# MaDMAN: Detection of Software Attacks Targeting Hardware Vulnerabilities

Nikolaos Foivos POLYCHRONOU[1], Pierre-Henri THEVENON[1], Maxime PUYS[1], Vincent BEROULLE[2]

[1]*Univ. Grenoble Alpes, CEA, LETI, LSOSP*, Grenoble, France
Firstname.Name@cea.fr
[2]*Univ. Grenoble Alpes, Grenoble INP, LCIS*, 26000, Valence, France
Firstname.Name@lcis.grenoble-inp.fr

***Abstract*—The increasing complexity of modern microprocessors created new attack areas. Attackers exploit these areas using Software Attacks Targeting Hardware Vulnerabilities (SATHV) such as Cache Side-Channel, Spectre, and Rowhammer attacks. These attacks target the microarchitecture to extract privileged information. As their target is the hardware, antivirus programs cannot detect them. But, they modify the normal behavior of the microarchitecture. Modern systems are equipped with hardware performance counters (HPCs), which measure events related to hardware components. Designers can take advantage of these counters to monitor and protect the system. In the literature, there exist many solutions that use HPCs to detect SATHV. But, due to the limited number of counters, proposed solutions only protect the microprocessor against a limited set of SATHV. In contrast, we propose MaDMAN, a Malware Detector, which gathers information from HPCs to detect a large set of SATHV. MaDMAN uses a Logistic Regression classifier. In our threat model, we include Cache Side-Channel, Rowhammer, and Spectre SATHV. Our detection mechanism succeeds to detect these attacks with 98.96% accuracy, 96.3% F-score, and 0% false positive rate. In addition, MaDMAN works in noisy environments and can detect successfully evasive malware.**

***Index Terms*—security, hardware performance counters, attacks, malware, microarchitecture, detection**

## I. Introduction

In the past, attackers exploited software attacks targeting software vulnerabilities or hardware attacks targeting hardware vulnerabilities. Software attacks exploit flaws or defects in the software code. They allow attackers to exploit the Operating System (OS) or applications in the system, to obtain some privileges. On the other hand, hardware attacks target flaws present in the hardware components of the system. Hardware attacks allow attackers to directly exploit the interaction with the system electronic components, without relying on a software vulnerability and independently of the OS. Apart from traditional hardware attacks such as fault attacks or side-channel attacks, attackers also found ways to exploit hardware vulnerabilities using software code. We refer to this new class of attacks as Software Attacks Targeting Hardware Vulnerabilities (SATHV).

SATHV exploit various mechanisms in the system. Some examples are the Branch prediction, Out of Order (OoO) execution, Dynamic Voltage and Frequency Scaling (DVFS), Direct Memory Access (DMA), and cache memories. Attackers exploit these hardware mechanisms to extract information from the system. As SATHV target hardware vulnerabilities, they are difficult to patch. Fixing these vulnerabilities often require redesign of the microarchitecture or of some hardware macro-components.

To defend against software attacks targeting software vulnerabilities, classical tools such as antivirus tools secure the system. However, antivirus tools can not detect and secure the system against attacks targeting hardware vulnerabilities. This is because SATHV behave as normal software applications and do not leave traces in system log files. In contrary, SATHV target hardware vulnerabilities not observable by a software application. As SATHV can be performed remotely and being undetected, we need to find ways to address them.

To address SATHV researchers proposed online detection mechanisms [1]–[3] that utilize *Hardware Performance Counters* (HPCs). HPCs are special-purpose registers integrated in most modern architectures. They store the counts of hardware specific events, such as branch mispredictions, cache misses, memory accesses, load, and store operations. They are primary used for performance analysis, tuning, and debugging. The number of available counters and events is platform specific.

But all proposed detection mechanisms are specific to some SATHV and cannot detect all the possible SATHV applicable in the system. As the number of available attack vectors increases, attackers can bypass detection mechanisms by using attack variants not considered by the detection model. Such examples are SATHV using eviction strategies or evasive SATHV. Evasive SATHV try to avoid detection inserting nop instructions or *sleep()* during the attack. It is critical to consider attackers that will try to hide their malicious activity. If detection mechanisms take into account only a small set of SATHV, the system is unsafe against the remaining attacks. It is necessary to develop a mechanism that will be able to detect with high accuracy most of the SATHV in the targeted platform and not only a limited subset. This is a complex task, as the number of HPCs is limited and the available events are different from platform to platform. Moreover, finding HPC events that correlate for all the attacks is difficult.

Our contributions in this paper are the following: we study HPC events proposed in the literature for the detection of SATHV and we present how well they detect a large set

of SATHV. We show that using eviction instead of cache maintenance instructions modifies the expected behavior of proposed HPC events. In addition, we propose MaDMAN, (Malware Detector - Monitor, Act, Notify). As far as we know, MaDMAN is the first ARMv7 detection mechanism detecting Spectre, Rowhammer, and CacheCSA attacks. MaDMAN is a software based detection mechanism based on Logistic Regression classification algorithm [4]. MaDMAN utilizes data from the HPCs to decide on the presence of malicious behaviors. Our detection mechanism detects SATHV, including evasive SATHV.

The rest of the paper is organized as follows: In section II we provide background information on SATHV. After section III presents related work. Then section IV, we present our methodology regarding our data collection approach, detection methodology, and evaluation. Section V exposes the applicability of state of the art side effects for the detection of SATHV in our experiments. Also, we expose some new features we use for our SATHV detection and our mechanism detection capabilities. Finally, section VI summarizes our work and concludes.

## II. Background

Cache side-channel, Spectre, and Rowhammer attacks are well studied SATHV. In this section, we will briefly describe the attacks and targeted vulnerabilities.

Cache side-channel attacks target the latency of accessing the main memory or the cache memory. If the requested data are inside the cache memory, the access time is quicker than accessing data stored in the main memory. Most of the cache side-channel attacks target this feature to extract confidential information [5], [6]. But another cache attack presented in [7] targets the latency of the *flush* instruction. If the requested address to *flush* is in the cache memory, *flush* instruction takes more time to execute than when the requested address is not present.

Rowhammer attacks target the main memory [8]. By repeatedly accessing the same DRAM row, inside a time interval less than the refresh interval of the DRAM, there is the possibility of inducing bit flips in the neighboring rows. If we repeatedly access the two neighbors of our victim row, there is a higher probability of inducing bit flips than one sided hammering [9]. Using a *flush* instruction to bypass the cache, enables us to execute the attack loop faster, increasing the probability of inducing faults. Gruss et al. [10] showed eviction based rowhammering is possible.

Spectre [11] targets the branch predictor. When the branch predictor predicts the outcome of a branch, if finally, the outcome is correct the Central Processing Unit (CPU) continues its operation. If the prediction was wrong, the CPU flushes the pipeline, restores the context before the prediction, and continues with the correct data. But the residual data stay in the cache. Spectre involves inducing a victim to speculatively perform operations that would not occur during correct program execution. Then, victim confidential information leaks via a side channel to the adversary.

## III. Related work

In the literature, we find different methods related to SATHV detection. Li et al. [1] proposed a detection mechanism, which uses HPCs to detect Rowhammer and Spectre attacks. Their monitor uses the HPCs to gather the necessary information to train the machine learning classifier. Chiappetta et al. [2] and Cho et al. [3] use the HPCs and machine learning to detect Cache Side-Channel attacks (CacheCSA). Chiappetta et al. implement a detection mechanism that could be run in user space, read the HPCs and feed the data in a classifier to decide for the existence of CacheCSA. Two of the three methods they proposed are based on machine learning. Cho et al. use multi-label classification to detect three variants of CacheCSA. Collecting data from the HPCs induces relative low performance overhead, is easier and less complex compared to higher level features of the OS and applications. The proposed mechanisms show the potentials of using HPCs and online analysis to detect with good accuracy a set of SATHV. But, all the related work consider the detection of a limited set of SATHV.

## IV. Methodology

From previous works, we can see it is possible to detect SATHV using HPCs. We benefit from HPCs to measure the stress that malicious applications induce in the hardware components of the microprocessor. We developed a testbench [12] that allows experimenting and evaluating different side effects. In this case, the side effects represent the differences in the behavior of HPCs in the presence or not of a malicious vector. In [12], we presented the testbench which allows us to measure the efficiency of State Of The Art (SOTA) solutions in different architectures. We studied simple threshold-based implementations targeting to detect only CacheSCA and Rowhammer. In contrast, in this paper we use Machine Learning implementations. Also, we chose SATHV that are most suitable, representative, and feasible in our platform. To compare the different HPCs behaviors between normal and malicious applications, we chose normal applications that best illustrate applications running in our target.

### A. Data collection

To collect the information from the HPCs, our monitoring module is executed parallelly to the applications. The monitoring Algorithm in Listing 1 is a loop that resets the HPCs, sleeps for the specified monitoring interval, reads the information from the HPCs, and writes the results in a .csv file. We extract and use the obtained results to analyze how the attack vectors modify the behavior of the different available hardware events. Besides, we use this dataset to train and test our detection mechanism. During the reading of the HPCs and the writing of the .csv files we disable the counters to eliminate the effects of our monitoring algorithm.

### B. Event selection

Because the available HPCs are limited, one must choose carefully which events to monitor. Further, we must determine

**Algorithm 1** Monitoring algorithm to extract information from the HPCs

```
configure HPCs
while running application dataset do
    reset HPCs
    nanosleep(monitoring interval)
    disable HPCs
    read HPCs and write to csv file
    re-enable HPCs
end while
disable HPCs
```

the monitoring interval between consecutive measurements, and the detection technique applied to decide if the sample is malicious or not. All together must provide a detection mechanism with a low performance overhead, high accuracy, and a very low false positive rate.

### C. Machine Learning Classifiers

We use machine learning algorithms to predict the class of the set of HPCs information. In our case, we have two classes or labels i.e., malicious or normal. The classification predictive modeling is a task that approximates a function *f* that maps an input dataset *x_input* to an output dataset *y_output*. This is a category of supervised learning as we provide the targets with the input data [13]. Our detection mechanism uses pre-labeled data for training. Except for the training, there is the testing phase, to evaluate the detection capability of our classifier. For our mechanism, we chose a Multi-linear Logistic Regression [4] classifier because of its simplicity. We can visualize logistic regression as an one-layer neural network [14] as we can see from Fig. 1. Logistic regression tends to limit the cost function between 0 and 1. The inputs $x_i$ are the HPC values and the $w_i$ are the weights associated with each input. The normalized output y is between 0 and 1. Then, a threshold is chosen with normalized outputs greater than the threshold labeled as malicious. The final output is a binary decision, *'0'* for normal and *'1'* for malicious applications.

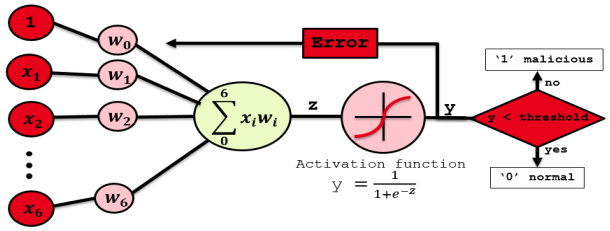


Fig. 1: Visualization of a Logistic Regression classifier.

### D. Exponentially Weighted Moving Average

For our online detection mechanism, we employ a similar method such as in [1], [15]. The Exponentially Weighted Moving Average (EWMA) technique helps to distinguish between normal and malicious behaviors and reduces false positives. EWMA is an algorithm that continuously computes a type of average for a series of measurements, as the measurements arrive. After a value in the series is added to the average, its weight in the average decreases exponentially over time. This biases the average towards more recent data. EWMAs are useful for several reasons, chiefly their inexpensive computational and memory cost. As well, they represent the recent central tendency of the series of values [16]. The EWMA algorithm requires a decay factor, $\alpha$. The larger the $\alpha$, the more the average is biased towards recent history and discounts older observations faster. The $\alpha$ must be between 0 and 1. We calculate EWMA as follows:

$$S_t = \begin{cases} Y_1, & \text{t = 1} \\ \alpha Y_t + (1-\alpha) \cdot S_{t-1}, & \text{t >1} \end{cases} \quad (1)$$

where $Y_t$ is the value at a time period t and $S_t$ is the value of the EWMA at any time period t. As so, EWMA behaves as the sliding window of size N in Fig. 2 for $\alpha = 2/(N+1)$. In [15] they showed that classification over windows of execution can also separate malicious from normal programs. With the segmented data of the sliding window, we calculate the average of consecutive decisions in the current window. If the average is above a certain threshold, we can conclude a malicious process is in progress. We use binary decisions which consist of *0's* for a normal and *1's* for a malicious application. The sliding window size is crucial for the detection accuracy of our online detection mechanism. A small window could result in high false positives, and a big window could result in high false negatives because in a bigger window it is more likely to miss the malicious behavior. To further reduce noise from our measurements, we also apply EWMA in the HPCs measurements.

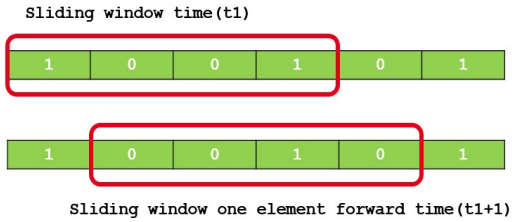


Fig. 2: Sliding window example for decision making

### E. Evaluation of the detection performance

The primary criterion to compare the detection mechanisms or to identify if a mechanism is effective or not should be detection accuracy. To calculate detection accuracy, we must initially identify if our measurements are labeled correctly. True Positives ($T_P$) are the observations corresponding to actual attacks predicted as attacks. True Negatives ($T_N$) are the observations corresponding to normal operations predicted as normal operations. False Positives ($F_P$) or False alarms are the observations corresponding to normal operations predicted as attacks. False Negatives ($F_N$) are the observations corresponding to actual attacks predicted as normal operations. Finally, accuracy is defined as:

$$Accuracy = \frac{T_P + T_N}{T_P + T_N + F_N + F_P} \quad (2)$$

Another metric is the F-score, which measures the global trade-off between precision and sensitivity, and is defined as:

$$F_{score} = \frac{2 * (precision * sensitivity)}{precision + sensitivity} \quad (3)$$

where precision is the proportion of positive observations that truly are positive and sensitivity (or true positive rate) is a metric that measures how well we identify True positives:

$$precision = \frac{T_P}{T_P + F_P} \quad (4) \quad sensitivity = \frac{T_P}{T_P + F_N} \quad (5)$$

This metric is used as some mechanisms are very good at detecting an attack, but they are too much sensitive, adding a high number of False alarms. Additional metrics which will assist us are the Specificity (or true negative rate) and $G_{MEAN}$ and are defined as:

$$specificity = \frac{T_N}{T_N + F_P} \quad (6)$$

$$G_{MEAN} = \sqrt{specificity * sensitivity} \quad (7)$$

Specificity is a metric that measures the proportion of normal operations identified correctly. $G_{MEAN}$ is a metric that searches for a balance between sensitivity and specificity.

Another feature we can use to evaluate the performance of our mechanism is the Receiver Operating Characteristics (ROC) curve. We use ROC curves to visualize the comparison of different classification models. We plot the true positive rate and the false positive rate for different thresholds, and we create the curve plotting the scores in a line of increasing thresholds. ROC shows the trade-off between the true positive rate and the false positive rate. The area under the ROC curve (AUC) is a measure of the accuracy of the model. A model closer to the diagonal line is less accurate and a model with 100% accuracy has an AUC of 1 [13]. In other words, the optimum result would be the point (0, 1) indicating 0% false positives and 100% true positives. Besides, the ROC curve can help in deciding the optimum threshold. From the ROC curve, we seek the threshold with the highest $G_{MEAN}$.

## V. Results

In the following section, we will present our experimental platform, the HPC event selection, and MaDMAN. Finally, we evaluate the detection performance of MaDMAN.

### A. Experimental platform

Our experimental platform is based on the evaluation board Zybo Z7-20. Zybo Z7-20 is an embedded software and digital circuit development board based on the Xilinx Zynq-7000 family. It integrates a dual-core ARM Cortex-A9 processor running at 667MHz. It is equipped with 1GB of DDR3L memory and a Debian GNU/Linux 10. The ARM Cortex-A9 processor is a 32-bit processor core implementing the ARMv7-A architecture. In ARMv7 cache maintenance operations can only be executed in privileged modes. From userspace we need to use eviction techniques [17] to *flush* a targeted address from the cache memories.

Our attack libraries include CacheCSA, Spectre, and Rowhammer attacks. The CacheCSA include Evict+Time, Prime+Probe, and Evict+Reload targeting the AES T-Table implementation. Our attack library includes both one and double sided Rowhammer attacks.

For our normal applications library we include MiBench [18] and PARSEC [19] suites. MiBench suite illustrates applications used in embedded systems, targeting different areas of the embedded market. Some of these are industrial control, networking, security, and telecommunications applications. On the other hand, PARSEC suite illustrates applications on emerging workloads, such as financial, deduplication, computer vision applications. These two suites combined represent a diverse range of applications.

### B. Event selection

We use HPCs to extract information. HPCs act in our case as sensors in the microprocessor. The ARM Cortex-A9 core implements 55 events, but only allows counting 6 events simultaneously. As we cannot monitor more than 6 side effects concurrently, we have limited information to try to detect multiple SATHV. This is our major limitation and drives us to carefully choose which hardware events to monitor.

Before the selection of the hardware events to monitor, we needed to answer what is the optimum time interval between our measurements. Too slow and the malicious application could hide its behavior, too quick, and the performance overhead of our detection mechanism would be unacceptable. We tested monitoring intervals less than 50ms. We did not experiment with greater monitoring intervals, as we need to react quicker than the attack can succeed. The attack that needs the least time to succeed in harming the system is the Rowhammer attack, which can induce bitflips in less than 64ms. We adopted the monitoring interval of 1ms for the following reasons: first, decreasing the monitoring interval to less than 1ms, we observe no supplementary information. The side effects tend to decrease proportionally. Second, the least amount of time a process can run is defined by *sysctl_sched_min_granularity* and is by default 750µs. Timing intervals less than 750µs give us more information than we need. Third, we use the *nanosleep()* to create the monitoring interval. We observed monitoring intervals less than 1ms were noisy. The time between consecutive measurements was different from the one we defined. To eliminate the extra noise, we prefer monitoring intervals greater than 1ms. Finally, to cover the worst-case scenario, we must use a monitoring interval closer to the minimum granularity. In this way, the attackers cannot hide their activity executing with the minimum granularity. If multiple processes execute concurrently with the minimum granularity, and we monitor with a too large monitoring interval of 5ms, then our measurements could be a mix of multiple processes. The aforementioned reasons let us choose the monitoring interval of 1ms.

To find the optimum set of hardware events to monitor, we examined the different behaviors of each event to malicious and normal applications. To visualize the interquartile range

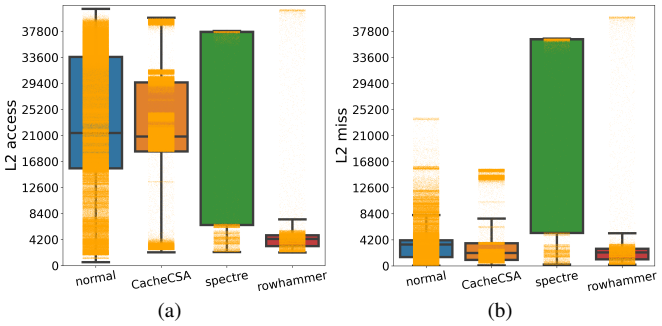


Fig. 3: (a) L2 accesses, (b) L2 misses

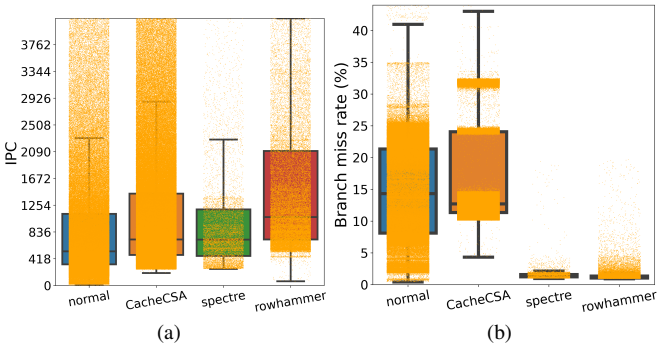


Fig. 4: (a) IPC, (b) Branch miss rate.

(IQR)(Fig. 3a the boxes) and median (Fig. 3a the horizontal line inside the boxes) of the measured HPC values for each individual SATHV, we use boxplots [20]. From the boxplots, we can get a first indication for the feasibility of detection using the targeted HPC events.

The starting point was hardware events proposed in the literature. First, we used hardware events proposed in [1], [3]. The two mechanisms propose side effects for CacheCSA, Spectre, and Rowhammer attacks. They are both implementations in Intel x86 platforms and both use attack vectors based on *flush* instruction. We analyze if the proposed set of hardware events perform as well in our platform using eviction based attacks. Cho et al. [3] proposed the use of the Instructions Per Cycle (IPC), Level 1 Cache (L1) misses, and Level 2 Cache (L2) misses. From Fig. 4a the median of the IPC is slightly increased during the attacks but we see no observable difference. From Fig. 3b, the median of L2 misses is increased for Spectre compared to normal applications. On the other hand, the median decreases for CacheCSA and Rowhammer compared to normal applications. We observe a lot of false positives, as applications such as the automotive applications in MiBench increase the IPC and L2 misses. Another point is that L2 misses and L2 miss ratios increase during the context switch or when an application runs for the first time. Relying only on cache statistics increases the false positives. Li et al. [1] proposed the branch miss rate and Last Level Cache (LLC) miss rate for the detection of Spectre and Rowhammer. As we observe from Fig. 4b, the median branch miss rate of Spectre and Rowhammer attacks is indeed decreasing compared to normal applications. As well, the median of the L2 miss rate is also increased during Spectre and Rowhammer compared to normal applications. With a first look, we can say that the L2 and branch miss ratio seem to be good indicators for the detection of Spectre and Rowhammer. But again, we observe that some MiBench applications exhibit a low branch miss rate and a high L2 miss ratio, which increases the False positives. In addition, we need 4 hardware events to calculate these two indicators. Furthermore, we can see in Fig. 3, 4b, and 5a that they are not reliable indicators for CacheCSA detection, and this limits our detection capability for CacheCSA to 2 hardware events. Fig. 5b shows the obtained ROC curve for a logistic regression classifier trained with this set of hardware features

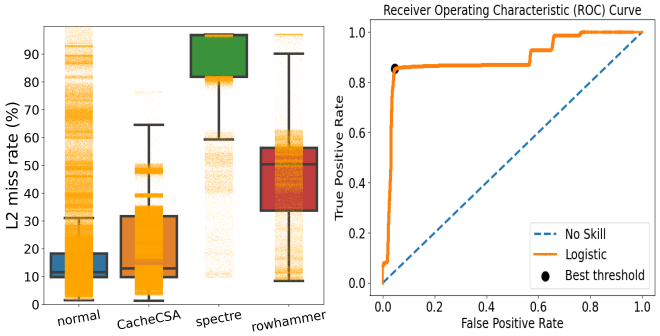


Fig. 5: (a) L2 Cache miss rate, (b) The receiver operating characteristic curve obtained for a logistic regression classifier using the hardware events used in [1].

for the detection of Spectre, Rowhammer, and CacheCSA attacks. We can observe the classifier using this set of events has a true positive rate of less than 90%.

To detect CacheCSA, Rowhammer, and Spectre attacks we observe that the instruction cache misses median decrease compare to normal applications. We can observe in Fig. 6b that the median of normal applications is higher than the attacks. We added in the figure the median values because the median line in the CacheCSA boxplot is not visible. Further, Spectre and Rowhammer attacks have small variations as their boxplots are very narrow. CacheCSA have a wider variation, but this is because we label as malicious the final analysis. The analysis part of the attack script performs normal operations, which add instruction cache miss measurements closer to the median of normal applications.

In Fig. 7a, we can observe that the median of the percentage of Translation Lookaside Buffer (TLB) allocation due to a data TLB request by L2 misses is very low for all the attack vectors compared to normal applications. Moreover, the boxplots of normal and malicious applications do not overlap. This is because attacks target specific address ranges and they do not need to bring a new translation frequently in the TLB. When the attack vectors miss in all levels of cache, they miss in data addresses that the translations exist in the TLBs and they are frequently used. On the other hand, normal applications are

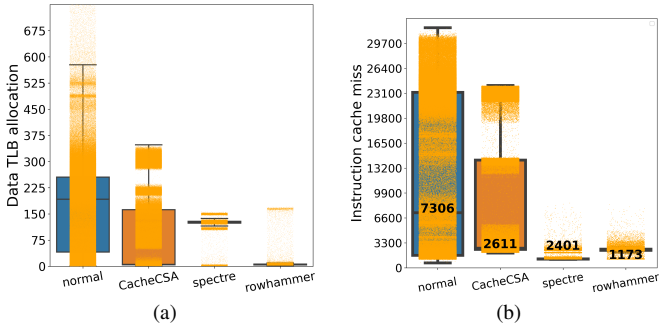


Fig. 6: (a) Translation Lookaside Buffer (TLB) allocation due to a data TLB request, (b) L1 Instruction cache misses.

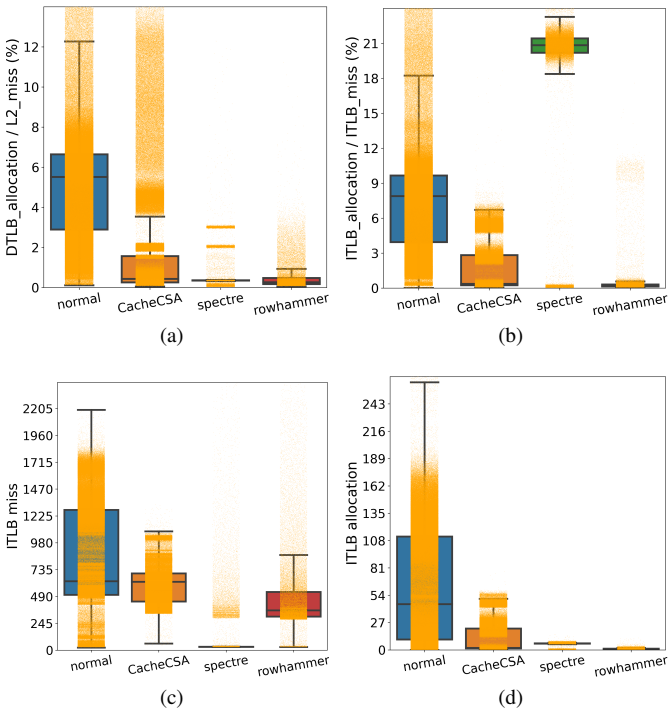


Fig. 7: (a) percentage of TLB allocations due to a Data TLB request (DTLB) by L2 misses, (b) percentage of TLB allocations due to an Instruction TLB request (ITLB) by Instruction TLB misses, (c) Instruction TLB misses, (d) TLB allocations due to an Instruction TLB request.

more versatile, and when missing in all levels of caches, it is more probable to be for a data address not used before. The TLB must then allocate the missing translation information. We also observe, that for Rowhammer and CacheCSA attacks the median of the percentage of TLB allocations due to an instruction cache request by the instruction TLB misses decreases. On the other hand increases for Spectre attack. For Rowhammer and CacheCSA attacks, this is because the attacks use a small code. For Spectre increases despite the small code, as during Spectre, we evict all the data from the cache, which evicts instructions as well. If the translation is evicted from the TLB due to the replacement algorithm, when we need to use the instructions of the loop, we will need to bring again the translation information in the TLB. This indicator is performing excellently, as the boxplots of malicious and normal operations do not overlap. The two events can substitute the branch miss rate and L2 miss rate, as the one shows how small the instruction code is, and the other shows that malicious code miss in all levels of cache accessing frequently used data addresses. We also observed, during context switch, both features increased reducing the false positive rate in this case.

After our analysis, the 6 hardware events we choose to monitor are L2 misses, Instruction cache misses, Data cache misses, Instruction micro TLB miss, Instruction TLB allocation, and Data TLB allocation. We calculate also the two percentages mentioned before i.e the percentage of TLB allocations due to a Data TLB request divided by the number of L2 misses, and the percentage of TLB allocations due to an Instruction TLB request divided by the Instruction TLB misses. To eliminate unnecessary noises in our measurements, we applied EWMA.

### C. MaDMAN implementation

MaDMAN is a multivariable logistic regression classifier. To train MaDMAN, we use 30% of our measurements set and we keep 70% for testing it. In addition, from our normal application pool, we randomly choose half of the applications for our analysis, and the rest we used for online testing. We do this as logistic regression is susceptible to overfitting.

We tested our classifier both in noiseless and noisy environments. We refer to a noiseless environment when only one application is running in the core, and another can only execute after the previous finishes. This is more realistic in IIoT devices. A noisy environment, on the other hand, is an environment when multiple processes contest the core for execution time. To demonstrate an environment like this, we randomly execute normal applications, from our application pool. We experimented with a maximum of 4 contesting applications.

To choose the decision window, we experimented with different sizes. When we use small decision windows, we observe a high False Positives Rate (FPR) during execution in a noisy environment. We can see in Fig. 8 the False positive rate (blue line) is 7% for small window sizes. When we use small decision windows, we assign more weight to current decisions. If our classifier makes a wrong prediction, EWMA assigns more weight to the wrong prediction increasing the False positive rate. On the other hand, when we increase the decision window, current measurements have less weight and our final output depends on the past measurements as well. Increasing the decision window, we observe that the False positive rate decreases to less than 1%. Increasing the decision window size impacts the False negative rate. When we increase the decision window size, the false negative rates increase as we can observe in Fig. 8 with the red and purple lines. This is because it is likely we miss the malicious behavior when multiple outputs from the classifier are used to calculate the

final decision. Our decision window will also include normal decisions, shifting the average closer to a normal application than malicious. Apart from this, a malicious code includes normal operations apart from the malicious operations as well. We chose the window size that balances the two ratios in the noisy environment. Fig. 9 shows the F-score for the different sizes of decision and measurements window. The red circles highlight the two points that gave us a F-score of 100%. We choose the decision window of 4 and the measurement window of 9 for our detection mechanism. This set of window sizes gives an F-score of 100%, as we see highlighted in red in Fig. 9.

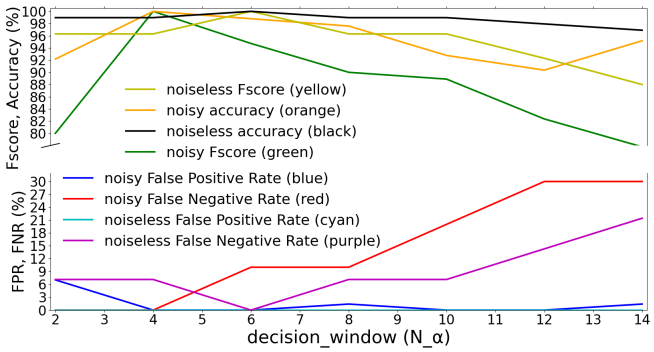


Fig. 8: Accuracy, False Negative, and False Positive rates regarding the decision window size in the noisy and noiseless environments.

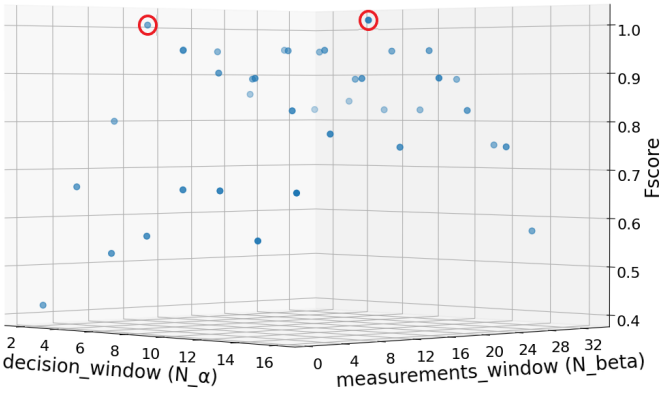


Fig. 9: F-score depending on the measurement window and decision window. The two red circles highlight the window sizes (decision_window= 4, measurement_window = 9) and (decision_window = 16, measurement_window = 9)

### *D. Evaluation of the detection performance of MaDMAN*

Our classifier performs with high accuracy both in noiseless and noisy environments. From Fig. 10, we can see the ROC curve for our detection mechanism under a noiseless environment. Our mechanism has an AUC of 0.983 and F-score of 98.72%. Because we label as malicious all the SATHV code execution, to calculate the evaluation metrics we use metrics at an application level. For example, if an application is labeled as malicious or normal and not the obtained measurements. The accuracy and F-score for the noisy environment is 100% for the decision window of 4. For the same decision window, in the noiseless environment the detection mechanism has an accuracy of 98.96% and F-score of 96.3%. This difference is because our metrics are sensitive to context switching, performing better in a noisy environment. Despite this, the false positive rate remains at 0%.

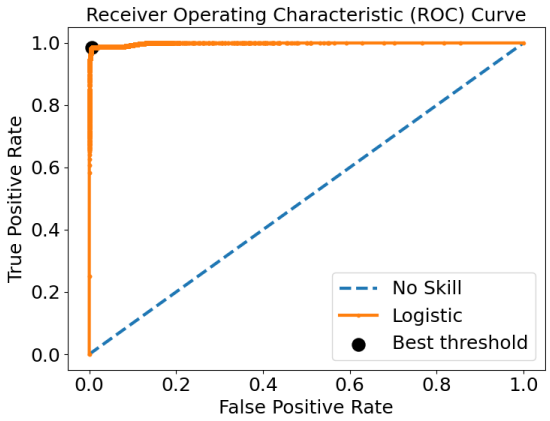


Fig. 10: The receiver operating characteristic curve obtained for our classifier.

To test our classifier against attack variants, we had the following approaches as in [21]. To begin with, we trained our classifier by using the quickest eviction strategy we found for our platform. Next, we created some attack variants with slower eviction strategies and some based on the insertion of nop instructions and random sleep functions during code execution. The insertion of nop instructions and sleep functions was placed carefully during atomic tasks, so as the attacks still succeed. By inserting this extra code, we modify the expected behavior of the hardware event. The goal is to obtain a behavior close to nominal. The detection mechanism of Li et al. [1], [21], detect with 0.23% false negatives for Rowhammer and 3.83% false negatives for Spectre, using Logistic Regression. When they tested their detection mechanism with an evasive Spectre attack, their detection accuracy fell less than 90%. Our classifier was capable to detect all the evasive SATHV, with no false negatives. The difference between our mechanism and the one in [1], except the hardware events, is the time interval between consecutive measurements. Li et al. used *perf* tools to extract the hardware features. Using *perf* we can monitor with the minimum time of 100ms, as the researchers do. As we monitor 100 times faster, we are more likely to detect malicious activities, in the presence of evasive techniques.

The performance overhead of a proposed detection mechanism is a major concern since it affects all the applications running on the system. We evaluated our detection mechanism in noisy and noiseless environments. We run the same set of applications with and without the detection mechanism, measuring the execution time using the Linux *time* command. The performance overhead of our detection mechanism is 1.3%.

## VI. Conclusion

SATHV can be dangerous for our systems. They gain popularity, as they can bypass current software protections and as it can be very difficult to patch a system with hardware vulnerabilities. This study presented MaDMAN, a detection mechanism that implements multivariable logistic regression to classify normal and malicious measurements. MaDMAN extracts the measurements from hardware performance counters, which provide information about the behavior of the system under test. MaDMAN, an ARMv7 based security mechanism, detects SATHV based on eviction techniques. Our threat model includes CacheCSA, Spectre, and Rowhammer attacks. Prior works did not consider eviction based approaches, as cache maintenance instructions were available to userspace. Eviction based techniques modify the expected behavior of proposed hardware events in the state of the art. We presented some new hardware events and indicators that help MaDMAN detect with high accuracy SATHV, and reduce the false positive rate during context switching. MaDMAN performs in both noiseless and noisy environments. It can detect attack variants and evasive malware with an accuracy of 98.3% and 1.3% performance overhead. MaDMAN decides in regards to past observations using EWMA. This sliding window technique further improved the false positive rate.

As there exist many more SATHV, a detection mechanism should be able to detect with high accuracy all of them and preferably independently of the platform. Because the information we can receive is limited to the available HPCs, the detection of all the SATVH is challenging. Our platform is currently limited to malicious codes that use eviction based techniques and is specific to ARMv7 architecture. Our goal is to mitigate in a platform that supports more SATHV variants, such as Flush+Flush attack. *Flush* based attack vectors are well studied in the literature. Adding eviction based attack vectors on top of *flush* based approaches will cover a greater attack surface than the proposed mechanisms. Further, we plan to experiment on different platforms. The goal is the implementation of a mechanism that can migrate to different platforms and stay effective.

## References


[1] C. Li and J. Gaudiot, "Detecting malicious attacks exploiting hardware vulnerabilities using performance counters," in *43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, Milwaukee, WI, USA, July 15-19, 2019, Volume 1*, V. Getov, J. Gaudiot, N. Yamai, S. Cimato, J. M. Chang, Y. Teranishi, J. Yang, H. V. Leong, H. Shahriar, M. Takemoto, D. Towey, H. Takakura, A. Elçi, S. Takeuchi, and S. Puri, Eds. IEEE, 2019, pp. 588–597. [Online]. Available: https://doi.org/10.1109/COMPSAC.2019.00090

[2] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Appl. Soft Comput.*, vol. 49, pp. 1162–1174, 2016. [Online]. Available: https://doi.org/10.1016/j.asoc.2016.09.014

[3] J. Cho, T. Kim, T. Kim, and Y. Shin, "Real-time detection on cache side channel attacks using performance counter monitor," in *2019 International Conference on Information and Communication Technology Convergence, ICTC 2019, Jeju Island, Korea (South), October 16-18, 2019*. IEEE, 2019, pp. 175–177. [Online]. Available: https://doi.org/10.1109/ICTC46691.2019.8939797

[4] M. Maalouf, "Logistic regression in data analysis: an overview," *International Journal of Data Analysis Techniques and Strategies*, vol. 3, no. 3, pp. 281–299, 2011.

[5] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, K. Fu and J. Jung, Eds. USENIX Association, 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[6] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 549–564. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp

[7] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., vol. 9721. Springer, 2016, pp. 279–299. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_14

[8] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 245–261. [Online]. Available: https://doi.org/10.1109/SP.2018.00031

[9] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, p. 71, 2015.

[10] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 2016, pp. 300–321.

[11] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 1–19. [Online]. Available: https://doi.org/10.1109/SP.2019.00002

[12] N.-F. Polychronou, P.-h. THEVENON, M. PUYS, and V. Beroulle, "Securing iot/iiot from software attacks targeting hardware vulnerabilities," in *19th IEEE Interregional NEWCAS Conference, Toulon, France, June 13–16, 2021*. IEEE, 2021, accepted to NEWCAS 2021.

[13] A. Sidath. (2018) Machine learning classifiers. [Online]. Available: https://towardsdatascience.com/machine-learning-classifiers-a5cc4e1b0623

[14] S. Raschka. Machine learning faq what is the relation between logistic regression and neural networks and when to use which? [Online]. Available: https://sebastianraschka.com/faq/docs/logisticregr-neuralnet.html

[15] M. Ozsoy, C. Donovick, I. Gorelik, N. B. Abu-Ghazaleh, and D. V. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*. IEEE Computer Society, 2015, pp. 651–661. [Online]. Available: https://doi.org/10.1109/HPCA.2015.7056070

[16] VividCortex, "ewma," https://github.com/VividCortex/ewma, 2017.

[17] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 39–54.

[18] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, pp. 3–14.

[19] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[20] R. McGill, J. W. Tukey, and W. A. Larsen, "Variations of box plots," *The American Statistician*, vol. 32, no. 1, pp. 12–16, 1978.

[21] C. Li and J.-L. Gaudiot, "Challenges in detecting an evasive spectre," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 18–21, 2020.