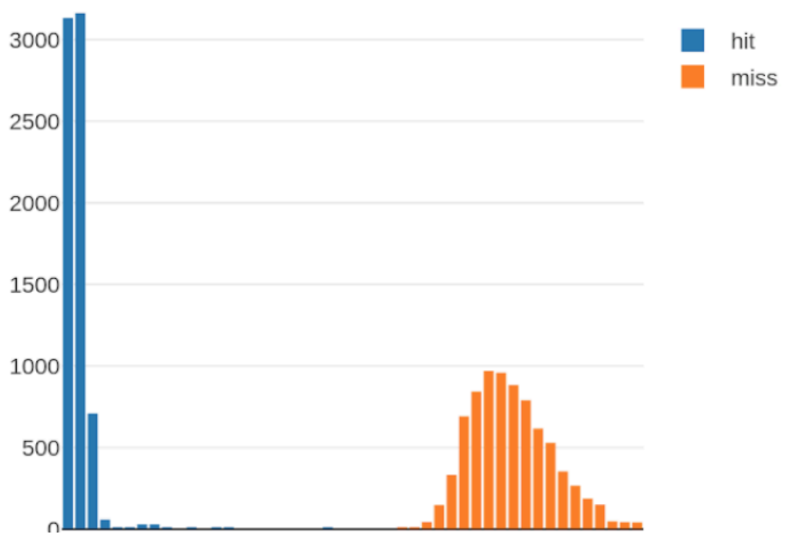
[IMPORTANT!] The entire assignment must be written in C. In particular, assembly and inline assembly may not be used. Use the C wrappers to the assembly instructions provided in `asm.h` instead. These suffice to complete the assignment (you do not need to use them all). Note: you may not add additional inline assembly to `asm.h`. Only use what is provided.

Lab 2 - Manual

TASK 0: Cache HIT vs Cache MISS Histogram

The goal of this task is to distinguish a cache hit from a cache miss based on timing. This is the core idea of a timing cache side-channel attack which you will implement in the following task. Create a histogram plotting the access times of 10K cache hits and 10K cache misses. The result should look something like this:



TASK 1: FLUSH+RELOAD

The goal of this task is to implement the Flush+Reload attack to leak an 8-byte secret "secret" from the provided shared library "libcrypto.so". The library implements a single function, "encrypt_secret_byte(buf, stride, index)", that indexes into the buffer "buf" with a stride of "stride" at index "secret[index]". As a side-effect, this encodes the value of "secret[index]" into the cache. Retrieve each byte of "secret" from the cache via the Flush+Reload cache timing side-channel attack. Implement Task 1 in "flush_reload.c".

You can define a custom key to test your Flush+Reload code by defining the "SECRET" compiler flag when building your program as follows:

```
$ CFLAGS=-DSECRET=0x6162636465666768 make flush_reload
```

Rethink the possible complications from all levels of the compute stack (cf. manual Lab 1). Does Flush+Reload incur new challenges compared to DRAMA?

TASK 2: Meltdown TSX

Now that you have implemented Flush+Reload, use it to implement Meltdown. Your victim will be the “wom” (Write-Only Memory) kernel module. Its source code is included in the assignment: it maps a buffer in memory and allows userland write-only access. Nevertheless, using Meltdown, you will **read** its contents.

The wom kernel module is already loaded on all nodes in the cluster (cf. “lsmod | grep wom” or “ls /dev/wom”). Note that it exposes different functionalities which you could use to manipulate the secret string.

Implement Task 2 in “meltdown.c”. You may assume that the secret is 32 bytes long – although you can write (and leak) more bytes into(/from) /dev/wom. Use TSX for suppressing the page fault (cf. Intel’s TSX documentation and programming help).

BONUS TASK: Meltdown Meets Spectre

In “spectre.c”, implement a variation of Task 2, using branch misprediction to suppress the page fault (instead of TSX). In other words: instead of wrapping the core of your Meltdown code inside a TSX transaction, wrap it inside a transient execution window caused by branch misprediction. You are free to choose what type of branch predictor you target: à la Spectre-v1 (PHT), Spectre-v2 (BTB), Spectre-v5 (RSB), etc.

For this bonus task (and **only** for this bonus task), you are allowed to use (inline) assembly.

Testing

You can find the “test_meltdown_spectre.sh” script which you can use to test your implementation of both Meltdown-TSX and the bonus task. The script will change the content of wom’s secret buffer and see if your program can leak it correctly. Assume you want to leak the first 32 bytes of the buffer and that they are all hex digits.

Upon running `$ ./test_meltdown_spectre.sh`, the script executes one time your implementation of Meltdown-TSX and Meltdown-Spectre and compares the leaked secret with the generated one. The script tells you at the end of each test whether your corresponding implementation is a PASS or FAIL

Upon running `$ ./test_meltdown_spectre.sh batch`, the script executes 100 times your implementation of “./meltdown” and “./spectre” and tells you the success rate of each binary.

Make sure that the implementation of Meltdown (and the optional bonus task) PASS the “test_meltdown_spectre.sh” script.

Notes

Make sure to test your implementations of *all tasks* on the HWSec cluster before submitting your code. We are going to grade your submissions on the same cluster.

Grades

- **2 Points: (TASK 0)** Distinguish between a cache hit and a cache miss.
- **4 Points: (TASK 1)** Recover the secret through Flush+Reload.
- **4 Points: (TASK 2)** Recover the second secret from kernel memory by implementing the Meltdown attack using Intel TSX.
- **1 Point: (BONUS)** Recover the secret from kernel memory by implementing Meltdown attack using Spectre branch misprediction.

Deliverable

A zip file named after your vunetid (eg “xyz123.zip”) containing:

- TASK 0: A histogram plotting 10K cache hits and 10K cache miss and the corresponding access times.
- TASK 1: The source code of “flush_reload.c” that implements the Flush+Reload cache attack leaking the secret contained in libcrypto.
- TASK 2: The source code of “meltdown.c” that implements the Meltdown attack using Intel TSX and leaks the content of the secret buffer created by the WOM kernel module.
- BONUS TASK: The source code of “spectre.c” that implements the Meltdown attack using Spectre fault suppression and leaks the content of the secret buffer created by the WOM kernel module.
- (Optional) A short README file with remarks about your implementation, possibly explaining what works and what doesn’t work, and what you think might be the problem.

Deadline

You're required to submit before **Fri, November 7th 2025 @ 23:59**.
Every late day causes 1 penalty point on the grade (max. delay of 3 days).