**Due date:** Wednesday March 1st 11:59:59 PM ET

**Collaboration policy:** You may discuss the lab at a high level with other classmates, but you may only work and code together with a single partner.

**Getting started:** Log in to our lab machine at the ip address: 152.2.135.251. To connect via ssh, run:

```
ssh YourOnyen@152.2.135.251
```

Your password will be emailed to you; please change it after logging in using `passwd`. You will need to access the machine from the campus network, or use the UNC VPN if you are working remotely.

**Submission:** You will download the lab from github, and write your solutions and code in-place within the downloaded directories. The lab will ask to record answers in files of a given name in certain locations. To submit your lab, you will zip up the entire base directory (called "lab1-cacheattacks") that you downloaded and submit the zipped up file through Canvas.

## Introduction

This lab explores the practical exploitation of hardware side channels. In the first part, you will sample the access latency of different cache levels in the memory subsystem. The latency numbers collected in part 1 will be helpful in creating a cache side/covert-channel. In the second part, Dead Drop, you will see how these side channels can allow you to bypass process isolation guarantees by creating a covert channel to send information between two processes that are unable to communicate using traditional means. In the third part, Capture the Flag, you will fine-tune the concepts you used in Dead Drop to steal a secret from a victim program by simply observing the cache.

The lab will help you to leverage the theory of the attacks that you learnt from the class to build practical ones that work on real processors. There are several challenges to tackle that we have not covered in the class. The discussion questions and hints below will help to address these challenges.

To get started with the lab, clone the repo at:

https://github.com/andrewmikwong/hwsec-course

and see the README.

**Please do not use VS Code to connect to the server.** VS Code's remote ssh plugin introduces a large degree of noise, and may cause your attack to fail.

**Assembly, C/C++, Python.** In most labs for this course, including Lab 1, you will be asked to write programs in C/C++. C/C++ are low-level languages that give you more control over the hardware than high level languages do. The programs written in those languages can be compiled to machine code and directly execute on the hardware without other abstraction layers. When working on microarchitectural attacks, having a high degree of control over the exact instructions being executed is essentially a requirement.

In this lab, we will not ask you to write assembly code. Instead, we use inline assembly to wrap assembly code as C/C++ functions which you can use directly.

We use Python to automate the process of launching attacks and plotting results.

# 1 Timing Analysis (20%)

This part will guide you through some common techniques and instructions that we can use to measure latency on processors, which is a prerequisite for timing covert/side channels.

## 1.1 Setting Up Your Environment

You will be given SSH access to a lab machine to run your code on and you will be assigned two CPUs on this machine. After logging into the lab machine and cloning the lab materials repo, you will need to modify the SENDER_CPU and RECEIVER_CPU variables in the cpu.mk file to your assigned CPUs. **IMPORTANT: You must set SENDER_CPU and RECEIVER_CPU to your assigned values before running code for any portion of this lab.**

**Exercise 1:** Modify the `SENDER_CPU` and `RECEIVER_CPU` variables in the `cpu.mk` to your assigned CPUs.

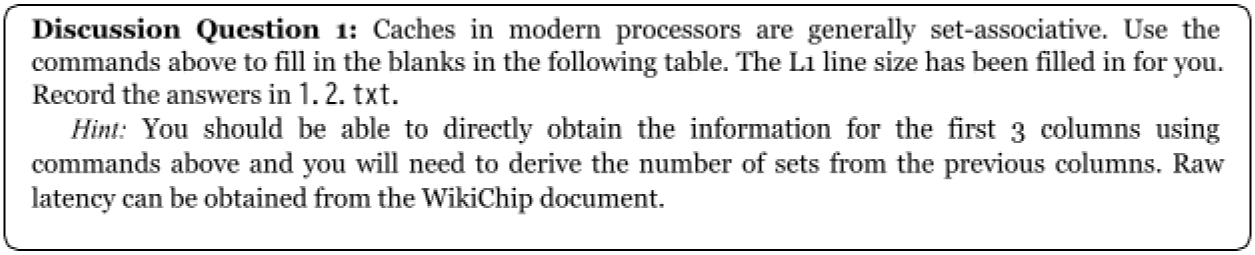
Once you have configured these variables, you can remove the $error line from cpu.mk as well.

## 1.2 Determine the Machine Architecture

Before exploiting any hardware side channels, you must first understand the machine architecture. There are a few commands that can help with this on Linux.

- `lscpu`: provides information on the type of processor and some summary information about the architecture of the machine.
- `less /proc/cpuinfo`: provides detailed information about each logical processor in the machine.
- `getconf -a | grep CACHE`: displays the system configurations related to the cache. This will provide detailed information about how the cache is structured. The numbers that are reported using this command generally use Byte (B) as the unit.

In addition, WikiChip provides a lot of information specific to each processor and architecture. You can find a detailed architecture description of the lab machine in Skylake architecture, which additionally provides the raw latency number for accessing different levels of caches.



**Discussion Question 1:** Caches in modern processors are generally set-associative. Use the commands above to fill in the blanks in the following table. The L1 line size has been filled in for you. Record the answers in `1.2.txt`.

*Hint:* You should be able to directly obtain the information for the first 3 columns using commands above and you will need to derive the number of sets from the previous columns. Raw latency can be obtained from the WikiChip document.

| | Cache Line Size | Total Size | Number of Ways (Associativity) | Number of Sets | Raw Latency |
|---|---|---|---|---|---|
| L1 | 64 Bytes | | | | |
| L2 | | | | | |
| L3 | | | | | |

Table 1: Cache Parameters

## 1.3 Warm-up with C Syntax

Before we actually get started with timing measurements, you need to familiarize yourself with C syntax and several useful x86 instructions. This section gives a brief explanation for the code in *utility.h* and *main.c*. Please read the code in the file utility.h and understand the following functions. In the comments, you can find detailed explanations of the x86 instructions used in each function.

- `rdtscp` and `rdtscp64`: Read the current timestamp counter of the processor.
- lfence: Perform a serializing operation.
- `measure_one_block_access_time`: Measure the latency of performing one memory access to a given address.
- `clflush`: Flush a given address from the cache - later accesses to the address will load from DRAM.
- `print_results` and `print_results_for_python`: print the collected latency data in different formats.

An extremely powerful tool used in C programs to interact with memory is the *pointer*. Here are a few C methods to manipulate memory using pointers.

- `malloc(size):` Allocate size bytes of uninitialized storage on the heap. On success, malloc returns a pointer to the beginning of some newly allocated memory. To avoid a memory leak, the returned pointer must be deallocated with free(). On failure, malloc returns a null pointer (denoted with the keyword NULL). You should always check to ensure the returned pointer is not equal to NULL!
- `free(ptr)`: Returns the space previously allocated by malloc(). If ptr is a null pointer, the function does nothing. Once you have free'd some memory, you should ensure never to use it again, otherwise you will introduce a Use after Free (UaF) vulnerability!
- `&var`: Returns the memory address of var, also known as a pointer to var. On our 64-bit lab machine, addresses are 64-bit integers. The & operator is called the "address-of" operator.
- `*ptr:` Dereference the pointer `ptr`. This allows you to read or write the data pointed to by `ptr` just as you would any other variable.
- `ptr[index]`: Treat ptr as an array and retrieve the entry at index index. This is the same as `*(ptr+index × sizeof(type))`. If `ptr` points to an 8-bit character, then `sizeof(type)` is 1 byte. Remember that arrays in C start at 0.
- `y = (type) x:` change the type of variable x and assign it to variable y. It is common to convert an 64-bit value back-and-forth between the integer and pointer format. For example,

```c
uint64_t x = 0x12345678; //x is a 64-bit unsigned integer
uint8_t *y = (uint8_t *) x; // cast x to a pointer (treat it as an address)
uint8_t z = *y; // access the data pointed to by pointer y.
// z now contains the data from address 0x12345678
```

## 1.4 Timing a Memory Access

In this part, you will time and report the latency of accessing a cache line located in the (a) L1 cache, (b) L2 cache, (c) L3 cache or (d) the DRAM. We will guide you through this process through the following exercise. At a high level, your code should perform the following:

1. Put a cache line into a given structure (L1, L2, L3 cache, or DRAM).
2. Measure the access latency using `measure_one_block_access_time`.

To help you to make sure that you are on the right track, we have provided a reference implementation in the executable file *reference*. To see the reference output, use command make run-reference.

**Measure L1 latency.** We have provided example code in *main.c* to measure L1 latency. The idea is to (1) perform a load operation on a target address to bring the corresponding cache line into the L1 cache, and then (2) measure the access latency by counting the number of cycles it takes to re-access the same target address. You can compile the starter code using the command: `make`, and then run it with: `make run`.

**Exercise 2:** Compile and run the starter code and observe the L1 latency measurements.

**Measure DRAM latency.** You will now need to figure out how to measure the DRAM access latency. Place an address into DRAM and then use measure_one_block_access_time to measure its latency. *Hint:* You can leverage the instruction `clflush` to achieve this goal.

**Exercise 3:** Fill in the code in `main.c` to populate the array `dram_latency` and report the medium DRAM latency. Compare your results with the reference implementation.

**Measure L2 and L3 latency.** Next, you should move on to measure L2 and L3 cache latency. Similarly, you need to put a target address to the L2 cache and L3 cache. Simply accessing the target address will make the address reside in the L1 cache. Therefore, you need to access other addresses to evict the target address from the L1 cache.

*Hint:* You should be careful with the mismatch of access granularities. The smallest operational unit in cache is the cache line size. However, a `uint64_t` value is 64 bits, i.e., 8 Bytes, which is smaller than the cache line. Accessing two integers that fall into the same line (more precisely, fall within the same cache line size aligned region of memory) will result in a cache hit, and won't cause an eviction.

**Discussion Question 2:** How many cache lines do you need to access to fill up the entire L2 cache and L3 cache respectively? You can derive the answer from Table 1. Put your answer in `1.4.txt`.

*Hint:* You may not be able to reliably evict the target address due to noise. This is common due to the following reasons. (1) Recall that the cache is set-associative, sometimes you may not get enough addresses that are mapped to the same cache set as the target address. It is possible because the cache mapping uses physical addresses, while the software uses virtual addresses. (2) The cache replacement policy in modern processors is more advanced than what we learnt from the class. It may smartly decide to keep the target address instead of evicting it.

To bypass the two scenarios above, you could try (1) use a buffer at the size that is 1.5× of your calculated
size; (2) access the eviction buffer multiple times.

**Exercise 4:** Fill in the code in `main.c` to populate the array `l2_latency` and array `l3_latency`. Report the medium L2 and L3 latency. Compare your results with the reference implementation.

## 1.5 Visualize Latency Distribution

Micro-architectural side channels are notoriously noisy. The noise comes from various sources and it is unlikely to come up with a complete list. Fortunately, there exist several effective methods to combat noise, and the most commonly used method is *repetition*. Specifically, the attacker can repeat the measurement many times to collect a large amount of data and use statistical methods to accurately decode the secret, as long as the noise is not substantial. Such statistical methods can range from simply computing the mean of the collected data, to signal processing techniques and even machine learning methods.

We have provided two Python scripts, *run.py* and *graph.py,* to show you an example of how to automatically launch multiple measurements and visualize these measurements.

*run.py* - A python script that will generate 100 runs from the reference binary. It will create a folder (if one doesn't already exist) called data, and it will store all the generated samples there in json format.

The script will overwrite the folder if it already exists.

*graph.py* - A python script that will plot the histogram of the samples collected from run.py. It will read the data from the folder data and generate a pdf file of the histogram in a folder called graph.

There exist several knobs- feel free to adjust them however you want.

The macro SAMPLES in utility.h is the number of samples to collect for each run.

- The variable executable_filename in run.py is the name of the binary file to be launched. You can change this variable to "['make', 'run']" to run your own code. Note that your code must print samples using the print_results_for_python function.
- The variable num_runs in run.py is the number of times to run the binary file.

Note that the SkyLake architecture has very similar L1 and L2 latency; hence it may be difficult to distinguish L1 vs L2. However, the {L1, L2} vs L3 vs DRAM must be clearly distinguishable!

**Exercise 5:** Generate the histogram pdf file using your binary code and compare it with the reference implementation. Include the histogram of the reference implementation and your implementation in your `solutions.pdf` document. See the submission guidelines in Section 1.6.

**Discussion Question 3:** Based on the generated histogram, report two thresholds, one to distinguish between L2 and L3 latency and the other to distinguish between L3 and DRAM latency. Put your answer in `1.5.txt`.

## 1.6 Submission and Grading

There is no checkoff for Part 1. This part is graded manually based on the following submitted materials under the "Part1-Timing" directory: code, the named question text files, and a pdf file.

- Code. Your code should be able to reproduce the histogram you submitted. Due to noise, we will run your code multiple times (around 5 times) and grade based on the best results. You should feel comfortable to submit your code as long as it can generate the expected results most of the time.
- The contents of 1.2.txt, 1.4.txt, and 1.5.txt.
- A pdf named solutions.pdf, containing your histograms from part 1.5.. Note that this pdf file will also be used by you to provide answers for the questions in Part 2 and Part 3.

# 2 Dead Drop: An Evil Chat Client (50%)

With all the preparation in Part 1, now you are ready to build Dead Drop: an evil chat client that can send messages between two processes running on the same machine. To make your life easier, you only need the ability to send integers from 0-255 (inclusive).

Unlike Part 1, in Part 2 you will have a lot of freedom in how you perform your attack. There are only two rules for the evil chat client.

1. The sender and receiver must be *different processes*.
2. The sender and receiver may only use syscalls and shared library functions *directly accessible from the provided* util.h except system(). Both the sender and receiver may use any x86 instruction except for `clflush`.

**Hardware covert channel:** With Dead Drop, you must implement cross-process communication using hardware covert channels. Suppose the sender and receiver run on the same physical machine at the same time. Processes on this machine share hardware resources (including caches, DRAM, processor pipelines, etc). If the sender process uses a hardware resource, it creates contention with other processes trying to use that same resource. Coupled with a mechanism to measure contention (such as a timer), this can be turned into a reliable way to send information from process to process that violates software-level process isolation. We suggest you build a Prime+Probe covert channel targeting the L2 cache. The reason is that (1) according to the analysis in Part 1, we have already observed that L2 and L3 latency can be clearly distinguished; (2) The L2 cache is private to each core, and thus is relatively easier to exploit compared to the L3 cache, which is shared across cores. You are also welcome to be creative to try other types of covert channels, though we do not recommend it for starters.

**Allowed code:** The sender and receiver may only use syscalls and the functions accessible from the provided *util.h* except for system(). There is no way to set up a shared address space between the sender and receiver, nor is there a way to call any obviously-useful-for-chat functions such as Unix sockets.

In addition, you may use any x86 instruction in your final submission except for `clflush`. The point of limiting your code usage is not to force you to re-implement convenience functions. Because we are unable to create a shared memory space between the two processes, traditional Flush+Reload or Evict+Reload attacks cannot be used. Instead, we can explore the possibility of using a Prime+Probe style attack.

**Expected behavior:** The Dead Drop client should behave in the following way. In two different terminals running on the same machine, you should be able to run the following commands:

```
1: Terminal B: $ make run_receiver

2: Terminal B:   Please press enter.

3: Terminal A: $ make run_sender
4: Terminal A:   Please type a message.

5: Terminal B: $
6: Terminal B:   Receiver now listening.

7: Terminal A: $ 47

8: Terminal B:   47
```

In other words, in lines 1 and 2, you start the receiver process in a terminal, and it prompts you to press enter to start listening for messages. Until you press enter, the receiver is not expected to echo messages from the sender. In lines 3 and 4, you start the sender in another terminal. Lines 5 and 6 are you pressing Enter to start the receiver process. It should now be ready to receive messages. In line 7, you type a message and hit enter on the sender side. Line 8 shows the same message appearing on the receiver side.

## 2.1 Getting Started

The files included for the Dead Drop portion of the lab are under Part2-DeadDrop.

- sender.c, receiver.c - template code for your sender and receiver respectively
- util.h, util.c - utility functions and library imports you are permitted to use (except for `clflush`)

Similar to Part 1, you need to compile your code using the command `make`. To test your chat client, you can open up two terminals on the lab machine and run `make run_sender` in one and `make run_receiver` in the other to demo your exploit. With these two commands, your processes will be running on two specific CPUs, SENDER_CPU and RECEIVER_CPU. You should have set the two variables to your assigned value. This is done to simplify the conditions for this lab.

The two CPUs you have been assigned are a pair of SMT (aka Hyperthreading) thread contexts which run on the same physical core and share all the hardware resources on that core, such as private L1 and L2 caches. Your implementation will only need to work on your assigned cores. This is because a successful exploit across any arbitrary pair of cores would require a resource shared between all cores on the machine, which would make this problem significantly more challenging. Percival describes the threats associated with SMT in more detail in *Cache Missing for Fun and For Profit* [1].

## 2.2 Transmitting A Single Bit Across Processes

In case you have no clue on how to get started building a covert channel to communicate an 8-bit integer, this section provides you a tip: start with communicating a 1-bit value first. You should find it useful reading this section to understand how to design a communication protocol, and several common pitfalls when you actually implement a protocol.

The discussion question and the exercise in this section are optional, meaning that you can get full credit for Part 2 if you directly implement an 8-bit chat client. If your 8-bit chat client does not work properly, you can still receive partial credit (10% of Lab 1) by completing a single-bit chat client.

**Communication protocol:** To build a chat client, we suggest you first construct a communication protocol, which includes: (1) Encode: what is the sender's action for sending bit 1 and bit 0; (2) Receive: what is the timing measurement strategy used by the receiver(e.g. the number of addresses to access);
(3) Decode: what is the threshold to be used to distinguish between bit 1 and bit 0.

You should come up with a tentative plan for (1) and (2), and then *experimentally* derive the threshold used in (3). If you are unable to derive the threshold, meaning that there does not exist a threshold to decode the bit, you will need to revisit your sender and receiver strategy.

*Hint:* To build a single-bit chatting client, you can use the whole L2 cache as the communication media. The receiver can measure the latency of accessing multiple cache lines.

**Discussion Question 4 (Optional):** Describe your communication protocol to communicate a single- bit value.

**Common pitfalls:** If the receiver needs to measure the latency of multiple memory accesses, you should pay attention to the following micro-architectural features that can introduce substantial noise to your communication channel.

- **Cache line size:** Be careful with the mismatch of the size of an integer and a cache line. Repeatedly accessing the same cache line will result in cache hits, and will not help you to detect the actual cache states.
- **Hardware prefetch:** Modern processors have hardware that can prefetch data into the cache before it is required as it attempts to anticipate future accesses. This may cause spurious cache evictions or mask evictions caused by the victim. For example, if an address has been evicted by the sender and is supposed to locate in L3, but it is prefetched by the hardware to the L2, your measurement will mistakenly consider this cache line has not been evicted before.

  Fortunately, the hardware prefetcher is not that smart. Generally, in most processors, it is only triggered by linear, fixed-stride access patterns within a page. Therefore, as long as your access pattern does not show a clear pattern, you can confuse the prefetcher to disable the prefetch effects.
- **Cache replacement policy:** While least recently used (LRU) is the most common example of a cache replacement policy, practical processor implementations often use much more complex policies. You may need to experiment a bit to roughly figure out how eviction and measurement works on your processor.

  Furthermore, be sure to carefully consider how you are accessing your cache lines. With Prime+Probe attacks, it is common to encounter *cache-thrashing* or *self-conflict*, a situation in which the attacker primes the cache set and evicts their own data with subsequent accesses while probing. One way to avoid this problem is to prime the cache by accessing the eviction set in one direction, and then probing in the reverse direction, as described by Tromer et al. [2].
- **Noisy or inconsistent results:** Because side channels exploit shared resources in unintended ways, the resulting covert channels can be quite noisy. Modern processors often contain optimizations that make them behave differently from the simplified architectures taught in class. This lab requires experimentation to find working approaches and values. You should not expect your solution to work on the very first attempt, so be sure to incrementally build up your solutions

and verify that each step is working before proceeding.

**Exercise 6 (Optional):** Derive the threshold for the decode operation. Implement a single-bit chatting client.

## 2.3 Transmitting An 8-bit Integer Across Processes

To transmit an 8-bit integer across two isolated processes, you can build a covert channel using the L2 cache since the sender and receiver will share the L2 via SMT. There is more than one way to do so, and we describe two possible methodologies here. Feel free to explore other methodologies.

- A naïve approach to transmit an 8-bit integer is by sending each bit sequentially. If you have already learned how to transmit a single bit in the above subsection, you can re-use your method to transmit 8 bits sequentially. In case you decide to go along this path, we have provided a string to binary conversion function and a binary to string conversion function in utils.h and utils.c.

  However, such a naïve approach requires the two processes to synchronize, and from experience the synchronization implementation is non-trivial.

- The second approach to encode an 8-bit integer is by using set-indices in the L2 cache. Since an 8-bit integer can have any value between 0-255, you can encode all possible values using 8 independent sets in the L2. This approach does not require a non-trivial synchronization, but on the other hand, requires that you implement a prime-evict-probe sequence at set-granularity between the sender and receiver. We would like to recommend this approach.

  Implementing set-granularity prime-evict-probe sequences requires a deeper understanding of the virtual address abstraction, and huge-pages. In case you decide to go along this second path, we have also provided an allocation of a huge page in the starter code in sender.c. We next briefly discuss the virtual address abstraction, and then give a primer on huge pages. You can skip the discussion on huge pages if you do not plan to use this approach.

- There exist many other approaches. Several communication protocols have been discussed in “An Exploration of L2 Cache Covert Channels in Virtualized Environments” [3].

**Linux Hugepages:** Keep in mind that the addresses you are dealing with in your C code are virtual addresses. However, physical addresses (which you will not have access to within your code) may be used when indexing into lower level caches. You may need to consider how address translation interacts with cache indexing/tagging as well as run some experiments to see if this presents a problem for your covert channel implementation.

The default page size used by most operating systems is 4K ($2^{12}$) bytes. You may find it useful to use a larger page size to guarantee consecutive physical addresses.

In particular, Linux supports *Huge TLB pages* for private anonymous pages, which allow programs to allocate 2 MiB of contiguous physical memory, ensuring $2^{21}$ consecutive physical addresses. This feature is enabled on the lab machine. The following sample code demonstrates how to allocate a single hugepage:

**Monitoring Hugepage Usage:** You can see if your huge page is being allocated or not by watching the status of /proc/meminfo. Namely, if you run cat /proc/meminfo | grep HugePages_, you should see the number of HugePages_Free decrease by 1 when your code is using one.

**Discussion Question 5 (Optional):** Given a 64-bit virtual address, fill in the table below. The table is intended to help you to figure out how to find addresses that map to the same L2 cache set.

```
void *buf= mmap(NULL, BUFF_SIZE, PROT_READ | PROT_WRITE,
                MAP_POPULATE | MAP_ANONYMOUS | MAP_PRIVATE |
                MAP_HUGETLB,
                -1, 0);

if (buf == (void*) - 1) {
   perror("mmap() error\n");
   exit(EXIT_FAILURE);
}

*((char *)buf) = 1; // dummy write to trigger page allocation
```

Listing 1: Allocating a hugepage.

| Page Size | 4KB | 2MB |
|---|---|---|
| Page Offset Bits | | |
| Page Number Bits | | |
| L2 Set Index Bits | | |
| L2 Set Index Fully Under Control? | | |

Table 2: Address calculation.

**Discussion Question 6:** Describe your communication protocol. Please refer to Section 2.2 for the components involved in a protocol.

**Exercise 7:** Implement a chatting client that can communicate an 8-bit integer. Please refer to the "Common Pitfalls" paragraph in Section 2.2 to debug your chat client.

### 2.4 Submission, Grading and Checkoff

You will submit the following items for this part under the "Part2-DeadDrop" Directory.

- submit your code
- In the solutions.pdf document, include your answers for the discussion questions (Discussion Question 6 is required, and all the others are optional in this part).

The grading of this part is based on your checkoff. You will also set up a 10-minute checkoff with me, during which you will open up two terminals on the lab machine and run make run_sender in one and make run_receiver in the other to demo your exploit. As mentioned before, if the 8-bit chat client does not work, we will give partial credit (10% of the whole lab) if you have completed a single-bit chat client.

## 3 Capture the Flag (30%)

In this part of the lab, you will be attempting to extract a secret value from a victim program. You will get a taste of a Capture the Flag (CTF) problem. The future labs will follow a similar pattern. Note that, we have deliberately designed Part 3 to be a relatively simple CTF problem. If you finished Part 2, this part should not take you a lot of time.

To keep the lab manageable, we have created a simple program, victim-N.c, that performs a secret-dependent memory access (pseudocode in Listing 2). The N will depend on how many accesses the victim makes to the cache set with an index of the flag. For now, we explain the victim code assuming the victim accesses 4 addresses. We explain at the end the other victim binaries. Your task will be to write a program, attacker.c that determines this secret value.

```
// Set flag to random integer in the
// range [0, NUM_L2_CACHE_SETS)
int flag = random(0, NUM_L2_CACHE_SETS);
printf(flag);

// Allocate a large memory buffer
char *buf = get_buffer();

// Find N addresses in buf that all map to the cache set
// with an index of flag to create a partial eviction set
char *eviction_set[N];
get_partial_eviction_set(eviction_set, flag);

// Main loop
while (true) {
    for (int i = 0; i < N; i++) {
        // Access the eviction address
        (*(eviction_set[i]))++;
    }
}
```

Listing 2: Victim pseudo-code.

**Exercise 8:** Complete the code in `attacker.c` to successfully extract the secret values from `victim-{2, 3, 4}`.

Using tmux, screen, or simply two SSH connections, you should be able to run make run_victim-N in one terminal and make run_attacker in another terminal. These two make recipes will compile your code and start the two processes on the correct CPUs using taskset. If you have problems running the victim binary, you may need to run chmod +x victim-N within your lab directory.

**Submission, Grading and Checkoff:** As with Dead Drop, you will place your code under the "Part3-CTF" directory. In your scheduled checkoff time, you will open up two terminals on the lab machine, run make run_victim-N on one, and run make run_attacker on the other to demo your exploit and show that the two printed flag values are the same.

**Partial Credit** For Part 3, we have provided 3 different binaries of the victim: namely, victim-2, victim-3, and victim-4. The binaries access different numbers of eviction addresses that map to the same cache set as the index of the flag. For example, victim-4 will access 4 eviction addresses, while victim-2 will access only 2 eviction addresses. Hence, the number of accesses the victim makes will act as a proxy to the difficulty of Part3 of the lab. You will get 50% partial credit for making your attacker successfully capture the flag for victim-4, 75% for victim-3, and full credit (100%) for victim-2.

## Acknowledgments

This lab was heavily adapted from MIT's 6.5950.

The original Dead Drop lab (Part 2 of this lab) was developed by Christopher Fletcher for CS 598CLF at UIUC. The starting code and lab handout are both heavily adapted from his work.

## References

[1] Colin Percival. "Cache missing for fun and profit". In: *Proc. of BSDCan 2005*. 2005.

[2] Eran Tromer, Dag Arne Osvik, and Adi Shamir. "Efficient Cache Attacks on AES, and Countermeasures". In: *Journal of Cryptology* 23.1 (Jan. 2010), pp. 37–71. issn: 0933-2790, 1432-1378. doi:

10.1007/s00145-009-9049-y. url: http://link.springer.com/10.1007/s00145-009-9049-y (visited on 09/18/2020).

[3] Yunjing Xu et al. “An Exploration of L2 Cache Covert Channels in Virtualized Environments”. In: *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*. CCSW ’11. Chicago, Illinois, USA: Association for Computing Machinery, 2011, pp. 29–40. isbn: 9781450310048. doi: 10. 1145/2046660.2046670. url: https://doi.org/10.1145/2046660.2046670.