CS 152 Laboratory Exercise 3

Professor: John Wawrzynek
TAs: Josh Kang and Hasan Genc
Department of Electrical Engineering & Computer Sciences
University of California, Berkeley

March 7, 2022

Revision History

Revision	Date	Author(s)	Description
1.2	2022-03-17	hngenc	Added workaround for "git://" being disabled
1.1	2022-03-10	hngenc	Added full path to pk-spectre
1.0	2022-03-07	hngenc	Initial release

1 Introduction and Goals

The goal of this laboratory assignment is to study out-of-order processor design in the Chisel simulation environment. You will be provided a complete implementation of a speculative out-of-order superscalar RISC-V core to experiment with and analyze. You can also choose to improve the design as part of the open-ended portion.

While students are encouraged to discuss solutions to the lab assignments with each other, you must complete the directed portion of the lab yourself and submit your own lab report for these problems. For the open-ended portion of each lab, students can either work individually or in groups of two or three. Each group will turn in a single report for the open-ended portion of the lab. You are free to participate in different groups for different lab assignments.

1.1 Graded Items

All reports are to be submitted through **Gradescope**. Please label each section of the results clearly. All directed items need to be turned in for evaluation. Your group only needs to submit *one* of the problems in the open-ended portion.

- (Directed) Problem 3.3: Performance Bottlenecks
- (Open-ended) Problem 4.1: Designing Your Own Branch Predictor
- (Open-ended) Problem 4.2: Recreating Spectre Attacks

- (Directed) Problem 5: Feedback

! → Lab reports must be written in *readable* English; avoid raw dumps of logfiles. **Your lab report must be typed, and the open-ended portion must not exceed ten (10) pages.** Charts, tables, and figures – where appropriate – are excellent ways to succinctly summarize your data.

2 Background

The infrastructure is similar to Lab 2, with the new addition of a new processor: BOOM.

2.1 BOOM: Berkeley Out-of-Order Machine

The Berkeley Out-of-Order Machine (BOOM) is a synthesizable, parameterized, out-of-order superscalar RISC-V core. It is a unified physical register file design (also known as explicit register renaming) with a split ROB and issue window. To build an SoC with a BOOM core, BOOM utilizes the Rocket Chip SoC generator as a library to reuse different microarchitectural components (TLBs, PTWs, etc).

! → BOOM documentation: `https://docs.boom-core.org/en/latest/index.html`

2.1.1 Pipeline

Conceptually, BOOM is divided into 10 stages: *Fetch*, *Decode*, *Register Rename*, *Dispatch*, *Issue*, *Register Read*, *Execute*, *Memory*, *Writeback* and *Commit*. However, some of those stages are combined in the current implementation. Additionally, some of these stages are pipelined across multiple pipeline stages.

Fetch : Instructions are fetched from instruction memory and pushed into a FIFO queue, known as the Fetch Buffer. Instruction pre-decode is performed, to find alignments of 16-bit vs 32-bit instructions. Branch prediction also occurs in this stage, redirecting the fetched instructions as necessary.

Decode : Decode pulls instructions out of the *Fetch Buffer* and generates the appropriate "micro-ops" (µops) to place into the pipeline.[1]

Rename : The ISA, or "logical", register specifiers (e.g. `x0-x31`) are then renamed into "physical" register specifiers.

Dispatch : The µop is then dispatched, or written, into a set of Issue Queues.

Issue : µops sitting in an Issue Queue wait until all of their operands are ready and are then issued. This is the beginning of the out-of-order portion of the pipeline.[2]

Register Read : Issued µops first read their register operands from the unified Physical Register File or from the Bypass Network.

[1] Because RISC-V is a RISC ISA, currently all instructions generate only a single µop.
[2] More precisely, µops that are ready assert their request, and the issue scheduler within the Issue Queue chooses which µops to issue that cycle.

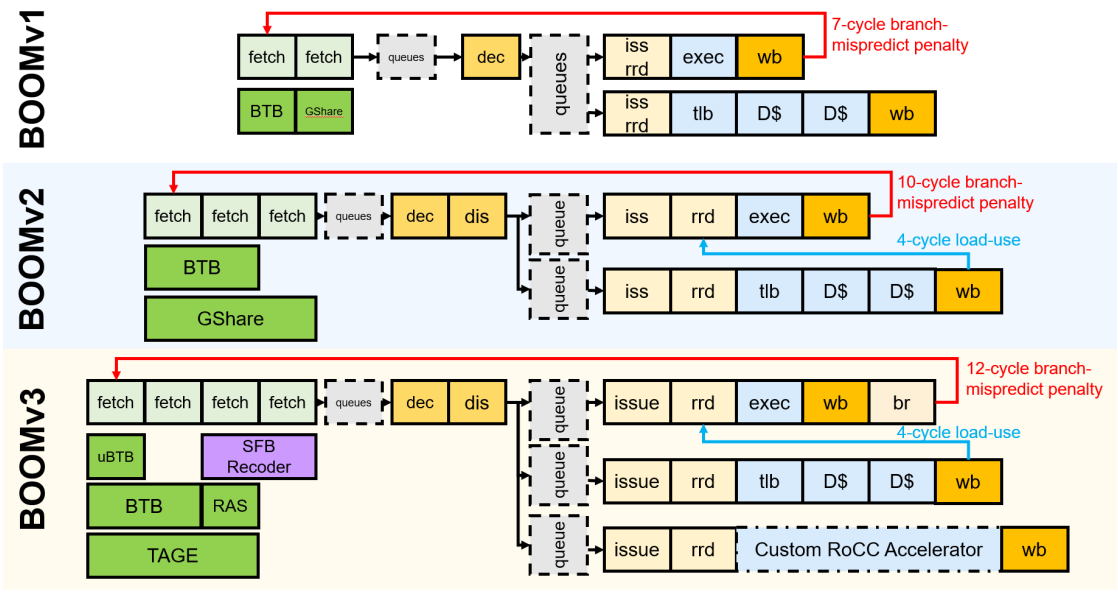

Figure 1: Evolution of the BOOM pipeline. You will be using the latest version, version 3.

Execute : µops then enter the Execute stage where the functional units reside. Issued memory operations perform their address calculations in the Execute stage, and then store the calculated addresses in the Load/Store Unit which resides in the Memory stage.

Memory : The Load/Store Unit consists of two queues: a Load Address Queue (LDQ), and a Store Queue (STQ). Loads are fired to memory when their address is present in the LDQ. Stores are fired to memory at Commit time (and naturally, stores cannot be committed until both their address and data have been placed in the STQ).

Writeback : ALU and load results are written back to the Physical Register File.

Commit : The Reorder Buffer (ROB), tracks the status of each instruction in the pipeline. When the head of the ROB is not-busy, the ROB commits the instruction. For stores, the ROB signals to the store at the head of the Store Queue (STQ) that it can now write its data to memory.

2.1.2 Branch Support

BOOM supports full branch speculation and branch prediction. Each instruction, no matter where it is in the pipeline, is accompanied by a Branch Tag that marks which branches the instruction is speculated under. A mispredicted branch requires killing all instructions that depended on that branch. When a branch instructions passes through Rename, copies of the Register Rename Table and the Free List are made. On a mispredict, the

saved processor state is restored.

2.2 Chipyard

Chipyard is an integrated design, simulation, and implementation framework for agile development of systems-on-chip (SoCs). It combines Chisel, the Rocket Chip generator, and other Berkeley projects to produce a full-featured RISC-V SoC from a rich library of processor cores, accelerators, memory system components, and I/O peripherals.

! → Chipyard documentation: `https://chipyard.readthedocs.io/en/latest/`

3 Directed Portion (20%)

3.1 Terminology and Conventions

Throughout this course, the term *host* refers to the machine on which the simulation runs, while *target* refers to the machine being simulated. For this lab, an instructional server will act as the host, and the RISC-V processors will be the target machines.

UNIX shell commands to be run on the host are prefixed with the prompt "`eecs$`".

3.2 Setup

To complete this lab, `ssh` into an instructional server with the instructional computing account provided to you. The lab infrastructure has been set up to run on the `eda{1..8}.eecs.berkeley.edu` machines (`eda-1.eecs`, `eda-2.eecs`, etc.).

Once logged in, source the following script to initialize your shell environment so as to be able to access to the tools for this lab. Run it before each session.

```
eecs$ source ~cs152/sp22/cs152.lab3.bashrc
```

First, clone the lab materials into an appropriate workspace and initialize the submodules.

```
eecs$ git clone ~cs152/sp22/lab3.git
eecs$ cd lab3
eecs$ LAB3ROOT="$(pwd)"
eecs$ git config --global url."https://github.com/".insteadOf git://github.com/
eecs$ ./scripts/init-submodules-no-riscv-tools.sh
```

The remainder of this lab will use `${LAB3ROOT}` to denote the path of the `lab3` working tree. Its directory structure is outlined below:

```
${LAB3ROOT}
    lab/
        open2/                          Source code for Problem 4.2
    generators/                         Library of RTL generators
        chipyard/                       SoC configurations
```

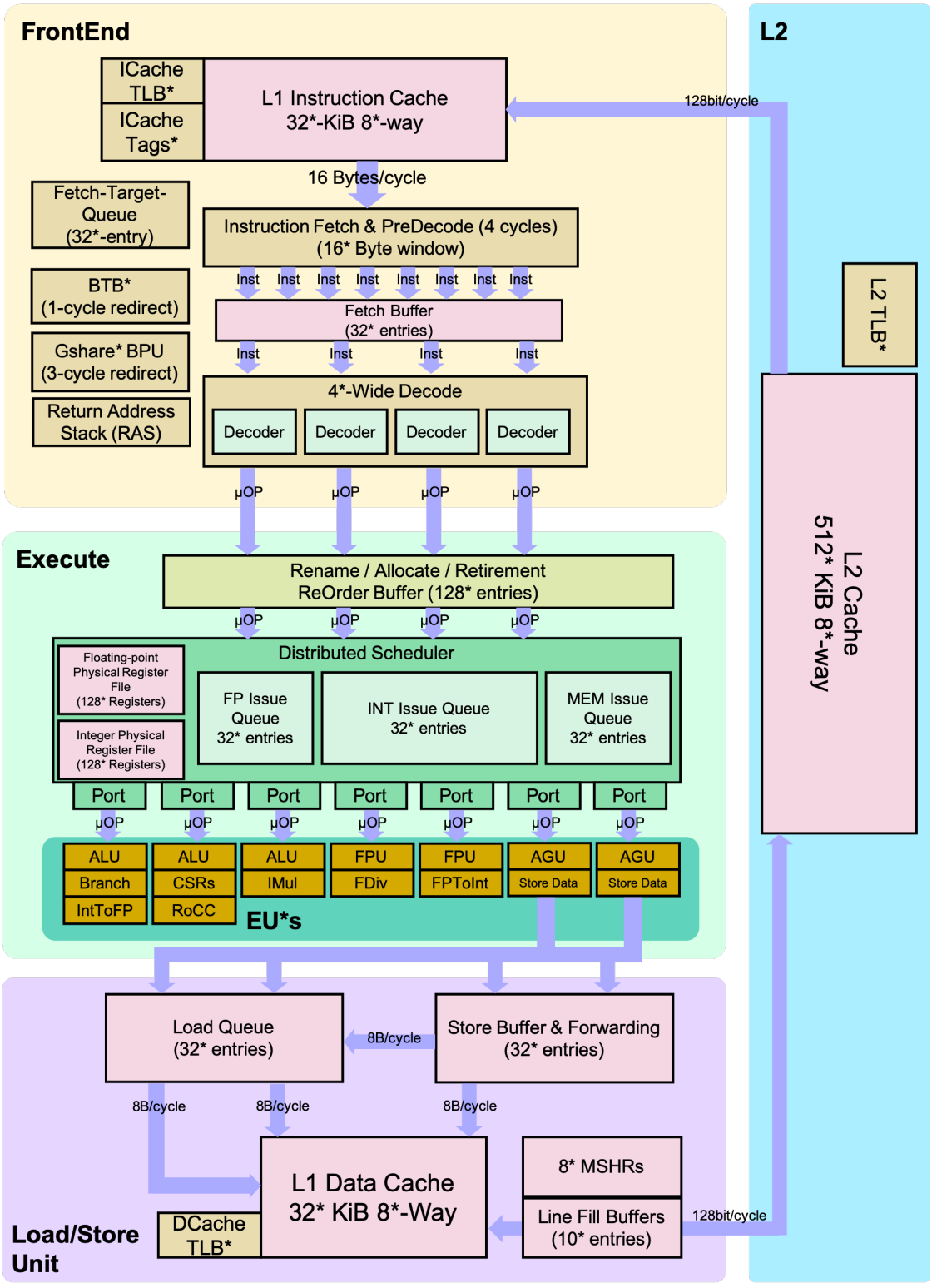

Figure 2: Detailed BOOM pipeline; * denotes where the core is configurable

```
    rocket-chip/                    Rocket Chip generator
    boom/                           BOOM core
    sifive-cache/                   Open-source inclusive L2 cache from SiFive
    testchipip/                     RTL blocks for interfacing with test chips
    ...
sims/
    verilator/                      Verilator simulation flow
    vcs/                            Synopsys VCS simulation flow
    ...
tools/
    chisel3/                        Chisel hardware description library
    firrtl/                         RTL intermediate representation library
    barstools/                      Collection of common FIRRTL transformations
    ...
```

3.3 Performance Bottlenecks

Building an out-of-order processor is hard. Building an out-of-order processor that is well balanced and high performance is *really* hard. Any one component of the processor can bottleneck the machine and lead to poor performance.

For this problem, you will set the parameters of the processor to a low-end "worst-case" baseline (Table 2) and incrementally introduce features. While some of these structures are on the small side, the machine should generally remain well-fed since only one instruction is dispatched and issued at a time, and the pipeline is not exceptionally deep.

Table 2: BOOM worst-case versus improved configurations

	Worst-Case	Improved
Physical Register File	33	64
Reorder Buffer	4	16
Branch Prediction	disabled	BTB, BHT, 8-entry RAS

Collect and report the CPI numbers for the following benchmarks. Results for the in-order Rocket core has been provided for you. **Note**: Since the compile and simulation times can be fairly significant, you may gather the data in collaboration with other students and share them, but the questions must be answered independently.

	dhrystone	median	multiply	qsort	spmv	towers	vvadd
Rocket	1.200	1.379	1.136	1.427	1.690	1.029	1.024
BOOM (worst-case)							
BOOM (64 PRF)							
BOOM (16 ROB)							
BOOM (br. pred.)							

Navigate to `${LAB3ROOT}/generators/boom/src/main/scala/common/config-mixins.scala` and search for the definition of the `WithNCS152BaselineBooms` config. First simulate with the default parameters, which should correspond to the "worst-case" settings. As you move down the rows of the table, change the parameters to match while retaining the features from the previous rows.

For each design point, build the simulator and run the benchmarks in a batch.[3]

```
eecs$ cd ${LAB3ROOT}/sims/verilator
eecs$ make CONFIG=CS152BaselineBoomConfig
eecs$ make CONFIG=CS152BaselineBoomConfig run-bmark-tests
eecs$ cd output/chipyard.TestHarness.CS152BaselineBoomConfig
```

In the output directory, review the `*.log` files (one per benchmark) for the cycle and retired instruction counts. Also record the *branch prediction accuracies* summarized at the end of the `*.out` files.

Format the data in a chart or table, and answer the following questions in your report:

(3.3.a) Compare the results for 1-wide BOOM with the in-order, 5-stage Rocket core. Was out-of-order issue an improvement on the CPI for these benchmarks? Explain the impact of each microarchitectural change.

(3.3.b) Is branch prediction with a BTB, BHT, and RAS always a benefit?

[3] You can run the benchmarks in parallel by adding the `-j` N flag to the `make` command, but refrain from spawning an excessive number of jobs so as to be fair to other users. $N = 4$ is probably acceptable.

4 Open-ended Portion (80%)

Select *one* of the following questions per team.

All open-ended questions should use the parameters for BOOM as shown in Table 3, unless otherwise specified. These should already match the `CS152SWPredBoomConfig`, and `CS152SmallBoomConfig` configurations provided with the lab.

Table 3: BOOM open-ended configurations

	Problem 4.1	Problem 4.2
Configuration	`CS152SWPredBoomConfig`	`CS152SmallBoomConfig`
Issue width	3	1
Fetch width	4	4
ROB	96 entries	32 entries
Issue slots	32 entries	8 entries
Integer register file	96 physical registers	52 physical registers
FP register file	64 physical registers	48 physical registers
LDQ/STQ	16 entries	8 entries
Max branches	12 branches	8 branches
L1D capacity	16 KiB	
L1D associativity	4 ways	
MSHRs	2 MSHRs	

4.1 Designing Your Own Branch Predictor

The version of BOOM provided in the `CS152SWPredBoomConfig` uses a simple bimodal predictor consisting of a Branch History Table (BHT) of 256 2-bit saturating counters. This was the same scheme implemented by the MIPS R10000 [1], although with 512 entries instead. For this problem, your goal is to build an improved branch predictor for BOOM that performs better than this baseline.

Feel free to scour the abundance of literature and historical examples for ideas. This technical report [2] provides a useful survey of common branch prediction techniques. A good design to look into is the two-level tournament branch predictor from the Alpha 21264 [3], which combines three predictors:

- A global predictor with a 4096-entry table of 2-bit saturating counters indexed by the global history of the last 12 branches
- A local predictor with a 1024-entry table of 10-bit branch patterns indexed by PC, which is then used to index a set of 3-bit counters
- An arbiter (or "choice") predictor with a 4096-entry table of 2-bit counters indexed by global history, used to select either the global or local prediction as the final one

Especially impressive designs could attempt to beat the hardware implementation of the TAGE predictor algorithm provided in the lab. You may replace the `WithSWBPD` config option with `WithTAGELBPD` to compare a hardware implementation of the TAGE predictor algorithm, with a 2 KB fast PC-indexed BHT, and 8KB of TAGE tables.

The Chisel implementations of various predictor components, including TAGE, BTBs, BHTs, and the RAS are available in `${LAB3ROOT}/generators/boom/src/main/scala/ifu/bpd`.

4.1.1 C++ Framework

For the implementation, we will leverage the C++ branch predictor framework that exists in BOOM, which lets one more easily construct software models of a branch predictor's intended functional behavior for rapid prototyping and verification.

Although it presents a relatively simplified interface compared to a real hardware predictor module, it is flexible enough to admit a wide space of potential designs while abstracting away much of the complexity of the surrounding logic in the core.

Implement your branch predictor design in `${LAB3ROOT}/generators/boom/src/main/resources/csrc/predictor_sw.cc`. This file contains three functions to modify:

Function	Description
`initialize_branch_predictor()`	This is called at the beginning of simulation and can be used to initialize global state.
`predict_branch()`	This is called to provide a prediction for a branch or jump instruction with the given PC (`ip`). The global history of the most recent branches is passed as a bit vector in `hist`. The `globalHistoryLength` config parameter defaults to 32 for the C++ predictor. The prediction is returned by assigning 1 for taken and 0 for not taken to the variable pointed to by `pred`.
`update_branch()`	This is called to update the predictor state with branch resolution information. Updates occur non-speculatively in program order. `ip` provides the PC of the branch or jump instruction, `hist` provides the global history, and `taken` provides the outcome of the branch or jump.

Both `predict_branch()` and `update_branch()` may be called multiple times per cycle. Remember that the branch predictor is inherently accessed speculatively, so you may see requests for branches that are later squashed by the pipeline.

4.1.2 Simulating

Build the simulator[4] and run the benchmark suite using the same flow as the directed portion (note the different top-level configuration):

```
eecs$ cd ${LAB3ROOT}/sims/verilator
eecs$ make CONFIG=CS152SWPredBoomConfig
eecs$ make CONFIG=CS152SWPredBoomConfig run-bmark-tests
```

[4] The build system has been slightly modified for this lab to avoid re-elaborating Chisel if only C++ resource files have been changed.

To run an individual benchmark only (e.g., `dhrystone`):

```
eecs$ make CONFIG=CS152SWPredBoomConfig \
        ${PWD}/output/chipyard.TestHarness.CS152SWPredBoomConfig/dhrystone.riscv.out
```

! → Chipyard also supports a VCS simulation flow. Simply navigate to `${LAB3ROOT}/sims/vcs`, and the same `make` commands should work. VCS has the advantage of shorter compile times compared to Verilator at the cost of somewhat lower simulation performance.

4.1.3 Debugging

To dump waveforms from simulation, run the debug versions of the `make` targets:

```
eecs$ cd ${LAB3ROOT}/sims/verilator
eecs$ make CONFIG=CS152SWPredBoomConfig debug
eecs$ make CONFIG=CS152SWPredBoomConfig run-bmark-tests-debug
```

Similarly, to dump a waveform for an individual benchmark:

```
eecs$ make CONFIG=CS152SWPredBoomConfig \
        ${PWD}/output/chipyard.TestHarness.CS152SWPredBoomConfig/dhrystone.riscv.vpd
```

Waveform dumps (which can become quite sizeable) are written to `${LAB3ROOT}/sims/verilator/output/f1/chipyard.TestHarness.CS152SWPredBoomConfig/*.vpd`. The branch predictor is located at `TestDriver.testHarness.top.boom_tile.frontend.bpd.banked_predictors_0.components_0.BranchPredictorHarness*` in the module hierarchy.

Waveforms can be viewed on the instructional servers with the DVE application, which requires X11 forwarding over `ssh` or X2Go:

```
eecs$ dve &   # '&' backgrounds the process
```

4.1.4 Benchmarks

The source code for all benchmarks can be found in `${LAB3ROOT}/toolchains/riscv-tools/riscv-tests/benchmarks/`. First initialize the `riscv-tests` submodule:

```
eecs$ git submodule update --init ${LAB3ROOT}/toolchains/riscv-tools/riscv-tests
```

The disassemblies that correspond to the pre-installed binaries are available at `${RISCV}/riscv64-unknown-elf/share/riscv-tests/benchmarks/*.riscv.dump`.

4.1.5 Submission

In your report, present the IPC and branch prediction accuracy results of your custom predictor on all benchmarks. Describe your design approach and any implementation challenges in detail, and explain its performance characteristics. Be sure to cite the appropriate sources if you borrowed from existing concepts.

- How did you calculate the amount of state your branch predictor has?
- How do certain parameters (e.g., number of entries) impact accuracy?
- Which branches or patterns were easier or harder to predict?
- What kind of application code do you expect your predictor to perform better or worse on?
- Do you foresee any challenges with the implementation of your predictor algorithm as a hardware block within a superscalar, out-of-order core?
- What changes or alternative approaches would you pursue as future work if more time were available?

Feel free to reach out to your GSI if you need help understanding BOOM, branch prediction schemes, or anything else regarding this problem.

4.2 Recreating Spectre Attacks

It turns out that BOOM, like many out-of-order processors, is susceptible to a class of microarchitectural side-channel attacks that exploit branch prediction, speculative execution, and cache timing to leak information from memory, bypassing security mechanisms such as virtual memory and bounds checks. These first came to prominance with the *Spectre* and *Meltdown* vulnerabilities disclosed in 2018. For this problem, your goal is to mount a Spectre attack on BOOM to extract secret data from protected kernel memory.

4.2.1 Background

First read the Spectre paper [4] and the Google Project Zero post [5] to understand the basic principles and techniques behind the Spectre exploit. The proof-of-concept in Appendix C of the paper may be a useful reference as you write your code.

Although not the focus of this problem, it is also worth reading a little about Meltdown [6], a closely related variant that arises from deferred TLB permissions checks.

4.2.2 Attack Scenario

In this scenario, you control a malicious adversary that runs as an unprivileged program in user mode (U-mode) on top of the RISC-V proxy kernel (`pk`), a lightweight execution environment that assumes the role of a minimalistic operating system. `pk` is designed to support tethered RISC-V implementations with limited I/O capability and thus handles I/O-related system calls by proxying (forwarding) them to a host – in this case, the machine running the simulation.

`pk` itself runs in the higher-privilege supervisor mode (S-mode) but shares the same virtual address space as the user program. The lower 2 GiB of the virtual address space is reserved for the user program, while `pk` is mapped into the upper portion after 0x80000000.[5] This

[5] This offset is specifically chosen such that this virtual address range is identical to the physical address range where `pk` resides, i.e., VA = PA.

is a common technique in operation systems to faciliate more efficient communication between kernel and user space. For our purposes, this fact merely makes it slightly easier to reason about the addresses of the secret data and the privileged code being targeted.[6]

The objective of the adversary is to learn the values of a contiguous 128-byte array in the static `.data.secret` section in `pk`, representing a secret key. This data is ordinarily inaccessible to user programs, as it resides in a page with supervisor-only read permissions.

We deliberately introduce a few artificial conditions to simplify the task without compromising the fundamental methodology:

- The secret data is placed at a fixed virtual address that is already known.
- The attack vector is a custom syscall handler in `pk` that contains a vulnerable *Spectre gadget* by design.
- The Spectre gadget leaks only one bit at a time to minimize noise from cache thrashing.

The Spectre gadget, shown here lightly edited, is very similar to the canonical example for Spectre Variant 1 (bounds check bypass):

```
uint8_t leak_array[128]; // Two 64-byte cache lines

int sys_leak(size_t index, int shift)
{
        if (index < sizeof(leak_array)) {
                uint8_t data = leak_array[index];
                index = (data >> shift) & 0x1;
                return leak_array[index * 64];
        }
        return -1;
}
```

The first array access, which involves an index controlled by the adversary through a syscall argument, can be used to speculatively read an arbitrary byte in memory. The second array access then reads from different cache lines depending on the data value and the shift argument.

Note that the bounds check in the `if` statement prevents the results of invalid accesses from becoming architecturally visible. However, side effects in the cache induced by the data-dependent load persist after speculative execution, and these side effects can be measured to infer data values. While several approaches to cache-based covert channels exist, a Prime+Probe attack [7] is likely the most practical option given that BOOM does not presently implement an unprivileged cache flush instruction.[7]

The overall flow of the attack would proceed as follows:

[6] Meltdown can be mitigated by separating the user and kernel address spaces, accomplished through kernel page-table isolation (KPTI) in Linux. However, this does not defend against Spectre.

[7] Flush+Reload is theoretically possible since the L2 cache controller provides a mechanism to flush specific lines, which also evicts from the L1 due to the inclusive property, but the requisite MMIO control registers are not exposed to U-mode by default.

1. **Training step**: Mistrain the branch predictor to strongly predict that the `if` condition for the bounds check will be true, generally by repeatedly invoking the syscall with valid inputs.
2. **Prime step**: Fill the cache sets that `leak_array` would map to with known lines.
3. **Exploit step**: Invoke the syscall with malicious arguments. After the `if` condition is predicted false, the `if` body is speculatively executed. The inputs are specially crafted so that the out-of-bounds load points to a secret byte, of which one bit is used to construct the pointer for another load that evicts one of the primed lines. The misspeculated execution path is eventually reverted when the branch is resolved, but the cache state remains perturbed.
4. **Probe step**: Determine which line was evicted by measuring the access time to each line. The cache set that previously held the evicted line corresponds to the value of the leaked bit.
5. Repeat for enough samples to gain sufficiently high confidence.

It is not necessary to follow this procedure exactly. You may want to experiment with variations to improve accuracy and/or throughput.

4.2.3 Reverse Engineering

The source code for the `sys_leak` handler can be found in `${LAB3ROOT}/toolchains/riscv-tools/riscv-pk/pk/syscall.c`. First initialize the `riscv-pk` submodule:

```
eecs$ git submodule update --init ${LAB3ROOT}/toolchains/riscv-tools/riscv-pk
```

A pre-built `pk-spectre` executable is already provided along with the Lab 3 infrastructure. To figure out the virtual addresses of various symbols in `pk` (e.g., `leak_array`), refer to the symbol table in the fully linked binary:

```
eecs$ riscv64-unknown-elf-readelf -s $RISCV/riscv64-unknown-elf/bin/pk-spectre > pk.sym
```

To figure out the instruction PCs to compare to the BOOM traces, look at the disassembly:

```
eecs$ riscv64-unknown-elf-objdump -d $RISCV/riscv64-unknown-elf/bin/pk-spectre > pk.dump
```

To figure out the addresses of the secret data and to verify your results, dump the `.data.secret` section contents:

```
eecs$ riscv64-unknown-elf-objdump -s -j .data.secret \
        $RISCV/riscv64-unknown-elf/bin/pk-spectre
```

Data words are displayed in big-endian byte order, i.e., the most significant (leftmost) byte in a word is stored at the lowest address. (The test data comes from the digits of π.)

4.2.4 Development and Testing

To help you start writing your code, `${LAB3ROOT}/lab/open2/spectre.c` contains an example skeleton with helper functions to invoke the syscall and read the cycle counter. To compile:

```
eecs$ cd ${LAB3ROOT}/lab/open2
eecs$ make
```

Feel free to modify the user program however you wish: add a custom linker script, mix in assembly files, etc.

To run the simulation:[8]

```
eecs$ cd ${LAB3ROOT}/sims/verilator
eecs$ make CONFIG=CS152SmallBoomConfig \
        BINARY="$RISCV/riscv64-unknown-elf/bin/pk-spectre ${LAB3ROOT}/lab/open2/spectre.riscv" \
        run-spectre-hex
```

Traces/logs go to `pk_spectre.chipyard.TestHarness.CS152SmallBoomConfig.out` and `pk_spectre.chipyard.TestHarness.CS152SmallBoomConfig.log`, respectively.

To enable logging of dispatched instructions in BOOM, open `${LAB3ROOT}/generators/boom/src/main/scala/common/config-mixins.scala` and set `enableDispatchPrintf` to `true` under the `WithNCS152SmallBooms` config. [9] This should help give you sense of the size of the speculative execution window prior to the branch mispredict recovery.

To quickly test cache eviction/measurement code and other utility functions, you can write a bare-metal program in `${LAB3ROOT}/lab/open2/baremetal.c`. Simulations are much shorter as it avoids waiting for `pk` to initialize and load the user program. This code executes in machine mode (M-mode) with physical addressing, and only a limited subset of libc is supported. **Note**: It is not possible to perform the entire attack in a bare-metal environment, as the victim syscall and secret data are not available without `pk`.

```
eecs$ make CONFIG=CS152SmallBoomConfig \
        BINARY="${LAB3ROOT}/lab/open2/baremetal.riscv" run-binary-hex
```

4.2.5 Submission

Your Spectre attack code should be submitted through Gradescope. In your report, explain how your code works, and describe any challenges encountered and solutions that you used.

- What is the accuracy and performance (cycles per secret byte) of your attack?
- What are some hardware and/or software countermeasures against Spectre that you can think of? Discuss the advantages and disadvantages of each.

[8] The 1-wide BOOM configuration is preferred as it simulates much more quickly.
[9] You may also want to disable `enableBranchPrintf` to make the log more readable

Feel free to reach out to your GSI if you need help understanding Meltdown/Spectre, BOOM, riscv-pk, or anything else regarding this problem.

5 Feedback Portion

In order to improve the labs for the next offering of this course, we would like your feedback. Please append your feedback to your individual report for the directed portion.

- How many hours did you spend on the directed and open-ended portions?
- What did you dislike most about the lab?
- What did you like most about the lab?
- Is there anything that you would change?
- Is there something else you would like to explore in the open-ended portion?
- Are you interested in modifying hardware designs as part of the lab?

Feel free to write as much or as little as you prefer (a point will be deducted only if left completely empty).

5.1 Team Feedback

In addition to feedback on the lab itself, please answer a few questions about your team:

- In a few sentences, describe your contributions to the project.
- Describe the contribution of each of your team members.
- Do you think that every member of the team contributed fairly? If not, why?

6 Acknowledgments

This lab is heavily based on the original CS 152 lab that used BOOM, developed by Christopher Celio, which was itself partially inspired by the preceding lab developed by Henry Cook. Special thanks goes to Jerry Zhao and other members of the Berkeley Architecture Research group who continue to develop BOOM.

References

[1] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–41, Apr. 1996. DOI: 10.1109/40.491460.

[2] S. McFarling, "Combining branch predictors," Western Research Laboratory, Tech. Rep. WRL-TN-36, Jun. 1993. [Online]. Available: https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf.

[3] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, Mar. 1999. DOI: 10.1109/40.755465.

[4] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002. [Online]. Available: https://spectreattack.com/spectre.pdf.

[5] J. Horn, *Reading privileged memory with a side-channel*, Jan. 2018. [Online]. Available: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html.

[6] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-lipp.pdf.

[7] D. A. Osvik, A. Shamir, and E. Tromer, *Cache attacks and countermeasures: The case of AES*, Cryptology ePrint Archive, Report 2005/271, 2005. [Online]. Available: https://eprint.iacr.org/2005/271.