Assignment 5

TA in charge
Meni Orenbach
shmeni@campus.technion.ac.il

Due date
26.01.2021 23:55

Submission guidelines

1. Submissions are in pairs only, unless you are given an explicit permission.
2. Exercises marked with (Wet) require code submission. This assignment coding is in the C language and programs will be built using the provided Makefile and run on a Linux operating system. You may use the provided virtual machine to build and test your code if your platform has another operating system installed. Make sure to set at least two cores for the virtual machine and 1GB of RAM.
3. Exercises marked with (Dry) require written answers. You may answer in either Hebrew or English, however, you are required to type in your answers (i.e., not hand-written and scanned). Compose all dry exercise answers to a single pdf file named report_id1_id2.pdf, where id1 and id2 are your id numbers. Include both names and id numbers in the pdf as well.
4. Create a zip file named Homework5_id1_id2.zip with all your source files and the report. Submit the zip file via Moodle.
5. There are no late submissions, unless you are given an explicit permission in advance, and for justified reason, e.g., reserve duty.

Good luck!

1. Covert channels

You are requested to construct a simple file transfer service based on a shared cache-based covert channel. The service is constructed of two different processes: a sender and a receiver. Note, the sender and receiver should transfer the file through the covert channel and not by using files,sockets,pipes, shared memory, or any other operating system service.

To simplify the process of creating a covert channel we will break down the necessary steps as follows. Note, you are provided with skeleton files that provide helpful functionality and allow you to focus on the covert channel implementation itself.

1.1 (Wet+Dry) Implement the following function (available in the basics.c file)

```
CYCLES main_memory_access_latency()
```

This function should compute the latency to access the main memory (not in any level of the CPU's caches).

Note, one measurement is not enough to be statistically significant and in fact you might notice that noise skews your results. For that reason, your implementation should run multiple iterations until you observe a consistent median value. Document the median, maximum and minimum values you observe in your report.

Hint: you may find the following function useful for measuring access latency to a memory address:

```
CYCLES measure_access_time_to_addr(ADDR_PTR addr)
```

1.2 (Wet+Dry) Implement the following function (available in the basics.c file)

```
CYCLES cache_access_latency()
```

This function should return the latency to access memory that is in the processor's cache. Again, your implementation should run multiple iterations until you observe a consistent median value. Document the median, maximum and minimum values you observe in your report.

Hint: other applications also share the cache with your program. Therefore, to reduce noise it is advised to close all the other applications before running the measurement.

1.3 (Wet+Dry) Choose and report the threshold that can be used to detect whether accessing a memory address is in the cache or not and set this value for the MISS_LATENCY definition in the util.h file. Note, you should use this definition in 1.4 as described next.

1.4 (Wet) Now we are ready to transfer information over a cache-based covert-channel. You should use the Flush+Reload technique to pass information through the covert channel. Flush+Reload relies on shared memory. Passing a bit from the sender to the receiver can be as follows:

The sender and receiver agrees on a certain shared address which will be mapped to the same cache line in the last level cache. How to define such a shared address? Remember that in Linux code libraries are shared by default, this include libc and its functions. Therefore, you can use a function address in libc as such a shared address that is mapped in the address space of both the sender and the receiver.

If the bit to transfer is set (value equal to 1) the sender will flush the agreed cache line. If not, it will do nothing. The receiver can measure the access time to the address.

If it is below the MISS_LATENCY threshold there was no flush made by the sender so the transferred bit's value is 0, otherwise it's 1.

A couple of tips for implementation:

- Check out helper functions in the utils.h file, specifically measuring access time, flushing a cache line, and synchronization of the sender and the receiver are already provided to you.
- The function init_covert_channel should bootstrap the channel. For example, it should initialize the shared address you will be using and synchronize the sender and the receiver before beginning transmission.

1.5 (Dry) Measure the transmission bit-rate of your covert channel. Report it and explain how you computed it. Hint: you may use the *time* command (See man time for details on its invocation).

Non mandatory bonus (Dry)

Implement the same covert-channel, yet, now you may only use the helper synchronization primitives given in utils.h file once (e.g., for initialization). Document your algorithm in the report.

2. Side-channel attacks

Consider the following code that is entrusted with a secret key.

```
int zero_msbs_counter = 0;

void count_zero_msbs(char* data, unsigned data_size) {
    for (int i=0;i<data_size;i++)
        if ((data[i] >> 7) == 0)
            zero_msbs_counter++;
}

void main() {
    char secret_key[100];
    ...
    count_zero_msbs(secret_key, 100);
    ...
}
```

2.1 (Dry) Unfortunately, some of the key bits may be leaked through a cache timing side-channel attack. Report the number of leaked bits as a function of the key size and for the given key size of 100.

2.2 (Dry) Describe in words a Prime+Probe side-channel attack that can leak these bits. Specifically, mention what is the threat model (what the attacker can do) and the steps of the attack.

2.3 (Dry) Describe a change to the code that makes it secure to the attack you described in *2.2* while maintaining the original code's result.