**CptS 426 Project #8**

**Leak in the “Cone of Silence” – Cache Covert Channel**

*Spring 2025*

*Instructor: Shih-Lien (Linus) Lu*

*(15 % of Grade)*

**Project Summary & Goal: Covert Channel Creation**

With all the preparation in the previous lab, we are ready to build a covert channel to break the “Dome of Silence” (<https://www.youtube.com/watch?v=g1eUIK9CihA>). Imagine a mole agent with top secret security clearance in a national defense institution. She, Eve, wants to leak information she knows to another person, Bob, who has no security clearance. She wants to achieve this goal without letting anyone knows about it. She decided to utilize a shared computer in the institution which both have accounts and can run programs on it. The method she employed utilizes cache timing as a side channel to communicate. To demonstrate the ability we will only need to send one byte at a time. It will be easy to send more information once this is demonstrated.

We suggest you use **Flush+Reload to build your covert channel**. This is due to:

1. According to the analysis in our previous lab, we have already observed that accessing cache levels (L1/L2/L3) and DRAM 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, which we do not recommend for beginners.
3. Flush and Reload is an easier exploit than prime and probe

**Rules for this Assignment:**

Unlike the previous lab you are free to explore ways for this attack. We only require you to follow the following rules:

1. The sender and receiver must be *different processes.* That is, they are two programs which are run at the same time.
2. The sender and receiver may only use syscalls
3. You are not allowed to set up a shared address space between the sender and receiver to communicate through it.
4. You are not allowed to call any obviously-useful communication functions such as Unix sockets or pipe.
5. You are not allowed to use any shared memory/file (eg. /proc/shm).
6. Obviously, you may not copy any pre-packaged code from online resources verbatim. At least, for your own learning, you should understand what the code is doing before you use it. If you would like to use some convenience code that stays within the spirit of the assignment, please say so in the report.
7. You may use any x86 instruction in your final submission.

**Expected Behavior of Your Completed Assignment:**

At the end you should be able to demonstrate on your PC with the following functions.

1. You will have two terminals opened.
2. The first one runs a program called “sender” (name this Terminal S)
3. The second one runs a program called “receiver” (name this terminal R)
4. On the first terminal you should be able to type a message. This message can be a single character (byte).
5. On the second terminal you should be able to receive this message (one byte) and display it on the screen.

Here is an example screen display:

Terminal R: $ run receiver (or just receiver) // you start the receiver process in a terminal

Terminal R: Please press enter to start: // the receiver prompts you to press enter to start listening for messages

Terminal S: $ run sender (or just sender) // you start the sender in another terminal

Terminal S: Type a message.

Terminal R: $ // you press Enter in the receiver's terminal

Terminal R: Receiver starts listening.

Terminal S: $ 4 // you type a message (a char) and hit enter in the sender's terminal

Terminal R: 4 // receiver should generate the same message as you entered on the sender's side

If you have time you can create a buffer of a fixed size and store the message in a buffer to send off. The behavior would look like:

Terminal R: $ run receiver (or just receiver) // you start the receiver process in a terminal

Terminal R: Please press enter to start: // the receiver prompts you to press enter to start listening for messages

Terminal S: $ run sender (or just sender) // you start the sender in another terminal

Terminal S: Type a message.

Terminal R: $ // you press Enter in the receiver's terminal

Terminal R: Receiver starts listening.

Terminal S: $ I am sending you the secret you asked for // you type a message (a char) and hit enter in the sender's terminal

Terminal R: I am sending you the secret you asked for // receiver should generate the same message as you entered on the sender's side

**Hints:**

1. **Transmitting a Single Bit Across Processes (Optional)**

We suggest that you start with building the framework of sending one bit of information. You can start by flushing the cache if you want to send a “1” bit and do nothing for sending a “0” bit from the sender process. On the receiver process you should preload a data into cache and wait to see if the data is still in the cache or it has been evicted by sender (flushed). This requires you to measure the time of accessing the data you have previously loaded into the cache. There are several common difficulties when you actually implement a protocol and we will discuss those in the following section. While, this first step is not required, it is recommended to complete this part first to get partial credit at least if you don’t finish the assignment on time.

1. **Things to think about**

We suggest the following steps for you to complete the project:

1. **Encoding scheme**: What is the sender’s action for sending a “1” bit and a “0” bit as mentioned above? How do you flush the cache?
2. **Decoding scheme**: How do you find out which bit (0 or 1) is sent? How do you measure access time? What is the threshold used to distinguish between 1 and 0?
3. **Establish a way to fine tune time:**  There are noises on the channel. How do you plan to filter out the noise? Do you have a way to fine tune the boundaries for distinguish “0” and “1”? You should decide on your encoding and decoding schemes first. Then *experiment* with time measurement.
4. **Communication Protocol**

To build a complete communication channel, we suggest that you construct a communication protocol, which may include a start symbol or a stop symbol.

**Common Mistakes and Some More Helpful Tricks**

If the receiver needs to measure the latency of multiple memory accesses, you should pay attention to the following features that can introduce substantial noise to your communication channel.

* **System Calls:** Be careful issuing system calls (such as printf) while measuring latencies. System calls have a large amount of overhead, both adding additional latency, and potentially causing significant cache evictions during execution.
* **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 very smart. Generally, in most processors, it is only triggered by linear, fixed-stride access patterns within a page.

* **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 you have learned. This lab requires experimentation to find working approaches and values. You should not expect your solution to work on the first attempt, so be sure to incrementally build up your solutions and verify that each step is working before proceeding.
* A simple approach to transmit an 8-bit (byte) data is by sending each bit sequentially. If you have already figured out a way to send and receive a single bit, then transmitting a byte is straight forward.
* **Linux Huge Pages**

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. You may need to dust off your notes took for Operating Systems class or refer to this Youtube video on the [Basics of Virtual Memory](https://youtu.be/8yO2FBBfaB0?si=ea9guTyJKkaujwP5) for a refresher.

The default page size used by most operating systems is 4K (212) 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 MB of contiguous physical memory, ensuring 221 bytes consecutive physical addresses. Here is a link to help you [enable Hugh Pages](https://xmrig.com/docs/miner/hugepages) on your system (both Windows and Linux). I am able to achieve a covert channel without changing the page size on EME136 machines.

**Submission and Grading**

This lab is graded manually based on the following:

1. Submit all your codes and how you compile and run your codes detailly (50%)
   1. Please include a flowchart of your approach
   2. List all parameters for your system’s cache setup
   3. Description on how you measure time
2. A report which can document the working of your covert channel (20 %). The report should include:
   1. How far have you been able to achieve the convert channel. – just sending 0s and 1s or a character a buffer containing ASCII bytes
   2. Screen captures of running the sender and receiver in two terminals
   3. It would be a plus if you could use a video to document the running of your codes
3. Include in your report the following discussion: (30%)
4. A summary of what is cache side-channel
5. Things you tried and mistakes you made and how you discovered them and overcome them
6. Other lessons you have learned