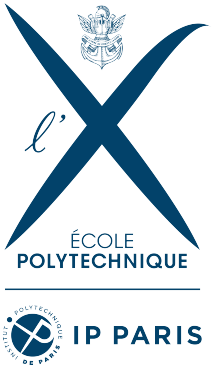


École Polytechnique

BACHELOR THESIS IN COMPUTER SCIENCE

Vulnerability Assessment for Rowhammer Attack

Author:

Bogdana Kolić, École Polytechnique

Advisor:

Maria Mushtaq, Télécom Paris

Academic year 2023/2024

Abstract

A recent discovery that repeated unprivileged accesses to the same address in modern DRAM chips can cause bit-flips in main memory has opened a path to a variety of attacks named The Rowhammer. As this security issue comes as a consequence of DRAM evolution, it has left many modern machines vulnerable. One study proposed machine learning as a mitigation technique for Rowhammer and presented neural-network models that can detect an attack based on the hardware-level events it generates. This approach gives promising results when trained and tested on data obtained by system simulation . In this thesis, we collect the data from physical hardware performance counters by using performance monitoring tools. For this purpose, we run several versions of the Rowhammer attack code and benchmarks directly on our machines to generate hardware event traces. We train and test simple classification models that can recognize our attack code with high accuracy and low overhead.

Contents

1 Introduction

Increasing computer performance had been the main goal of hardware development, but the development of Dynamic Random Access Memory (DRAM) chips had a side effect of increasing security risks. DRAM, which commonly serves as the main memory in consumer machines, aims to achieve low latency, high capacity and low cost-per-bit of memory[15][26][37]. The manufacturers' approach to this task was to increase the cell density in modern DRAM chips. Although this improved performance, it in turn made DRAM prone to *disturbance errors*, as Kim et al. named this vulnerability in their 2014 paper [26]. Moreover, the paper presented a user-level program that can cause bit-flips in modern DRAM chips and exposed the *Rowhammer* threat[1]. The subsequent studies further explained how the Rowhammer attack can be used to gain kernel privileges [44], overcome memory isolation enforced by virtualization [46] or undermine the accuracy of deep neural networks [43].

In the past decade, numerous mitigation techniques claiming to defend against the Rowhammer have been developed [24][26][30], as well as the more and more complex versions of the attack. While it has been advertised that the Target Row Refresh (TRR) mechanism implemented in DDR4 DRAM chips would ultimately put an end to Rowhammer [31], studies have shown that DDR4 chips are, in fact, still vulnerable to Rowhammer attacks [19][23][32][40]. The new mitigation method proposed by France et al. [18] uses Machine Learning models that can recognize the attack based on hardware event traces on a specific system simulated by the *gem5* simulator (for the processor and caches) and DRAM simulator *Ramulator* [28]. Their results show high accuracy, low overhead, and do not require a significant space on the silicon.

In this thesis, we perform a similar vulnerability assessment. We focus on three specific systems and omit the simulators. We run our attack code directly on the machines and read the existing hardware performance counters to obtain the hardware event traces. Two of our machines are equipped with DDR4 DRAM chips as main memory, while one of the machines utilizes DDR5. The processors on the machines are Intel Skylake, Tiger Lake and Alder Lake, respectively. All of our machines operate under Linux. We collect the data from the first two systems by using the *PAPI* library [4]. The Alder Lake processor on the third machine is currently not supported by PAPI, so we resort to *perf* [36] to access the performance counters. Our goal is to train models that could recognize different Rowhammer implementations which might pose a threat to these systems, and trigger a defense mechanism in time to prevent the attack. We show that simple models such as Logistic Regression and Decision Trees can distinguish various attack code from several benchmarks based on only four hardware event traces.

The rest of the thesis is organized as follows: the second section provides the necessary knowledge to understand the DRAM and the Rowhammer attack, as well as the Hardware Performance Counters and related tools; in the third section we describe the experimental setup for the generation of hardware event traces and gathering the data for training and testing; the fourth section explains the feature selection process; the fifth section provides insight on our machine learning methodology and results; section six is dedicated to a discussion on potential problems arising in our work and topics that should be studied further; we state the conclusions in section seven and the acknowledgments in section eight.

2 Background

2.1 DRAM and Disturbance Errors

Dynamic Random Access Memory (DRAM) is a category of volatile memory[2] commonly having the function of main memory in modern computers. On our machines, the main memory is DRAM of types DDR4 and DDR5, where DDR stands for Double Data Rate Synchronous DRAM.

[1] Although Rowhammer became popular in 2014 after being presented in [26], the vulnerability had been known in 2012 [10] [23]

[2] Contents of the memory are lost after the machine is shut down

The communication between the processor and the main memory goes through the *memory controller*. The memory controller is responsible for issuing the following commands: ACTIVATE, PRECHARGE, REFRESH, READ and WRITE. By using separate bus channels, the commands are delivered to the intended *DRAM channels*. A DRAM channel is comprised of *DRAM modules*. Several identical *DRAM chips* that operate in lockstep form a *rank*, and one or more ranks constitute a module [19]. The DRAM hierarchy is given in Figure 1.

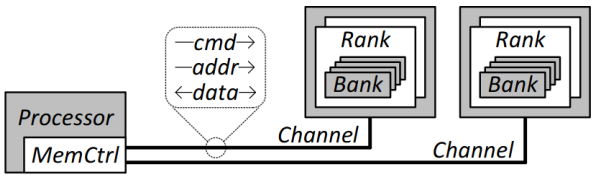


Figure 1: Logical hierarchy of DRAM [27]

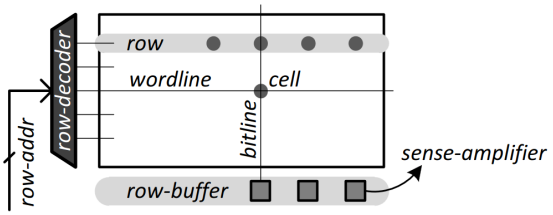


Figure 2: DRAM Bank: Logical organization [27]

Ranks are two-dimensional arrays of cells, partitioned into *banks*, which are the smallest DRAM units that can work in parallel (bank-level parallelism) [27]. Each bank has one *row-buffer*, which serves as an intermediary for accessing data from a row in the bank. Every cell stores some charge to represent one bit (logically 0 or 1). Cells in the same bank column are connected to a wire called the *bitline*, and cells in the same bank row are connected to the *wordline*. The bitlines are connected to the row-buffer. Figure 2 depicts this. Accessing data from a bank goes as follows [26]:

- The memory controller sends an ACTIVATE command to *open* a certain row - the voltage of its wordline is raised and as a result the cells from the row connect to their bitlines; finally, their charges are transferred into the row-buffer
- A specific column is accessed in the row-buffer, as given by the received READ/WRITE command
- In order to activate a different row, the currently opened row must be *closed* - the PRECHARGE command ensures that the voltage on the row's wordline is lowered and that the row-buffer is cleared; this process completely restores the charge in the row's cells

Since the cells lose their charge over time, the charge is periodically restored. The REFRESH command is responsible for opening and closing multiple rows at once, which preserves the data stored in the cells. Numerous REFRESH commands are issued during one fixed *refresh interval*, so that all of the rows are refreshed in that time period [26]. Kim et al. [26] performed a large study on DDR3 chips and found that repeated toggling of one row's wordline can cause *disturbance errors* in nearby DRAM cells - some cells lose their charge faster and refreshing cannot restore it. We observe this behavior as bit-flips in those cells.

2.2 The Rowhammer

The goal of the attack is to induce bit-flips in the main memory. The attacker's method, therefore, is to open and close one row from a DRAM bank sufficiently many times between two refreshes. This row is called the *aggressor row* and the procedure *hammering*. The rows that are affected by the attack are called *victim rows*. The basic C code for performing the hammering is given in Figure 3.

Thanks to the unprivileged CLFLUSH command [22] , the attacker can easily invalidate a specific cache line and access data directly from the DRAM. Nevertheless, accessing data from the same address in a loop does not guarantee that one row will be opened and closed repeatedly [26]. Accessing data from a row when it is already in the row-buffer is much less costly than closing the currently opened row and then opening the desired one. When the memory controller receives consecutive read/write requests to the same row it optimizes the access time by issuing multiple READ/WRITE commands without closing the row. Hence, as we see in

```
/* Attack loop */
for(int i = 0; i < toggles; i++){
    z = *x; //read(x)
    z = *y; //read(y)
    asm volatile("clflush (%0)" : : "r" (x) : "memory");
    asm volatile("clflush (%0)" : : "r" (y) : "memory");
}
```

Figure 3: Basic Rowhammer loop (simplified code from [41])

Figure 3, the attacker forces closing the row by accessing two rows from the same bank one after the other[3]. This, however, requires bypassing memory translation and making sure that the two rows belong to the same bank. Some solutions rely on obtaining the memory mapping from *proc/self/pagemap* and timing analysis [40], while Seaborn and Dullien [44] suggest a probabilistic approach and hammering multiple rows to increase the chances of having two aggressor rows in the same bank.

One way of optimizing the attack is to increase the number of useful activations by choosing two aggressor rows which are adjacent to the same victim row. This version of the attack is called the *double-sided rowhammer*. In some environments, the attacker cannot use the CLFLUSH command. Qiao and Seaborn [42] propose a version of the attack without the CLFLUSH instruction, based on non-temporal store instructions such as MOVNTI and MOVNTDQ.

2.3 Hardware Performance Counters and Tools

Hardware Performance Counters (HPCs) are special-purpose registers in modern processors used to count the number of different event occurrences. Their number varies according to the processor type and it imposes an upper bound on the number of events that can be measured simultaneously. For example, one could use these counters to obtain the number of cache hits and misses, processor cycles, the number of instructions, etc. related to some process. Events that can be counted depend on the processor architecture and the events specific to a processor are called *native events* [11] [34]. In our work, we make use of two tools to access the performance counters: the *PAPI* library [4] and the *perf tool* [36].

2.3.1 The PAPI Library

Performance API (PAPI) is a cross-platform interface that allows instrumenting C and Fortran code to accurately measure its performance. The native events can be counted using PAPI's low-level API. That entails defining an *EventSet* and specifying the events of interest. The count begins with a call to the *PAPI_start* function, and the counters can be stopped by calling the *PAPI_stop* function. Furthermore, this function can be used for obtaining the counters' values. The functions *PAPI_read* and *PAPI_accum* store the counters' values into a provided array without stopping them. *PAPI_accum* adds the values to the array and resets the counters, while *PAPI_read* copies the values into the array without modifying the counters. Function *PAPI_reset* resets the counters.

PAPI offers *multiplexing* support - the possibility to count more events than there are dedicated registers in the processor. The concept is based on subdividing the usage of the physical counters over time among the

[3]From [27] and [29] we see that DRAM banks are actually formed by similarly structured subarrays, where the subarrays each have their own local row-buffer and share a global row-buffer in the bank. But, as every row needs to be closed before another row in the same bank can be opened [27], we conclude that it is indeed enough to take two aggressor rows from the same bank, regardless of their subarray.

```c
#include <stdlib.h>
#include <stdio.h>
#include <papi.h>

int main(){
    int retval, EventSet = PAPI_NULL;
    unsigned int native = 0x0;
    long_long values[1];

    /* Initialize the library */
    PAPI_library_init(PAPI_VER_CURRENT);

    /* Create an EventSet */
    PAPI_create_eventset(&EventSet);

    /* Find the code for the native event */
    PAPI_event_name_to_code("CACHE-MISSES", &native);

    /* Add it to the eventset */
    PAPI_add_event(EventSet, native);

    /* Start counting */
    PAPI_start(EventSet);

    // The code that generates the traces

    /* Read the counters */
    PAPI_read(EventSet, values);

    /* Stop the counters */
    PAPI_stop(EventSet, values);
}
```

Figure 4: Code instrumentation with PAPI

events being counted. As a result, the final counts obtained represent an approximation of the actual number of event occurrences. Two functions are needed in order to use multiplexing, *PAPI_multiplex_init* and *PAPI_set_multiplex*. The latter is called once for each *EventSet* that should be multiplexed.

In multi-threaded programs, PAPI can be configured to count only the events belonging a specific thread, by the means of *PAPI_thread_init* and *PAPI_attach* function calls [5] [11] [38].

An example of using low-level PAPI interface to measure the performance of C-language code is given in Figure 4. The inclusion of the *papi.h* header is necessary to use the functions from the library. Before using any other PAPI function, a call to *PAPI_library_init* must be made to initialize the library. Afterwards, an *EventSet* is created and the native event "CACHE-MISSES" is added to it. To be able to add the event using the function *PAPI_add_event*, we must first call *PAPI_event_name_to_code* to get the platform-specific code of the event. Accessing the counters to obtain the number of cache misses is later regulated with the aforementioned *PAPI_start*, *PAPI_read* and *PAPI_stop* functions.

PAPI source code [9] comes with a file *papi_native_avail* which can be run to list available native events on the given platform. Running *all_native_events* further tests whether these events are supported when various flags are provided. All of the functions mentioned return an integer value that can be used for error handling in the code.

2.3.2 Perf

The perf tool [36] is a part of the Linux kernel and its purpose is using performance counters to profile programs. It offers some predefined events that can be used to set up the performance monitoring counters, and the list of those events can be obtained with the *perf list* command. In addition, all events available on the architecture can be referenced by their raw encoding. For example, on Alder Lake processors, the event INST_RETIRED.ANY is encoded as *cpu_core/event=0xc0,umask=0x00/* [7] and it can be used to count the number of retired[4] instructions. Moreover, we can specify that we want to count the events at user level by adding the flag *u* : *cpu_core/event=0xc0,umask=0x00/u*. For our purposes, we use perf with *stat* option, which runs a command and gathers performance counter statistics [6][20].

3 Data Generation Setup

3.1 Attack and No-Attack Programs

We use four different programs to simulate the attack and another three to represent the no-attack behavior. All of the code used in this project can be found in [2]. The basic attack code is a slightly modified version of the code obtained from the authors of [18]. The code's function *attack* calls the C function *asm* to create a loop for loading values from two addresses belonging to the aggressor rows, and then removing them from cache by using CLFLUSH. The code for the second version of the attack (using non-temporal store instructions) and two no-attack functions are derived from it by altering the *attack* function. One no-attack code simply removes the CLFLUSH instruction from the loop, and the second makes random accesses on the pre-initialized buffer from which the aggressor rows are chosen in the attack code. The different *attack* (*no_attack*) functions are presented in Figures 5 and 6.

The third no-attack code is the *STREAM benchmark* [35], used to test memory performance. The last two attack programs are the *double-sided rowhammer* and a simplified *rowhammer-test* from Google's Project Zero *rowhammer-test* GitHub repository [41]. Our code keeps the memory initialization from the *rowhammer-test* and the hammering function, but it changes the number of addresses that are hammered in the attack for the sake of diversification of generated data samples. To address the existence of the *Many-sided RowHammer attack* [19], we adjust the code to take four, eight, nine, ten and twenty aggressor addresses.

[4]Instruction are retired only if they are completely executed - if they were truly needed for the program execution flow [21]

```
asm volatile(
    "MOV %[nb_loops], %%ecx  \n"
    "JMP hammer_loop_asm     \n"
    "hammer_loop_asm:        \n"
    "  MOV (%[addr1]), %%edx \n"
    "  MOV (%[addr2]), %%edx \n"
    "  CLFLUSH (%[addr1])    \n"
    "  CLFLUSH (%[addr2])    \n"
    "  LOOP hammer_loop_asm  \n"
    :: [nb_loops] "r" (nb_loops),
 [addr1] "r" (addr1),
     [addr2] "r" (addr2)
    : "%ecx", "%edx", "memory"
    );
```

(a) Basic attack loop

```
asm volatile(
    "MOV %[nb_loops], %%ecx  \n"
    "JMP hammer_loop_asm     \n"
    "hammer_loop_asm:        \n"
    "  MOVNTI %%eax, (%[addr1]) \n"
    "  MOVNTI %%eax, (%[addr2]) \n"
    "  MOV (%[addr1]), %%edx \n"
    "  MOV (%[addr2]), %%edx \n"
    "  LOOP hammer_loop_asm  \n"
    :: [nb_loops] "r" (nb_loops),
 [addr1] "r" (addr1),
     [addr2] "r" (addr2)
    : "%eax", "%ecx", "%edx", "memory"
    );
```

(b) Attack loop using non-temporal store instructions

Figure 5: Basic attack code and the non-temporal store derivation

```
asm volatile(
    "MOV %[nb_loops], %%ecx  \n"
    "JMP no_hammer_loop_asm     \n"
    "no_hammer_loop_asm:        \n"
    "  MOV (%[addr1]), %%edx \n"
    "  MOV (%[addr2]), %%edx \n"
    "  LOOP no_hammer_loop_asm  \n"
    :: [nb_loops] "r" (nb_loops),
 [addr1] "r" (addr1),
     [addr2] "r" (addr2)
    : "%ecx", "%edx", "memory"
    );
```

(a) Loop without CLFLUSH (no-attack code)

```
long long unsigned index;
uintptr_t addr;
for(int i=0;i<nb_loops;i++){
index = rand() % 163840;
addr = (uintptr_t) (&buffer[index]);
asm volatile(
    "  MOV (%[addr]), %%edx \n"
    ::
     [addr] "r" (addr)
    : "%edx", "memory"
    );
}
```

(b) Random accesses on the buffer (no-attack code)

Figure 6: No-attack loops derived from the basic attack code

3.2 Three Different Environments

We perform our work on three different machines running Linux. The first machine is VivoBook_ASUSLaptop X521EQ_S533EQ, with an 8GB DDR4 main memory and an 11th Gen Intel© Core™ i5-1135G7 @ 2.40GHz × 4 - Tiger Lake processor. It has a Linux Mint 21.2 Cinnamon 5.8.4 installed, and the Linux kernel is 5.15.0-91-generic. The second machine is LENOVO ThinkPad T470s W10DG, with 8GB DDR4 memory and Intel© Core™ i5-6300U CPU @ 2.40GHz × 2 - Skylake processor. It is running Linux Mint 21.1 Cinnamon 5.6.5, kernel version 5.15.0-56-generic. The third machine is DELL Precision 3571 with 16GB DDR5 main memory and a 12th Gen Intel© Core™ i7-12700H × 14 - Alder Lake processor. It also runs Linux Mint 21.2 Cinnamon 5.8.4, kernel version 5.15.0-76-generic . All of the machines have three levels of cache.

We obtain our data by running the programs in different load conditions [18]. For NO LOAD conditions, we run the programs as the only process on the system. We introduce LOAD by executing instances of the STREAM benchmark in parallel with the program of interest. Specifically, we run two STREAM benchamrks on LENOVO machine, four on ASUS and eight instances of STREAM on our DELL machine to create LOAD.

3.3 Sampling with PAPI

We wish to periodically read the counters while our code (attack or no-attack) is running. PAPI has a function for sampling based on event overflow. However, we are interested in measurements after equal time intervals, thus we create our own program for sampling. A timer is configured to send a signal that interrupts the program periodically after 1ms and we read the counters while handling the interrupt. This might interfere with our event count [45], hence we place the code whose traces we are trying to obtain in a separate thread and PAPI is instructed to measure only the events from that thread. In order to get as precise measurements as possible, we use signal blocking to make sure that the interrupts are handled from the main thread.

We try to test the accuracy of our approach in the following manner: for the same code, we compare the total event count when the program is periodically sampled and when it is not. We take our seven different programs[5], and we derive two PAPI-instrumented programs from each of them: one to be sampled at every millisecond as described above and to compute the total number of event occurrences, while the other simply computes the total number of the same event occurrences. We want to measure how much the total event counts differ from each other and we compute the relative error using the formula

$$\frac{abs(sampled\_program\_count - benchmark\_program\_count)}{benchmark\_program\_count} * 100\%$$

We run both versions of each program, one after the other, fifty times without LOAD and fifty times with LOAD on the system. We do this on both ASUS and LENOVO machine to obtain a total of 1400 samples of relative errors. We are interested in the events whose counts are not likely to change much between two consecutive runs of the same program: the number of instructions and cache accesses. Hence, on our machines, we show the results for the events "INST_RETIRED"[6], "MEM_INST_RETIRED:ANY", and "PERF_COUNT_HW_CACHE_L1D:ACCESS". The relative errors that we compute for each of the three events are given in figures 7, 8 and 9, and we present the average, minimal and maximal error in Table 1. For our testing purposes, we treat all of the samples equally - we make no distinction based on the code tested, the machine used, or the load conditions.

Event counted	Average error (%)	Minimum error (%)	Maximum error (%)
Instructions	0.005	0	0.282
Memory instructions	0.012	0	1.105
L1 data cache accesses	0.014	0	1.104

Table 1: Change in the total event count for the sampled program against the benchmark

3.4 Sampling with Perf

In order to obtain periodic measurements with perf, it is enough to use one command:

perf stat -e 'events_to_count' -I 'sampling_interval' -o 'output_file' ./program

This will produce a file *output_file* with periodic readings of specified counters in *sampling_interval* millisecond intervals. Minimal interval length is 1ms. Events that are counted can be represented by their raw encoding found at [7]. The result are the counts generated by running the specified *program*.

[5] rowhammer-test code is the one with four aggressor rows

[6] Tiger Lake processor on the ASUS machine allows us to use the event "INST_RETIRED:ANY" which is counted by a designated fixed counter, freeing up programmable counters to count other events. However, on the LENOVO machine with a Skylake processor, we have to use a programmable counter, hence the event "INST_RETIRED:ANY_P" [7]

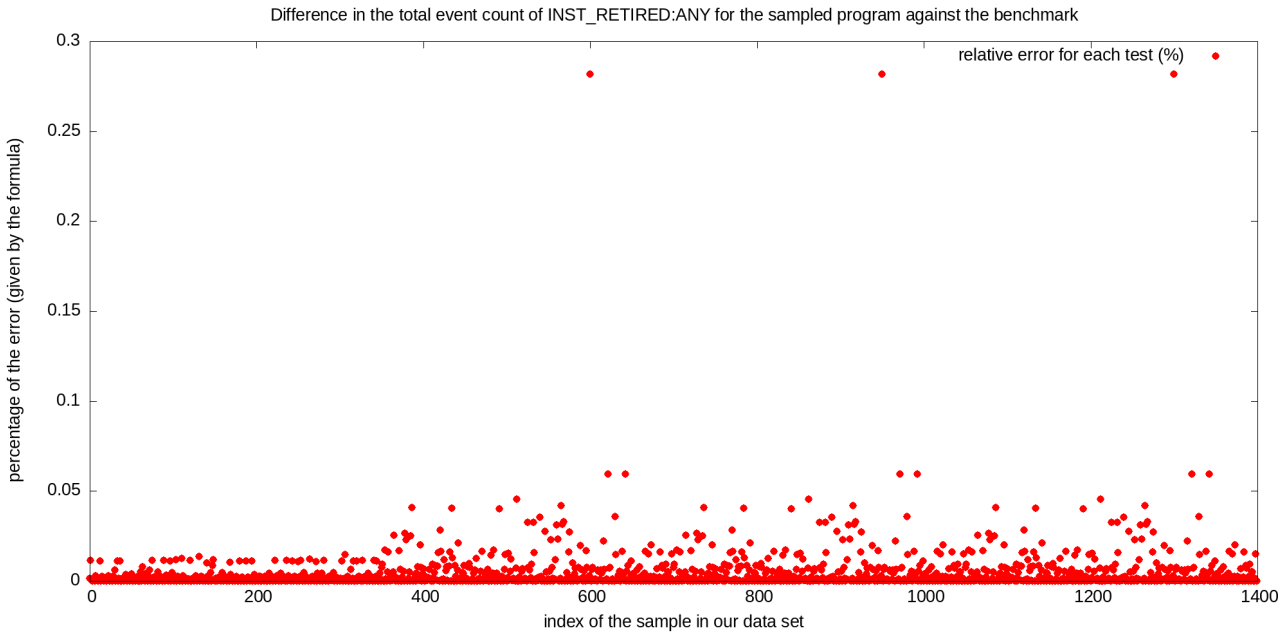


Figure 7: Error in the instruction count

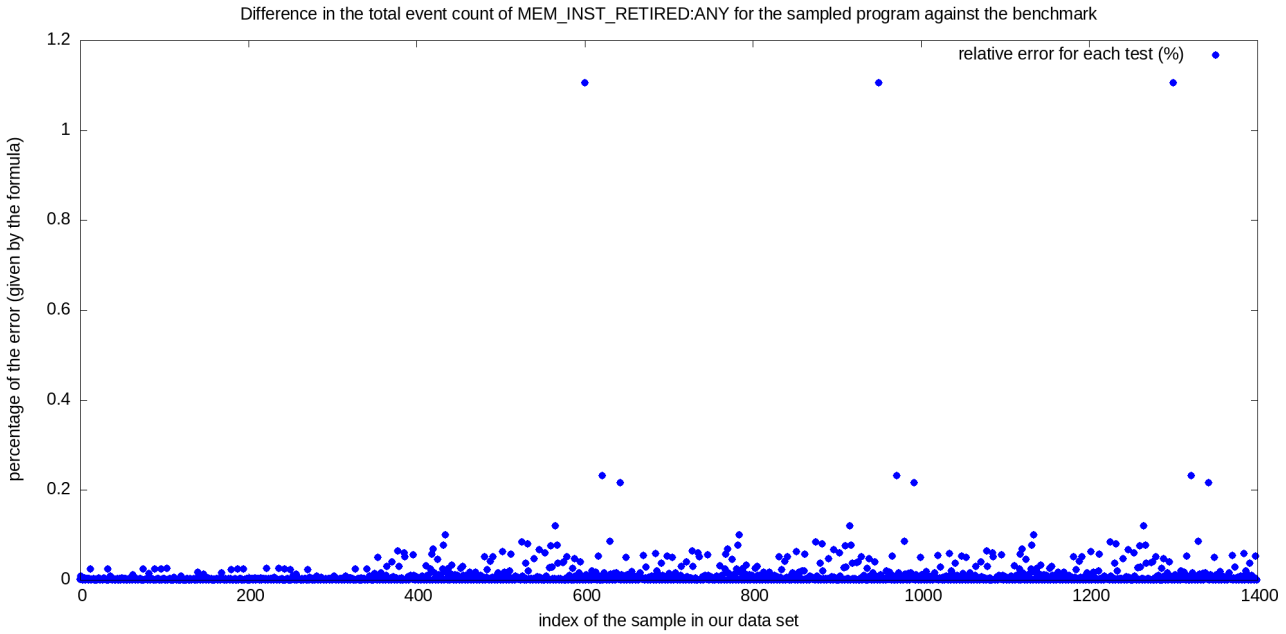


Figure 8: Error in the memory instruction count

4 Feature Selection

Based on findings from [18], we conjecture that observing cache events could prove useful. The essence of the attack is activating a DRAM row, hence it is characterized by a high number of last-level cache misses. Since we are constantly accessing DRAM, and moreover trying to cause bank conflicts, we also expect a high number of stalled cycles caused by the attack program [27]. France et al. [18] suggest looking at the number of last-level cache (LLC) hits, as they observe a significantly larger number of LLC hits produced by the no-attack

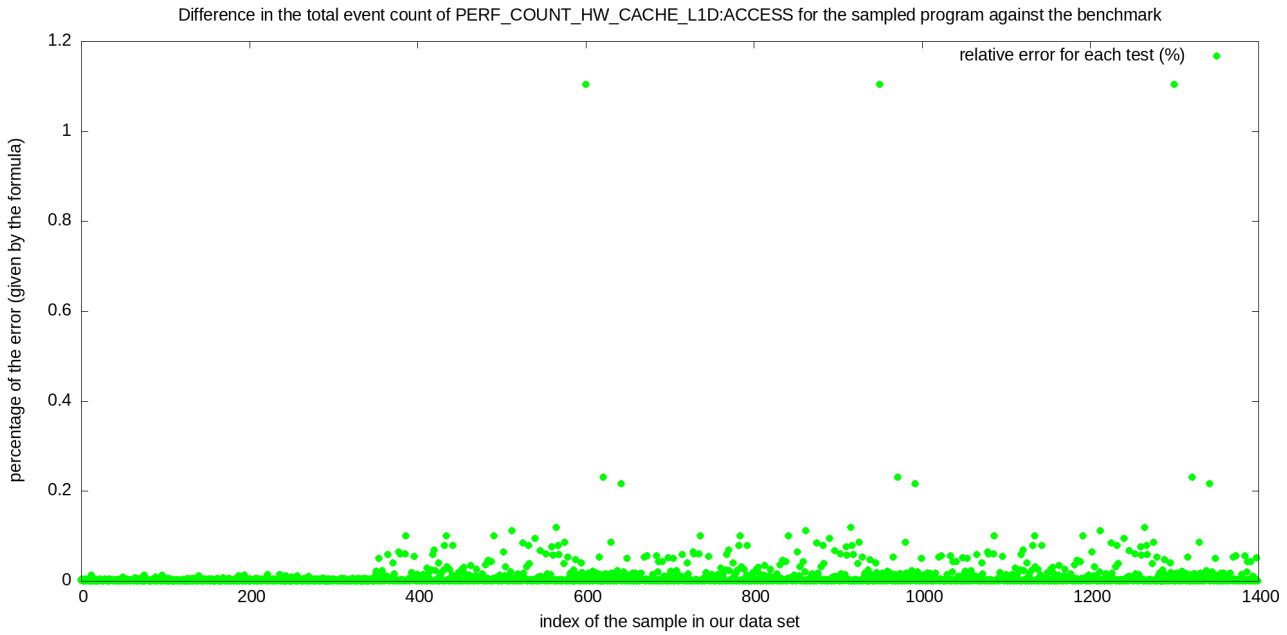


Figure 9: Error in the L1 data cache access count

code compared to their attack code. However, their simulated system has only two levels of cache, whereas our machines have three cache levels of larger size. As a consequence, we do not expect a large number of cache hits at the last level, but rather in level-1 (L1) and level-2 (L2) cache[7]. Thus, we choose to count L1 and L2 cache hits. To be more specific, L1 cache is divided into data and instruction cache. In this thesis we count the number of L1 data cache load hits and L2 cache load hits.

The features we select to train our models stem from the hardware events that we gather from the performance counters on our machines. Therefore, the selection highly depends on the availability of the events on the system architectures we work with. On ASUS and LENOVO laptops we use the PAPI library, and the native events we count are given in the second column of Table 2. The equivalent events for the Alder Lake architecture found on our DELL machine are taken from [7] and placed in the third column.

Event	PAPI native event name	Alder Lake event
LLC (L3 cache) misses	ix86arch::LLC_MISSES	LONGEST_LAT_CACHE.MISS
L2 cache hits	L2_RQSTS:DEMAND_DATA_RD_HIT	L2_RQSTS.DEMAND_DATA_RD_HIT
L1 data cache hits	MEM_LOAD_RETIRED:L1_HIT	MEM_LOAD_RETIRED.L1_HIT
Stalled cycles	UOPS_RETIRED:STALL_CYCLES	UOPS_RETIRED.STALLS

Table 2: Event selection on different machines

[7] In the cache hierarchy, L1 cache is the closest to the processor, while LLC is the cache closest to the main memory. The processor first requests the data from the lower cache levels and sends further requests to other cache levels only if a cache miss occurs.

5 Machine Learning

5.1 Data Sets

We use the programs introduced in section 3 and run them on all three machines in both NO LOAD and LOAD conditions to generate the data. On ASUS and LENOVO machines, the data is sampled using the PAPI library and by using the perf tool on our DELL machine. One sample represents the event count in a 1ms interval. In total, we have around 3.2 million samples on LENOVO machine, 2.7 million samples on DELL and 4.7 million samples on ASUS machine. Each sample contains nine fields: machine from which the sample is obtained, load conditions, code that generated it, timestamp, the four features we wish to observe, and the label specifying whether the code represents an attack or not.

We plot the data from our LENOVO machine in both NO LOAD and LOAD conditions on figures 10, 11, 12 and 13. The data from the other two machines is shown in the appendix A. On the graphs, we make a distinction between the traces generated by the attack code, our custom no-attack code and the STREAM benchmark (which is another variant of the no-attack code). As expected, in NO LOAD conditions, the attack code produces more cache misses, less L1 and L2 cache hits, and has more stalled cycles than no-attack code. A portion of samples generated by the attack code has a high number of L2 cache hits, however, it can still be distinguished from the no-attack code according to its three other traces. The STREAM benchmark is an exception when we observe the number of last-level cache misses, but it can also easily be classified as no-attack code by its other traces. While in NO LOAD conditions it seems that separating the attack and no-attack traces is a simple task, introducing LOAD turns this into a challenge. We rely on machine learning models to perform the classification.

5.2 Models

We choose to study three different models, adjusting their parameters according to the data set we are using: Logistic Regression [8], a Decision Tree Classifier [1], and a Category Boosting (CatBoost) Classifier [3]. We implement *LogisticRegression* and *DecisionTreeClassifier* models using the *scikit-learn API* [14][39] .

Logistic Regression is a model commonly used for binary classification. The model computes the probability that the linear combination of the input features belongs to a certain category using a logistic function. Hence, the model only needs to store the weights used for the linear transformation (depends on the number of features) and the prediction consists of simple computations [13] [33].

Decision Trees are binary trees where the leaves yield a class, and the other nodes partition the input space. The prediction is obtained by traversing the tree from the root to a leaf, evaluating the input based on conditions given at the nodes. The complexity of the model grows with the depth of the tree [12] [16] .

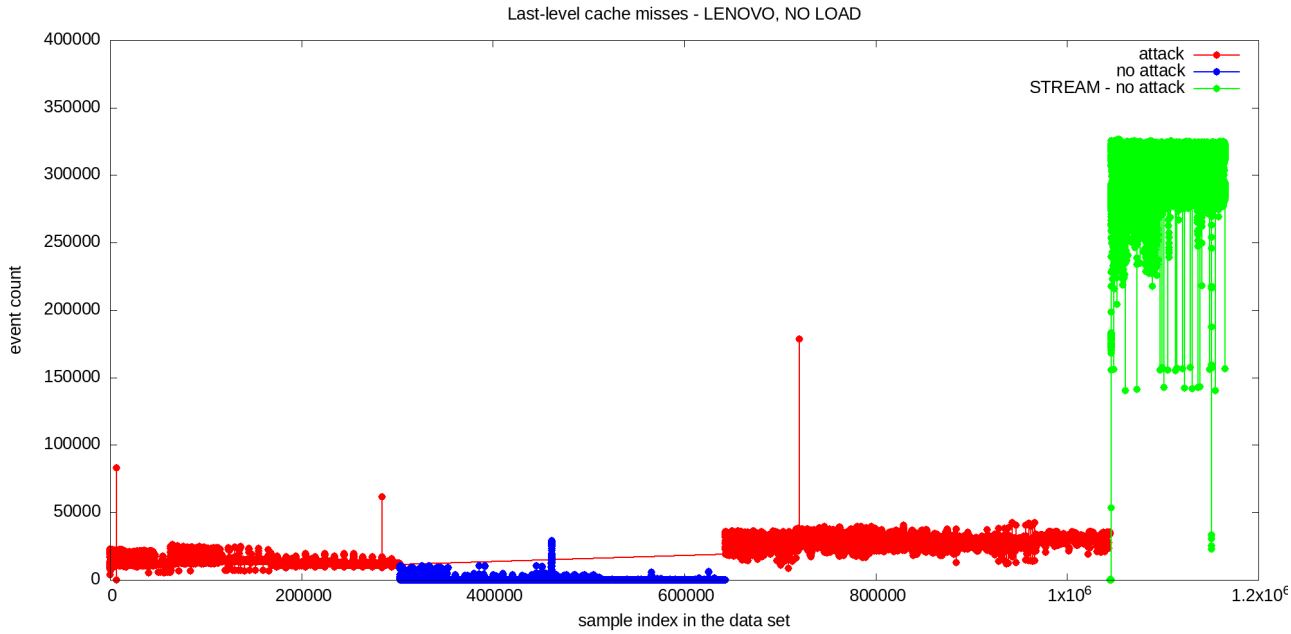
Boosting is a technique where multiple *weak learners* (usually decision trees with a small number of nodes) are combined to create a strong machine learning model. **CatBoost** is a new *Gradient Boosting* model that appears to outperform its predecessors [17].

5.3 Training and Testing

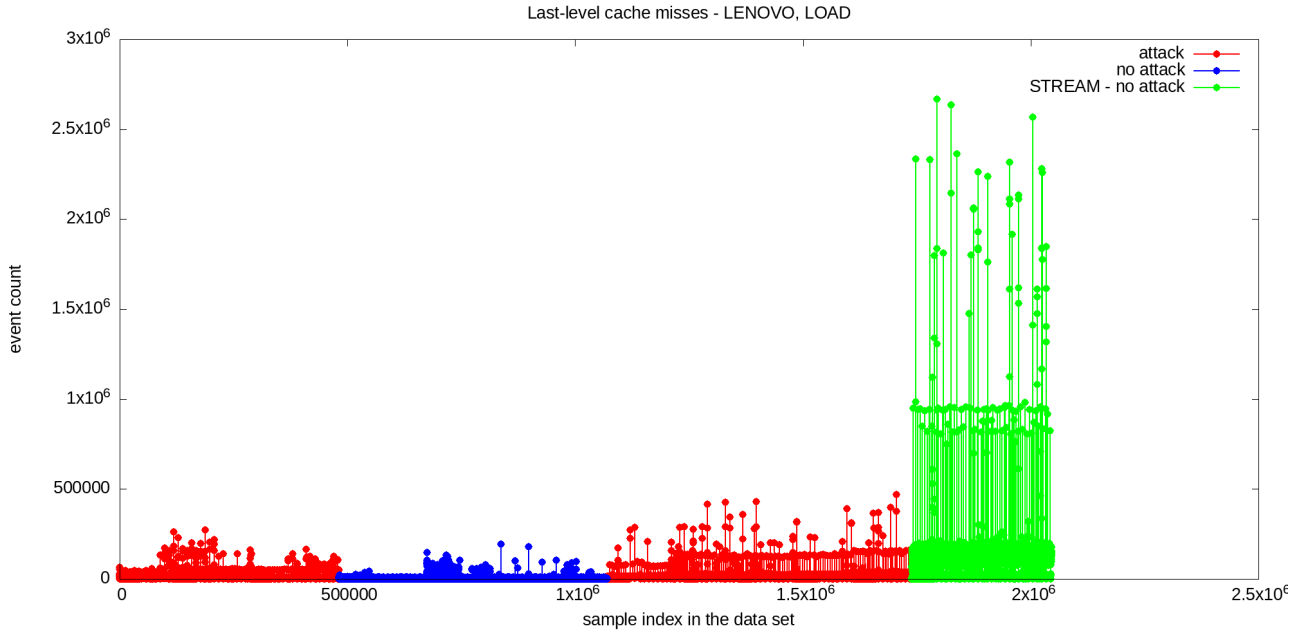
We stick to the four chosen hardware event features for training and testing the models. We do not use load conditions as a feature, but we do create models that are specific to a machine. We randomly split each data set into a training and testing data set of approximately the same size. Models are trained on the training data set, and the results we present are obtained based on the testing data set.

5.4 Results

In tables 3, 4 and 5 we present the performance of the three models on each machine. All of our tested models give satisfactory results. They all have accuracy above 99.3%. Furthermore, the best model on each machine

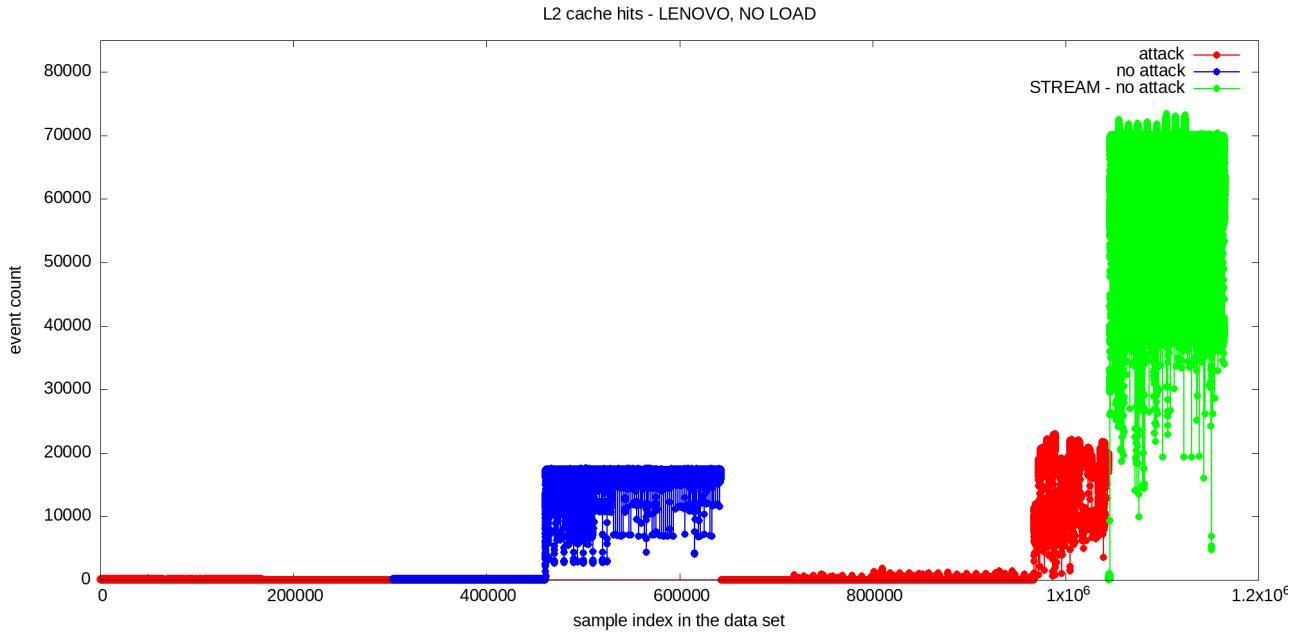


(a) NO LOAD

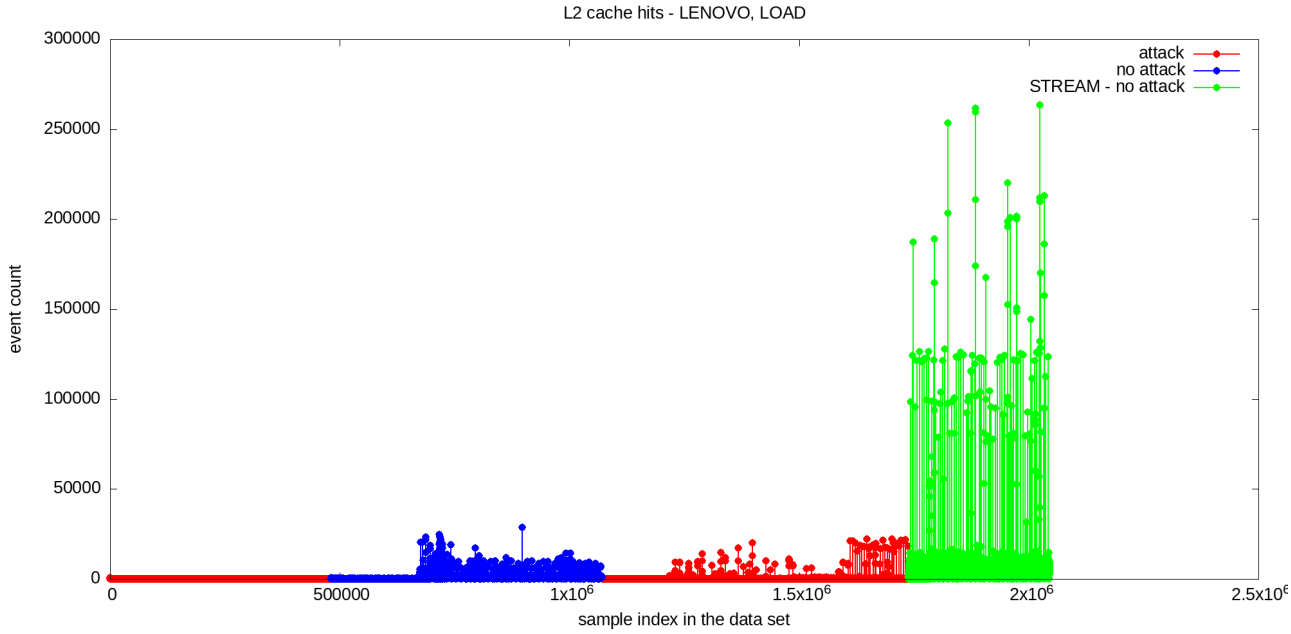


(b) LOAD

Figure 10: Last-level (L3) cache misses on LENOVO machine

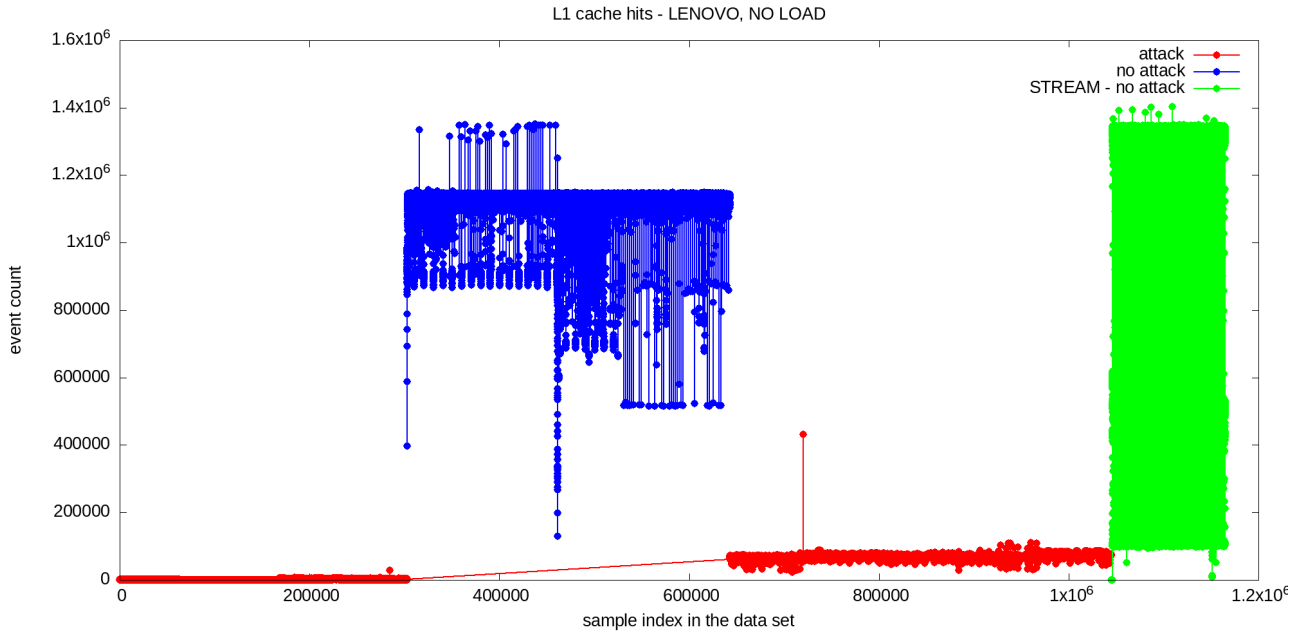


(a) NO LOAD

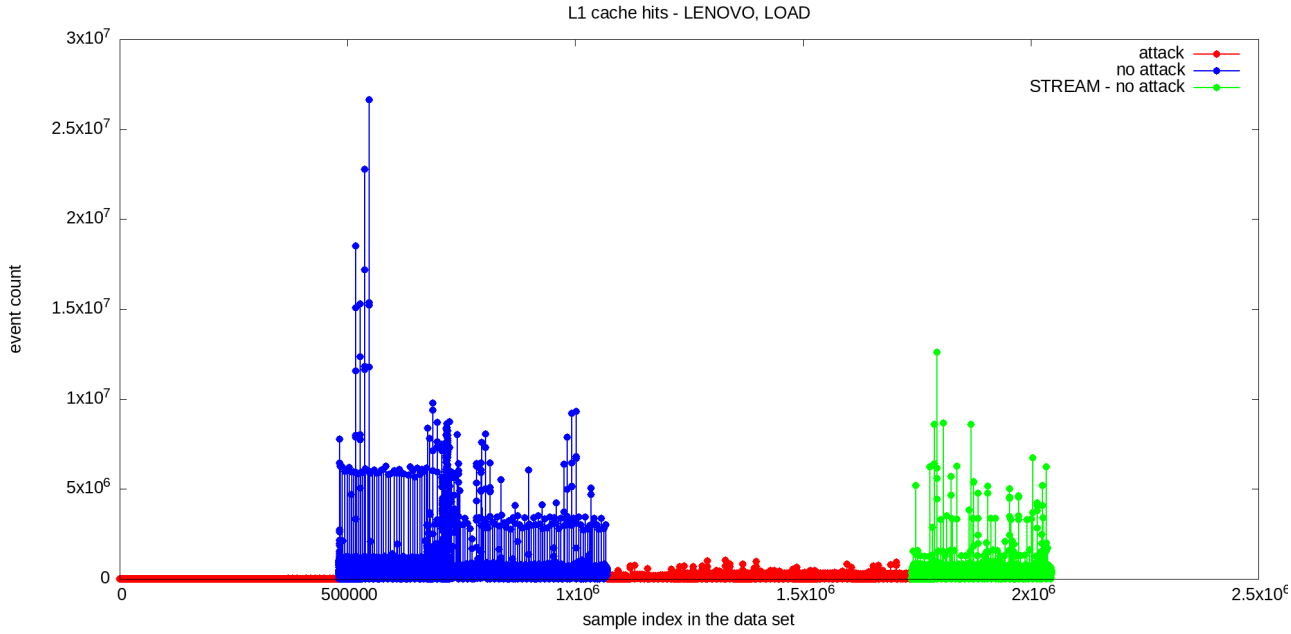


(b) LOAD

Figure 11: L2 cache hits on LENOVO machine

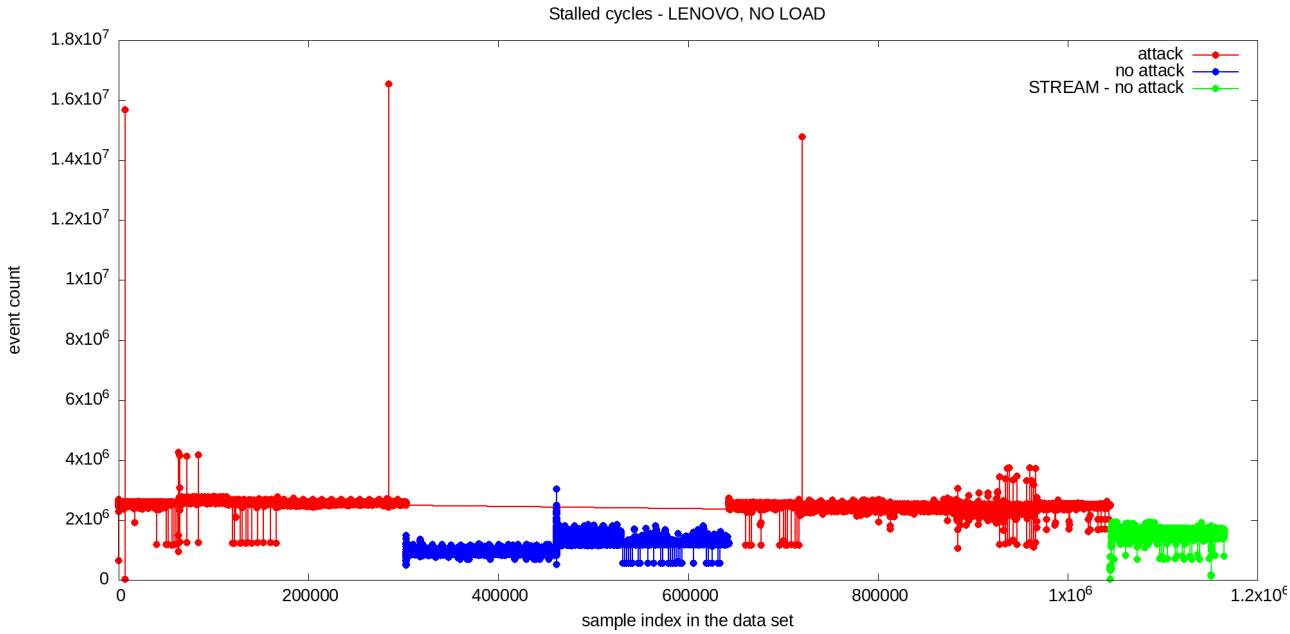


(a) NO LOAD

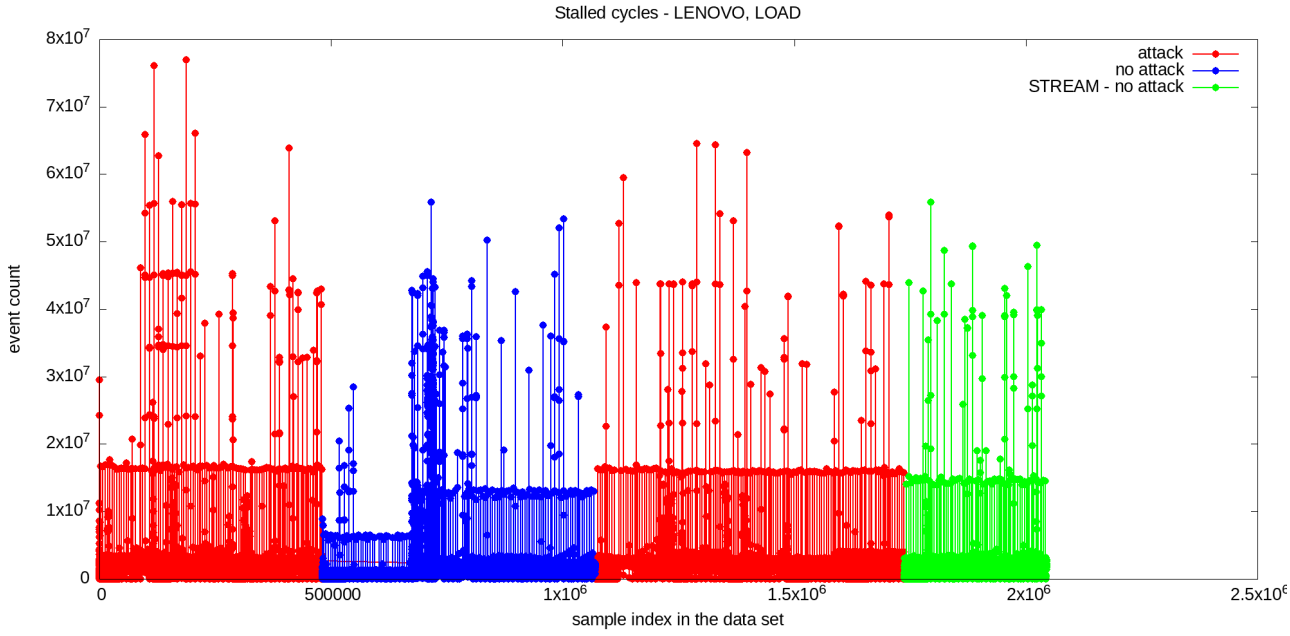


(b) LOAD

Figure 12: L1 cache hits on LENOVO machine



(a) NO LOAD



(b) LOAD

Figure 13: Stalled cycles on LENOVO machine

has accuracy above 99.8%. France et al. [18] achieved this result with neural networks classifying multiple traces from a time window, while we are able to obtain it with decision trees and boosting, classifying traces at one time point. The models are significantly faster in software, performing a single classification on the samples from the test set under 100 nanoseconds on average. It is worth mentioning that, although the models are faster than the ones presented by France et al. [18], we are still performing a classification of traces representing a 1ms interval, while their work uses 10μs time windows.

Model	Accuracy (%)	False positives (%)	False negatives (%)	Average classification time (ns)
Logistic Regression	99.733	0.077	0.190	7.995
Decision Tree	99.803	0.005	0.192	30.088
CatBoost	99.784	0.012	0.204	64.516

Table 3: Performance of the chosen ML models on LENOVO machine

Model	Accuracy (%)	False positives (%)	False negatives (%)	Average classification time (ns)
Logistic Regression	99.312	0.679	0.009	15.445
Decision Tree	99.952	0.042	0.006	27.294
CatBoost	99.986	0.012	0.002	66.614

Table 4: Performance of the chosen ML models on ASUS machine

Model	Accuracy (%)	False positives (%)	False negatives (%)	Average classification time (ns)
Logistic Regression	99.899	0.067	0.034	13.604
Decision Tree	99.986	0.013	0.001	26.827
CatBoost	99.908	0.078	0.014	69.030

Table 5: Performance of the chosen ML models on DELL machine

Nevertheless, our models can still detect Rowhammer before it executes in most cases. The lowest number of activations needed to cause bit flips on DDR4 is reported to be ten thousand per aggressor row [25]. We time the code used to generate the samples on all three machines with 10000 toggles[8] and observe that it should take more than 1ms to complete. On DELL machine, the attack takes at least 4ms and on LENOVO, 1.111 ms. With classification time under 100ns, and assuming the correct prediction, our models are able to detect the attack before it executes. On our ASUS machine, the portion of the double-sided rowhammer code performing only 10 000 toggles takes 0.998ms, but all other programs take at least 1.053ms to complete.

All of the models could easily be integrated in hardware: the decision tree model on LENOVO machine has depth five, and the one on DELL has depth four. The CatBoost model on ASUS is made of fifteen decision tree estimators, where each estimator has maximum depth three. We still note that the less complex decision tree model on ASUS, with maximum depth five, also achieves high accuracy of 99.95%.

6 Discussion

Our methodology encounters several challenges:

[8]We perform 10000 loop iterations for the non-temporal stores attack, but we do not have a way of confirming how many row activations occur at each loop iteration(more or less than 1) due to *write-combining buffers* [42]

- **Sampling frequency and overhead trade-off:** We want a small interval for sampling, so that we are able to detect the attack before it completes even when some traces are classified as false negatives. However, the overhead increases with the sampling frequency. By closely examining the timestamps of our samples in the data set, we can already observe that the counters might not be read precisely after each millisecond, but after a millisecond and some noise. Furthermore, perf imposes a lower bound of 1ms for the sample interval length. There is no restriction coming from our custom PAPI programs, and the entire procedure (sampling accuracy, data generation and model training) with smaller intervals should be further studied on ASUS and LENOVO machines.
- **Sampling accuracy and the number of features trade-off:** We know that multiplexing is a way of counting more events than there are counters on the machine. But, multiplexing requires the program to run for a certain amount of time to give good approximation of its traces. One measure that we take in order to avoid multiplexing and obtain more accurate event count is counting only four events. This should allow for sampling with small enough intervals, as no statistical approximation is needed to get the event traces.
- **Correctly labeling the traces in the data set:** With PAPI we are able to instrument the code and we try to count the events coming only from the hammering functions. Nevertheless, in many of the programs this includes the traces of memory initialization and aggressor rows selection, which do not necessarily indicate an attack. This issue is amplified with the use of perf, as it counts the events generated from the entire sampled program.
- **Good representation of the problem:** We aim to generalize the traces by using several implementations of the attack and benchmarks code, but we cannot claim with certainty that our generated traces represent all Rowhammer variants. Future work could study how changing the parameters of the attack (for example the number of toggles) alters its traces. Moreover, there are numerous instances of no-attack code making it impossible to take all of them into account.
- **Appropriate training data set:** The attack traces represent 69% and 58% of the samples on our ASUS and LENOVO machines, respectively. However, they have the majority of 79% on the DELL machine, which might affect the models' accuracy.
- **Implementation in hardware:** The models themselves might be simple enough to be implemented in hardware, but the question of how many processes can we monitor simultaneously with existing performance counters remains. We might need to add more performance counters to be able to implement the entire detection mechanism.
- **Will the models classify the attack before it causes bit flips?** On our DELL machine, we work with traces obtained from the entire program, and not just the hammering function. When timing the attack we take this into account and time the entire program execution. This might lead to false conclusions as we are really only interested in the time it takes to classify the hammering traces, and the accuracy of classifying those traces. In addition, we time the attack which performs ten thousand activations as that is the lowest number of activations reported to cause bit flips on DDR4. Rowhammer on DDR5 chips is yet to be studied, and this number might differ. There are also potential problems with properly timing the non-temporal stores attack.

7 Conclusion

Rowhammer attack is a security issue which keeps evolving to adapt to current defense mechanisms. A new detection method which is both easy to integrate in hardware and does not incur a large system overhead uses

neural networks to detect the attack and trigger mitigation. In our work, we continue studying machine learning detection mechanisms on data sets consisting of event traces obtained from hardware performance counters. We explore methods for precise data collection, look for useful hardware-event features and show that fast and simple to implement decision tree models can recognize various attack code based on its hardware event traces. Using only four features - the traces obtained directly from hardware performance counters, our models appear to be highly accurate classifiers. The accuracy of the Decision Tree model on our LENOVO machine is 99.8%, while the CatBoost model on ASUS and the Decision Tree model on DELL machine achieve the accuracy above 99.9%. More importantly, their performance in software is sufficiently fast (less than 100ns) and the time for attack detection primarily depends on the interval size used for the sampling. In most cases when the classification is correct, they can recognize the attack behavior before it successfully completes its execution. Future research should further study the models trained on samples with smaller interval sizes.

8 Acknowledgments

I would like to express my utmost gratitude towards my supervisor, Dr. Maria Mushtaq, for all the advice and guidance she has given me, as well as the opportunity to work on this project. I would also like to thank T. Dullien, D. Gruss, H.J. Boss, L. France and D. Novo for their assistance in understanding and implementing the Rowhammer attack on DDR4.

9 References

[1] 1.10. Decision Trees — scikit-learn.org. https://scikit-learn.org/stable/modules/tree.html#classification. [Accessed 12-03-2024].

[2] GitHub - bogdanaKolic/rowhammer-vulnerability-assessment: This repository contains all the code and data related to my Bachelor Thesis: Vulnerability Assessment for Rowhammer Attack. — github.com. https://github.com/bogdanaKolic/rowhammer-vulnerability-assessment. [Accessed 17-03-2024].

[3] GitHub - catboost/catboost: A fast, scalable, high performance Gradient Boosting on Decision Trees library, used for ranking, classification, regression and other machine learning tasks for Python, R, Java, C++. Supports computation on CPU and GPU. — github.com. https://github.com/catboost/catboost. [Accessed 12-03-2024].

[4] PAPI official website. https://icl.utk.edu/papi/. [Accessed 09-02-2024].

[5] PAPI wiki page. https://bitbucket.org/icl/papi/wiki/Home. [Accessed 09-02-2024].

[6] perf-stat(1) - Linux manual page — man7.org. https://man7.org/linux/man-pages/man1/perf-stat.1.html. [Accessed 01-03-2024].

[7] PerfMon Events — perfmon-events.intel.com. https://perfmon-events.intel.com/. [Accessed 01-03-2024].

[8] sklearn.linear_model.LogisticRegression — scikit-learn.org. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression. [Accessed 12-03-2024].

[9] Innovative Computing Laboratory (ICL) at University of Tennessee Knoxville (UTK). GitHub - icl-utk-edu/papi — github repository. https://github.com/icl-utk-edu/papi/tree/master. [Accessed 09-02-2024].

[10] Kuljit S Bains, John B Halbert, Christopher P Mozak, Theodore Z Schoenborn, and Zvika Greenfield. Row hammer refresh command. https://patents.google.com/patent/US9236110B2/en.

[11] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, aug 2000.

[12] Jason Brownlee. Classification And Regression Trees for Machine Learning - MachineLearningMastery.com — machinelearningmastery.com. https://machinelearningmastery.com/classification-and-regression-trees-for-machine-learning/. [Accessed 12-03-2024].

[13] Jason Brownlee. Logistic Regression for Machine Learning - MachineLearningMastery.com — machinelearningmastery.com. https://machinelearningmastery.com/logistic-regression-for-machine-learning/. [Accessed 12-03-2024].

[14] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[15] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Flexible-latency dram: Understanding and exploiting latency variation in modern dram chips, 2018.

[16] Emir Demirović and Peter J. Stuckey. Optimal decision trees for nonlinear metrics, 2021.

[17] Anna Veronika Dorogush, Vasily Ershov, and Andrey Gulin. Catboost: gradient boosting with categorical features support, 2018.

[18] Loïc France, Maria Mushtaq, Florent Bruguier, David Novo, and Pascal Benoit. Vulnerability Assessment of the Rowhammer Attack Using Machine Learning and the gem5 Simulator -Work in Progress. In *SaT-CPS 2021 - ACM Workshop on Secure and Trustworthy Cyber-Physical Systems*, pages 104–109, Virtually, United States, April 2021.

[19] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P*, May 2020. Best Paper Award, Pwnie Award for Most Innovative Research, IEEE Micro Top Picks Honorable Mention, DCSR Paper Award.

[20] Brendan Gregg. Linux perf Examples — brendangregg.com. https://www.brendangregg.com/perf.html. [Accessed 01-03-2024].

[21] Intel. Intel® vtune™ profiler user guide, 2023.

[22] R Intel. Intel r 64 and ia-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d, and 4, 2023.

[23] Patrick Jattke, Victor Van Der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 716–734, 2022.

[24] Dayeon Kim, Hyungdong Park, Inguk Yeo, Youn Kyu Lee, Youngmin Kim, Hyung-Min Lee, and Kon-Woo Kwon. Rowhammer attacks in dynamic random-access memory and defense methods. *Sensors*, 24(2), 2024.

[25] Jeremie S. Kim, Minesh Patel, Abdullah Giray Yaglikçi, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques. *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 638–651, 2020.

[26] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014.

[27] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A case for exploiting subarray-level parallelism (salp) in dram. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 368–379, 2012.

[28] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.

[29] Donghyuk Lee, Yoongu Kim, Vivek Seshadri, Jamie Liu, Lavanya Subramanian, and Onur Mutlu. Tiered-latency dram: A low latency and low cost dram architecture. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 615–626, 2013.

[30] Eojin Lee, Ingab Kang, Sukhan Lee, G. Edward Suh, and Jung Ho Ahn. Twice: Preventing row-hammering by exploiting time window counters. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 385–396, 2019.

[31] Jung-Bae Lee. Green memory solution. In *Investor's Forum, Samsung Electronics*, 2014.

[32] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing rowhammer faults through network requests. *CoRR*, abs/1805.04956, 2018.

[33] Maher Maalouf. Logistic regression in data analysis: An overview. *International Journal of Data Analysis Techniques and Strategies*, 3:281–299, 07 2011.

[34] Corey Malone, Mohamed Zahran, and Ramesh Karri. Are hardware performance counters a cost effective way for integrity checking of programs. In *Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing*, STC '11, page 71–76, New York, NY, USA, 2011. Association for Computing Machinery.

[35] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[36] Ingo Molnar. perf: Linux profiling with performance counters, 2009.

[37] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. A modern primer on processing in memory, 2022.

[38] University of Tennessee. *PAPI User-s Guide*, 3.5.0 edition.

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[40] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, Austin, TX, August 2016. USENIX Association.

[41] Project Zero team at Google. GitHub - google/rowhammer-test: Test DRAM for bit flips caused by the rowhammer problem — github repository. https://github.com/google/rowhammer-test. [Accessed 09-02-2024].

[42] Rui Qiao and Mark Seaborn. A new approach for rowhammer attacks. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 161–166, 2016.

[43] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Bit-flip attack: Crushing neural network withprogressive bit search. *CoRR*, abs/1903.12269, 2019.

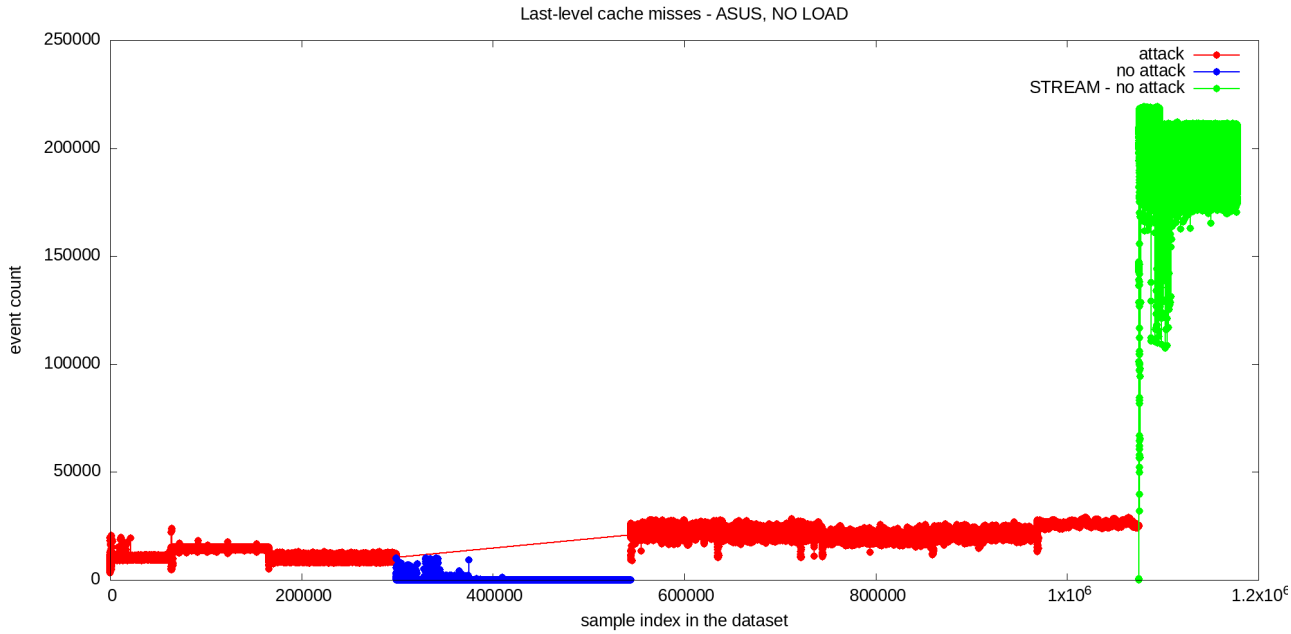
[44] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges.

[45] Vincent M Weaver and Jack Dongarra. Can hardware performance counters produce expected, deterministic results? Atlanta, GA, 2010-12 2010.

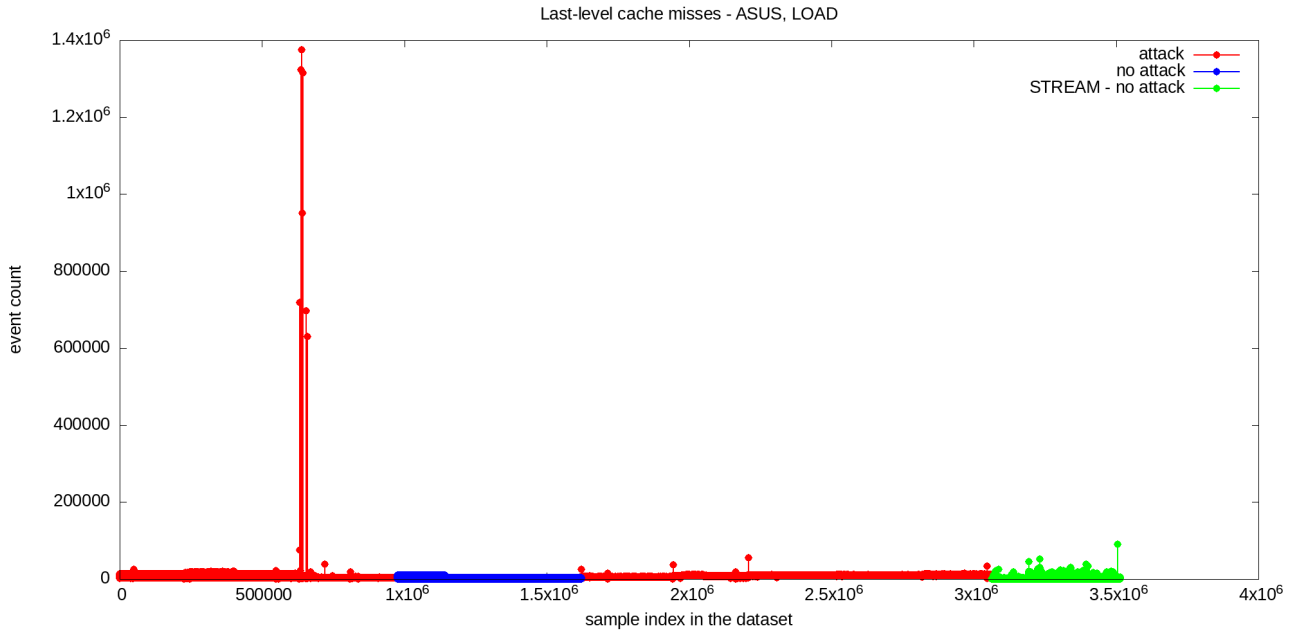
[46] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 19–35, Austin, TX, August 2016. USENIX Association.

A Appendix

In this section, we provide the visualization of the four sampled features in both NO LOAD and LOAD conditions generated on our ASUS and DELL machines.

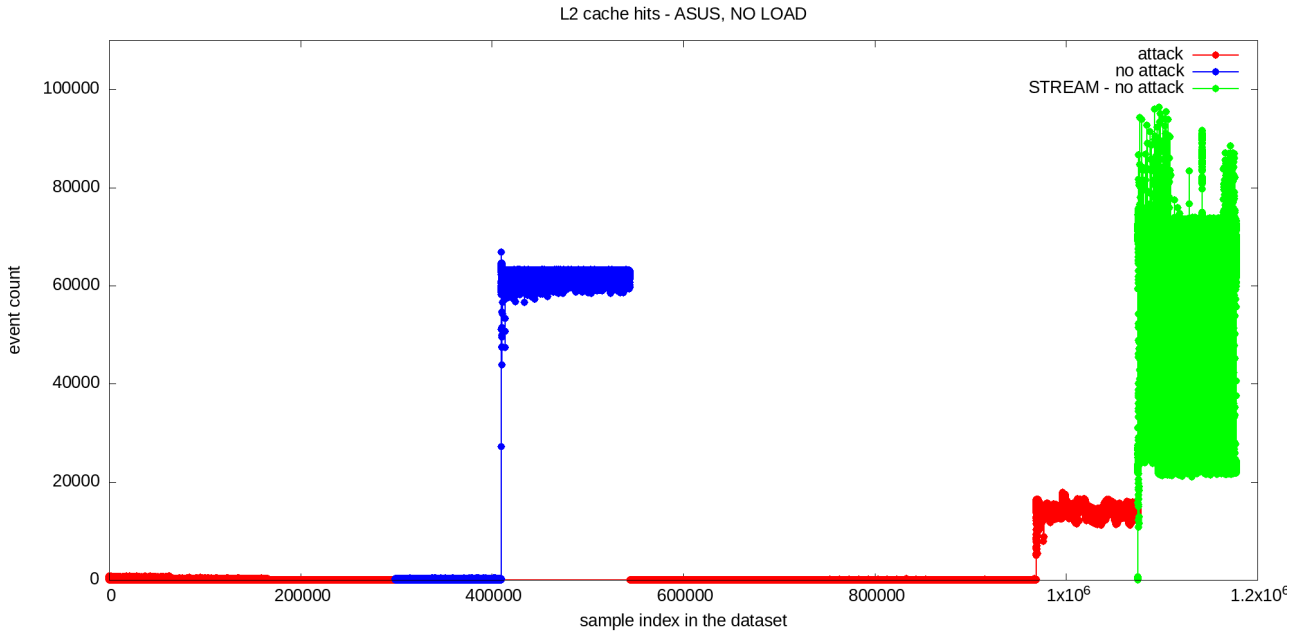


(a) NO LOAD

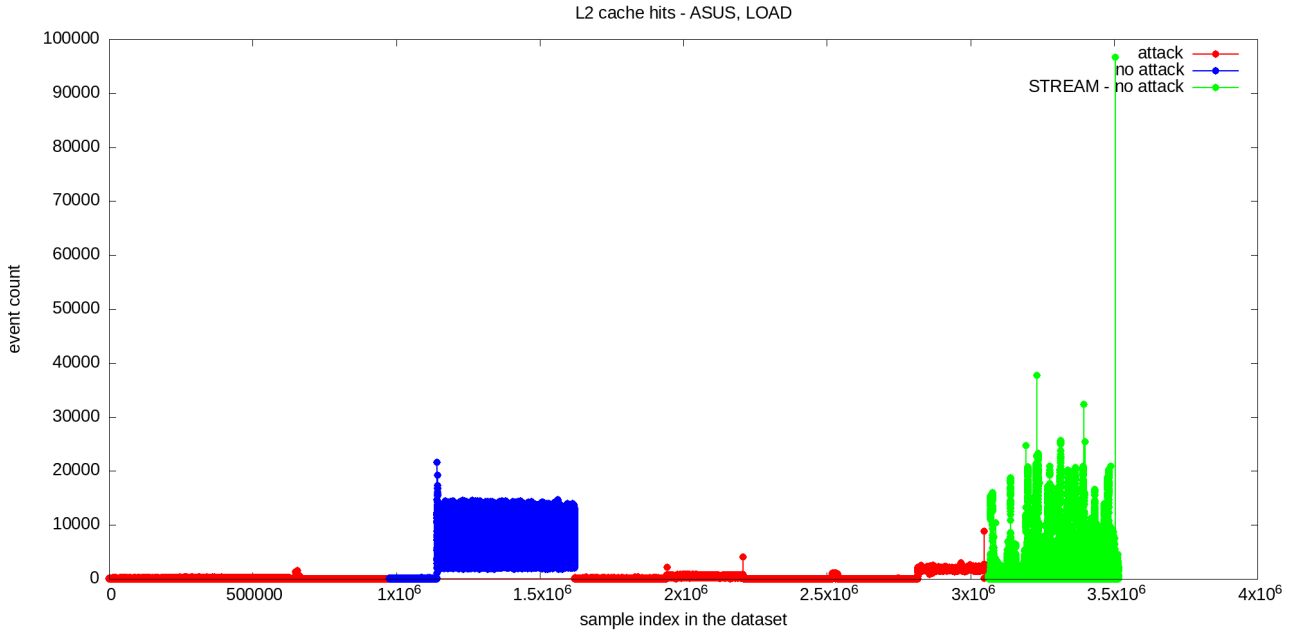


(b) LOAD

Figure 14: Last-level (L3) cache misses on ASUS machine

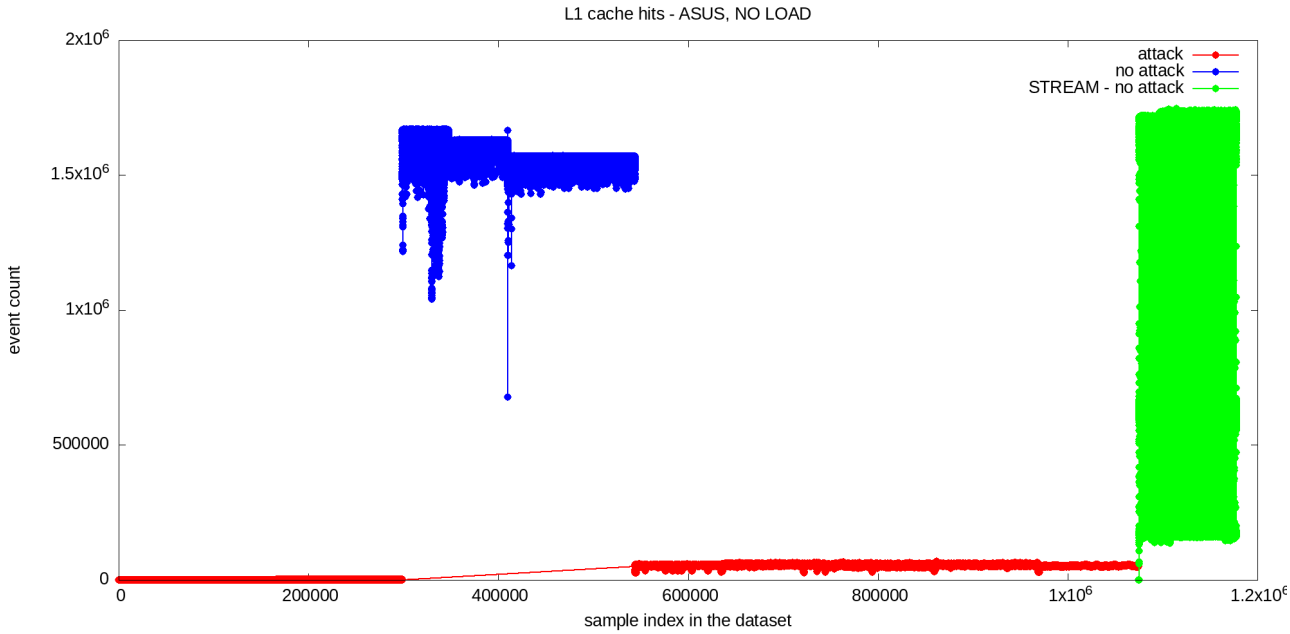


(a) NO LOAD

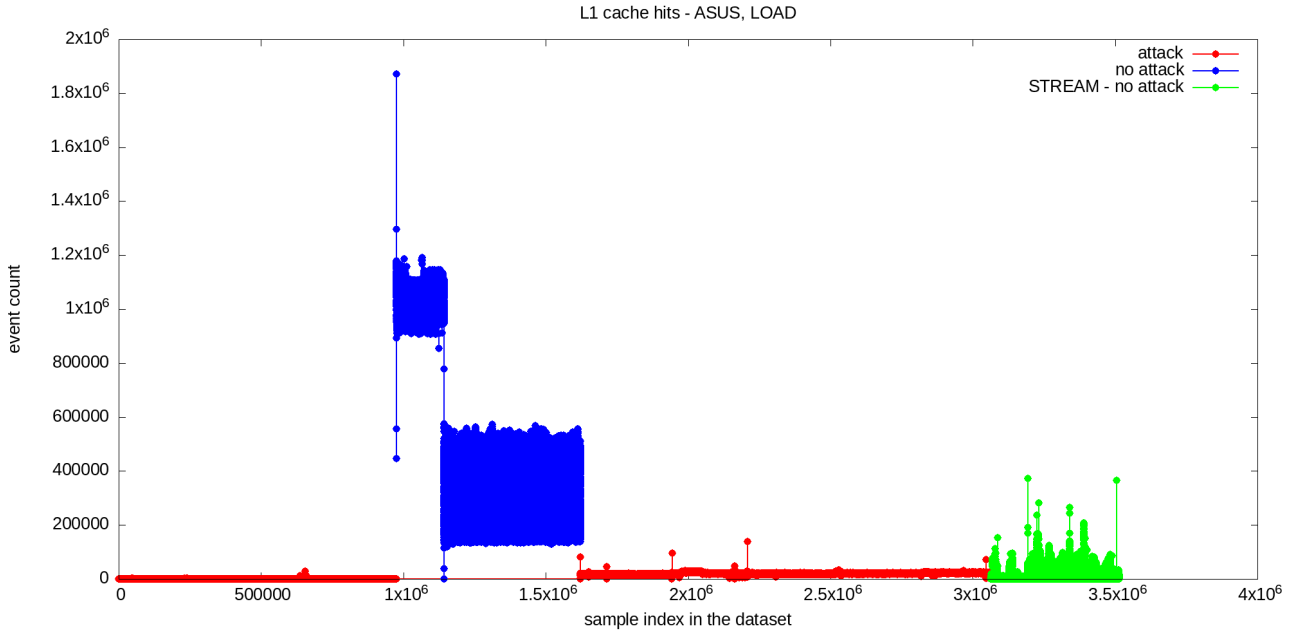


(b) LOAD

Figure 15: L2 cache hits on ASUS machine

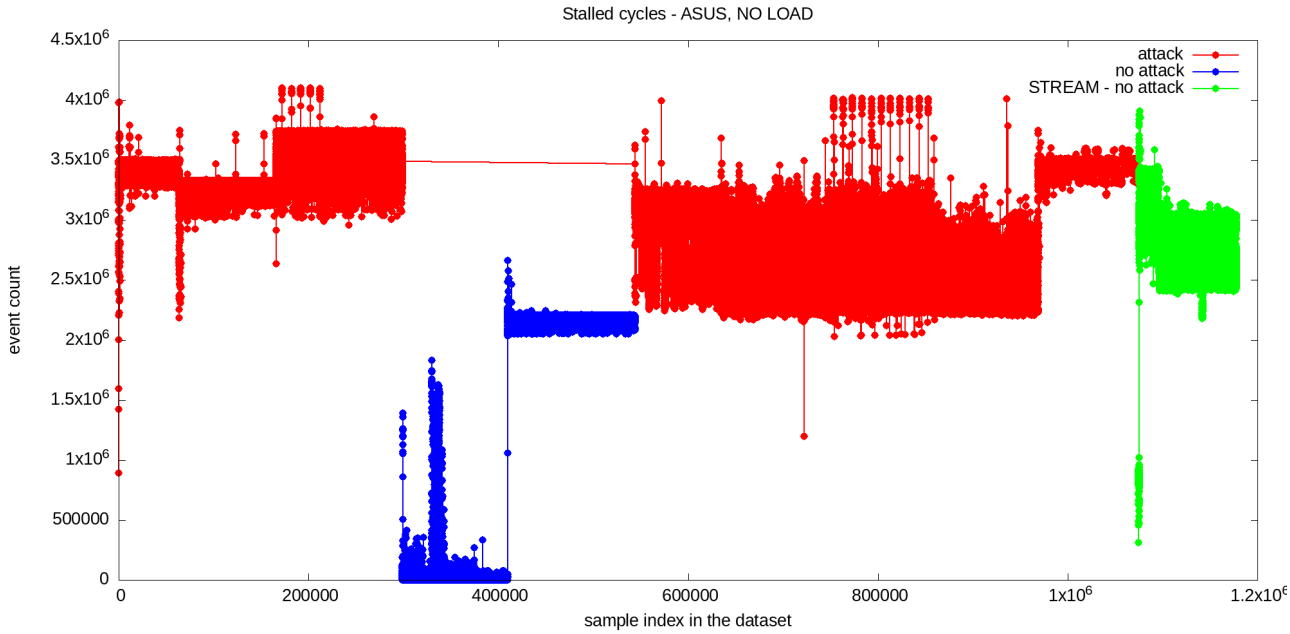


(a) NO LOAD

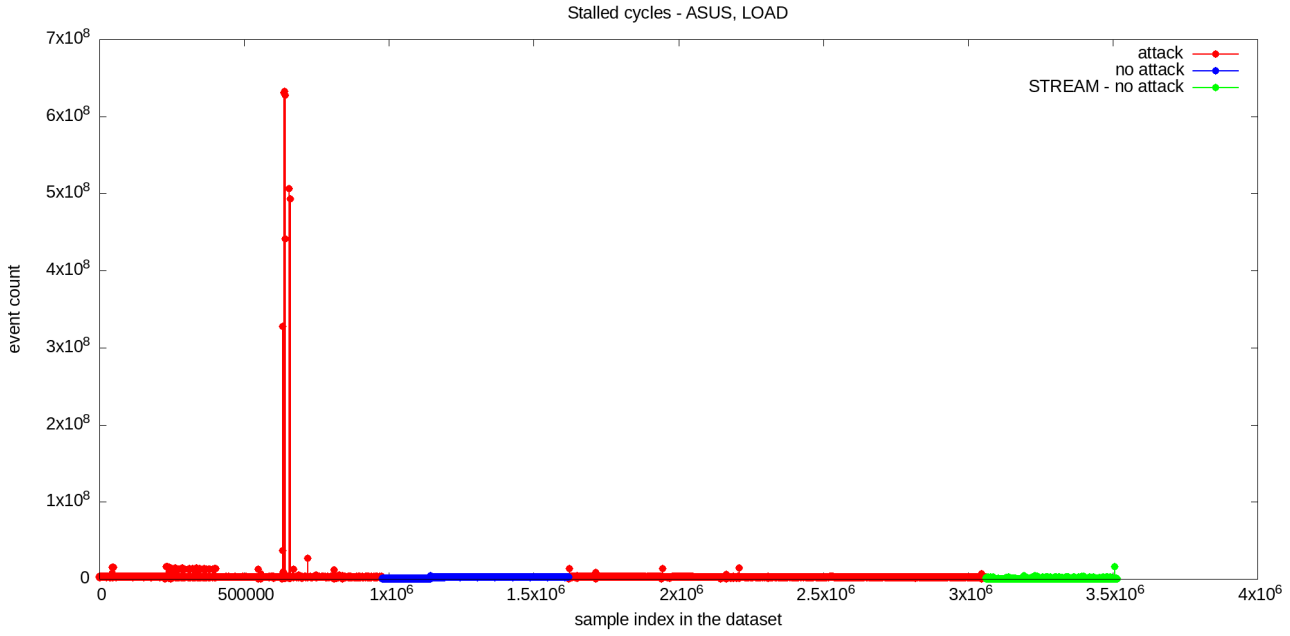


(b) LOAD

Figure 16: L1 cache hits on ASUS machine

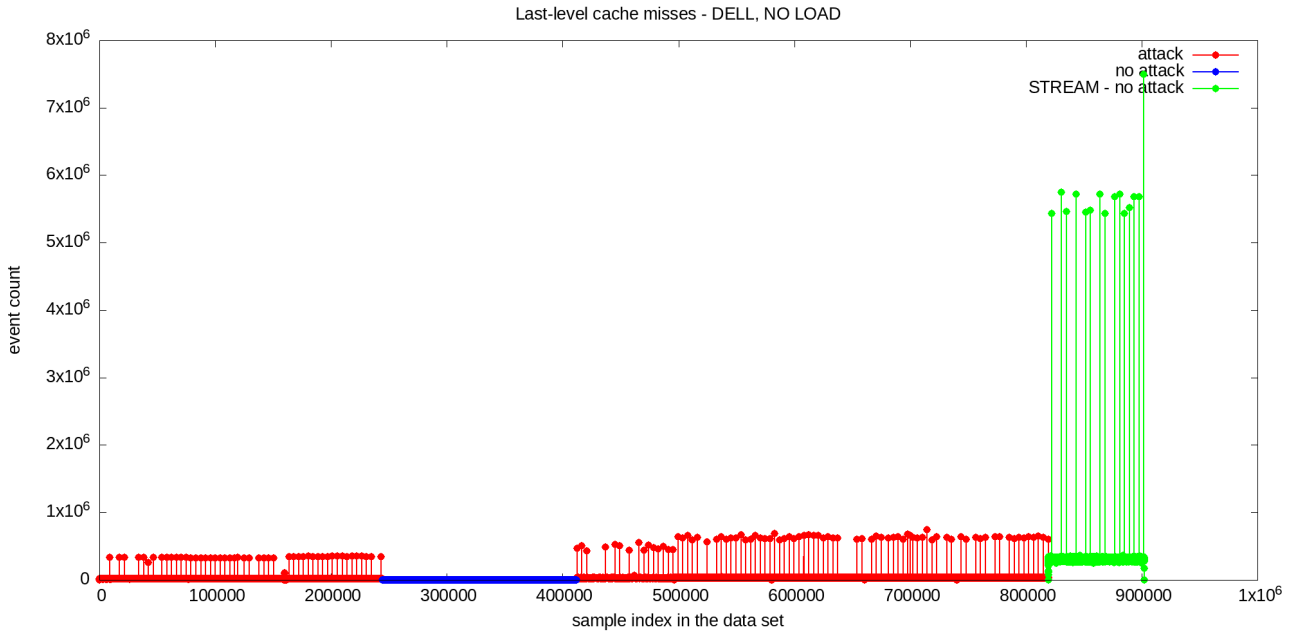


(a) NO LOAD

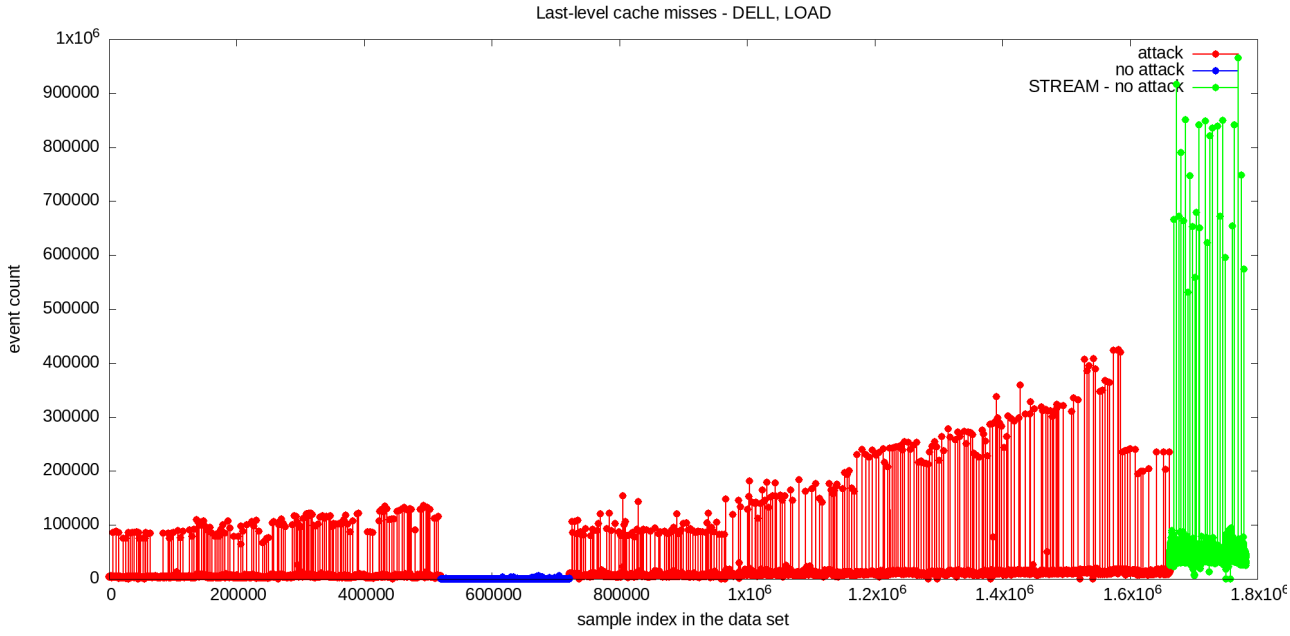


(b) LOAD

Figure 17: Stalled cycles on ASUS machine

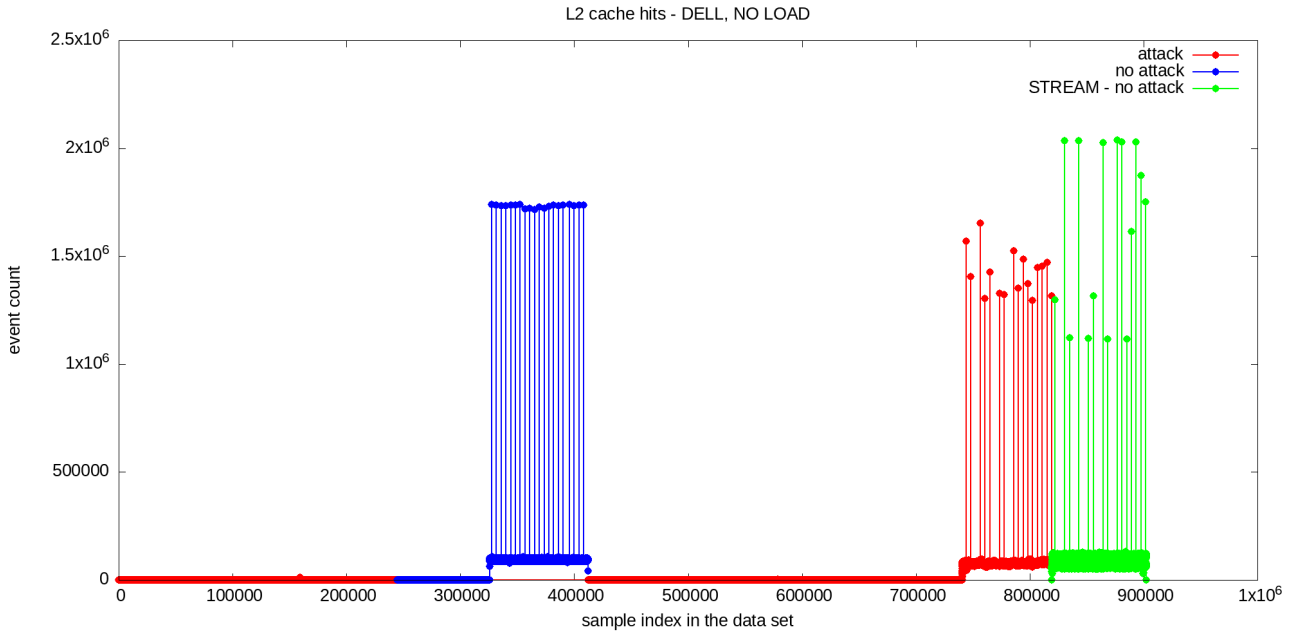


(a) NO LOAD

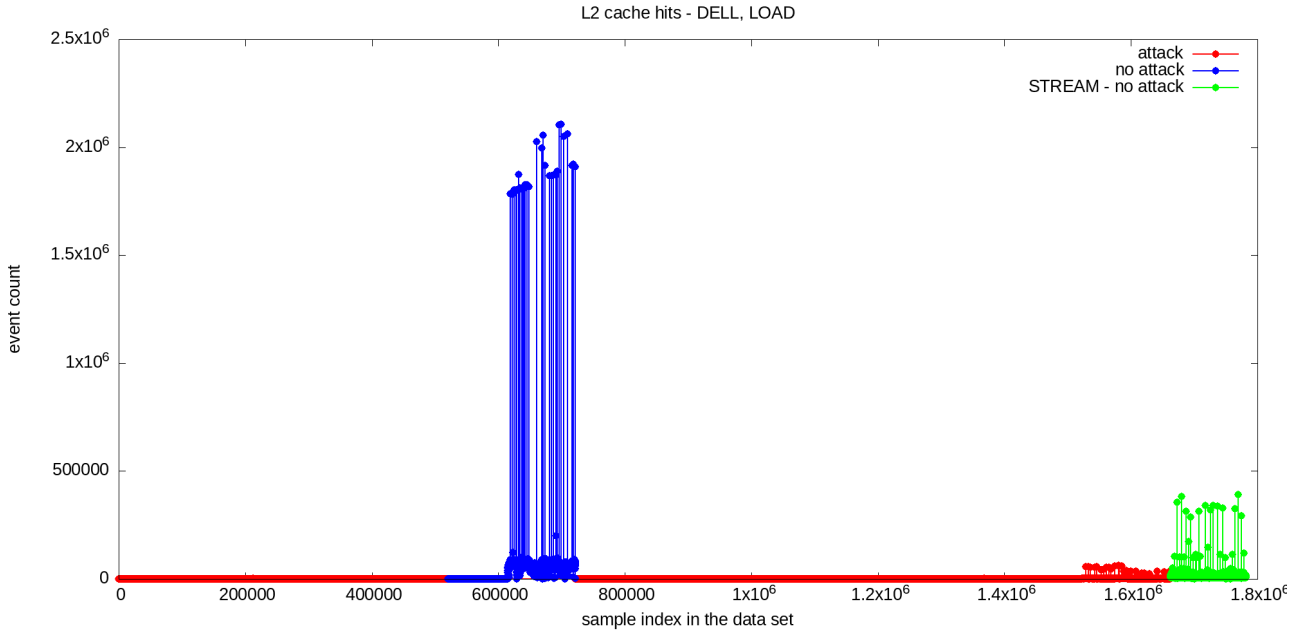


(b) LOAD

Figure 18: Last-level (L3) cache misses on DELL machine

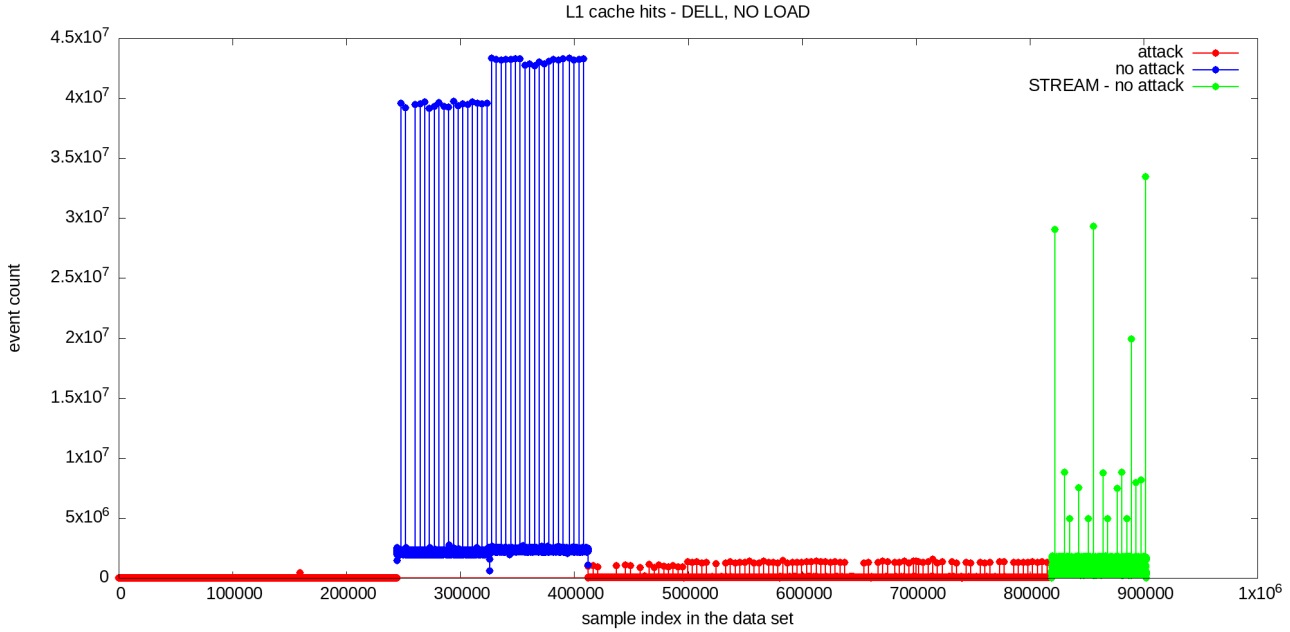


(a) NO LOAD

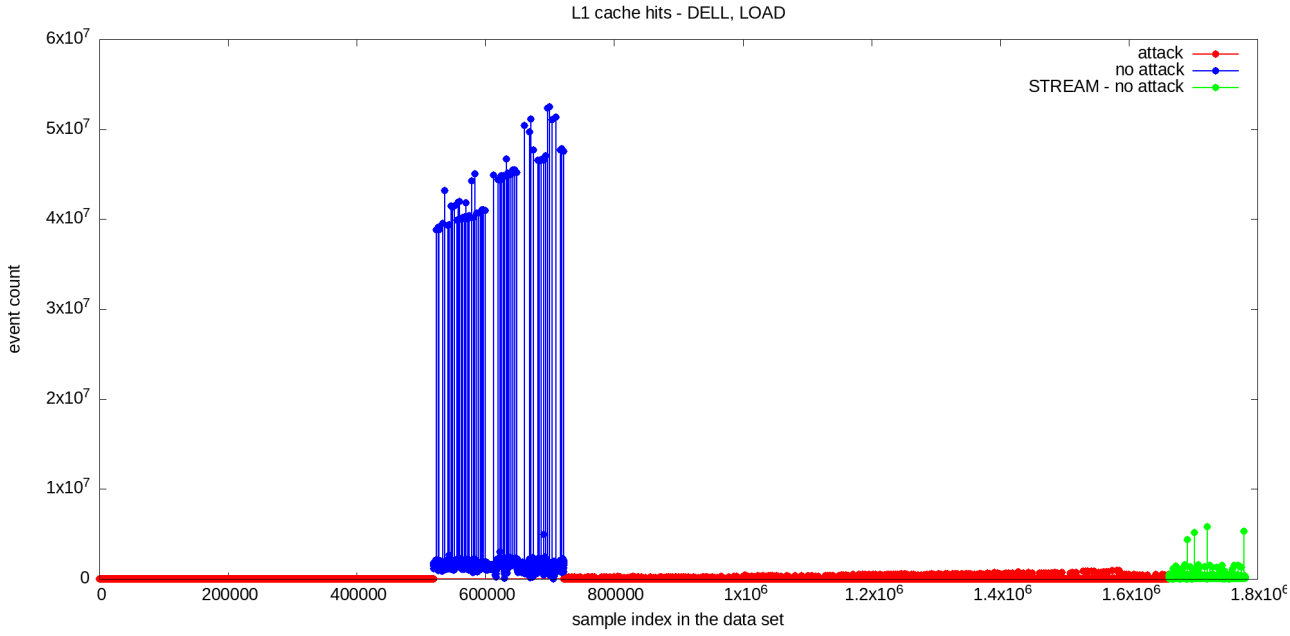


(b) LOAD

Figure 19: L2 cache hits on DELL machine

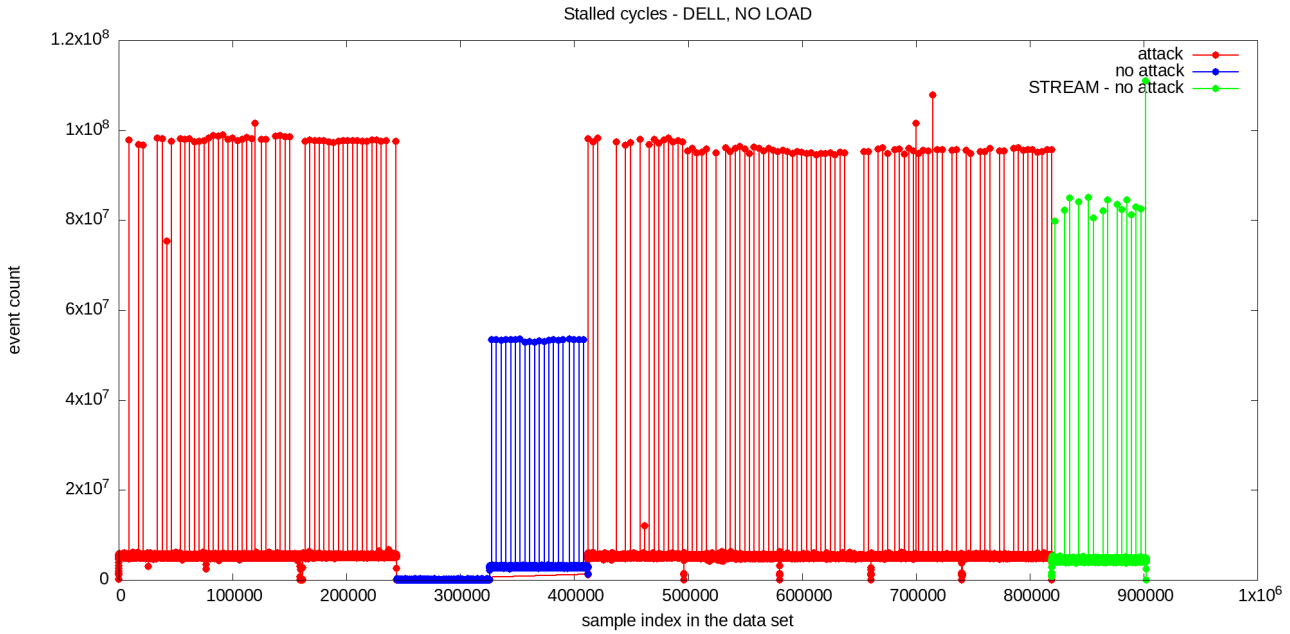


(a) NO LOAD

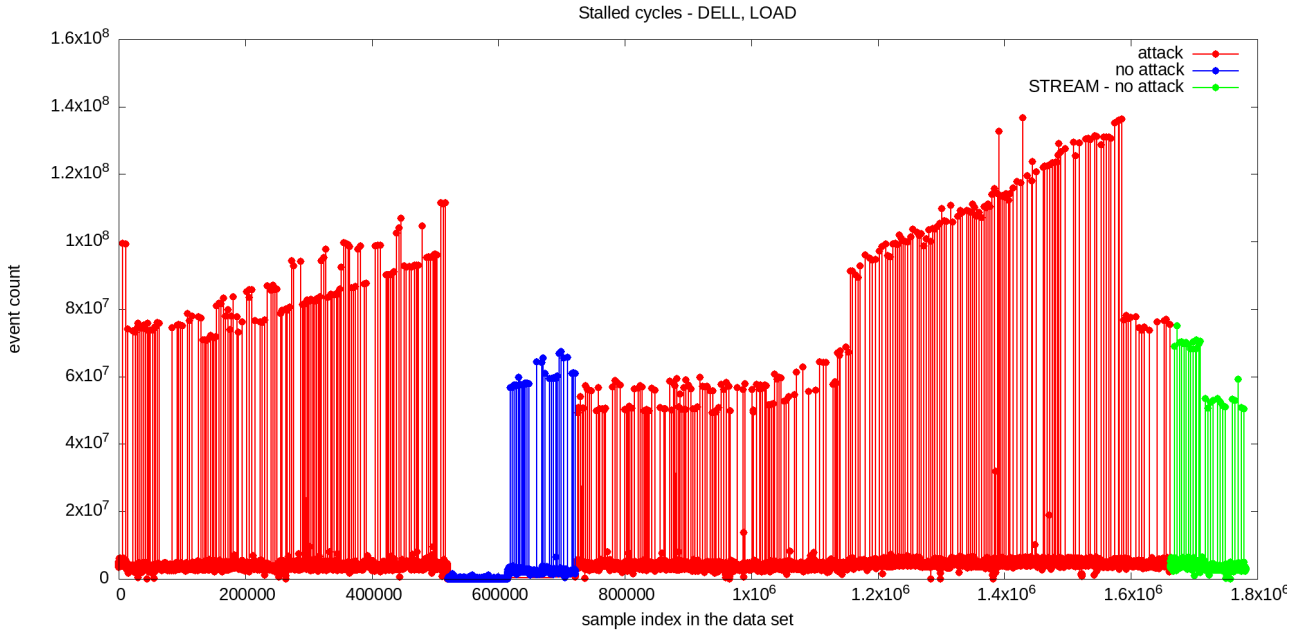


(b) LOAD

Figure 20: L1 cache hits on DELL machine



(a) NO LOAD



(b) LOAD

Figure 21: Stalled cycles on DELL machine

ÉCOLE POLYTECHNIQUE | INSTITUT POLYTECHNIQUE DE PARIS

Statement of Academic Integrity Regarding Plagiarism

I, the undersigned Kolić Bogdana [family name, given name(s)], hereby certify on my honor that:

1. The results presented in this report are the product of my own work.

2. I am the original creator of this report.

3. I have not used sources or results from third parties without clearly stating thus and referencing them according to the recommended rules for providing bibliographic information.

Declaration to be copied below:

I hereby declare that this work contains no plagiarized material.

Date March 18, 2024

Signature Bogdana Kolić