Vulnerability Assessment for the Rowhammer Attack Using Hardware Performance Counters and Machine Learning

Bogdana Kolić, Maria Mushtaq

Vulnerability Assessment for the Rowhammer Attack Using Hardware Performance Counters and Machine Learning

Bogdana Kolić[1,2] and Maria Mushtaq[2]

[1] École Polytechnique, Palaiseau, France
bogdana.kolic@polytechnique.edu
[2] LTCI, Télécom Paris, Institute Polytechnique de Paris, Palaiseau, France
maria.mushtaq@telecom-paris.fr

Abstract

Numerous machines using DRAM chips as main memory are vulnerable to the *Rowhammer* attack, which can be used as a tool for privilege escalation. The existing mitigation techniques either require complex hardware implementation or have a high performance cost. A potential improvement would be to implement a detection mechanism and trigger performance-costly mitigation only in the case of attack detection. In this paper, we study this defence method on three systems using Intel Skylake, Tiger Lake and Alder Lake processors, and DDR4 and DDR5 DRAM chips as main memory. We execute four variants of the attack code on these machines and observe their traces in the hardware. We use the *PAPI* library and *perf* to periodically read the generated traces from the machines' hardware performance counters. Finally, we train machine learning models such as *logistic regression* and *decision trees* to distinguish attack and no-attack behaviour. Our best models achieve accuracy above 99.6% and perform the classification of both 50μs and 1ms samples in software fast enough (less than 0.5μs per sample) to detect the attack before completion.

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 main memory in consumer machines, aims to achieve low latency, high capacity and low cost-per-bit of memory[15][25][35]. As a solution, manufacturers increased 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 [25]. 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 [42], overcome memory isolation enforced by virtualisation [44] or undermine the accuracy of deep neural networks [41]. In the past decade, numerous mitigation techniques

[1] Although Rowhammer became popular in 2014 after being presented in [25], the vulnerability had been known in 2012 [10] [22]

claiming to defend against the Rowhammer have been developed [23][25][28], 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 [29], studies have shown that DDR4 chips are, in fact, still vulnerable to Rowhammer attacks [19][22][30][38]. Furthermore, the Rowhammer on DDR5 chips is yet to be studied. A new mitigation method proposed by France et al. [18] uses Neural-Network models that can recognise 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* [27]. Their results show high accuracy, low overhead, and do not require a significant space on the silicon.

In our work, we continue studying machine learning models that can be used as a detection mechanism for the Rowhammer. We omit the simulators and run the attack code directly on three different machines. We generate the data by reading their existing hardware performance counters. Two of our machines are equipped with DDR4 DRAM chips as main memory, while one of the machines utilises 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* [34] to access the performance counters. 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. We make the following contributions:

- We present detection models that take into account diverse implementations of the Rowhammer attack (namely four attack variants)
- We train our models based on data obtained directly from the hardware performance counters of three real systems
- We propose highly-accurate models for three real systems, which perform a fast classification in software

2 Background

2.1 DRAM and Disturbance Errors

Dynamic Random Access Memory (DRAM) is a hierarchy of DRAM channels, modules, ranks, chips and banks. Banks are the smallest units that can work in parallel [26]. They are two-dimensional arrays of cells, where the charges stored in the cells represent logical bits (0 or 1). Each bank is equipped with a row-buffer, which serves as an intermediary for accessing data from a DRAM bank row. Commands for accessing DRAM are issued by the *memory controller* [25].

Kim et al. [25] performed a large study on DDR3 chips and found that repeated *opening* (also called *activating*) and *closing* one row - transferring the row into the row-buffer and back, can cause *disturbance errors* in nearby DRAM cells - some cells lose their already non-persistent charge faster than it is automatically restored (*refreshed*). We observe this behaviour as bit-flips in those cells.

2.2 The Rowhammer

The goal of the attack is to induce bit-flips in 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*. An example of code for hammering written in C is given in Figure 1.

```
/* 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 1: Rowhammer loop (simplified code from [39])

The attack targets main memory and therefore, the attacker has to bypass the cache. Thanks to the unprivileged CLFLUSH command [21], the attacker can easily invalidate a specific cache line and access data directly from the DRAM. In some environments, however, the CLFLUSH command is unavailable. Qiao and Seaborn [40] propose a version of the attack without the CLFLUSH instruction that is based on non-temporal store instructions such as MOVNTI and MOVNTDQ. Nevertheless, accessing data from the same address in a loop alone does not guarantee that one row will be opened and closed repeatedly [25]. When the memory controller receives consecutive read/write requests to the same row it optimises the access time by issuing multiple READ/WRITE commands without closing the row. Hence, as we see in Figure 1, the attacker can force closing the row by accessing two different rows from the same bank one after the other. 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 [38], while Seaborn and Dullien [42] suggest a probabilistic approach and hammering multiple rows to increase the chances of having two aggressor rows in the same bank.

One way of optimising the attack is to increase the number of useful accesses by choosing two aggressor rows which are adjacent to the same victim row. This version of the attack is referred to as the *double-sided rowhammer*.

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] [32]. In our work, we make use of two tools to access the performance counters: the *PAPI* library [4] and the *perf tool* [34].

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 [5] [11] [36].

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.

2.3.2 Perf

The perf tool [34] 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 our purposes, we use perf with *stat* option, which runs a command and gathers performance counter statistics [6][20]. The main difference between PAPI and perf is that with PAPI we obtain traces of specific portions of code, while perf counts the events generated by the entire program execution.

2.4 Related Work

Many mitigation techniques for the Rowhammer attack have been developed, but they either cause a significant performance overhead or cannot easily be implemented in the hardware. France et al. [18] suggest that a need-based detection mechanism could be a solution. In their work, they look at the hardware-level events characteristic to the attack execution (for example, a high number of cache misses) and present neural-network models that can detect the attack based on its traces on time to prevent its completion. They generate the traces by running the attack and no-attack code in the gem5 simulator and they use *Ramulator* [27] to simulate main memory. Their models give promising results when trained and tested on these data in software. To obtain an online detection mechanism, the models would have to be implemented in the hardware and the system architecture might have to be modified to allow them to count specific hardware-level events. In this paper, we focus on three real systems and run the code directly on the machines to generate the traces, then use performance monitoring tools to read their existing hardware performance counters. Our models use system-specific events as input features, based on their availability on the architectures. Furthermore, unlike the system simulated by France et al. [18], all our systems have three levels of cache. In addition, we try to address the diversity of the Rowhammer by using several different implementations of the attack.

3 Methodology

Our goal is to obtain machine-specific classification models that can easily be implemented in the hardware, and which analyse the traces of a process to categorise it as attack or no-attack behaviour. We also want a short classification time, so that we can trigger a defence mechanism in case of an attack detection. First step in our method is generating the data sets containing hardware-event traces of different attack and no-attack programs. Then, we study the data and train models that can perform a fast and accurate classification.

We start the creation of data sets by taking three different machines to get a variety of processors and DRAM chips. Then, we study the implementation of the Rowhammer attack and choose several instances of the attack and no-attack code to run on the machines. We also try to measure the minimal attack execution time[2] on these machines to select the frequency at which we will read the traces from the HPCs. After we have chosen the features (hardware events) of interest, we set up the counters, start running the code which generates the traces, and make periodic readings to the counters while the programs are executing. We use performance monitoring tools to access the HPCs on our machines. Each reading gives us one sample in the data set. Once we have the data sets, we train and test different models to make a binary classification - output the label attack or no-attack, based on the count of hardware-event traces.

3.1 Machine Specifications

We perform our work on three different machines running Linux. The first machine is Vivo-Book_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 understand that in reality the attack may or may not be running as the only process on the system, so we obtain our data by running the programs in different load conditions [18]. For NO LOAD conditions, we try to 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 benchmarks on LENOVO machine, four on ASUS and eight instances of STREAM on our DELL machine to create LOAD.

3.2 Attack and No-Attack Programs

On our DELL machine, and for the feature-selection purposes, we use four different programs to simulate the attack and another three to represent the no-attack behaviour. All of the code (as well as the entire generated data set and graphs) can be found in [2]. The first variant of the attack is a slightly modified version of the code obtained from the authors of [18], written for the gem5 simulator. 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

[2]The lowest number of activations needed to cause bit flips on DDR4 is reported to be ten thousand per aggressor row [24]. We time the code used to generate the samples on all three machines with 10000 toggles.

from cache by using CLFLUSH. The code for the second version of the attack (using non-temporal store instructions [40]) 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-initialised buffer from which the aggressor rows are chosen in the attack code. The different *attack* (*no_attack*) functions are presented in Figures 2 and 3.

The third no-attack code is the *STREAM benchmark* [33], 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 [39]. Our code keeps the memory initialisation 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. For data generation on ASUS and LENOVO, we in addition have no-attack code derived from the *double-sided rowhammer* and the *rowhammer-test* by removing the CLFLUSH instruction from their hammering functions.

```
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) Base 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 with non-temporal store instruction

Figure 2: CLFLUSH 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 3: No-attack loops derived from the basic attack code

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, but as we are interested in measurements after equal time intervals, we create our own program for sampling. A timer is configured to send a signal that interrupts the program periodically after 50μs and we read the counters while handling the interrupt. This might interfere with our event count [43], 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.

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 counts are generated by running the specified *program*.

3.5 Feature Selection

Based on previous work [18], we assume that observing cache events could prove useful. The essence of the attack is activating a DRAM row, hence it is characterised by a high number of last-level cache misses. Since we are constantly accessing DRAM, and moreover trying to access different rows from the same DRAM bank, we also expect a high number of stalled cycles caused by the attack program [26]. France et al. [18] suggest looking also at the number of last-level cache (LLC) hits, as they find that their no-attack code produces significantly more LLC hits than 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. 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 paper 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. After some experiments where we counted the various events available, generated by both the attack and no-attack code, we finally decided to keep the four events described above. On ASUS and LENOVO laptops we use the PAPI library, and the native events we selected to count are given in Table 1. The equivalent events for the Alder Lake architecture found on our DELL machine are taken from [7] and given in Table 2.

3.6 Data Generation

We run the programs introduced in section 3.2 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

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

Table 1: Event selection on ASUS and LENOVO machines

Event	Alder Lake event
LLC (L3 cache) misses	LONGEST_LAT_CACHE.MISS
L2 cache hits	L2_RQSTS.DEMAND_DATA_RD_HIT
L1 data cache hits	MEM_LOAD_RETIRED.L1_HIT
Stalled cycles	UOPS_RETIRED.STALLS

Table 2: Event selection on the DELL machines

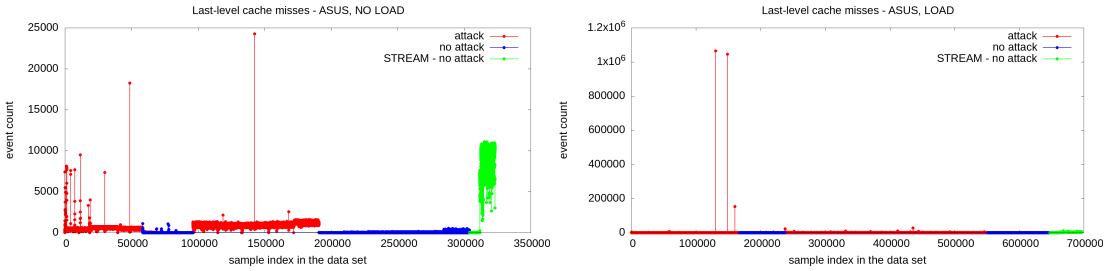
the event count in a 1ms interval for the data set generated on the DELL machine, while samples from ASUS and LENOVO data sets stand for 50μs intervals. In total, we have around 1.3 million samples on LENOVO machine, 2.7 million samples on DELL and 1 million samples on ASUS machine. Each sample contains 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.

4 Machine Learning

We plot the data from our ASUS machine (LLC misses and L1 cache hits) in both NO LOAD and LOAD conditions in figures 4 and 5. The graphs representing L2 cache hits and stalled cycles, as well as the data from the other two machines, can be found in the repository [2]. 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 higher number of L2 cache hits than some of the no-attack code, however, it can still be distinguished from the no-attack code according to its three other traces (for example, the LLC misses). The STREAM benchmark is an exception due to its very high 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 thus rely on machine learning (ML) models to perform the classification.

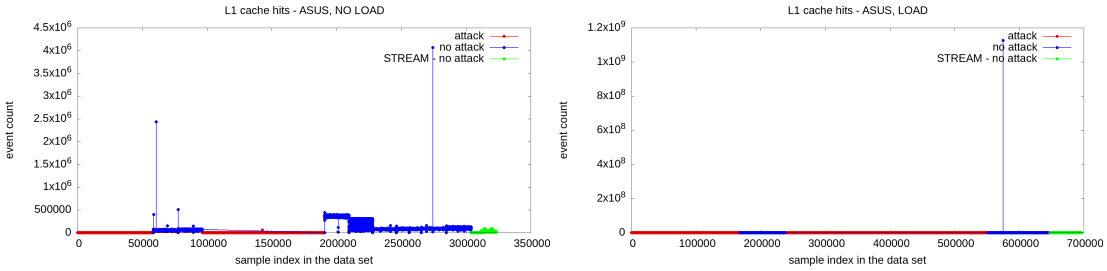
4.1 Models

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]. *LogisticRegression* and *DecisionTreeClassifier* models are implemented using the *scikit-learn API* [14][37] .



(a) NO LOAD (b) LOAD

Figure 4: Last-level (L3) cache misses on ASUS machine



(a) NO LOAD (b) LOAD

Figure 5: L1 cache hits on ASUS machine

Logistic Regression is a model commonly used for binary classification. The model computes the probability that a 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] [31].

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].

4.2 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.

4.3 Results

In tables 3, 4 and 5 we present the performance (Accuracy, False positives - *FP*, False negatives - *FN* and the average time a model takes to make a classification of one sample - *Avg time*) of the three models on each machine. All of our tested models give satisfactory results. They all have accuracy above 92%. Furthermore, the best model on each machine has accuracy above 99.6%. France et al. [18] achieved an accuracy above 99.8% 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. This means that our models could classify the data obtained directly from the HPCs. The models are significantly faster in software, a single classification on the samples from the test set takes less than 0.5 microseconds on average. It is worth mentioning that, although the models are faster than the ones presented by France et al. [18], on our DELL machine we are still performing a classification of traces representing a 1ms interval, while their models use 10μs time windows.

Model	Accuracy (%)	FP (%)	FN (%)	Avg time (ns)
Logistic Regression	92.580	6.357	1.063	34.920
Decision Tree	99.606	0.043	0.351	72.023
CatBoost	99.617	0.039	0.344	234.004

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

Model	Accuracy (%)	FP (%)	FN (%)	Avg time (ns)
Logistic Regression	99.229	0.363	0.408	20.705
Decision Tree	99.972	0.022	0.006	37.870
CatBoost	99.966	0.022	0.012	176.649

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

Model	Accuracy (%)	FP (%)	FN (%)	Avg 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 should still detect Rowhammer before it executes on all three machines. Timing the attack on our machines gives the following results: we observe that on DELL machine, the attack takes at least 4ms, 0.998ms on ASUS and 1.111 ms to execute on LENOVO machine (for 10 000 attack-loop iterations). With classification time under 0.5μs, and assuming the correct prediction, our models are able to detect the attack before it executes. The small enough intervals needed for collecting the data increase our chances of detecting the attack before completion even with some false negative predictions. We argue that our models could easily be integrated in hardware: the Decision Tree model on ASUS machine has depth five, and the one on DELL has depth four. The CatBoost model on LENOVO is made of fifteen

decision tree estimators, where each estimator has maximum depth four. We still note that the less complex Decision Tree model on LENOVO, with maximum depth five, also achieves high accuracy of 99.6%.

5 Discussion

Our methodology encounters several challenges:

- **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 on ASUS and LENOVO might not be read precisely after 50 microseconds, but after 50 microseconds and some noise. Furthermore, perf imposes a lower bound of 1ms for the sample interval length.

- **Correctly labelling 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 initialisation 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 generalise 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. In our work, we also do not study whether changing the parameters of the attack (for example the number of toggles) alters its traces. Representing no-attack behaviour is an even more challenging task.

- **Appropriate training data set:** We aim to have a good balance of the attack and no-attack samples in our data sets. The attack traces represent 62% and 38% of the samples on our ASUS and LENOVO machines, respectively, and they have the majority of 79% on the DELL machine. We do not know if this difference has an effect on our 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.

6 Conclusion

Rowhammer attack is a security issue which keeps evolving to adapt to current defence 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 data collection, look for useful hardware-event features and show that fast and simple to implement decision tree models can recognise 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 and CatBoost models on our LENOVO machine is 99.6%, while the Decision Tree models on our ASUS and DELL machine achieve the accuracy above 99.9%. More importantly, their performance in software is sufficiently fast (less than 0.5μs needed to classify one sample) and the time for attack detection primarily depends on the interval size used for sampling. In the case when the classification is correct, they should recognise the attack behaviour before it successfully completes its execution.

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-PAPER — github.com. https://github.com/bogdanaKolic/ROWHAMMER-VULNERABILITY-ASSESSMENT-PAPER. [Accessed 30-04-2024].

[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] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

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

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

[30] 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.

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

[32] 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.

[33] 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.

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

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

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

[37] 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.

[38] 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.

[39] 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].

[40] 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.

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

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

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

[44] 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.