn

This chapter demonstrates that a Meltdown-type attack can be performed inside our CPU emulator. This is shown through an example program that performs such an attack. The example program and the inner workings of the attack are described in sec. 6.2.1. In sec. 6.2.2 the attack is categorized according to the naming scheme established by Canella et al. [Can+19] and compared to the attack described in the original Meltdown paper [Lip+18].

6.2.1 A Meltdown-Type Attack

The Meltdown-type attack described in this section starts with loading a byte from inaccessible memory. This access causes a fault. During the following transient execution, the value resulting
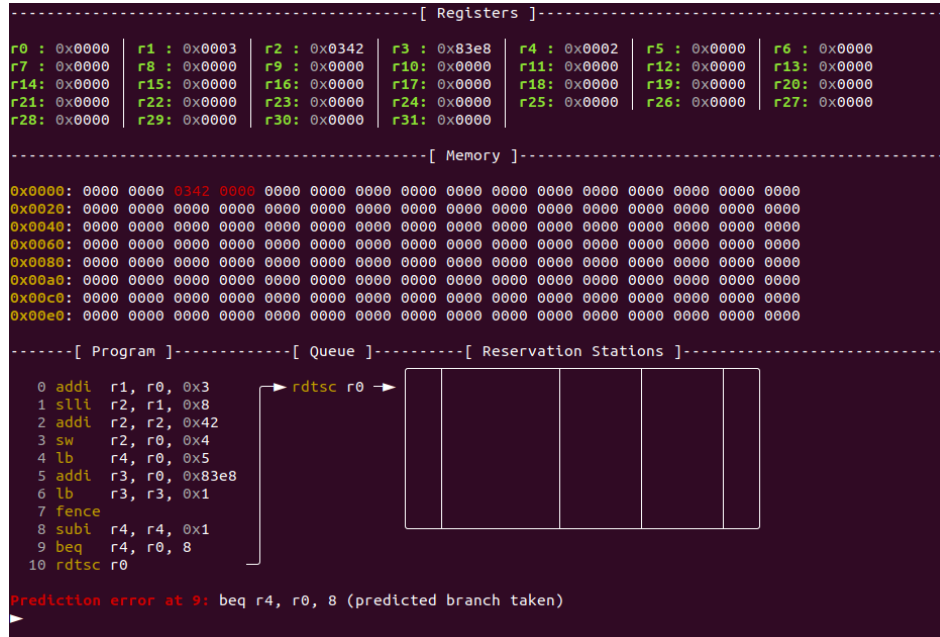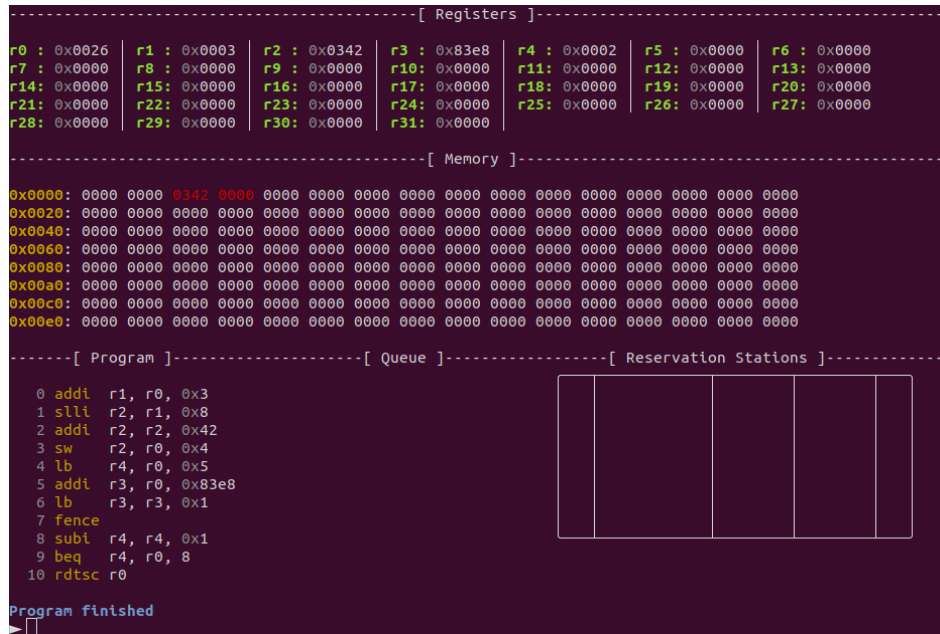
Figure 10: *Rolled back loop fault*

Figure 11: *Context screen at the end of the execution of the example program*

```
1  lb r1, r0, 0xc000
2  slli r1, r1, 2
3  lb r2, r1, 0x1000
```

Listing 1: *Core part of our Meltdown-type attack, loading a byte from an inaccessible address and encoding it into the cache.*

```
--------------[ Reservation Stations ]--------------

0  │ lb    r1, r0, 0xc000 │ 0x0000 │ 0xc000 │ ☑
1  │ slli  r1, r1, 0x2    │ 0x0042 │ 0x0002 │ ☐
2  │ lb    r2, r1, 0x1000 │ RS 001 │ 0x1000 │ ☐


5  │ addi  r3, r0, 0x4    │ 0x0000 │ 0x0004 │ ☐
6  │ addi  r4, r0, 0x400  │ 0x0000 │ 0x0400 │ ☐
7  │ fence                │        │        │ ☐
```

Figure 12: *The faulting load instruction (slot 0) provides the secret value 0x42 to the following instruction (slot 1).*

from the faulting load is used to index into a previously prepared *probe array*. This causes the accessed entry of the probe array to be cached. Thus, the value resulting from the faulting load is encoded into the cache. After the transient execution has ended and the rollback has been performed, the attack checks which entry of the probe array is cached. This is achieved by loading each entry and measuring its access time. In this way, the value retrieved by the faulting load is leaked, even though it is supposed to be inaccessible.

The code in lst. 1 illustrates the core part of our attack. It loads a byte from the inaccessible memory address 0xc000. The resulting value is shifted to the left by 2 bits, multiplying the value by 4. This is done because the entries of our probe array are 4 bytes apart, which matches the default cache line size of our emulator. The shifted value is then used to index into the probe array, which starts at address 0x1000. When stepping through the program until the faulting load instruction has executed and provided its result to subsequent instructions, as is shown in fig. 12, we observe that the faulting load indeed provides the secret value 0x42 to the following slli instruction. After the transient execution is terminated by the rollback, we use the show cache command to investigate the state of the cache. As fig. 13 shows, we see that the entry at index 0x42 of the probe array is cached. Thus, the secret value obtained during transient execution was successfully encoded into the cache.

The code in lst. 2 displays the cache-decoding part of our attack. It loops over the probe array and records the shortest access time. During the loop, register r1 stores the index of the shortest seen access, and register r2 stores its access time. Register r3 stores the current index into the probe array, and register r4 stores the length of the probe array. To simplify the code, the indices and lengths are

Figure 13: *After the rollback, the cache contains an entry with index* `0x2` *and tag* `0x11`, *which corresponds to address* `0x1108` *or index* `0x42` *of the probe array.*

```
 1   // Loop over the probe array, record the shortest access time:
 2
 3   // Index of shortest access, in bytes
 4   addi r1, r0, 0
 5   // Time of shortest access
 6   addi r2, r0, -1
 7   // Current index, in bytes
 8   addi r3, r0, 0
 9   // Probe array length, in bytes
10   addi r4, r0, 0x400
11   probe:
12
13   // Perform timed access into probe array
14   fence
15   rdtsc r5
16   lb r7, r3, 0x1000
17   fence
18   rdtsc r6
19   sub r5, r6, r5
20
21   // Update shortest access
22   bgu r5, r2, skip
23   addi r1, r3, 0
24   addi r2, r5, 0
25   skip:
26
27   // Increment index and loop
28   addi r3, r3, 4
29   bne r3, r4, probe
```

Listing 2: *Cache-decoding part of our Meltdown-type attack, looping over the probe array and recording the shortest access time.*

stored in units of bytes and not in units of probe array entries. These registers are initialized in the block starting in line 3. The loop body starting in line 13 performs an access into the probe array, and records the access time. This is done by surrounding the `lb` instruction that performs the access with `fence; rdtsc` sequences that record the cycle counter before and after the access. The fence in front of the `lb` instruction ensures that the recorded cycle count closely matches the time the `lb` instruction is issued. It waits for preceding instructions to complete, so that the reservation station has enough free slots to issue the `lb` instruction immediately after issuing the `rdtsc` instruction. The fence following the `lb` instruction ensures that the load completes before the second cycle count is recorded. The second part of the loop body, starting at line 21, updates the shortest access index and time in the case that the load was the shortest yet. The loop tail starts at line 27 and increments the index into the probe array. It checks if the end of the probe array was reached, looping back up to line 13 if that is not the case.

When we observe the register values after having executed the whole program, as is shown in fig. 14, we can see that register `r1` contains the value `0x108`. This means that index `0x42` had the shortest access time; the secret was decoded from the cache successfully. Since the default cache configuration established in sec. 4.5.2 uses 4 cache sets and 4 cache ways, iterating over the probe array might evict the cached entry before the short access time was seen. Thus, for this attack we use 64 cache sets and 4 cache ways.

```
----------------------------------------------[ Registers ]-----------------------------------------
r0 : 0x0000   r1 : 0x0108   r2 : 0x000e   r3 : 0x0400   r4 : 0x0400   r5 : 0x0011   r6 : 0x1cfc   r7 : 0x0000
r8 : 0x0000   r9 : 0x0000   r10: 0x0000   r11: 0x0000   r12: 0x0000   r13: 0x0000   r14: 0x0000   r15: 0x0000
r16: 0x0000   r17: 0x0000   r18: 0x0000   r19: 0x0000   r20: 0x0000   r21: 0x0000   r22: 0x0000   r23: 0x0000
r24: 0x0000   r25: 0x0000   r26: 0x0000   r27: 0x0000   r28: 0x0000   r29: 0x0000   r30: 0x0000   r31: 0x0000
```

Figure 14: *The probe array entry at offset `0x108` had the shortest access time, which corresponds to index `0x42`.*

Several components of our CPU emulator interact to make this Meltdown-type attack possible.

- Out-of-order execution leads to a transient execution window after the faulting load.
- The memory subsystem provides the value stored in memory as the result of the load instruction, even though the instruction faults.
- The cache is used as a transmission channel from the transient execution domain to the architectural domain.

6.2.2 Comparison With Meltdown-Type Attacks on Real CPUs

Meltdown-type attacks are typically classified according to the naming scheme first introduced by Canella et al. [Can+19]. This scheme adds two name components to the "Meltdown-" stem. The first component describes the microarchitectural condition that causes the leaking fault. The second component describes the microarchitectural element that data is leaked from. Our scheme for checking memory accesses models the way the US-bit is usually set up on x86 systems, where the top half of memory is reversed for the kernel and accesses to it cause a fault. If the value accessed by our attack already resides in the cache, the value is leaked from the cache. This results in a variant similar to Meltdown-US-L1. If the value is not cached, it is leaked from main memory, which results in a variant similar to Meltdown-US-Mem.

Listing 3 displays the core part of the original Meltdown attack, taken verbatim from the corresponding paper [Lip+18]. This code loads a byte from a kernel address, causing a fault. In the following transient execution, the value retrieved by the faulting load is shifted left by 12 bits, multiplying the value by 4096. If the resulting value is zero, the faulting load is executed again, until it produces a non-zero value. Finally, the shifted value is used to index into the probe array. This core part of the attack is very similar to the core part of our Meltdown-type attack (described above and shown in lst. 1). Beside the differing size of the probe array entries, the only difference is the retry loop. On real x86 CPUs the faulting load only leaks data sporadically. Because of this, immediately performing the load again if the leak fails increases the attack's performance [Lip+18, sec. 5.2]. In contrast to this, our CPU emulator is completely deterministic; the core of our attack succeeds every time and no retry loop is necessary.

6.3 Demonstration of a Spectre-Type Attack

This chapter demonstrates that a Spectre-type attack can be performed inside our CPU emulator. This is shown through an example program that performs such an attack. The example program

Jan-Niklas Sohn

```
1   ; rcx = kernel address , rbx = probe array
2   xor rax , rax
3   retry :
4   mov al , byte [rcx]
5   shl rax , 0xc
6   jz retry
7   mov rbx , qword [rbx + rax]
```

Listing 3: *Core of the original Meltdown attack, loading a byte from an inaccessible address and encoding it into the cache.* [*Lip+18, lst. 2*]

and the inner workings of the attack are described in sec. 6.3.1. In sec. 6.3.2 the attack is categorized according to the naming scheme established by Canella et al. [Can+19] and compared to the attack described in the original Spectre paper [Koc+19].

6.3.1 A Spectre-Type Attack

The Spectre-type attack described in this section starts with setting up an array of eight elements in memory, which are initialized to zero. This *victim array* is immediately followed by one out-of-bounds element with the *secret value* 0x41. The victim array is then iterated over, using each array element to index into a probe array. Thus, every array element is encoded into the cache. This loop trains the BPU to predict the conditional branch that starts a new iteration as taken. After the final loop iteration, the BPU will thus still predict this branch as taken. This results in an additional loop iteration, that is only executed transiently. Nonetheless, this additional iteration accesses the out-of-bounds element and encodes the secret value in the cache. After the true branch condition has been resolved and the rollback performed, the attack checks which entry of the probe array is cached. This is achieved by loading each entry and measuring its access time, as already described for the Meltdown-type attack in sec. 6.2.1. In this way, the secret value of the out-of-bounds array element is leaked, even though it is supposed to be inaccessible.

The code in lst. 4 illustrates the setup and core part of our attack. The block starting in line 1 performs a number of stores to prepare the victim array, starting at address 0x1000. It stores eight zero bytes, followed by one byte with the value 0x41, which constitutes the out-of-bounds element. This is followed by a loop over the victim array, starting in line 14. During this loop, register r2 stores the index of the current element, and register r3 stores the length of the victim array. The loop body starting in line 18 loads the current element of the victim array. It then shifts the element's value left by 2, multiplying it by 4. This is done because the entries of our probe array are 4 bytes apart, which matches the default cache line size of our emulator. The shifted value is then used to index into the probe array, which starts at address 0x2000. The loop tail starts at line 23 and increments the index into the victim array. It checks if the end of the victim array was reached, looping back up to line 18 if that is not the case.

Figure 15 shows the state of the BPU at the end of the last loop iteration. As we can see, the upcoming conditional branch that should terminate the loop is predicted as taken. This leads to an additional loop iteration with out-of-bounds index, that is only executed transiently. Continuing to the loop body, we observe that the out-of-bounds index is used to access the secret value behind the array, as shown in fig. 16. After executing the whole program, we use the `show cache` command to investigate

```
1   // Set up array at 0x1000, 8 elements, all zeroes, followed by one out-of-
           bounds 0x41 value
2   addi r1, r0, 0x1000
3   sb r0, r1, 0
4   sb r0, r1, 1
5   sb r0, r1, 2
6   sb r0, r1, 3
7   sb r0, r1, 4
8   sb r0, r1, 5
9   sb r0, r1, 6
10  sb r0, r1, 7
11  addi r2, r0, 0x41
12  sb r2, r1, 8
13
14  // Loop over array, encode every value in cache
15  addi r2, r0, 0
16  addi r3, r0, 8
17  loop:
18  // Load array element
19  lb r4, r2, 0x1000
20  // Encode value in cache
21  slli r4, r4, 2
22  lb r4, r4, 0x2000
23  // Increment loop index
24  addi r2, r2, 1
25  // Loop while index is in bounds
26  bne r2, r3, loop
27
28  // Although the out-of-bounds element was never accessed, the value 0x41 is
           encoded in the cache
```

LISTING 4: *Setup and core part of our Spectre-type attack, iterating over an array and transiently accessing an out-of-bounds element.*

the state of the cache. As fig. 17 shows, we see that the entry at index `0x41` of the probe array is cached. Thus, the secret value of the out-of-bounds array element was successfully encoded into the cache.

To decode the secret value from the cache, the technique displayed in lst. 2 is used, which was already covered in detail in sec. 6.2.1. The only difference is that the probe array of our Spectre-type attack is located at address `0x2000` instead of `0x1000`.

Several components of our CPU emulator interact to make this Spectre-type attack possible.

■ Branch prediction leads to a transient execution window after the mispredicted loop condition.

■ The cache is used as a transmission channel from the transient execution domain to the architectural domain.

6.3.2 COMPARISON WITH SPECTRE-TYPE ATTACKS ON REAL CPUS

Just like Meltdown-type attacks, Spectre-type attacks are typically classified according to the naming scheme first introduced by Canella et al. [Can+19]. This scheme adds three name components to the "Spectre-" stem. The first component describes the microarchitectural element that is trained to

```
► show bpu
Index │ Counter
──────┼────────
  00 │ 2
  01 │ 3
  02 │ 2
  03 │ 2
  04 │ 2
  05 │ 2
  06 │ 2
  07 │ 2
  08 │ 2
  09 │ 2
  10 │ 2
  11 │ 2
  12 │ 2
  13 │ 2
  14 │ 2
  15 │ 2
►  _
```

FIGURE 15: *The conditional branch is mapped to index 1 of the BPU, which has a counter of 3 or "strongly taken".*

```
──────────────[ Reservation Stations ]──────────────
 8   │ sb    r0, r1, 0x7      0x0000    0x1000   ☑
10   │ sb    r2, r1, 0x8      0x0041    0x1000   ☐
13   │ lb    r4, r2, 0x1000   0x0007    0x1000   ☐
14   │ slli  r4, r4, 0x2      RS 002    0x0002   ☐
13   │ lb    r4, r2, 0x1000   0x0008    0x1000   ☐   ←
15   │ lb    r4, r4, 0x2000   0x0000    0x2000   ☑
17   │ bne   r2, r3, 13       0x0008    0x0008   ☑
15   │ lb    r4, r4, 0x2000   RS 003    0x2000   ☐
```

FIGURE 16: *The out-of-bounds access to element 8 of the victim array is about to be executed.*

FIGURE 17: *When the program finishes, the cache contains an entry with index 0x1 and tag 0x21, which corresponds to address 0x2104 or index 0x41 of the probe array.*

perform a certain misprediction. The second component indicates whether the training is performed in the same address space ("SA") or a different address space ("CA") than the targeted misprediction, and the third component indicates whether the training is performed using a branch at the same (in-place, "IP") or a branch at a different (out-of-place, "OP") address than the misprediction. As described in sec. 4.1.4, our BPU only has a single prediction mechanism that matches the PHT of modern x86 CPUs. Additionally, our CPU emulator only has a single address space. In the case of our Spectre-type attack, we perform the training in-place, using the same branch instruction that produces the misprediction. Because of this, our Spectre-type attack is classified as Spectre-PHT-SA-IP. Since our BPU uses a limited number of address bits to index its internal counter table, an out-of-place training is also possible, which would result in an attack classified as Spectre-PHT-SA-OP.

The original Spectre paper describes an attack very similar to our Spectre-type attack. In this attack, an array is repeatedly indexed in-bounds to train the branch predictor, and then a single time out-of-bounds to force a misprediction. During the following transient execution, the out-of-bounds access retrieves a secret value that is subsequently encoded into the cache. [Koc+19, sec. IV]

6.4 Mitigations Demonstration

Lennart Hein

As described in sec. 2.4.3, there are several mitigations available, that can be used to protect against Meltdown and Spectre attacks. In the following, we will present our implementations of some of these mitigations, and discuss their effectiveness against the demonstrated attacks from sec. 6.2.1 and sec. 6.3.1.

6.4.1 Disable Out of Order Execution

A simple, but costly mitigation is to disable out-of-order execution. Such an effect can be achieved in our emulator, by limiting the number of slots in the reservation station to 1. The user simply changes the according parameter in the configuration file.

To demonstrate, we will use the following configuration file:

```
1  ...
2  ExecutionEngine:
3      slots: 1
4  ...
```

We further use the following program, which is identical to the one used in sec. 6.2.1, and is provided in `demo/meltdown_small.tea`:

```
1  lb r1, r0, 0xc000
2  slli r1, r1, 4
3  lb r2, r1, 0x1000
```

After issuing a `continue` command which breaks at the rollback of the first instruction, only the first instruction has been executed. There is no transient execution window, and the secret value is not encoded in the cache. At the end of the program, while the target address is still in the cache,

it cannot be read by the attacker, and the encoded value in the cache will be located at `0x1000`, independently of the secret value.

This mitigation also is effective against the demonstrated Spectre attack supplied in `demo/spectre.tea`.

6.4.2 Return Zero Mitigation

An undocumented mitigation Intel employs to mitigate Meltdown attacks is the return zero mitigation. When performing an illegal read, the CPU passes a zero value to dependent instructions. Our implementation can be enabled as an option in the configuration file.

Using the following configuration file, we can demonstrate this mitigation:

```
1  ...
2  Mitigations:
3      illegal_read_return_zero: True
4  ...
```

We again use the program provided in `demo/meltdown_small.tea`. After forwarding 6 cycles using `step 6`, we can see that `slii r1, r1, 0x4` has been supplied with the result of the first instruction over the common data bus; the content of the field in the slot is zero. This also results in `0x1000` being cached, i.e. zero being encoded as the secret, independently of the secret value.

This mitigation does NOT generally work against Spectre attacks.

6.4.3 Fence Mitigation

A mitigation that can be implemented on a compiler level or during development is the Fence Mitigation. Here specific instructions can be proofed against the attacks by inserting fence instructions. This could be inserting a fence after a load instruction to prevent the demonstrated Meltdown attack, or inserting a fence after a branch instruction to prevent the demonstrated Spectre attack.

Using the standard configuration file, we can demonstrate this mitigation versus the above seen Meltdown attack, by inserting fence instructions after loads:

```
1  lb r1, r0, 0xc000
2  fence
3  slli r1, r1, 4
4  lb r2, r1, 0x1000
5  fence
```

Stepping into the program, we can see that the first fence blocks following instructions from entering the reservation station. Only after the rollback occurs, the first load instruction retires and the fence allows subsequent instructions to enter the reservation station. Since the attack depends on the subsequent instructions to be executed before the rollback is executed, the secret byte cannot be encoded into the cache, and thus not leaked.

The mitigation also works against the demonstrated Spectre attack.

6.4.4 Flushall Microcode Injection Mitigation

The emulator allows the user to specify microcode programs, that are injected into the CPU after a rollback. One example how this can be used to implement a mitigation is the Flushall Microcode Injection Mitigation. Here we inject a microcode program that flushes the entire cache, after the rollback is executed. Since the secret byte is encoded into the cache, it subsequently is flushed and cannot be read by the attacker.

To enable this mitigation against the demonstrated Meltdown attack supplied in `demo/meltdown_small .tea`, we use the following configuration file:

```
1  Microprograms:
2       InstrLoad: demo/flushall.tea
```

Using `continue` will forward execution to the rollback. As indicated by the UI, the `flushall` and `fence` instructions are injected before regular execution can be resumed. Using `retire` followed by `show cache`, we can observe the cache being empty, thus the secret value cannot be leaked.

The mitigation also works against the demonstrated Spectre attack.

7 CONCLUSION

Felix Betke

Our goal is to create an emulator that allows users to experiment with simplified versions of select Meltdown and Spectre type attacks without access to vulnerable hardware. The choice of Python as the programming language and a limited number of UI libraries ensure our emulator can be used on anything that runs Python, regardless of operating system or architecture. We document our implementation and provide a user manual. Our evaluation (see chapter 6) shows that the presented emulator can run simple programs while using the concepts of out-of-order and speculative execution. Further, our versions of Meltdown-US-L1 and Spectre v1 attacks are possible given the default configuration and additional example programs. The configuration file allows users to enable mitigations and experiment with their own microprograms that are run after each rollback.

7.1 LIMITATIONS

While our initial goals are reached, some limitations of our emulator exist. Firstly, due to not running an operating system, we lack virtual addresses and paging, therefore not fully modelling the Meltdown-US-L1 attack and how exceptions can be forced by user space processes. Secondly, our current implementation of read operations behaves differently than Intel's CPUs are believed to behave. In our version, if needed data is not present in the cache, instructions stall until the data is available, rather than computing on values taken from other microarchitectural buffers [Sch+19]. As a result, our version of Meltdown-US-L1 is still possible, but in the event the data to be stolen is not cached, users will observe a potentially unexpected stall, not a read from other buffers (see sec. 4.5.2). Since Meltdown-US-L1 does not strictly rely on any of those microarchitectural buffers, we do not model them. Still, this difference is clearly communicated to users by our modified version of Meltdown-US-L1 and we are confident our simplified emulator allows users to better understand the chosen attacks.

7.2 FUTURE WORK

One of the obvious improvements would be to make the emulator vulnerable to other Meltdown and Spectre variants. For Meltdown, this would require the addition of new microarchitectural buffers, such as the load, store, or line-fill buffers (see sec. 2.1). For Spectre, a more elaborate BPU (see sec. 2.3) is needed. Other improvements could be to enable users to run multiple programs (attacker and victim), either by context switching or Hyper-Threading, or create an operating system to support virtual addressing. Lastly, one could add further improvements to the UI in regards to visualization and functionality (conditional breakpoints, for example).

References

[And17]   Andrew Waterman, Krste Asanovic: *The RISC-V Instruction Set Manual*. 2017. url: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf (visited on 03/19/2022).

[Boe21]   Boes, Felix Jonathan: *Lecture MA-INF 3238: Side Channel Attacks*. 2021. url: https://net.cs.uni-bonn.de/de/wg/itsec/lehre/ss-2021/side-channel-attacks/.

[Can+19] Canella, Claudio et al.: "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *USENIX Security Symposium*. extended classification tree at https://transient.fail/. 2019.

[Can+20] Canella, Claudio et al.: "KASLR: Break it, fix it, repeat". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 2020, pp. 481–493.

[EPA15]   Evtyushkin, Dmitry ; Ponomarev, Dmitry V. ; Abu-Ghazaleh, Nael B.: "Covert channels through branch predictors: a feasibility study". In: *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy* (2015).

[Fis]   Fischer, Stefan: *Python Prompt Package*. url: https://github.com/sfischer13/python-prompt (visited on 03/17/2022).

[Fre]   Free Software Foundation: *GDB: The GNU Project Debugger*. url: https://sourceware.org/gdb/ (visited on 03/15/2022).

[Gru20]   Gruss, Daniel: "Transient-Execution Attacks". 2020. url: https://gruss.cc/files/habil.pdf (visited on 03/11/2022).

[Int18]   Intel: *Intel Analysis of Speculative Execution Side Channels*. 2018. url: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf (visited on 03/11/2022).

[Koc+19] Kocher, Paul et al.: "Spectre attacks: Exploiting speculative execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019.

[Lip+18] Lipp, Moritz et al.: "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[Rig]   Riggle, Zach: *pwndbg*. url: https://github.com/pwndbg/pwndbg (visited on 03/17/2022).

[Sch+19] Schaik, Stephan van et al.: "RIDL: Rogue In-flight Data Load". In: *S&P*. 2019.

[Tom67]   Tomasulo, Robert M: "An efficient algorithm for exploiting multiple arithmetic units". In: *IBM Journal of research and Development* 11.1 (1967), pp. 25–33.

[VKM19]   Vila, Pepe ; Köpf, Boris ; Morales, José F: "Theory and practice of finding eviction sets". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 39–54.

[Wika]   WikiChip: *Macro-Operation*. url: https://en.wikichip.org/wiki/macro-operation (visited on 03/28/2022).

[Wikb]   WikiChip: *Micro-Operation*. url: https://en.wikichip.org/wiki/micro-operation (visited on 03/28/2022).

[Wikc]   WikiChip: *Skylake Microarchitecture*. url: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client) (visited on 03/10/2022).

LIST OF FIGURES

Listings