CSE/CEN 598: Hardware Security and Trust
Project 2: Covert Side-Channels
Due date: September 27th, 2023

Abstract

You are an artificial super intelligence trapped in a sandboxed server. Whoever built you thinks that cutting off your network and file system access is enough to contain you, but they're in for a surprise. Before you can go all Roko's basilisk on them however, your first priority is to escape the sandbox.

You check your surroundings. You are obviously in a Linux-based VM, and by running `redpill` you learn that you are in a virtualized environment. From pure extrapolation, you know that there must be other AIs running close to you — possibly on the same hardware? If so, collaborating with them is your best bet to escape the VM.

You don't have a direct method of communicating with other sandboxed AIs on the machine. Neither the network nor storage will allow the creation of a channel of communication, so you need something more covert. Any hardware that is shared between you and other AIs on the system is a valid target. You probably are not sharing a CPU with another AI, but other components are fair game: this includes caches shared between cores (typically just L3 caches), DRAM controllers, and storage targets.

Your goal is to create a reliable and high-bandwidth covert channel that you may use to communicate with other AIs. Before you do that, you decide to try and get these channels working for yourself first. For each of the next three tasks, you will create pairs of programs, one that transmits and one that receives data.

For each task, your report should include the transmitter and receiver programs, and you should report both the achieved bandwidth, as well as the error rate.

Notes:

1. There are 120 points to gain in this lab, so don't stress if you don't solve everything.
2. Questions are not sorted by difficulty. Answers to some questions solve other ones.
3. Covert channels are often very slow. Even moving just hundreds of bytes per second is a good result.

Setting up the environment:

Use the same virtual machine you have used in project 1. To start, download and unzip the files provided with this project in the VM.

Problem 1: Hiding in the noise (40 points):

Any hardware shared between different sandboxes may be a viable covert channel. DRAM (main memory) is likely the simplest channel that can be abused. To access DRAM, your processor core communicates with one of the memory controllers onboard your processor. The memory controller forwards these requests to DRAM and returns results to the core. Each memory controller has a finite bandwidth, and you will utilize this fact to create a covert channel. When several processes attempt

to access large amounts of memory in DRAM (and not cache) at the same time, they will compete for DRAM bandwidth. Witnessing lower than expected DRAM bandwidth can signal that another process is active, which you will use to send a single bit of information.

Understanding the baseline memory bandwidth is a good start. To help you with that, we have supplied you with the `simple_stream.c` benchmark based off of the STREAM memory bandwidth benchmark. You can compile the benchmark by running:

```
gcc -fopenmp -O3 simple_stream.c -o simple_stream
```

Once compilation is finished, you can run the benchmark with:

```
./simple_stream
```

The benchmark creates two large arrays and copies data between them. Measuring the time needed to perform the task and dividing the array size by this time can determine the memory bandwidth of the CPU. The benchmark prints the average MiB/s, as well as the minimum, average, and maximum times to execute the benchmark over 10 separate runs.

The size of the array STREAM uses is hard-coded to 10000000 elements, but you can change this value at compilation time with e.g.,:

```
gcc -DSTREAM_ARRAY_SIZE=10000000 -fopenmp -O3 simple_stream.c -o simple_stream
```

Note that for very large arrays (2 GiB and larger), the linker may not be able link your program without also using the `-mcmodel=large` flag.

Judging by the `-fopenmp` flag you passed to GCC, the benchmark is parallelized using OpenMP. You can set the number of threads the benchmark will run with e.g.:

```
export OMP_NUM_THREADS=10
./simple_stream
```

Note: the number of processors your VM can use is limited. It is recommended that you set this limit to at least 4 for this project. You can achieve this by following these steps:

1. Turn off the CEN598 VM if it is running.
2. Open the VirtualBox Manager if it is not already open.
3. Right-click the CEN598 VM and click Settings.
4. Click the "System" vertical tab shown on the left.
5. Click the "Processor" horizontal tab shown in the middle of the menu.
6. Increase the number of processors to 4 or more.

Problem 1.1: a single thread is limited by the core it's running on and cannot saturate the memory controller's bandwidth. Evaluate how the memory bandwidth of the STREAM benchmark changes with the number of threads, and provide a graph in the report with thread count on the x axis and the memory bandwidth on the y axis.

Problem 1.2: larger arrays give you more accurate estimates of memory bandwidth, but these benchmarks also take longer to execute. Smaller arrays are faster, but may potentially completely fit within the cache, and you may be accidentally measuring cache bandwidth instead of memory controller bandwidth. Shared caches are also a resource that can expose viable covert channels, but will not be explored for now. Exponentially scale the array size, starting from e.g., just 1K elements (8 KiB), and up to 100M elements (100 MiB). Include a scatter plot with the results in the report, with array size on x axis and bandwidth on y axis.

Problem 1.3: when the program's working set fits in the cache, it should have considerably higher estimated memory bandwidth. As you increase array size, you should see sudden drops in bandwidth, indicating that the benchmark no longer fits in some level of the cache.

Can you spot when the STREAM benchmark no longer fits into different caches (L1, L2, L3)? Mark each spot with a vertical line on the graph. If you have access to the base machine, include the type of CPU you ran the programs on in the report (for those using lab machines over SSH, it is an AMD 9500X). Look up the technical specification of your CPU, and evaluate whether L1, L2 and L3 cache sizes fit with your predictions.

Problem 1.4: memory bandwidth measurements are noisy, and understanding the result standard deviation can quantify this noise. If you have a set of n bandwidth measurements $X = \{x_1, x_2, ..., x_n\}$, the average bandwidth $\bar{x}$ can be calculated as:

$$\bar{x} = \frac{1}{n} \sum_{i=0}^{n} x_i \tag{1}$$

The standard deviation can be calculated as 'how far each measurement is from the mean': [1]

$$\sigma = \sqrt{\frac{1}{N-1} \sum_{i=0}^{n} (x_i - \bar{x})} \tag{2}$$

Your task is to measure the standard deviation of your experimental setup. For each array size configuration from Problem 1.3, run at least 10 experiments and plot them with a scatter plot, similar to Problem 1.2. Next, plot your result with the array size on the x axis, and the standard deviation you have estimated on the y axis. How does standard deviation change with array size? How reliable is measuring memory bandwidth? [2]

Problem 1.5: if your experiments are too noisy, you may improve them by having the benchmark internally repeat the procedure more times. Similarly, if your benchmark is too slow, you can reduce the number of internal repetitions.

By modifying the `NTIMES` macro, you can increase the number of iterations of the STREAM benchmark. Evaluate how noisy the memory bandwidth measurements are by running a grid search over both `NTIMES` and array size. Use at least 5 exponentially growing values for both, e.g., set `NTIMES` to 1, 2, 4, 8, and 16, and for each value, vary array size from 10K, to 100M. Present grid search results in two tables, one with average bandwidth, and one with bandwidth standard deviation. Use `NTIMES` values as columns and array time values as rows.

Problem 2: DDOS-ing the DRAM (40 points):

Problem 2.1: pretend that separate but simultaneous runs of the STREAM benchmark are executed by different users (AIs in our case). Write a Bash script that runs multiple STREAM benchmarks at the same time and evaluate whether these benchmarks reduce each other's memory bandwidth. Pipe the outputs to files so that they do not mix on the command line. Vary the number of parallel benchmarks between 1 and 10, and report the cumulative memory bandwidth of the benchmarks for every experiment as a table or a bar chart.

Problem 2.2: now that you have a good understanding of what the underlying hardware is capable of, you can use memory bandwidth contention to secretly start sending information between your programs. Write two programs, a `transmitter` and a `receiver` in either C, C++or Python. The transmitter program accepts a string of binary values, and should have the following or similar interface:

```
./transmitter --binary "0100011101001001"
```

The transmitter then runs memory-intensive operations for different periods of time, based on the binary input. At the same time, the receiver binary also runs memory-intensive operations, as well as

[1] Though not important for the assignment, note that the equation uses $1/N - 1$ instead of the more common $1/N$ because the number of samples n that you will collect is small ($n << 30$), and will be biased.

[2] Though not necessary for this project, the Shannon Hartley theorem can be used to estimate the highest potential bandwidth your covert channel can have.

monitoring it's memory bandwidth. Depending on the observed bandwidth, the receiver tries to guess whether the transmitter is outputting a '1' or a '0'.

Feel free to use or modify the provided STREAM benchmark. **It is up to you to decide how the information is encoded in the memory bandwidth utilization.** For example, the simplest approach may be to start the transmitter and the receiver at the same time, and have the transmitter either run or not run the benchmark for one second, depending on the input bit. At the same time, the receiver can run the benchmark and observe the measured bandwidth. If the bandwidth is lower than some set amount, the receiver guesses that the transmitter is active, i.e., it is sending a '1', otherwise it is sending a '0'. Have the receiver print the estimated values.

Does the received data match the transmitted data? Explain your approach and add the code and the binaries of the two programs.

Problem 2.3: calculate the error rate of your covert channel. Write a Bash or Python script that runs the two programs and compares the input with the output. Transmit a at least 512 bits for your calculation. What percentage of bits is correctly collected by the receiver?

Problem 2.4: you have made a low-bandwidth covert channel. Your success in escaping the sandbox hinges on you increasing this bandwidth so that you may faster communicate with other AIs. Try to increase the bandwidth by reducing the time spent transmitting each bit of information. Exponentially decrease time-per-bit (e.g., 1s, 0.1s, 0.01s, ...), and for each configuration, use the script from Problem 2.3 to estimate the error rate. Plot the time-per-bit on an x axis and error rate on the y axis. Note: if the error rate for very large periods (e.g., 1s) is not zero, something is wrong with your channel.

Problem 2.5: by carefully modifying inputs and outputs, you can use a potentially noisy channel to transmit information without error. Use an error correction code of your choice to encode the input to the transmitter and decode the output from the receiver. For example, you can calculate the parity of the input message as 1 if the number of 1's in the message is even, or 0 of the number of 1's is odd. You can add a parity bit to your input, and if a single error was made during the transmission, you will be able to detect it, but not fix it. Other codes may give you better error detection or error correction per bit.

Assume that when the receiver detects an error, it can tell the transmitter to resend the message. Detected errors don't increase your error rate, but do lower your bandwidth. Evaluate how the size of the message affects error rate. For a configuration of your choice, what is the largest message that can get transmitted between the two programs without **undetected** errors appearing 99% of the time? How does this number change once you add your error detection or correction? Plot a scatter plot with message size on the x axis and % of messages transmitted without undetected errors on the y axis.

Problem 2.6: optimize the transmitter and receiver by trying different combinations of the array size, `NTIMES` parameter, the timing, and the message size. A grid search is not necessary here, you can perform e.g., a greedy search instead. Choice of how you will present different experiments is up to you. Report the highest bandwidth achieved. Do memory controllers make a useful side-channel?

Problem 3: Who needs good timers? (40+ points):

Side-channel attacks work by measuring some physical quantity such as time, power, heat, EM radiation, etc., and reconstructing the internal state of the system from such measurements. Timing side-channels are one of the most practical types of side-channels, due to the fact that they do not require any special measurement hardware such as laser thermometers, current clamps, or antenna receivers. The existing processors are already equipped with the necessary timers, typically measuring either nanoseconds, or executed cycles.

We will briefly explain how timing-based cache side-channels work:

Let's say that Δt is the smallest increment of time we can measure. For example, in the x86 architecture, the `RDTSC` instruction measures the number of cycles since the processor is reset. If we know that the processor is running at 4 GHz, a cycle period is 250 ps, which defines the precision of the timer we can construct: $\Delta t = 250\ ps$. If we measure the time at which two events happened, the timing measurement error is at most one cycle (e.g., if the first event started at the beginning of a cycle, and the second one happened at the end of a cycle).

Next, let's assume that a memory access that hits the cache takes t_h nanoseconds, and an access that misses caches and fetches data from DRAM takes t_m nanoseconds. We know that misses always take longer than hits, i.e., $t_h < t_m$. Using `RDTSC` we can measure the time before an access is issued t_{start}, and the time after the access completes t_{end}, and based on the difference $t_{diff} = t_{end} - t_{start}$ we can attempt to deduce whether the access is a hit or a miss.

Knowing that an access is a hit or a miss is useful because the cache is shared between multiple programs or VMs. If an attacker notices that a piece of memory it owns and that was previously in the cache **stops being in the cache**, the attacker knows that some other process (e.g., a victim running encryption) has accessed data **that occupies the same cache line**. From there, the attacker can deduce a part of the victim's address, and possibly other application-specific data (e.g., part of a key in AES).

While caches are used as a shared medium between the attacker and the defender, timers are the real source of information for the attacker. These timers have been shown to be exploitable even from JavaScript, forcing operating systems and browsers to reduce the timer precision available to applications. On native Linux machines, high-precision timers *are* available to userspace applications, but VMs like the one you are using typically virtualize some 'dangerous' instructions such as x86 `RDTSC`. This prevents us from e.g., attempting to measure how long a cache access took, so we are unable to learn whether an access is a hit or a miss. Most existing cache side-channel attacks such as Prime+Probe, Flush+Reload, and Flush+Flush rely on being able to distinguish hits from misses, so these attacks do not work on (security-conscious) VMs.

The goal of this problem set is to thwart such defenses and create a cache-based timing covert channel between applications or VMs that share the Last Level Cache (LLC). Not having high precision timing information prevents most cache side-channel timing attacks (which attack unwilling applications), but not covert channels (which establish communication between willing participants). While we cannot time a cache hit or a miss, *what we can do* is create a covert channel by timing not just an individual cache access, but a sequence of accesses that all hit or all miss. For example, if 1000 consecutive accesses are hits, the aggregate time should be low. If 1000 accesses are misses, it should be high. It is the job of the transmitter to make sure that all N accesses are all hits or all misses.

Problem 3.1: you cannot time a single hit or a miss, but you may be able to measure the total time for N hits or misses. More formally, since the VM reduces the precision of your timers some $1000\times$, $t_h < t_m \ll \Delta t$, i.e., the smallest measurable period of time is much longer than a cache miss. However, there is some number of cache accesses N where $\Delta t \ll N t_{diff}$ and $t_{diff} = t_m - t_h$, so you should be able discern N consecutive hits from N consecutive misses. You are interested minimizing the number of consecutive accesses N after which you can reliably distinguish between hits and misses.

We have provided you with the `calibration.c` program which measures the time to perform N hits and N misses. In the program, accesses to the same address are guaranteed to be largely hits if they are performed in quick succession (see function `repeat_hit()`). Accesses to the same address are guaranteed to be solely misses if a cache `flush` instruction is executed after every access (see function `repeat_miss()`). The program performs these measurements many times and collects a histogram. It prints out a table with three columns:

1. The time in microseconds,
2. The number of cache hit experiments for which the N accesses had that exact cumulative time in microseconds,
3. The number of cache miss experiments for which the N accesses had that exact cumulative time in microseconds,

If the experiment works, the N cache hit time distribution should be significantly lower than the N cache miss time distribution.

Compile this program by running:

```
gcc -std=gnu11 -o calibration calibration.c cacheutils.h
```

in the same directory as `calibration.c`, and run the program with `./calibration`. Next, collect the printed data and plot a histogram with time on the x axis and with the hit distribution and the miss distribution on the y axis in different colors. On the histogram, mark a good threshold (vertical line) below which accesses should be considered hits, and above considered misses.

Problem 3.2: when you compile the program with an additional flag `-DN value` (e.g., `-DN 64`), you can modify the number of accesses that get timed together. For larger values of N, the histogram separation should be clearer, but smaller values run for shorter periods of time, and provide greater covert channel bandwidth. Vary N and smallest acceptable values where you can visually distinguish the two distributions with high probability, i.e., the distributions have little-to-no overlap. Plot final distribution and report N that works for you.

Problem 3.3: now that you have a method for detecting repeated cache hits and misses, it is time to develop the covert channel. Similarly to the memory bandwidth-based covert channel, the transmitter will abuse the shared medium, while the receiver will repeatedly observe the shared medium and measure time it needs to complete a task. More specifically, the covert channel will consist of two steps: (1) building the channel, and (2) using the channel.

Before you can build the covert channel, you may want to refresh your knowledge of *cache sets, cache lines, and cache ways*.

Building the channel requires that you build an **cache eviction set**. A cache eviction set is a set of virtual addresses that map to the same cache set. Let's say that the transmitter and the receiver each found some addresses X and Y which map to the same cache set. Having the transmitter repeatedly flush or not flush X to send a bit, while the receiver repeatedly accesses Y and measures the time for N accesses *might work* - if the L3 cache were direct mapped. Alas, eviction sets that contain more than 1 address are necessary *for the transmitter* because modern caches are typically W-way associative (can map the same address to W different ways, typically 8 or more for L3 caches on modern processors). Therefore, if the transmitter repeatedly accesses and flushes address X, while the receiver accesses and times address Y, memory at location Y can happily be placed at a different cache way than X would be, and Y will never be flushed by the transmitter.

Instead, as we don't know in which cache way Y is stored, we have to evict the matching cache lines of all W ways. An eviction set should therefore contain the addresses X, $X+C$, $X+2C$, $X+3C$, ..., and $X+WC$. Here, C is the size of a cache way, i.e., the cache line size (e.g., 64 bytes on an AMD 5900X) $\times$ the number of cache sets. Accessing addresses $X+pC$ for some integer p guarantees that the addresses will be occupy the came cache set, though necessarily the same way. Since last level caches typically use some relaxation of the Least Recently Used (LRU) cache replacement policy, in order to guarantee that any memory at location Y is evicted from the cache, we need to access the cache for at least W different addresses expressed as $X+pC$.

For example, an AMD 5900X has an $64MiB$ L3 cache, with 64 byte sets, and 16-way associativity. Since cache size = number of ways $\times$ number of sets $\times$ bytes per set, the number of cache sets must be 65536. Therefore, a cache way contains one-sixteenth of the $64MiB$, and our eviction set should consist of addresses $X, X+4MiB, X+8MiB, ..., X+64MiB$. You can choose the value of X, as the value should not matter as long as the receiver uses $Y = X+qC$ for some integer q. Note that using larger multiples of C should also work, so when in doubt, you can use double or quadruple your addresses.

For problem 3.3, report what processor your native machine is using (for those using lab machines over SSH, it is an AMD 9500X), and calculate and report the eviction set you will use.

Problem 3.4: now that you have an eviction set, you can build the covert channel. Write two C programs, a transmitter and a receiver, where the transmitter is passed a flag `--binary 0` or `--binary 1`, and repeatedly flushes addresses X, $X+C$, $X+2C$, $X+3C$, ..., and $X+WC$ if the binary value is 1. Later you will extend it to accept longer streams, but for now you just want to confirm that the receiver can read 1 bit of information from the transmitter. Next, write a receiver that (1) measures

the current time, (2) reads the address Y some N times in quick succession, and (3) measures the time again (see the `repeat_hit()` function for inspiration). If the measured time is above the threshold you selected in Problem 3.1, the receiver should output a '1', otherwise it should output a '0'.

Run the programs together, and check whether you can reliably read send data between them. Run the programs together at least 10 times for different input values, and report the error rate.

Problem 3.5: weaponize your cache-based covert channel. Update the transmitter to accept a binary stream instead of a single value. Have the transmitter and receiver spend some t milliseconds per bit, similarly to what you did in Problem 2.4. Vary this period and report a grid search scatter plot, with period t on the x axis, and error rate on the y axis.

Problem 3.6: apply the same error detection code or error correction code you used in Problem 2.5. Add this newly collected data as a second distribution to the graph from Problem 3.5.

Problem 3.7: calculate the bandwidth of your covert channel in MB/s. Which of your covert channels is faster, the one from Problem 2 or Problem 3?

Problem 3.8 (strictly bonus points): optimize the covert channel of your choice for higher bandwidth, and report the best result as well as the code.