**Laboratory 7**

**Cache memory**

CPU cache refers to memory that is built directly onto your CPU. It is distinct from system memory (RAM), which is mounted to the motherboard and actually far slower than the CPU cache.

There are several types of cache memory:

* L1 Cache, or Level 1 Cache, is the primary CPU cache. It’s the fastest “level” of CPU cache and also has the smallest memory capacity. The faster/lower-level your Cache, the less CPU cycles it takes for your CPU to access it.

*Note:* The L1d cache stores data, while the L1i cache stores instructions for the cores to execute.

* L2 Cache, or Level 2 Cache, is the secondary CPU cache. It’s slower than Level 1 Cache, but still pretty fast, and offers a significant bump in memory capacity.
* L3 Cache is slower than Level 1 and 2 Cache but serves the purpose of making them both faster by providing the largest pool of on-CPU memory for them to utilize.

**Practic 1. (1 pt)** The *[lscpu](https://www.baeldung.com/linux/cli-hardware-info" \l "3-using-lscpu)* command is a useful command-line utility for obtaining insights into the CPU architecture and its features along with cache size. Run lscpu command and identify:

- how many cores does you CPU have?

- how much cache memory of each type are shown ?

- how many instances of each type of cache memory are shown and why?

**Practic 2 (4pt)** There are four small programs you can download from <https://github.com/CristianN> [CC/](https://github.com/CristianNCC/)[CacheExamples](https://github.com/CristianNCC/CacheExamples), each one of them are unoptimized regarding cache memory:

a) MatrixMultiplication – change the function that multiplies two large matrices A x B, to use the column from B already loaded into cache for all the spots in the result. Adjust matrices size if the execution is to slow >120sec.

b) MinMax – this is just an example in which we could use the data already loaded in cache for computing a certain operation (ex. minimum) to also compute other information about the data (like also getting the maximum value). Change the functions to compune both at the same time and see if there is a difference in the execution time.

c) Struct – the order of the members in a struct is important, this example shows how to optimize the amount of memory by reordering the members such as the padding to be minimal. Compare the amount of memory you see while running the program and check it with “top” command.

d) StructTwo – this is an example of a structure that could be used in a video game, some members like loot are less used, optimize the memory bu moving those members in a new structure which contains only these less used members. Include in the main struct only a pointer to your structure added so it could still be accessible.

**Cachegrind tool**

Install Valgrind tool (<https://valgrind.org/>) it includes several tools from which we will use Cachegrind (the cache memory profiler).

To install it on Ubuntu run: sudo apt-get install valgrind (you need to be in sudoer list).

To run a certain program under the profiler, run: valgrind tool=cachegrind ./your\_program

**Practic 3(1pt).** Run Cachegrind on the MultiplyMatrix project used for the previous exercise. Check the difference before and after your optimisations.

**Cache attacks examples**

1. Evict+Time – based on the execution time of victim’s programs, then the attacker evicts one specific cache set and measures the execution time of the victim again. If the execution time increased, the cache set was probably accessed.

2. Prime+Probe – First step, Prime, the attacker occupies specific cache sets. After the victim program has been scheduled, the Probe step is used to determine which cache sets are still occupied.

4. Flush+Reload – The attacker constantly flushes a cache line using the clflush instruction on an address. While victim’s program is running, the attacker measures the time it takes to reaccess the same address again.

5. Cache Template Attack – based on Flush and Reload, includes a profiling phase when the cache-hit ratio is stored in a cache template matrix and then the exploitation phase when events are detected from cache hits using the cache template matrix.

Some particularities of the L3 or last level cache makes it vulnerable:

* L3 cache is shared among all CPUs.
* Cache lines are shared among different processes.
* The operating system allows programs to map any other program binary or library into their own address space.
* all data which is cached within the L1 and L2 cache must also be cached in the L3 cache.

**Practic 4.(1 pt)** Download the Cache Template Attacks project (<https://github.com/IAIK/>[cache\_te mplate\_attacks](https://github.com/IAIK/cache_template_attacks)). Follow the calibration and profiling steps for “gedit” program, then check that the cahe template attack can detect an event (key pressed) done in gedit.

Bibliography:

[1] <https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-gruss.pdf>

[2] <https://github.com/IAIK/cache_template_attacks>