Vulnerability Assessment of the Rowhammer Attack Using Machine Learning and the gem5 Simulator -Work in Progress

Loïc France, Maria Mushtaq, Florent Bruguier, David Novo, Pascal Benoit

Vulnerability Assessment of the Rowhammer Attack Using Machine Learning and the gem5 Simulator — Work in Progress

Loïc France, Maria Mushtaq, Florent Bruguier, David Novo, Pascal Benoit
LIRMM, CNRS, University of Montpellier
Montpellier, France
{firstname}.{lastname}@lirmm.fr

Abstract

Modern computer memories have been shown to have reliability issues. The main memory is the target of a security attack called Rowhammer, which causes bit flips in adjacent victim cells of aggressor rows. Multiple mitigation techniques have been proposed to counter this issue, but they all come at a non-negligible cost of performance and/or silicon surface. Some techniques rely on a detection mechanism using row access counters to trigger automatic defenses. In this paper, we propose a tool to build a system-specific detection mechanism using gem5 to simulate the system and Machine Learning to detect the attack by analyzing hardware event traces. The detection mechanism built with our tool shows high accuracy (over 99.8%) and low latency (maximum 264µs to classify when running offline in software) to detect an attack before completion.

1 Introduction

Physical Memory is a key component to modern computing. Computers use Dynamic Random Access Memory (DRAM) as main memory, with multiple cache levels between the processor and the main memory to speed up the access to frequently used data. As computer technologies became more efficient and smaller, manufacturers have been able to put more memory in much smaller spaces, resulting in better performances and lower cost [12]. However, making DRAM smaller resulted in higher vulnerability to what Kim et al. [14] depicted as disturbance errors.

Attacks exploiting this error, named Rowhammer (RH) attacks, have rapidly appeared to precisely flip bits to gain kernel privileges [20], and it became a major threat to modern computer memories. It has been shown that this type of attack is cross-CPU compatible [19], can escape web page sandboxes [11] and is feasible using only network requests [17].

To counter this attack, multiple mitigation techniques, such as increasing the refresh rate [5], probabilistic neighbor refreshes [14] or row access counters [16], have been proposed. However, all-weather mitigation approaches are either performance costly or take non-negligible space on the silicon. Furthermore, new attacks aware of these techniques have been developed to make the implemented protections ineffective [10]. Some solutions would benefit from simply being need-based mitigations instead of permanently lowering the performances.

As Rowhammer attack tries to corrupt the memory with repeated accesses to DRAM memory while avoiding row buffer hits and cache hits, it can be distinguished from normal processes by looking at some events on the hardware level. Therefore, in this case, vulnerability assessment of microarchitectural components can serve as first-line-of-defense toward Rowhammer vulnerabilities.

In this paper, we present work in progress on a detection mechanism of Rowhammer attacks for a specific system using the gem5 simulator, based on the analysis of hardware events by a Machine Learning model. For the moment, the presented work has only limited test cases for the detection mechanism and does not evaluate the implementation cost of the proposed solution.

2 Related work

RH mitigation has been largely studied since its first appearance in literature [14]. Most of the existing mitigation techniques as of today are synthesized in [13].

Some protection mechanisms try to make the attack impossible to perform through hardware changes, such as an increase of the refresh rate or an additional refresh happening with a small probability after each access [14]. Those solutions have an impact on performances and power consumption even under normal operation (no attack), but have a relatively small impact on the hardware surface, as it does not need a lot of memory.

Some other mitigation techniques use some mechanism to detect the attack and act automatically when the attack is detected. This often requires counters to be used, e.g. as row access counters [3][16][21]. When the counter reaches a threshold value, the related rows are refreshed. Those solutions do not have an impact on performance as they rely on small processes running in parallel directly on the hardware or the memory controller. However, this requires extra memory to store the counters.

RH defense mechanisms do not often rely on Machine Learning to detect that an attack is happening. This type of solution has been barely studied to defend against RH attacks. In particular, two contributions stand out:

Chakraborty et al. [9] propose a software-based solution to predict bit-flips in the memory. The software monitors the LLC miss rate to detect suspicious processes, and then records the DRAM banks and rows accessed by this process to identify access patterns. A Convolutional Neural Network (CNN) then categorizes the access pattern as being from an attack process or not. This solution does not require any hardware modification and relies on address mapping reverse-engineering to precisely detect victim rows. However, the mechanism takes a lot of time (1.5s on average) to detect an attack process after it has begun. If the process already knows the virtual-to-physical address mapping, the bit-flip can happen in less than 10ms according to Kim et al. [14]. 1.5s is more than enough to corrupt the system.

Alam et al. [4] describe a three-step mechanism to detect micro-architectural Side-Channel Attacks (SCA) on encryption algorithms in real time. The first step is the detection of anomalies in the Hardware Performance Counters. When an anomaly is detected, the abnormal traces are passed to a classifier that uses a trained machine learning model to output the possible types of SCA. Finally, a Correlation module is used to detect correlations between the abnormal process and the encryption process. If the correlation factor is high enough, the attack can be termed as an SCA against the encryption algorithm. This paper focuses on the defense of encryption algorithms against Micro-architectural SCA. However, according to Alam et al. [4], attacks that do not target the protected encryption process are categorized as a safe state of the system.

To the extent of our knowledge, the RH attack is still an issue for modern DRAM. The most effective mitigation techniques rely on detection mechanisms that have a high silicon footprint and/or performance overhead. Based on our findings, Machine Learning (ML) has not been studied as a detection mechanism with performance in mind. Existing solutions only have a software implementation and a high processing time. The performances of ML as a Rowhammer detection mechanism have to be studied more in depth.

3 Methodology

3.1 Overview

In this paper, we propose a method to create an RH detection mechanism based on the categorization of hardware event traces as normal or abnormal (no attack or attack) by a Machine Learning (ML) model.

The creation methodology is divided into three steps: features selection, simulation, and training and testing the ML model. We also describe the final hardware implementation of the detection mechanism once it is built.

3.2 Features selection

We specify a list of hardware events that would be influenced by the RH attack. This list of features should allow the mechanism to distinguish between a benign behavior and an abnormal one. A good variety of features is important to have as much relevant information as possible to detect attacks. Using redundant features could make the implementation more expensive in terms of silicon surface and/or time to detect.

3.3 Simulation

Using gem5, we configure the architecture on which the mechanism will work. The simulator must be configured to output the desired features during a simulation.

A good variety of programs is chosen, including attack programs and hardware performance benchmarks to stress the components that would also be targeted by the attack (high memory usage).

Those programs are run either independently or sequentially on a single simulated processor, or concurrently on parallel simulated processors.

3.4 Machine learning offline training and testing

The feature traces are extracted from the gem5 output files. Logged events are transformed into fixed-duration samples, grouped into windows (fix-length buffers) with a percentage of overlap between two consecutive windows. Windows are labelled as attack or no attack.

Multiple ML models are trained and tested with the datasets. To select the model to use for the final implementation, the accuracy, time to classify when running in software and memory usage must be taken into account.

3.5 Detection mechanism implementation and operation

Once the ML model is able to distinguish normal behavior and attack, it can be implemented as an online detection mechanism and implemented in the system architecture. The created mechanism built with the proposed method is illustrated in Figure 1.

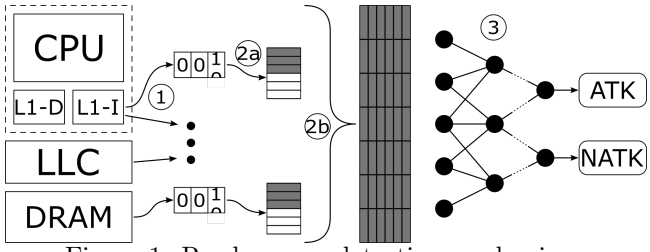


Figure 1: Rowhammer detection mechanism.

It can be separated into three steps noted by circled numbers in the figure: ① hardware counters, ② sample concatenation, and ③ classification.

①: Hardware counters have to be integrated in the system architecture to count how many times the selected features happen. This implies modifications on existing architectures to make features available to the mechanism.

②: Counters are regularly read, copied in predetermined-size buffers and cleared (a); once the buffers are full, they are copied in the input buffers for the ML model and partially cleared with respect to the overlap mentioned in Section 3.4 (b).

③: The implemented ML model classifies its input buffer as depicting a normal behavior or an abnormal one.

4 Experiments and results

4.1 System specifications

All the ML training and testing are performed directly on a laptop computer with an Intel® Core™ i7-8565U CPU @ 1.80GHz, 16GB RAM running Windows 10, version 19042.

The system simulation using gem5 (version 19) is running on a server. The gem5 system configuration we use, includes one or two DerivO3CPU or TimingSimpleCPU running at 1 GHz; two 32KB L1 caches per CPU: one instruction cache and one data cache; one 512KB L2 cache; and finally a DRAM at 2400 MHz as main memory, with a storage limit of 4 GB handled by the DRAM simulator Ramulator [15].

The number of CPU depends on whether we need to run two programs concurrently. The choice of the CPU model depends on our need for out-of-order CPU(s) or in-order CPU(s).

4.2 Features selection

The attack we want to detect is the RH attack. Its main goal is to disturb a DRAM row (the victim row) by repeatedly opening the neighboring rows (the aggressor rows). Aggressor rows have to be closed and reopened repeatedly in order to induce faults in the victim row. This translates on the hardware events level to a high number of row misses.

To access the aggressor rows, the attacker must first bypass all the cache levels, either using cache flush instructions if they are available to the attacker, or using cache eviction methods. At the hardware events level, cache flush instructions are seen as cache hits. So, there must be a high frequency of cache misses and, if the attack relies on cache flush, an equally high frequency of cache hits at the Last-Level Cache (LLC). As all LLC misses lead to DRAM row hit or row miss, monitoring LLC misses is redundant.

The first level cache is separated into an instruction cache (L1-I) and a data cache (L1-D). The L1-D will follow the LLC in terms of hits and misses, but as the

attack is typically a short loop of memory access instructions, once all the instructions of this loop are in the L1-I, there is no more L1-I miss. The number of L1-I misses will be very low compared to the number of L1-D misses.

Consequently, The set of features we choose to use for the detection is presented in Table 1.

Scope	Event	Alias
L1 Data Cache	Cache miss	L1-D-m
	Cache hit	L1-D-h
L1 Instructions Cache	Cache miss	L1-I-m
	Cache hit	L1-I-h
Last-Level Cache	Cache hit	LLC-h
DRAM	row buffer miss	DRAM-m
	row buffer hit	DRAM-h

Table 1: Features selection

4.3 Simulation on gem5

In order to create the datasets, we need a configurable system simulator, with the capability to output the desired features.

gem5 meets all these requirements. We can use it to configure the system by selecting the working frequency, CPU(s), caches and Random Access Memory (RAM). For the CPU, we mostly used the standard out-of-order CPU DerivO3CPU at 1GHz, but made some simulations with the in-order CPU TimingSimpleCPU at the same frequency. We used 2 levels of cache, and the DRAM simulator Ramulator. When using multiple cores, the CPU and the first level caches are duplicated for each core. A Simplified view of the system we created can be seen in Figure 2.

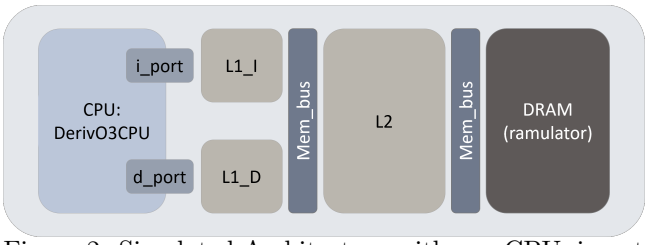


Figure 2: Simulated Architecture with one CPU. i_port and d_port are the ports for instructions and data, respectively.

For the simulated programs, we choose two different programs that will run either separately in two simulations, or concurrently on multiple CPUs. The first program is a combination of alternating attack and no attack loops presented in Source Code 1. The attack loop is very similar to the code presented by Kim et al. [14], with X and Y being constant addresses in two distinct DRAM rows. The no-attack loop is just a simple loop that accesses random addresses from a buffer. This loop runs much faster than the attack one due to a higher ratio of cache hits over cache misses. No-attack loops are run 30 times more on average to compensate and keep a comparable execution time.

This whole program runs multiple times (typically 10) with different loop duration to prevent time-aware models from learning the timing rather than characterizing the traces.

```
1  mov rand(), %ecx;random integer ∈ [5k; 25k]
2  atk_loop:
3      mov (X), %eax
4      mov (Y), %ebx
5      clflush (X)
6      clflush (Y)
7      loop atk_loop
8
9  mov rand(), %ecx;random integer ∈ [150k; 750k]
10 no_atk_loop:
11     ;random address in a 2^18-bytes array
12     mov rand(), %eax
13     mov (%eax), %ebx
14     loop natk_loop
```

Source Code 1: Pseudo-code for attack program.

The second program is the STREAM benchmark [18]. This program is made to test the performance of the memory. Both the attack code and the STREAM benchmark are memory-heavy programs, so the detection we are building will have to really recognize attack patterns and not only memory-heavy programs.

4.4 Training and testing datasets creation

The simulator generates a log file with the desired hardware events and their timestamps, and the program counter for every instruction. The event counters and the program counter are extracted from the simulator output file. The counters are used to generate datasets. Samples of fixed duration are created using the number of events happening in 100 ns, and concatenated into windows of 100 samples (10 µs). The last 50% of one window is used as the first 50% of the next window. The program counter is used to label the windows as attack or no attack.

4.5 Machine learning training and testing

At the time of writing, the models are trained and tested with datasets from multiple simulations in different load conditions: Isolated conditions and low load.

In isolated conditions, a single core is used to run either the custom program using source code 1, or a tuned STREAM benchmark [18], as seen in datasets 1 to 3 of Table 2. The generated dataset is labelled depending on the simulated program: the STREAM benchmark is labelled as no attack, and the custom program is labelled according to the loop being run.

In low load conditions, two cores are running in parallel, with the first core running the custom program and the second core running the STREAM benchmark, as seen in dataset 4 of Table 2. The generated dataset is labelled using the program counter of the first CPU running the attack program.

#	**simulation details**
1	CPU1 [in-order]: attack program (source code 1)
2	CPU1 [out-of-order (o3)]: attack program
3	CPU1 [o3]: STREAM benchmark
4	CPU1 [o3]: attack program CPU2 [o3]: STREAM benchmark

Table 2: Datasets Specifications

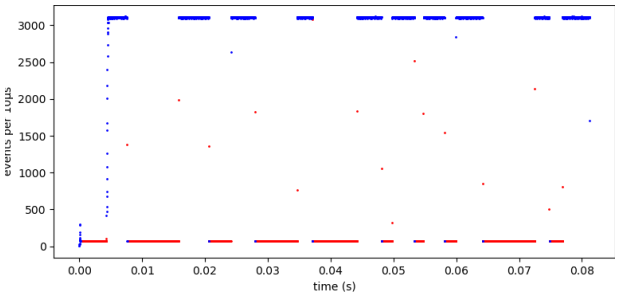
Load	**Datasets # used**	
	For Training	**For Testing**
Isolated	shuffle(1,2,3,4)	1,2,3
Low	shuffle(1,2,3,4)	4

Table 3: Datasets used for training and testing under different load conditions.

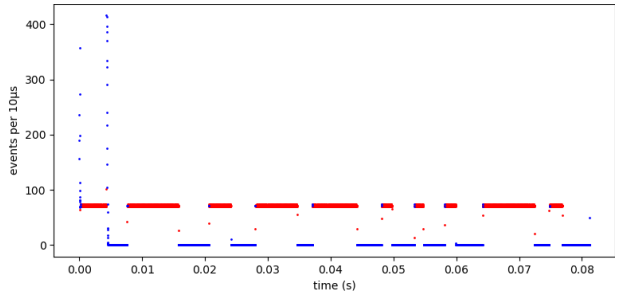
The samples for the LLC hits and misses and DRAM row hits and row misses from dataset 2 (attack program, isolated) of Table 2 are presented Figure 3. The same features for dataset 4 (low load) are presented Figure 4.

In isolated conditions, we can easily see on the traces that a simple threshold is enough to differentiate attack and normal behavior. However, when we introduce load, the feature patterns of attack and no attack are sometimes hard to distinguish, which makes simple threshold-based methods ineffective. ML models can distinguish the patterns they learned, which makes them great choices for this type of operation.

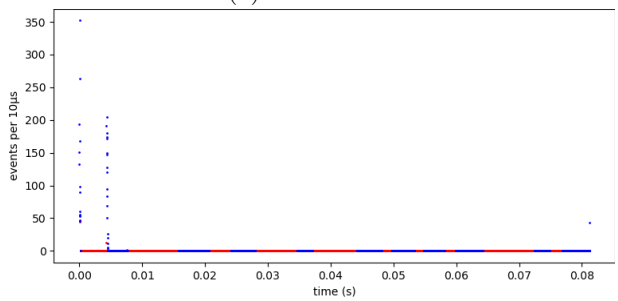
For training, sample windows from different simulations are shuffled to distribute the attack and no-attack



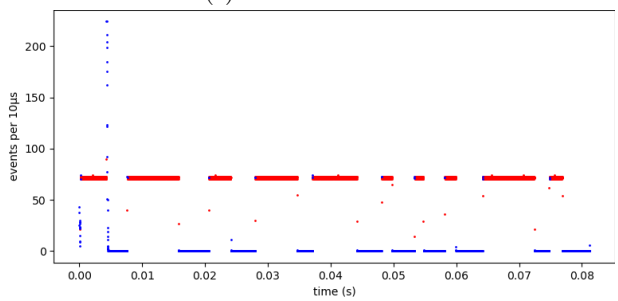
(a) LLC hits



(b) LLC misses

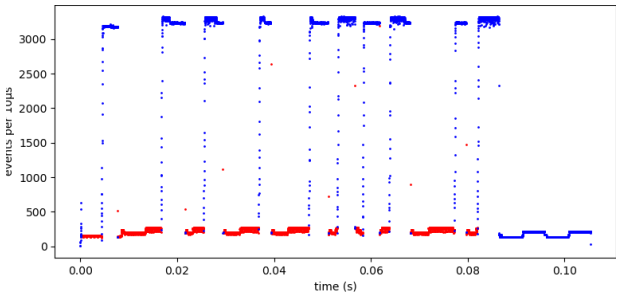


(c) DRAM row hits

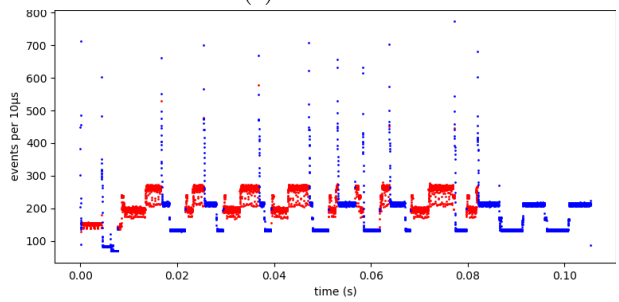


(d) DRAM row misses

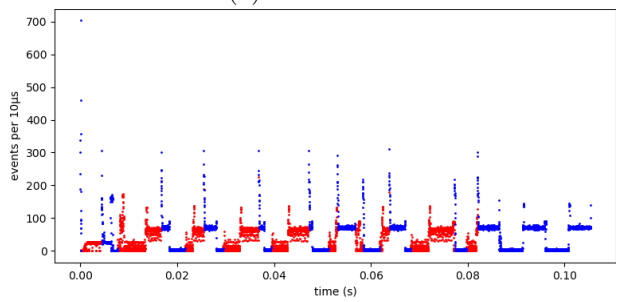
Figure 3: 10 µs samples evolution over time (in seconds) for custom program in isolated conditions. Red: an attack process is running; Blue: no attack process is running



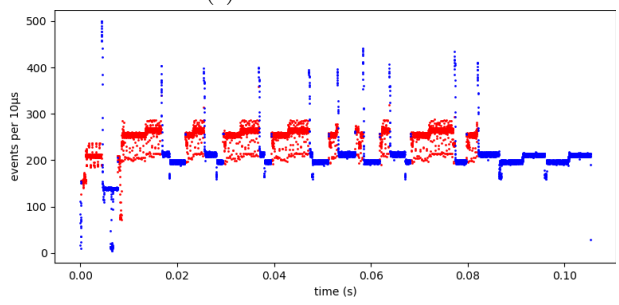
(a) LLC hits



(b) LLC misses



(c) DRAM row hits



(d) DRAM row misses

Figure 4: 10 µs samples evolution over time (in seconds) for custom program in parallel to STREAM benchmark. Red: an attack process is running; Blue: no attack process is running

portions evenly in the training set. When a set is used for both training and testing, only 50% of the windows are chosen randomly and used for training. Testing still uses 100% of the windows.

Three different ML models are tested: Long Short-Term Memory (LSTM), Multi-Layer Perception (MLP) and Convolutional Neural Network (CNN). All three models are modified models originally created by Jason Brownlee on Machine Learning Mastery [2]. LSTM and CNN models were originally intended for human activity recognition [8][6], and MLP for time series forecast [7]. The models have different characteristics in terms of performance and cost.

LSTM takes one sample (per number of features) as input, and recognizes the evolution of the different features across the window. It is easy to integrate in hardware because it does not scale with the window length, but takes much more time due to its recursive nature.

MLP is made from several fully-connected layers. Compared to the other ML models tested, it is the fastest when used in software, but can be hard to integrate in hardware due to its high number of interconnections.

CNN is a type of network used primarily to analyze visual images. One neuron in a layer is connected to a small region of the previous layer (its receptive field). Its speed in software is comparable to the MLP speed in software, and is much easier to integrate in hardware thanks to a relatively lower number of interconnections.

The models were built using Keras [1] with the python codes presented in Source Codes 2, 3 and 4. In these source codes, `n_timesteps` is the number of samples in a window (100), and `n_features` the number of different features logged by gem5 (7). All models take the 100×7 samples window buffer as input, and have 2 outputs (attack and no-attack).

```
model = Sequential()
model.add(LSTM(100,
        input_shape=(n_timesteps,n_features)))
model.add(Dropout(0.5))
model.add(Dense(100, activation='relu'))
model.add(Dense(n_outputs, activation='softmax'))
model.compile(loss='categorical_crossentropy',
        optimizer='adam', metrics=['accuracy'])
```

Source Code 2: Python code to build the LSTM model

```
model = Sequential()
model.add(Permute((2,1),
        input_shape=(n_timesteps, n_features)))
model.add(Dense(n_timesteps // 2))
model.add(Flatten())
model.add(Dense(128, activation="relu"))
model.add(Dense(n_outputs, activation="softmax"))
model.compile(loss='categorical_crossentropy',
        optimizer='adam', metrics=['accuracy'])
```

Source Code 3: Python code to build the MLP model

```
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=3,
        activation='relu',
        input_shape=(n_timesteps,n_features)))
model.add(Conv1D(filters=64, kernel_size=3,
        activation='relu'))
model.add(Dropout(0.5))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dense(n_outputs, activation='softmax'))
model.compile(loss='categorical_crossentropy',
        optimizer='adam', metrics=['accuracy'])
```

Source Code 4: Python code to build the CNN model

ML model	Load	Accuracy (%)	FP (%)	FN (%)	Software overhead
LSTM	I	99.970	0.028	0.002	216µs
	L	99.924	0.071	0.005	264µs
MLP	I	99.983	0.016	0.001	5.9µs
	L	99.881	0.119	0	7.1µs
CNN	I	99.975	0.024	0.001	38.5µs
	L	99.962	0.038	0	37µs

Table 4: ML models predictions performances

4.6 Results

Table 4 shows the results of the different ML models for different load conditions (I: Isolated, L: Low load). The accuracy is the percentage of windows the model is able to categorize correctly as attack or normal behavior. False Positives (FP, resp. False Negatives, FN) is the percentage of windows the models categorize as attack (resp. normal behavior), when it is a normal behavior (resp. an attack). The measured software overhead is the time necessary to categorize one window as attack or no attack on a single thread.

All three models show accuracy above 99.8%. Errors usually happen on one or two windows when transitioning between attack and no attack. This means that all three models are able to detect the attack when it starts using less than 50 µs of samples, and do not classify normal behavior as attack. This time is sufficiently lower than the minimum time necessary to flip a bit using an RH attack ($\backsim$10 ms). When implemented in software on the test machine, the slowest model (LSTM) takes less than 300 µs to categorize a 10 µs dataset window. The other models are faster with 10 µs and 40 µs for MLP and CNN respectively.

The hardware footprint and the speed to categorize one window will be taken into account when integrating the mechanism in hardware in a future work.

5 Conclusion

The RH attack has been a serious threat to memory security since its first appearance. Existing countermeasures are either performance-costly or require significant space on the silicon. Existing Machine-Learning based detection systems are not well studied. In this paper, we have introduced a vulnerability assessment mechanism which does the trace analysis using gem5 and binary classification of these behavior using Machine Learning. Our approach seems to offer very high accuracy for simple attacks with very low timing overhead when the system is run in software. The time it takes to detect the attack (maximum 264 µs per 10 µs window) is sufficiently lower than the minimum time to perform an attack ($\tilde{1}$0 ms). This solution provides a good RH detection mechanism that could be used to launch effective but performance-costly mitigation techniques.

6 Future work

At the time of writing, the proposed solution has only been tested with offline classification in software. We intend to extend our proof of concept to complex variants of RH vulnerabilities and we intend to perform online classification. We also plan to integrate our assessment platform at the hardware level by exploring vulnerabilities in different microarchitecture components i.e., branch predictors, prefetchers, etc.

Acknowledgements

The authors acknowledge the support of the French Agence Nationale de la Recherche (ANR), under grant ANR-19-CE39-0008 (project ARCHI-SEC). They also

acknowledge the French Ministère des Armées – Agence de l'innovation de défense (AID) under grant ID-UM-2019 65 0036.

References

[1] "Keras." [Online]. Available: https://keras.io/

[2] "Machine learning mastery," 2013. [Online]. Available: https://machinelearningmastery.com/

[3] "ARMOR: A hardware solution to prevent row hammer error in DRAMs," 2015. [Online]. Available: http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/

[4] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya, "Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks." *IACR Cryptology ePrint Archive*, 2017.

[5] "About the security content of Mac EFI security update 2015-001," Apple, 2015. [Online]. Available: https://support.apple.com/en-gb/HT204934

[6] J. Brownlee, "1D convolutional neural network models for human activity recognition," 2018. [Online]. Available: https://machinelearningmastery.com/cnn-models-for-human-activity-recognition-time-series-classification

[7] ——, "Deep learning for time series," 2018. [Online]. Available: https://machinelearningmastery.com/how-to-develop-multilayer-perceptron-models-for-time-series-forecasting/

[8] ——, "LSTMs for human activity recognition time series classification," 2018. [Online]. Available: https://machinelearningmastery.com/how-to-develop-rnn-models-for-human-activity-recognition-time-series-classification

[9] A. Chakraborty, M. Alam, and D. Mukhopadhyay, "Deep learning based diagnostics for rowhammer protection of DRAM chips," in *Proceeding of ATS*, 2019.

[10] P. Frigo, E. Vannacc, H. Hassan, V. v. der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the many sides of target row refresh," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2020.

[11] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," in *Proceedings of DIMVA*, 2016.

[12] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[13] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques," in *Proceedings of ISCA*, 2020.

[14] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *Proceedings of ISCA*, 2014.

[15] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, 2015.

[16] E. Lee, I. Kang, S. Lee, G. E. Suh, and J. H. Ahn, "TWiCe: Preventing row-hammering by exploiting time window counters," in *Proceedings of ISCA*, 2019.

[17] M. Lipp, M. Schwarz, L. Raab, L. Lamster, M. T. Aga, C. Maurice, and D. Gruss, "Nethammer: Inducing rowhammer faults through network requests," in *Proceedings of EuroS&PW*, 2020.

[18] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995.

[19] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-CPU attacks," in *Proceedings of USENIX Security Symposium*, 2016.

[20] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," *Black Hat*, 2015.

[21] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Mitigating wordline crosstalk using adaptive trees of counters," in *Proceedings of ISCA*, 2018.