

# Transient Execution Emulator

Meltdown and Spectre Behind the Scenes

---

Felix Betke, Lennart Hein, Melina Hoffmann, Jan-Niklas Sohn

2022-04-01

Rheinische Friedrich-Wilhelms-Universität Bonn



- Topic
- Background
- Our task & approach
- Implementation
- Demos:
  - Meltdown
  - Spectre
- Conclusion

- Lab builds on SCA lecture
- Meltdown and Spectre mostly patched
- Difficult to experiment with
  - Personal computer often times not usable
- Goal: Vulnerable CPU Emulator that runs on many systems
  - Should offer a gdb-like interface

## Background

---

- Frontend:
  - Fetches/Decodes instructions, maintains queue
  - Branch prediction
- Execution Engine:
  - Multiple sets of execution units
- Memory Subsystem:
  - Handles memory operations
  - Maintains L1 cache
  - Ensures data is loaded from other caches/memory

- Independent instruction streams
- Tomasulo algorithm:
  - Reservation stations
  - Common Data Bus
- Rollbacks

## Speculative execution

- Predict results of branch instructions
- Prevent stalls
- BPU maintains counters
- Rollbacks

- Abuses out-of-order execution
- Meltdown-US-L1:
  - Define oracle array
  - Perform illegal read to steal secret
  - Embed secret-dependent oracle entry into cache
  - Await rollback and measure oracle access times
- Small time window



- Abuses speculative execution
- Different variations. Here: prediction of conditional branch instrs.
- Spectre v1: Deliberately train BPU used by victim process
- Make victim leak secret into cache
- Direct consequence of speculative execution

- Disable out-of-order execution
- Intel's microcode mitigation
  - Microprograms
- OS mitigations

- Disable speculative execution:
  - Completely disable
  - fence instructions
- Flush entire cache after rollback

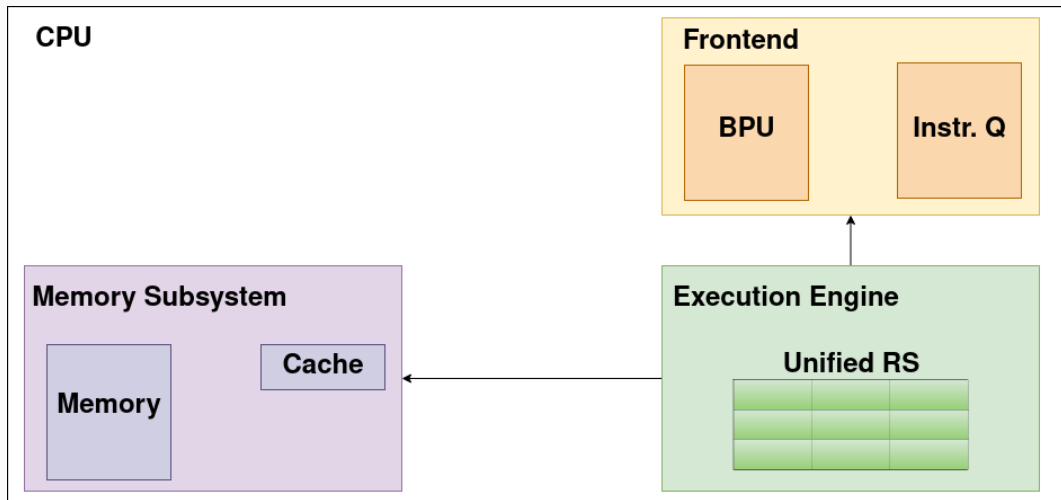
# Our task

- Develop graphical CPU emulator vulnerable to:
  - Our version of Meltdown-US-L1
  - Spectre v1
- Must support single step, out-of-order, and speculative execution
- Implement Intel's microcode mitigation
- Other mitigations via microprograms
- Target audience: SCA students
  - Or anyone with basic knowledge of TE attacks

## Our approach

---

- Must-haves, nice-to-haves, future work
- At time of Meltdown/Spectre publication: Skylake
- Filter components needed for our Meltdown/Spectre versions
- Build simplified CPU



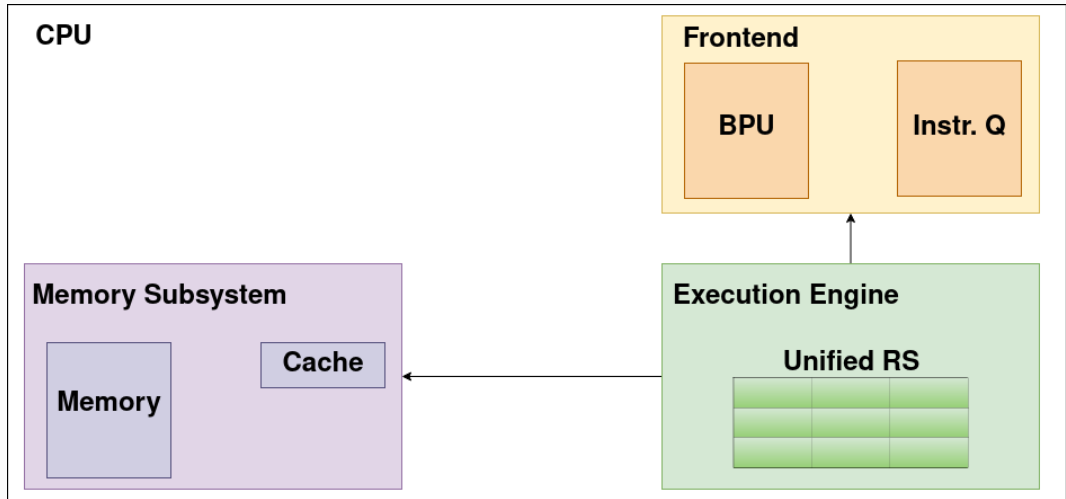
## Implementation of our emulator

---



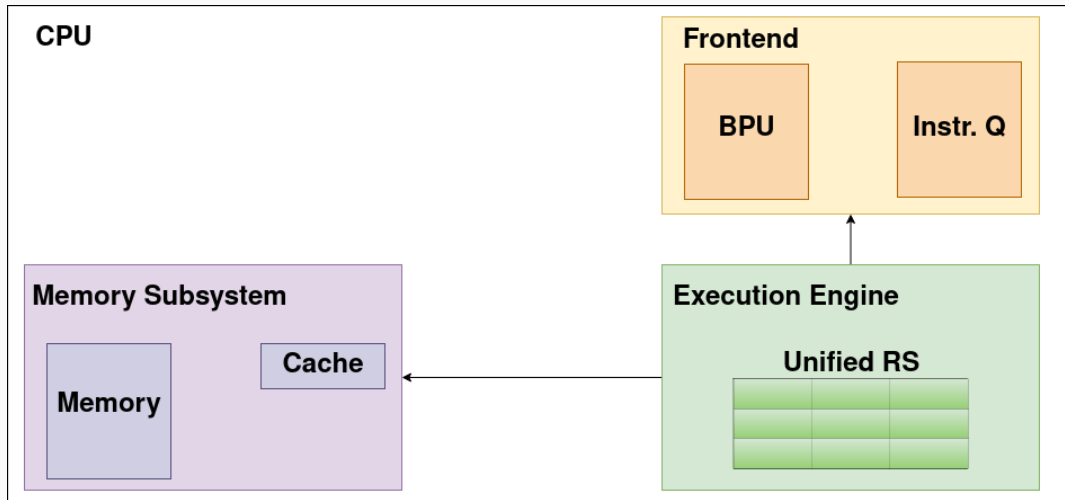
# Implementation of our emulator

- overview over our whole emulator
- Out-of-order execution
- Speculative execution
- Fault handling and rollbacks



- Assembler style source code
- Arithmetic, branch and memory instructions, fence, rtdsc
- Provides an instruction list
- Only one type of instructions

## CPU components



- Execution engine
- Tomasulos algorithm
  - Unified reservation station
  - Instructions wait for their operands
  - Keeping track of operands and results

- Resolve operands and target register
- Two kinds of register values: Word and SlotID
- Put register content into operand list
- Put SlotID into target register

## Example Reservation Station

-----[ Reservation Stations ]-----					
0	addi	r1, r0, 0x3	0x0000	0x0003	<input checked="" type="checkbox"/>
1	slli	r2, r1, 0x8	0x0003	0x0008	<input type="checkbox"/>
2	addi	r2, r2, 0x42	RS 001	0x0042	<input type="checkbox"/>
3	sw	r2, r0, 0x4	RS 002	0x0000	<input type="checkbox"/>
4	lb	r4, r0, 0x5	0x0000	0x0005	<input type="checkbox"/>

- Execute instructions in reservation station
- Broadcast the result over the CDB
  - Registers
  - Reservation station slots
  - At most once per cycle

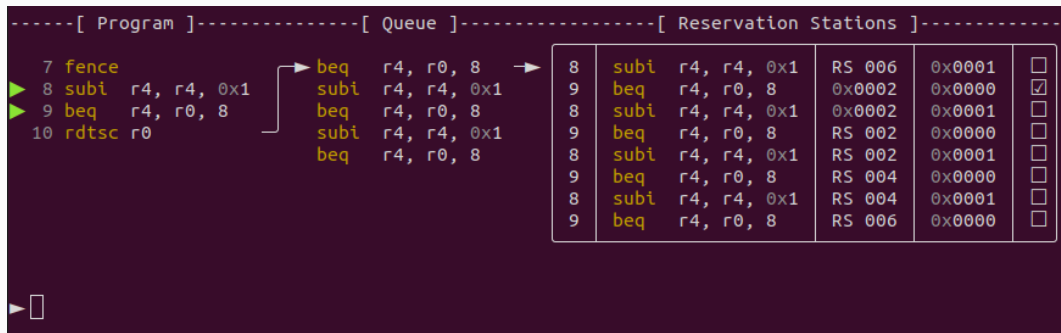


- Predict outcome of branch instructions
- Resume execution based on this prediction
- Two central components
  - Branch prediction unit (BPU)
  - CPU frontend with instruction queue

- Simplified version
- Array of predictions
- 2-bit-saturating counter to handle predictions

## CPU frontend with instruction queue

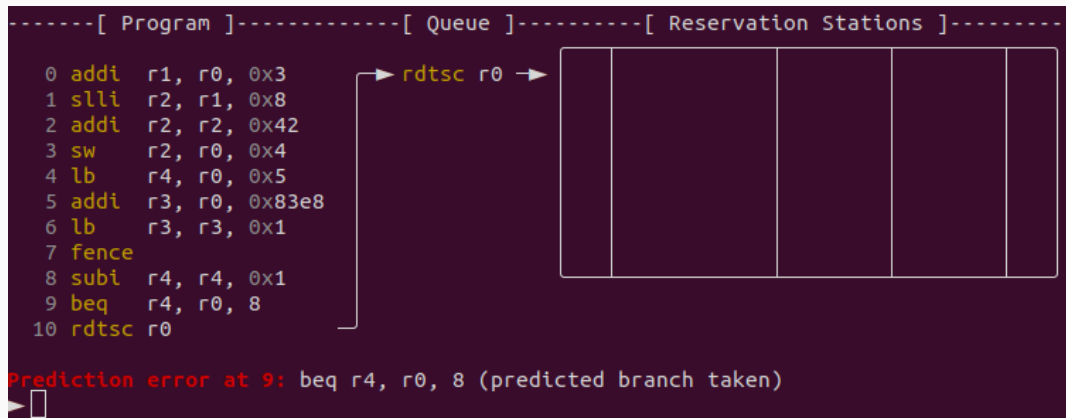
- Interface between instruction list and execution unit
- Involved in speculative execution
- Manages instruction queue



- Microarchitectural fault situation that has to be handled before we can resume our execution
  - Mispredicted branches
  - Attempt to access inaccessible memory
- Have to handle the effects of transient execution

- Only rollback the register state and the memory contents
- No rollback in Cache and BPU
- Restore register state via snapshots
- Prevent memory rollbacks by executing stores in-order
- Handle faults in program order

## Rollback after mispredicted branch



Thank you for your attention

Do you have any questions so far?

Demo





## Demo - Meltdown

```
lb r1, r0, 0xc000  
slli r1, r1, 4  
lb r2, r1, 0x1000
```

Encode byte into oracle array

```
fence  
addi r1, r0, 0  
addi r2, r0, 0xFFFF  
addi r3, r0, 0x0000  
addi r4, r0, 0x0FF0
```

r1	r2	r3	r4
shortest load	shortest load time	current offset	last offset

```
probe:  
fence  
rdtsc r5  
lb r7, r3, 0x1000  
fence  
rdtsc r6  
sub r5, r6, r5
```

Measure access time

```
bgtu r5, r2, skip  
addi r1, r3, 0  
addi r2, r5, 0  
skip:
```

update shortest

```
addi r3, r3, 0x10  
bgeu r4, r3, probe
```

increment and loop

```
slli r1, r1, 4
```

shift result

- Demonstrate mechanism behind Spectre-type attacks
- BPU can be trained for targeted misprediction
- Requires code sequence that encodes leaked value into cache

## Spectre-Type Attack: Overview

- Prepare victim array: 8 elements, all `0x01`
  - Followed by secret value `0x41`
- Victim loops over the array and encodes each value into the cache
  - BPU is trained to predict that the loop continues
- Final loop condition will be mispredicted
  - During transient execution: Additional iteration with out-of-bounds index
  - Secret value accessed and encoded into cache

## Spectre-Type Attack: Preparation

```
// Set up array at 0x1000, 8 elements, all 0x01
addi r1, r0, 0x1000
addi r2, r0, 0x01
sb r2, r1, 0
sb r2, r1, 1
sb r2, r1, 2
sb r2, r1, 3
sb r2, r1, 4
sb r2, r1, 5
sb r2, r1, 6
sb r2, r1, 7
// Followed by one out-of-bounds 0x41 value
addi r2, r0, 0x41
sb r2, r1, 8
```

## Spectre-Type Attack: Execution

```
// Loop over array, encode every value in cache
addi r2, r0, 0    // r2: Loop index
addi r3, r0, 8    // r3: Array length
loop:
// Load array element
lb r4, r2, 0x1000
// Encode value in cache
slli r4, r4, 4
lb r4, r4, 0x2000
// Increment loop index
addi r2, r2, 1
// Loop while index is in bounds
bne r2, r3, loop
```

## Attack Demo

---

- Flush cache after rollback
- Prevents using cache as transmission channel
- Implementation: Inject microcode after rollback
  - Inject `flushall` instruction after mispredicted branch

## Mitigation Demo

---



- Goal: CPU Emulator
  - Out-of-Order Execution
  - Branch Prediction
  - Transient Execution Attacks
  - Mitigations

