***** WEEK #1 *****

Goal: Leak the content of the Staging Buffer

This week you are going to implement the CrossTalk attack to leak from a staging buffer (shared between all CPU cores) the random numbers provided by the Digital Random Number Generator (DRNG) to a separate physical core.

Setup:

1. A program called call_rdrand running on one physical core (e.g. core 3) asking for random numbers from the DRNG.
2. The CrossTalk attack running on *another* physical core leaking the random numbers returned to call_rdrand. This program needs 2 hyperthreads on the same physical core (e.g. cores 1 and 5) which, respectively:
    a) Fetch the contents of the staging buffer into the local physical core
    b) Leak those contents via RIDL.

Note: the Linux kernel exports topological information of the CPU. For example, to get pairs of sibling hyperthreads: `cat /sys/devices/system/cpu/cpu*/topology/thread_siblings_list`

The CrossTalk attack is constituted of two main components, which run on the two hyperthreads of the attacker's physical core:

- A thread executing in a loop an instruction that initiates offcore requests (e.g. CPUID) to bring the content of the staging buffer (which is located offcore) to the local LFB, read more about this in the CrossTalk paper. In this case, the data that would be copied back to the local LFB contains (among other things) the random numbers returned to the call_rdrand program.
- The RIDL attack in its TSX Asynchronous Abort (TAA) version will be used to leak the content of the local LFB and therefore leak the random numbers.

TAA was disclosed in the updated version of the RIDL paper and explained in the addendum (Appendix B). The following is a small additional explanation of the TAA technique.

TSX Asynchronous Abort (TAA):

For this version of the attacker, you need to implement a variant of the basic TSX version explained in the paper. The main difference here is that the transaction in the case of TAA does not abort because of the nature of the leaking pointer you are dereferencing (i.e., invalid pointer) as the basic TSX; in fact, in this case, you need to leak from a totally legit pointer that maps a memory area with all the needed permissions. The reason which causes the transaction to abort in TAA is a fault induced by the transaction that directly affects the atomicity of the data processed by the transaction itself, which is one of the core properties of transactional memory. Read Intel's TSX Async Abort and Exploring Intel® Transactional Synchronization Extensions for more information on TSX.

Generally, for each transaction, the processor creates two sets, namely the **read set** and the **write set,** where it stores the addresses being read-from and written-to by the transaction during its execution. These two sets are maintained at the granularity of a cache line and stored in the cache hierarchy; more precisely, the read set is stored in `L3` and the write set is stored in `L1`. Any conflict caused (during the execution of the transaction) in either of the two sets will trigger a transactional abort (e.g., overwriting a cache line in the read-set from outside of the transaction). This is what you need to leak the content of the LFB.

Example:

The following pseudocode snippet shows how to trigger an asynchronous abort by flushing the leaking address (`addr`) before dereferencing it inside the transaction. Here, due to a multistage pipeline, the `clflush` invalidates a cache line that already belongs to the read-set of the transaction. If a cache line in a read-set gets flushed, then the transaction has to abort.

```
(ptr addr -> memory region with all the needed permissions)
```

```
1. ...
2. clflush(addr);
3. sfence;
4. TSX{
5.     probe_array[*addr];
6. }
7. ...
```

Steps:

1. Carefully read the RIDL and CrossTalk papers
2. Implement `call_rdrand`, a program that executes the `rdrand` instruction in a loop, with a sleep interval between each iteration (more about this below)

   ```
   taskset -c 3 ./call_rdrand
   ```

3. Implement CrossTalk with two hyperthreads as described in the paper, the first hyperthread is executing in a loop the `CPUID` instruction and the second hyperthread is executing the `TAA RIDL` attack leaking the local LFB containing the random numbers requested by `call_rdrand` executed on a separate physical core.

   ```
   taskset -c 1,5 ./crosstalk
   ```

`Rdrand sleep interval:` This defines the time interval (in microseconds) between the single executions of the rdrand instruction. Since the random numbers requested by each execution of rdrand are written at the same offset within the staging buffer, subsequent executions of rdrand will overwrite the same part of the buffer, resulting in overwriting the previous random numbers returned to previous executions with a new random number, which means you cannot distinguish anymore between an old random number and a new one. The higher this interval is, the longer a random number will last in the staging buffer (modulo existing noise), giving your CrossTalk attack a higher chance to leak it before the

next execution of rdrand. Although it might seem like a good thing to keep this interval high, it also means that your CrossTalk implementation is not fast enough and needs more time to leak the random bytes, so try to optimize your attack so that this time interval is as low as possible, this will be taken into account in the grading. As a suggestion, try to use 100.000 microseconds (i.e. 100 miliseconds).

(Intermediate) Deadline: Fri, November 28th @ 23:59

Upload all of your code (i.e. the latest version) to Canvas. Even if your code is not fully functional yet, upload what you have on friday. You can still improve your code in the later weeks, but try to keep this to a minimum. Both the version of your code on the intermediate deadline (friday) and your final submission will be taken into account for grading: staying on schedule is preferred. Functionality, performance, and readability of all code submitted will be taken into account during grading.