TODO

TODO

Bachelor Thesis

by

TODO

in fulfillment of requirements for degree

Bachelor of Science (B.Sc.)

submitted to

Rheinische Friedrich-Wilhelms-Universität Bonn

Institut für Informatik IV

Arbeitsgruppe für IT-Sicherheit

in degree course

Computer Science (M.Sc.)

First Supervisor: TODO
University of Bonn

Second Supervisor: TODO
TODO

Sponsor: TODO
University of Bonn

TODO

Abstract

TODO

Contents

1 Introduction

Contrary to 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 which branches are taken. With either technique, the CPU might encounter cases where the current CPU state must be rolled back to a previous one. For out-of-order execution, this happens if an instruction raises an exception (illegal memory access, for example). For speculative execution, this happens if a branch is mispredicted. A rollback causes 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 change the contents of caches and other buffers. The disclosure of both Spectre and Meltdown 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. And while the performance losses introduced by software and hardware mitigations are measurable, neither vulnerability can be exploited freely 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 effected 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.

We design and implement a graphical CPU emulator that supports single-step out-of-order and speculative execution and is vulnerable to select variants of Spectre and Meltdown. While it is a simplification in comparison to real hardware, the emulator 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 microprograms 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 supposed to be vulnerable. Secondly, chapter 3 further describes the target audience of the emulator and which variants of the vulnerabilities have been chosen, while chapter 4 documents the implementation of the emulator by describing each main component and explaining how rollbacks are implemented. Additionally, it contains an overview of 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, and determines how effective the implemented mitigations are. Finally, chapter 7 summarizes the other chapters, briefly reflects on how valuable the emulator might be to our target audience, and gives ideas for future improvements.

2 Background

This chapter briefly covers the theoretical background needed to use the presented emulator and continue its development. The reader is assumed to have an understanding of elementary CPU concepts, such as pipelining, caching, and virtual addressing.

2.1 CPU

TODO

2.2 Out-of-order execution

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]. 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 components, the first of which is the reservation station, which collects the operands of instructions until they are executed by the execution units. Crucially, the corresponding execution units do not need to wait until all operands are present and can instead compute other instructions. 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 written to registers first, just to be read again by other instructions that need them as operands.

Strictly speaking, according to Tomasulo, each set of execution units needs its own reservation station, however, more modern implementations by Intel use a single unified reservation station that handles all types of instructions, rather than just one [Tom67] [Wik].

2.3 Speculative execution

Speculative execution allows a CPU to predict the outcome of comparisons and other branch instructions. This prevents stalls when waiting for the instruction that determines which branch is taken to finish. Similarly to out-of-order execution, rollbacks are needed in some cases. However, in addition to exceptions, they also occur if a branch was mispredicted. [Gru20]

To predict the outcome of branch instructions, CPUs include a branch prediction unit (BPU) [Gru20]. While available in different configurations, many modern CPUs record the most recent outcomes of a branch with a counter [Gru20] that is either incremented if the branch is taken, or decremented.

2.4 Meltdown and Spectre

Meltdown [Lip+18] and Spectre [Koc+19] abuse out-of-order and speculative execution to leak data from memory addresses that are normally inaccessible to the attacker over the caches of the CPU.

2.4.1 Meltdown

On a high level, Meltdown [Lip+18] works by forcing exceptions when reading data inaccessible to the attacker and transiently encoding this data 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] 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. It is important that the data to be stolen is currently cached. 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 the scope of this work.

2.4.2 Spectre

Spectre [Koc+19], on the other hand, relies on the CPU mispredicting a branch and transiently executing instructions that are not part of the correct execution path. This misprediction in a victim process can be induced by maliciously configuring the BPU (sec. 2.3). 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 variant 1 of Spectre, which takes advantage of the CPU mispredicting the outcome of comparison instructions.

2.4.3 Mitigations

The mitigations for Meltdown are available in both software and hardware, some of which are implemented in the emulator to allow users to 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 normally inaccessible data into the cache.

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 KTPI (or KAISER) [Lip+18], are highly effective, but out of scope.

Unlike Meltdown, 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 questionable mitigation is to simply disable speculative execution [Koc+19]. Alternatively, Intel recommends potential victim's 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]. A fence instruction is provided as part of the emulator's instruction set.

An additional mitigation that works against both Meltdown and Spectre is to flush the entire cache after a rollback. While highly inefficient, it does prevent the encoding the inaccessible data during the transient execution phase of the attacks. This mitigation can be realized using the emulator by a sequence of instructions that are executed after each rollback.

3 Catchy Lab title

3.1 Abstract

3.2 Introduction (1 page)

```
general task/ goal
    CPU emulator with actual out of order execution, maybe speculative execution and comprehensible implementation and documentation
    implement different µ-architectural features like out-of-order exe., speculative exe. b.c. we want to observe and teach µ-arch. attacks
brief context of the task/ goal (e.g. a sentence on Meltdown and why it is important to understand it)
structure of the report
```

3.3 Brief Theoretical Background (2-3 pages)

```
premise according to Felix: reader knows SCA lecture -> brief recaps/ reminders to reference back to from the other chapters

### really brief introduction to CPU

    multiple components
        maybe mostly via picture

    how they work together

    maybe data-flow from instruction to result

    a lot of hypothticals due to trade secrets

    suggested literature:
        maybe Gruss Diss. and other works on cbsca
        maybe textbooks
        maybe CPU wiki for pictures

### out-of-order execution

    a bit more in depth b.c. this is not content of the sca lecture

    maybe why do we need it/ advantages
```

```
24
25      brief explanation of Tomasulo and how it works
26
27      suggested literature:
28          original Tomasulo paper from 1965
29          maybe a more recent/ didactically edited explanation (is this in
        texbooks?)
30
31      also something about speculative execution
32
33  ### Meltdown and Spectre
34
35      brief introduction to Meltdown (and Spectre) in general
36
37      slightly more in-depth introduction to the Meltdown attack we picked
        and want to run in our emulator
38
39      highlight the relevant parts of the CPU, maybe mention how they
        interact in Meltdown (and Spectre)
40
41      mitigations and maybe how successfull they are
42
43      suggested literature:
44          ggf. Dissertation Gruss if we mention basic cbsca
45          Gruss et al. papers on Meltdown and Spectre e.g. https://gruss.cc/
        files/meltdown.pdf or https://gruss.cc/files/meltdown_cacm.pdf
46          Canella, Gruss et al. on mititgations https://gruss.cc/files/
        transient-attacks.pdf
```

3.4 maybe Specification of the task (1 page)

```
1  emulator to execute and teach Meltdown
2      who is the target audience
3      which Meltdown attacks specifically
4      etc. further concretisations
5
6  are there existing solutions/ related works?
7      Felix' old emulator? citable?
8      suggested literature:
9          todo
```

3.5 Our Emulator Program/ backend (17-18 pages)

```
1  maybe general info e.g. that we wrote it in python, if we used special
       tools/ libraries
2  for everything we implemented:
```

```
3         which part of a real life cpu does this model/ how is this done in
      real life cpus
4         briefly how does it work
5         why did we choose to model it like we do
6             nice code structure/ code easy to understand by students
7             some features more or less relevant for meltdown
8             etc.
9         maybe what did we leave out
10        challenges?
11
12  ### CPU Components and our equivalents/ models (10-11 pages)
13
14      modular setup based on real life CPUs and nice coding conventions
15
16      #### CPU
17
18          contains/ controls the rest
19
20          preparation (loading the program, init the rest)
21
22          ticks
23
24          coordinate rollbacks?
25
26      #### program handling
27
28          instructions
29
30          parser
31
32      #### data handling
33
34          how we model and handle data
35
36          byte, word
37
38      #### scheduling
39
40          BPU
41
42          frontend
43
44      #### memory
45
46          mmu
47
48              how do we store data
49
50              how do we model data access (i.p. wrt Meltdown)
```

```
51
52          cache
53
54              what kinds of caches can we represent
55
56              how do we model if something is cached and the access times (i.
    p. wrt Meltdown)
57
58      #### execution
59
60          Reservation Station/ Slots
61
62          unlimited execution units
63
64  ### Tomasulo (2 pages)
65
66      our version of out-of-order execution
67
68      where in our program do we implement which components
69
70      what did we leave out/ do differently
71
72  ### Rollbacks/ Exception handling (2-3 pages)
73
74          in particular wrt making Meltdown possible
75
76          general goal/ concept of rolling back after a misprediction or an
    exception
77
78          snapshots and interaction of the CPU components
79
80  ### ISA (2 pages)
81
82      overview of ISA
83
84      µ-instr. vs. ISA
85          mixture of both in one: only µ-code, but more abstract than in real
     life
86          do not need to think about page faults etc. anyways, do not need
    overly complex instructions
87
88      reasoning behind the choice of instructions (manageable size and
    instructions (e.g. no divide by zero) balanced with functionality
    particularly wrt Meltdown)
89
90
91  ### config files (1 page)
92
93      what can be configured without changing the source code
```

```
94          why these variables? relevant for Meltdown?
```

3.6 Our Visualisation and Usage/ Frontend (10 pages)

```
ggf. gemäß Anmerkung von Lenni umsortieren: die idee das UI nicht mit in
    die 'unsere designetnscheidungen' zu nehmen, sondern im prinzip so als
    Manual abzukapseln finde ich eigentlich ganz gut. Müssen wir aber dann
    mal in der Praxis schauen. Soll ja keine didaktischen begründungen das
    manual zu sehr aufblähen, vllt wird das sinnvoll, dann einen teil des
    UI im "backend" kapitel einzubauen, und dann wirklich ein cleanes
    manual kapitel zu haben

### general concept

    goal: UI for the emulator with visualisation of CPU/ memory components
    and their contents

    in terminal
        maybe comparison to existing analysis tool/ debugger gdb

    triggers/ controls the actual emulator
        overview features, e.g. breakpoints, step-by-step and stepback

    maybe which tools/ libraries were used?

### features in more detail and their didactic purpose

    breakpoints, step-by-step, stepback and more

    challenges/ design choices during the implementation
```

3.7 Demonstration/ evaluation (7 pages)

```
what kind of system do we need/ did we use to run this?
    which python Version?

## general demonstration

    brief example program showing all the features in a "normal" execution,
     e.g. adding stuff

### Meltdown und Spectre demonstration

    #### Example Program Meltdown

        show example program
```

```
13
14          maybe compare to example program for real life architecture from
    SCA or literature (Gruss) if available
15
16          explain which Meltdown variant it implements
17
18          briefly highlight which components (we expect to) interact to make
    it work
19
20          how well does it work?
21
22      #### maybe example program spectre
23
24          same as Meltdown
25
26      #### mitigations
27
28          what is possible in our program as is
29              planned:    cache flush: microcode -> config file
30                          mfence im assembler (normally in compiler)
31                          aslr directly in program -> config
32                          flush IQ ->
33                          disable speculation und out of order (nice to have)
    -> config
34
35          how effective are these
36              in real life
37              in our program
38
39          what would be the necessary steps/ changes to the program for
    further mitigations
40              compare to changes in hardware by the manufacturers
```

3.8 Conclusion (1 Page)

```
1   recap goals
2
3   did we reach the goal of implementing a CPU emulator where a user can
        perform a Meltdown attack
4
5   how many Meltdown attacks are possible?
6
7   is Spectre possible?
8
9   mitigations
10      which mitigations did we implement
11      is this a good amount/ sample of real world mitigations or do we miss
        important ones?
```

```
12      how well do they perform (also in comparison to how they perform in the
         real world)
13
14  value to students:
15      how easy to use and convenient do we think our program is?
16      do we think this will be a good tool for teaching?
17      how accessible is it wrt different host architectures?
18
19  further work
20      more (detailed/ realistic) functionality for more Meltdown and Spectre
        variants
21      more mitigations
22      maybe nice to haves wrt to the visualisation/ general UI functionality/
         ISA?
```

References

[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.

[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.

[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.

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

Statement of Authorship

I hereby confirm that the work presented in this bachelor thesis has been performed and interpreted solely by myself except where explicitly identified to the contrary. I declare that I have used no other sources and aids other than those indicated. This work has not been submitted elsewhere in any other form for the fulfilment of any other degree or qualification.

TODO

TODO