



HAL
open science

A hybrid solution for constraint devices to detect microarchitectural attacks

Nikolaos Polychronou, Pierre-Henri Thevenon, Maxime Puys, Vincent Beroulle

► To cite this version:

Nikolaos Polychronou, Pierre-Henri Thevenon, Maxime Puys, Vincent Beroulle. A hybrid solution for constraint devices to detect microarchitectural attacks. EuroS&P 2023 - Security of Software/Hardware Interfaces (SILM 2023), located with the 8th IEEE European Symposium on Security and Privacy, Jul 2023, Delft, Netherlands. pp.259-269, 10.1109/EuroSPW59978.2023.00033. cea-04473740

HAL Id: cea-04473740

<https://cea.hal.science/cea-04473740>

Submitted on 22 Feb 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Hybrid Solution for Constraint Devices to Detect Microarchitectural Attacks

Anonymized

Abstract—We are seeing an increase in cybersecurity attacks on resource-constrained systems such as the Internet of Things (IoT) and Industrial IoT (IIoT) devices. Recently, a new category of attacks has emerged called microarchitectural attacks. It targets hardware units of the system such as the processor or memory and is often complicated if not impossible to remediate since it imposes modifying the hardware. In default of remediation, some solutions propose to detect these attacks. Yet, most of them are not suitable for embedded systems since they are based on complex machine learning algorithms.

In this paper, we propose an edge-computing security solution for attack detection that uses a local-remote machine learning implementation to find an equilibrium between accuracy and decision-making latency while addressing the memory, performance, and communication bandwidth constraints of resource-constrained systems. We demonstrate effectiveness in the detection of multiple microarchitectural attacks such as Rowhammer or cache attacks on an embedded device with an accuracy of 98.75% and a FPR near 0%. To limit the overhead on the communication bus, the proposed solution filters 99% of the samples during normal operation.

Index Terms—IoT, Security, HPCs, Edge-Computing, Local-Remote Detection, Microarchitectural Attacks

1. Introduction

In recent years, we have observed an increasing number of Internet of Things (IoT) and Industrial IoT (IIoT) devices. Millions of these devices are placed in critical locations and collect data that are later processed to make appropriate decisions and take actions. However, despite their use in critical applications (such as health, transport or smart cities), IoT devices often lack security guarantees and security support.

In parallel to malwares such as cryptolockers, worms or trojans, a new class of attacks has emerged that we refer to as microarchitectural attacks. Microarchitectural attacks include attacks such as Cache Side-Channel Attacks (CacheSCA) [9], Rowhammer [8], Spectre [16], and Meltdown [20] and target hardware units such as the processor or the memory. One major threat of these attacks is the complexity required to remediate them that often imposes the modification of the component itself or an important overhead on the performances. Thus, in default of remediation, some solutions based on the detection of these attacks were presented in the state of the art to circumvent this problem. Yet, most of them are not suitable for embedded systems. These solutions are based on a too complex machine learning algorithm that does not fit in a little processor or solutions that

create a very big overhead on the communication bus. In such systems, performance, memory, energy consumption, and communication bandwidth play an important role in adopting a security solution to detect microarchitectural attacks.

Since microarchitectural attacks directly exploit hardware, a number of works [10], [23], [33], [27] rely on Hardware Performance Counters (HPCs) to detect these attacks. HPCs are special core registers found in most modern processors. They enable the measurement of hardware-specific events with near-zero overhead for the CPU. Proposed detection solutions extract and analyse HPCs to characterize the behavior of a program using machine learning (ML). Despite the large number of published security solutions using the HPC-ML configuration, most of them are not suitable for IoT devices due to their high overhead induced by complex ML algorithms.

While we aim for a high attack detection rate, we also need to consider a minimum FPR (that is, avoiding false alarms during normal execution of the system). Due to the criticality of some IoT systems (health, industry, transport, etc), they will implement response mechanisms to prevent damage to human or environment. Thus, undetected attacks may obviously damage the system or extract sensitive data. However, frequently triggering response mechanisms due to a false alarm (such as a system recovery) would result in the device not operating properly. Though, a high FPR can also increase system overheads. Furthermore, as IoT devices often extract and transmit data to a remote server, the required bandwidth is a critical parameter for their normal operation.

Related works: In the literature we find two main implementation approaches: remote and local. The first approach implements the security mechanism locally within devices, either in software or hardware. This approach allows fast detection and reduces communication and memory overhead of storing and transmitting the HPC measurements remotely later. On the downside, this approach increases the performance overhead since we must extract system information and deploy the ML to decide for the presence or not of an attack. Simple ML models such as Logistic Regression (LR), Decision Trees (DT) etc, can reduce the induced overheads. However, due to their simplicity they have limited learning capacity which can result in low accuracy i.e., either high detection rate and increased FPR, or low detection rate and low FPR. This is why complex ML implementations are preferred.

To reduce the FPR while having high detection rate, in [23] the authors use ensemble ML. In an ensemble ML, two or more ML algorithms are used, while the final decision is made by a majority vote. The drawback of this solution is the bigger performance overhead (8%) of using multiple ML algorithms in combination and using

one specific model per attack vector. In another work [10], the authors propose Long Short-Term Memory networks (LSTM) for detection of microarchitectural attacks in desktop and server environments. They use the LSTM to learn the behavior of the system during normal operation, which enables them to detect any deviation from it as malicious. They manage to detect attacks with 99.70% F-score, 0.125% FPR and 3.5% performance overhead. However, LSTMs are resource-intensive MLs, and while a 3.5% performance overhead is not high for server or desktop environments, it can pose limitations when deployed in an IoT device, as resources are limited. Decreasing the frequency of HPC extraction and ML deployment can reduce the overheads, but as shown in [27], in this case the ML algorithms might be vulnerable to evasive attacks, i.e., attacks whose behavior is modified in a way that the extracted HPCs are closer to the normal behavior.

The second approach implements the security mechanism in a remote system such as a cloud server. This approach reduces the performance overhead of the local implementation of the detection mechanism, since it only needs to periodically extract HPCs, but significantly increases the memory and communication overhead since we must store this extracted information before transmitting them to a remote server. In [12], the authors use LSTM and Conditional Restricted Boltzmann Machine (CRBM) in a remote system. They succeed in detecting attacks with 99.97% accuracy and have an FPR of 0.5%. In [17], the authors use a one-class SVM and detect attacks with an accuracy of 100%. Both implementations extract HPC data every 1 ms, which significantly increases communication overhead. Considering the hundreds of other edge devices transmitting all of their extracted information, these solutions might lead to exceeding the networks capacity, which can result in slowdowns and unavailability.

Both approaches seek to minimize some overheads at the expense of others. On the other hand, we find a promising approach [33] for resource-constrained systems. In [33], authors use information from the HPCs in combination with a remote implementation of ML, to avoid overheads in the local system. They also try to reduce the communication overhead of sending all extracted HPC samples by sending only a compressed version. However, authors found that increasing the compression rate reduces the accuracy. A compression rate of 20% and 30% showed the best detection rate. Yet, their idea is limited in that it relies on signature checking, which attack authors can avoid [7]. Also, they rely on matrix multiplication for local data compression, which tends to increase execution overhead exponentially with the size of the data being compressed and linearly with the compression rate. The authors measure the performance overhead to be more than 5% at a compression rate of 30%.

From the works cited above, we observe that the use of complex ML models is preferred, due to their ability to provide a high detection rate, while minimizing the FPR. On the downside, implementing these solutions locally induces an increased performance overhead, which might be restrictive for IoT devices. While, remote implementations resolve the performance overhead issue but the required bandwidth required to periodically transmit all the extracted information should

be optimized, otherwise it could lead to network issues. Also, they will be totally ineffective if the network is down.

Contributions: We propose a HPC-based mechanism allowing to detect microarchitectural attacks and compliant with resource limited devices. This solution is based on a local-remote approach, featuring a simple ML classifier locally to the device, alongside complex and resource heavy ML located in a remote cloud. The local ML will first try to decide how likely a behavior is suspicious and send the samples to the cloud only if necessary for further analysis.

This solution is based on edge computing concepts and takes advantage of both local and remote approaches to accurately detect attacks, targeting a 0% FPR without imposing large overhead on the local system while minimizing bandwidth. This target of 0% FPR is introduced by the likelihood of automatic fallback mechanisms in critical IoT devices that will disrupt physical processes (e.g., health monitoring, power distribution, etc) and which should only be triggered for good reasons. Hence, our contributions are the following:

- We propose the first HPC-based local-remote ML mechanism to detect microarchitectural attacks in resource-limited devices. To decide when to send a sample to the remote ML, we implement a two-level detection threshold scheme.
- We evaluate and compare ML algorithm implementations in local, remote, and local-remote systems. We evaluate our solution against microarchitectural attacks and compare it to related works.
- Finally, we propose and evaluate a near zero false positives strategy (FP) based on an Isolation Forest algorithm used in combination with the local-remote implementation.

Outline: The remainder of the paper is organized as follows. In Section 2, we introduce our local-remote ML mechanism allowing to detect microarchitectural attacks and analyze it in detail. Then, in Section 3, we compare ML algorithms implemented locally, remotely, and finally the local-remote implementation. In Section 4, we adapt the approach of [29] to the needs of our proposed idea and successfully further reduce the FPR. Section 5 talks about the comparison between our solution with related works and Section 6 discusses limitations of our solution. Finally, Section 7 summarizes our work and concludes.

2. Proposed solution

We propose a solution that uses edge computing techniques to detect microarchitectural attacks in constraint devices such as IoT and decrease the use of the communication bandwidth. The global architecture is represented in Figure 1 and describes a first level of detection on the local IoT devices and a second on a remote server. The edge device preprocesses the extracted HPC data to filter out the ones with low value of maliciousness, i.e., the normal HPC measurements. To be somewhat independent of a remote security solution, the local ML is able to raise an alarm when it is convinced that an attack is being carried

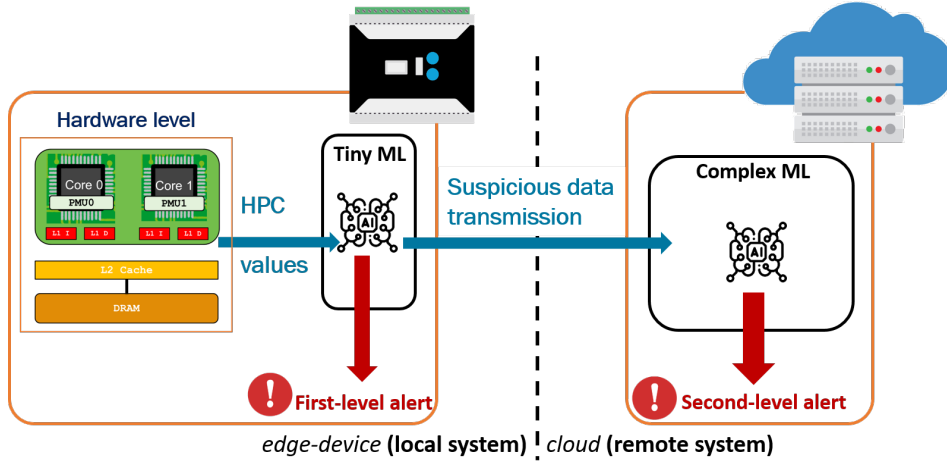


Figure 1. Global overview of the proposed solution.

out. This is useful in cases where the network is down and no connection to the remote solution is possible. HPC values that cannot easily be labeled neither as malicious or trusted are transmitted to a remote server for further analysis. In this case, a complex ML can accurately detect malicious activities. The proposed idea allows us to move parts of the storage and processing resources away from the remote center and closer to the local device. This idea overcomes many limitations of traditional cloud computing such as latency, service delays, network outages, and reliability.

2.1. Local ML implementation

In each edge device, the local ML algorithm which processes the HPC values can be supervised or unsupervised. It reads the HPC samples and outputs a probability between 0% and 100% for each sample to be a microarchitectural attack. The implementation, as shown in Figure 2, allows us to trust samples with a low probability of an attack. It sends samples with an intermediate probability to a remote ML for further evaluation and raises an alert if there are samples with a high attack probability. This is achieved by setting two thresholds.

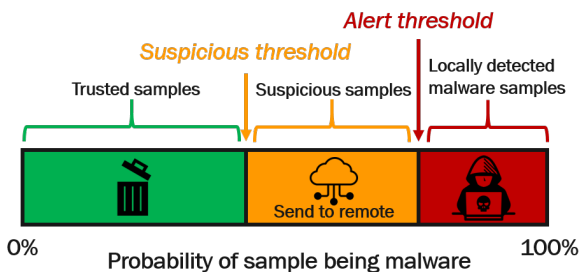


Figure 2. Two-level detection threshold implementation.

The first threshold (*alert threshold*) acts as a trigger for detecting samples that have a high probability of being a microarchitectural attack. We choose the alert threshold based on the maximum probability obtained using the local ML for only normal training samples, plus an extra

offset. This offset acts as an uncertainty level. By choosing the alert threshold as this, we are more certain that during evaluation the local ML will not raise false alerts.

The second threshold (*suspicious threshold*) is set to allow the samples that our local ML cannot identify with high confidence to be stored and later sent to the remote ML for further processing. MLs from related work use a probability of 50% to classify samples as malicious or normal. We chose to set the suspicious threshold to the threshold with the maximum G_{mean} , as shown in (1). G_{mean} is a metric that tries to find an equilibrium between FPR and True Positive Rate (TPR). Essentially, G_{mean} will give us the probability threshold that allow us to filter as many normal samples as possible, while also maximizing the suspicious samples to raise an alert locally or sent them to the remote for further evaluation.

$$G_{mean} = \sqrt{TPR * (1 - FPR)} \quad (1)$$

Subtracting an offset allows the local ML to label samples as suspicious a bit more loosely. This will allow to send malicious samples with behavior closer to normal to the remote ML, but also possibly future attacks based on zero-day vulnerabilities which may differ from learning data and have a lower suspicious score. The trade-off being that more normal traffic will also be sent for remote analysis.

2.2. Remote ML implementation

Suspicious samples are stored locally and then sent to a remote ML for further analysis. The local system sends suspicious samples every Δs to the remote system. This value is chosen empirically and more analyses will be required to evaluate its optimum. The remote ML is based on a complex ML, which is capable of learning complicated behaviors. It can be either a supervised or an unsupervised ML implementation. An unsupervised ML implementation, such as an LSTM, learns only normal behavior and detects any behavior that deviates significantly from the patterns our ML mechanism is trained to identify as a normal behaviour. Using MLs that learn only from normal behavior, we can potentially detect zero-day

attacks by hypothesizing that they deviate from normal behavior and that the local ML succeeds in labelling them as suspicious.

The remote system examines suspicious samples it receives from the local system for differences from normal behavior and flags them as abnormal if they deviate and notifies the local system. Examples of such complex MLs are Autoencoders, one-class SVM, and LSTM networks. LSTM is a network that uses information from past samples to make predictions for current or future samples. Thus, if a remote LSTM ML is used, the additional information about the past HPC samples increases the data sent to the remote.

Benefits of a local-remote approach: Local filtering allows us to reduce the memory requirements for local storage of the extracted samples while minimizing the communication and network overhead for transmitting the samples to the remote system each Δs . Using this local-remote approach, we can generally filter most of the extracted data under normal operation, while only transmitting remotely data under attack. In the following sections, we evaluate various local-remote implementations and present their overhead and detection capabilities.

3. Solution evaluation

In this section, we evaluate various local or remote ML models individually and their combination using several detection metrics. We also evaluate the overheads incurred in the local system and the data successfully filtered. Section 3.1 will introduce our experimental platform and the choice of HPC we use for monitoring. Section 3.2 will present our experimental results, justifying our final choice of ML algorithms. In Section 3.3, we evaluate the filtering succeeded by the two-level threshold approach and finally, in Section 3.4, we optimize the memory required to save the extracted HPC data labelled suspicious.

3.1. Experimental platform

Our experimental platform is based on a Raspberry Pi4 Model B running on Ubuntu 20.04.2 LTS. Its processor is widely used in embedded devices with high resources requirements such as I-IoT gateways and has four ARM Cortex-A72 cores running at 1.5GHz. Each core of the processor provides 84 HPCs and enables the extraction of measurements from six registers simultaneously. As proposed in [27], we extract measurements from the HPC registers each 1ms to be able to detect cache attacks integrating eviction techniques. A dedicated kernel module mainly based on assembly code was used to extract the samples in order to limit the performance overhead.

To demonstrate our idea, we use several microarchitectural attacks as test vectors. These include CacheSCA variants, Spectre variants, Meltdown, and Rowhammer. Also, we include obfuscated (evasive) variants as suggested in [27] by inserting *nop* or *sleep* instructions in the attack code to hide malicious activity. Most of these attacks are publicly available from github [24], [35], [14], [25], [6], [30], [5] and modified to be able to run on the targeted processor. For our benign applications, we use MiBench [11] and PARSEC [3], which are industry

standard embedded system benchmarks that cover a wide range of applications. Our library contains 29 attacks and 72 normal applications.

As the number of events that can be measured simultaneously is limited to six on our experimental platform, it is essential to select the optimal subset of HPC registers to monitor in order to obtain the best detection results. There are different ways to identify the best suitable HPC registers:

- A first solution is to use HPC registers related to different hardware units of the processors: cache memory, bus, TLB, etc.
- A second solution is to use generic HPC registers, that is to say, events that are implemented in most of the CPU cores (Intel, AMD, ARM, RISC-V).
- Feature extraction methods such as Pearson correlation [1] or Mutual Information [19] are mathematical methods allowing to calculate the amount of information provided by a hardware event in order to detect or classify data.
- A theoretical study was proposed in the review [26] to present the most interesting events to detect microarchitectural attacks.

Finally, we selected six HPC events using Mutual information method. This requires the extraction of all available HPCs in the ARM cortex-A72 processor core during the execution of normal and malicious applications. The experimental setup described in Figure 3 was used to extract these measurements. The methodology can be divided in multiple steps:

- **Step 1:** Configuration of the extraction of the HPC events and the monitoring interval.
- **Step 2:** Execution of benign and malicious applications.
- **Step 3:** Extraction of the raw traces of HPC values from a computer.
- **Step 4:** Use of Mutual Information method to evaluate and identify the 6 best HPCs events.

The top six features we obtained with this methodology for microarchitectural attacks detection in the context of IoT are the following:

- **ISB_SPEC:** This event counts each instruction barrier speculatively executed.
- **L1D_TLB_REFILL:** This event counts any L1 data TLB refill.
- **BR_IMMED_SPEC:** This event counts each branch speculatively executed.
- **L2D_CACHE_REFILL:** This event counts any L2 unified cache refill
- **BR_MIS_PRED:** This event counts any branch which is not correctly predicted.
- **MEM_ACCESS_LD:** This event counts memory accesses due to load instructions

3.2. Evaluation of different ML algorithms

For the training of the ML algorithms we used a 70/30 random split of the different applications. Since some applications have a longer execution time than others, we used time-series data augmentation techniques as proposed in [13] to balance the number of samples between

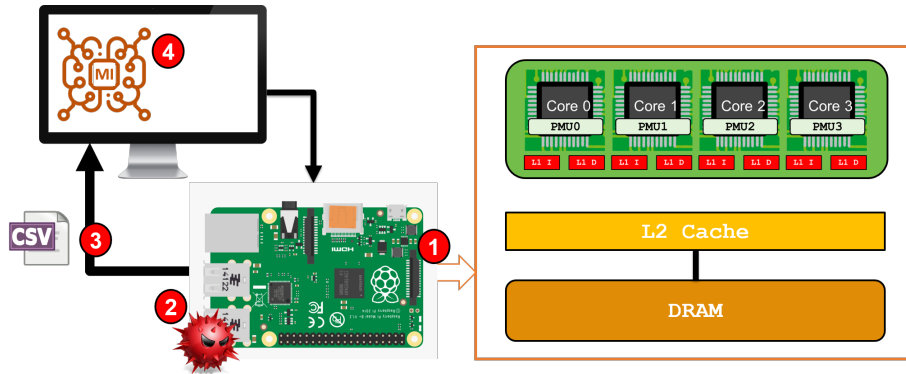


Figure 3. Testbed used to identify best features.

each application, either by upsampling or downsampling. Such techniques are time warping, magnitude warping, window warping, scaling, and window slicing. After augmenting our dataset we use 20% of the training data for validation using stratified sampling [36].

As a first step, we identify how each ML algorithm performs on our classification problem. Our motivation for evaluating the performance of each ML classifier is to show that simple local MLs are capable of identifying malicious samples but have a high FPR. This allows us to demonstrate the need to use complex MLs that have a lower FPR, and finally the advantages of a local-remote ML solution.

We selected two local ML classifiers for testing: (i) Linear SVC (LSVC), and (ii) AdaBoost using five Logistic Regression classifiers. We also select two remote ML classifiers: (i) LSTM, and (ii) LSTM Autoencoder (LSTMAE) with each of them using the seven past samples to predict the current one. For the evaluation, we use the following metrics: True Positive Rate (TPR) or detection rate, and FPR. TPR is a metric that measures how well we identify True Positives (TP) and FPR is the percentage of negative samples falsely classified as positive relative to the total number of negative samples. We pay significant attention to FPR because we argue that low FPR is as important as high TPR for resource-limited and critical devices.

TABLE 1. CLASSIFICATION METRICS FOR THE DIFFERENT ML ALGORITHMS SIMPLE (LINEARSVC & ADABOOST) AND MORE COMPLEX (LSTM7 & LSTM AUTOENCODER7).

Single Level Threshold	TPR	FPR
<i>LinearSVC</i>	99.75%	1.63%
<i>AdaBoost</i> (<i>Logistic Regression</i>)	99.70%	0.68%
<i>LSTM7</i>	72.99%	0.03%
<i>LSTM+</i> <i>AutoEncoder7</i>	92.41%	0.45%

In Table 1, we display results according to the different classification metrics for each of the classifiers. These results for the simple MLs are obtained by using as the classification threshold, the 50% probability of a sample being a microarchitectural attack as in other state of the art works. For the complex MLs, we use as the classification threshold, the quantile (99.85%) of the distribution of the

mean square of the reconstruction errors of the normal training dataset as in [28].

Also as we can see from Table 1, the local MLs have a high TPR ($\sim 99.70\%$) but also a high FPR (0.68% and 1.63%) while the more complex MLs have lower FPR (0.03% and 0.45%) than the simple MLs. The lower TPR of the complex ML is due to the fact that the simple MLs use classification, able to learn patterns from the input and knowing the final label, while the complex perform anomaly detection and detect as malicious any input that greatly deviates from the normal dataset. This is why the remote has lower FPR, since they are trained only with normal data, which helps the remote ML to recognize most normal samples. To recall, having a 0.5% FPR would mean that 5 samples out of 1000 would be erroneously detected as malicious. With a sampling rate of 1ms, this would potentially mean a false alarm every 1 seconds.

TABLE 2. CLASSIFICATION METRICS FOR THE LOCAL-REMOTE.

Two Level Threshold	TPR	FPR
<i>AdaBoost + LSTM7</i>	73.09%	0.02%
<i>AdaBoost + LSTM AutoEncoder7</i>	92.66%	0.13%

In Table 2, we observe how the combination of the simple local ML with the more complex remote ML responds to this classification problem. As we can see the TPR remains high (92.66%) and the FPR low (0.13%) when using the combination of AdaBoost and LSTM AutoEncoder7 algorithms.

Benefits of a local-remote approach: First minimizing the FPR is desirable since one FP is enough to label a whole application as malicious. As stated in Section 1, frequently taking actions due to an alarm during normal operation can make the device unusable. Furthermore, the TPR remains high, which allows us to capture as much malicious activity as possible and only need one timing-window with high anomaly score to detect a malicious app. The TPR is lower than the simple single-level threshold MLs, since the local part of the local-remote only raises alerts for high probability samples and transmits the suspicious for further evaluation in the remote. Also, attacks do not execute only malicious actions in their

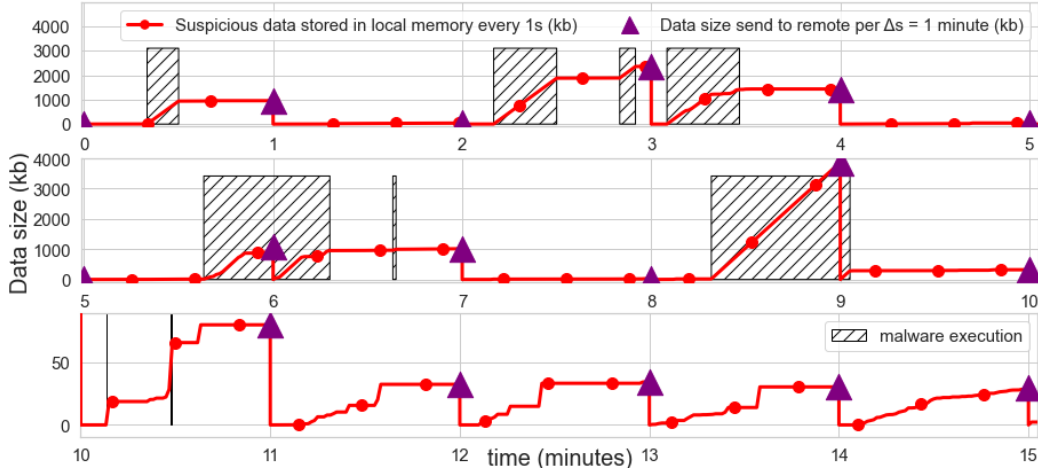


Figure 4. Data stored to the local memory per second (red dotted line) and sent to the remote ML for further evaluation each Δs 1 minute (purple triangles). The shaded areas indicate the period where the system is **under attack**.

whole execution, and the remote ML will recognize such parts as normal, and not as attacks as indicated in our labelling effectively lowering the TPR.

3.3. Edge-device Local ML filtering

In this section we evaluate the filtering succeeded by the two-level threshold approach. We focus on the filtering succeeded during normal operation only, since this is the state of the device in most of its operation and we do not want to overflow the communication interface with limited value data under this condition. From Table 3 we can observe the filtering succeeded under normal operation, when the local ML is configured as AdaBoost (same case than in Table 2), monitoring the system each 1ms, monitoring six 32 bit HPC counters per core, and 4 cores in total, while transmitting the data to the remote each 1 minute. The total size of data extracted from the system per minute of execution is 5760Kb.

Benefits of a local-remote approach: From Table 3, we can see that our approach can successfully filter more than 99% of the extracted HPC data under normal operation. This is due to the local ML being very successful to label as normal benign applications (keeping them under the suspicious threshold). This allows us to dramatically reduce the bandwidth required under normal operation where solutions containing only a remote ML will send all samples to the cloud.

TABLE 3. DATA SEND PER MINUTE UNDER NORMAL OPERATION AND PERCENTAGE OF FILTERING WHEN 5760KB OF HPC DATA ARE EXTRACTED PER MINUTE.

Local ML (filtering only)	Data send per minute	filtering percentage
AdaBoost (Logistic Regression)	39kb	99.32%

3.4. Memory & Detection Time Optimization

In this subsection we optimize the memory required to save the extracted HPC data labelled suspicious. Further-

more, we optimize the detection time, since under attack we would like to detect the attacks before the specified $\Delta s = 1$ minute and further waiting the remote ML to give us the final decisions. To succeed with the above, we make the two following observations from Figure 4 and Figure 5:

- Under normal only operation the saved suspicious data does not exceed a certain value, which in this case as seen in Figure 5 is not more than 50Kb.
- When an attack executes on the system, the rate of saving suspicious data in local storage increases. We can observe that in Figure 4 in periods 2-3, 5-6, 8-9.

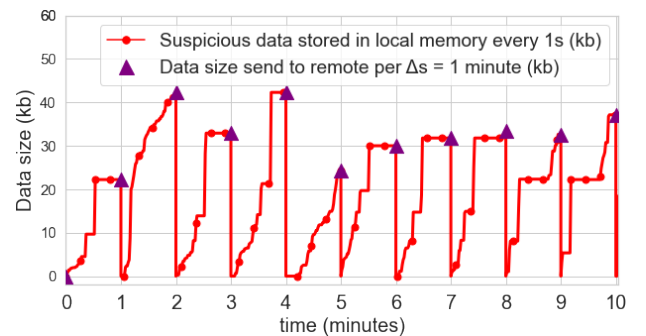


Figure 5. Data stored to the local memory per second (red dotted line) and sent to the remote ML for further evaluation each Δs 1 minute (purple triangles) **under normal operation**.

So to optimize the memory required and the detection time, we set the size of the local storage to the calculated maximum amount of data saved and transmitted under normal operation, so under normal operation we do not exceed this size. For AdaBoost this is 54kb. If under attack, we require more memory to save the extracted HPC samples, and rather than saving more data and as soon as the memory is full, we send the data directly to the remote to have a quicker detection time. As mentioned in the previous observations, under attack the rate of suspicious data saved is greater than under normal operation, which most probably will overflow the local storage.

This implementation choice allows us to only specify the necessary memory for local storage, rather than using

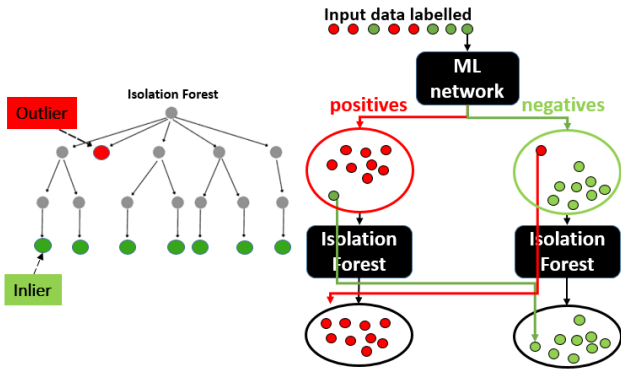


Figure 6. Isolation Forest used by Sadaf et al. [29] to increase classification metrics.

a 5760Kb memory, and also under attack we transmit the data faster in most cases, than waiting the specified Δs .

4. False Positive Minimization Using an Isolation Forest

In the previous section, we successfully reduced the required communication bandwidth and optimized the memory and detection time. But, as observed from Table 2, the FPR of the local-remote might be lower than both complex and single MLs used individually, but it can still be minimized to avoid unnecessary false alarms. To minimize the FPR, we adapt a solution proposed by [29] as can be seen in Figure 6. Sadaf et al. use an Isolation Forest [21] to optimize their classification metrics. An Isolation Forest is a technique allowing to detect/find outliers in a set of data as the samples that greatly differ from others. In their implementation as seen in Figure 6, the input samples are predicted by a ML. As their ML is not perfect, it has some FPs and some FNs. In their hypothesis, FPs are outliers among the rest of the TPs and FNs are outliers among the majority of TNs. By using then an Isolation Forest in the positives, they can find the FPs and change their final label to negative increasing the correct predictions and classification metrics.

In our case to minimize the FPR, we adapt their approach by also using the memory and detection time optimization presented before in Section 3.4. To do this we make the following hypotheses:

- If the remote ML received the data before the expected Δs , we trust its decisions since this most probably happened due to the execution of an attack.
- If the remote ML received data at the expected Δs , we can expect being under normal operation and must be careful for FPs. Thus, we double check the remote's ML positive decisions using an Isolation Forest.

In Figure 7, we can see both our pseudo-algorithm and our technique to adapt the approach proposed by Sadaf et al. When the remote received the data as expected at Δs , for each remote ML positive decision, we also check if the suspicious samples are inliers or outliers in the Isolation Forest. If the Isolation Forest indicates that the sample is an inlier, i.e., it predicts that it is similar with a large pool of normal samples used for the remote ML training, then

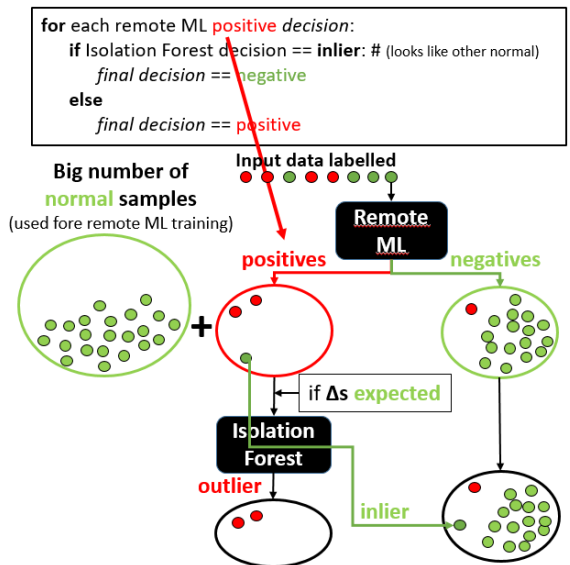


Figure 7. Isolation Forest technique used by Sadaf et al. adapted for our approach.

we reset the decision to negative, otherwise we notify the embedded device of the presence of an attack. As we can see from Table 4, the use of the Isolation Forest to double check remote's decisions, can successfully reduce the FPR to a near 0% compared to not using it, which results in 0.13%.

TABLE 4. CLASSIFICATION METRICS FOR THE LOCAL-REMOTE ML *with and without* THE ISOLATION FOREST FOR FP REDUCTION.















Two Level Threshold	TPR	FPR
<i>AdaBoost + LSTM AutoEncoder7</i>	92.66%	0.13%
<i>AdaBoost + LSTM AutoEncoder7 + Isolation Forest</i>	92.66%	~ 0%

5. State-of-the-Art Comparison

Table 5 presents how our work compares to other approaches in the SOTA. This is not a comparison on the same dataset, since we use different attack vectors and normal libraries, in addition to the different experimental platform, but it shows the contributions of our approach as all articles try to detect the same microarchitectural attacks.

As we can see from the table, when the threat model has limited attack vectors (as in rows 1-4), simple ML models such as Logistic Regression can succeed in detecting the attacks with high accuracy and low FPR. When the threat model has more attack vectors (rows 5-), then more complex MLs are used, such as ensemble ML and LSTM, which keep the accuracy high, while minimizing the FPR. On the downside, we can observe that WHISPER [23] has an overhead of 8%, and FortuneTeller an overhead of 3.5%. These overheads might not be significant in the targeted desktop/server systems, but using these solutions

TABLE 5. COMPARISON OF OUR LOCAL-REMOTE IMPLEMENTATION TO THE RELATED WORKS.

Detection Mechanism	Attacks	Accuracy	F-score	FPR or Precision	Overhead	System	Local or Remote
Mushtaq et al. [22] Logistic Regression (No Load)	Flush+Reload	99.51%		0.48% FPR	0.94%		Local
Mushtaq et al. [22] SVM (No Load)	Flush+Reload	98.82%		0.397% FPR	1.29%		Local
Mushtaq et al. [22] Logistic Regression (No Load)	Flush+Flush	91.73%		0% FPR	1.10%		Local
Mushtaq et al. [22] SVM (No Load)	Flush+Flush	97.42%		0% FPR	0.79%		Local
WHISPER [23] Ensemble Learning <i>One model per attack</i> (DT, RF and SVM) (No Load)	CacheCSA, (F+F, F+R, P+P), Spectre, Meltdown	>97.05%			<8%		Local
FortuneTeller [10] LSTM	Spectre, CacheCSA, (F+F, F+R, P+P), Meltdown, Rowhammer		99.70%	0.125% FPR	3.50%		Local
Wei et al. [34] OC-SVM	Prime + Probe, Spectre, Rowhammer, Evasive	<98.63%		<0.5% FPR			
Wei et al. [34] LSTM	Prime + Probe, Spectre, Rowhammer, Evasive	<99.06%		<0.5% FPR			
Kuruvila et al. [18] Random Forest	Flush + Flush, PNScan, Spectre, Meltdown, Rowhammer, BashLite	89.90%	89.91%	89.25% Precision	<1.22%		Local
Wang et al. [32] MPL	CacheCSA, (F+F, F+R, P+P), Spectre	<98.9%	<97%	5.3% FPR	<3.2%	 	
Wang et al. [32] Logistic Regression	CacheCSA, (F+F, F+R, P+P), Spectre	<98.9%	91.90%	14.9% FPR	<3.23%	 	
Ours AdaBoost + LSTM AutoEncoder7	Spectre, Rowhammer, CacheCSA (F+F, F+R, E+R, P+P), Meltdown, Evasive	98.75%	96.19%	~ 0% FPR	0.80%		Local Remote

on an embedded device could be challenging. Wei et al. [34], Kuruvila et al. [18], and Wang et al. [32] propose solutions for embedded systems. While Wei et al. succeed having high accuracy and low FPR, they use an LSTM, which is a resource demanding ML algorithm, and it could be challenging fitting it in a resource limited system. On the other hand, Kuruvila et al and Wang et al. use simpler MLs to reduce the overheads imposed on the embedded device, but as we see from the table the FPR is high (or precision is low). In comparison, our solution succeeds in keeping the detection accuracy high (98.75%), having minimal FPR ($\sim 0\%$) and imposing a minimal overhead (0.8%) on the edge-device.

Finally, compared to the solution proposed by [33], which was able to compress the extracted data by 20-

30% while keeping the overhead around 5%, we see that our solution can filter up to 99% while the overhead is less than 1%.

6. Discussion

In this section we discuss some implications of our findings and limitations of our approach and propose some ideas to improve future solutions.

Overhead in more constraint devices: In Section 5 and more precisely in Table 5, we mentioned that the overhead imposed by our solution in the Raspberry PI4 is less than 1%. However, this kind of platform has high computation resources and is considered as a complex

IoT device. To validate that our proposed solution can be integrated in more resources constrained devices, more experiments should be performed to evaluate the overhead induced by HPC monitoring with tiny ML.

Limited dataset: Botacin et al. [4] recently discuss about the use of limited dataset to detect microarchitectural attacks and other kind of malwares with ML algorithms. In order to circumvent this issue, we choose to use evasive attacks based on techniques proposed in [27]. These include the insertion of *NOP* or *sleep* instructions in between sensitive tasks to modify the output counts to range closer to normal behavior. We also use some data augmentation techniques to increase the size of some applications. We can still improve by finding more hidden strategies to obfuscate the attack scripts or use GAN (Generative Adversarial Network) based Augmentation as proposed in [15].

Improving the security of the solution: It was proven in [2] that an advanced attacker with knowledge of our simple local ML parameters, could hardly develop code that produces behavior similar to normal while still succeeding the attack. However, the current solution is implemented in software, which means that the local security supervisor including the HPC extraction, the tiny ML and the publishing of suspicious samples implementation and the other applications share resources. This issue still introduces a vulnerability that can be used by an attacker to circumvent or break the solution with a malware. An improved solution will be to implement the local detector as a trusted application in a TEE (Trusted Execution Environment) such as Trustzone or as a dedicated hardware implementation as it was proposed in [31].

Network availability: If the network is down or victim of a Denial of Service attack, our approach will only rely on the ability of the local ML to detect the attacks. Still, it provides a level of security since it can detect high probability samples. Depending on the criticality of the user application, the lack of communication between the IoT node and the control server can also be considered as an attack.

Periodicity to send data to control server: Another parameter we would like to investigate is Δs . In this work it was empirically set to one minute, but more research needs to be done that takes into account communication overhead, detection latency, encryption of suspicious data, and storage overhead. Sending data more frequently to the remote can reduce the detection time, but also increases the overheads, while decreasing Δs reduces the overheads but increases detection time.

Malwares detection: In this paper, we choose to focus on the detection of microarchitectural attack as this class of attacks is complex to circumvent. This kind of attacks can not really be considered as a class of malware but rather as a part of a malware used to gain privileges or steal sensitive data. As some malwares do not use these methods, it is required, to detect more categories of attacks to update our dataset to include complete IoT malwares

such as BashLite, IoTReaper or Mirai. Since these malwares do not stress as much as microarchitectural attacks the hardware, it will probably necessitate more complex ML algorithms on the local device and the remote server or samples from different sources than HPCs.

7. Conclusion

Embedded devices face an increasingly amount of microarchitectural attacks. Such attacks often require hardware changes to be circumvented and detection is generally preferred over mitigation. Several works have been proposed that effectively detect microarchitectural attacks, but without considering the limitations of embedded devices such as computing power, memory, and bandwidth. We have shown that local solutions can have a high detection rate but also a high FPR, while remote solutions are effective but require a large amount of resources and network availability.

This work proposes a solution for detecting attacks on low-resource devices. We propose a local-remote implementation that minimizes the performance, memory, and communication overhead in edge devices while having a high detection rate and minimal FPR. The overall approach benefits from the presence of complex ML algorithms in a cloud to take appropriate decisions but is fully able to work in case of network downtime. We also evaluate the proposed local-remote idea in terms of various metrics. We show that the local system successfully filters 99% of the normal extracted data, which reduces the bandwidth in normal operation. Furthermore, the local-remote increases precision compared to a purely single level threshold implementation, while the FP minimization strategy using an Isolation Forest, as applied in the context of this solution, can reduce FPR to near 0%, preventing unneeded attack responses. Finally, we discuss design choices and limitations of this work.

Multiple paths can be explored as future works. First, many parameters can be tweaked to further reduce overhead (Δs , changing local implementation to an FPGA/ASIC solution). This could also reduce the attack surface of the local detection mechanism itself. Also, this work is currently limited to microarchitectural attacks. Many other kinds of attacks target embedded devices and in particular IoT (malwares exploiting softwares flaws, denial of service botnets, cryptolockers, reverse shells, etc). As these attack will less exploit the hardware, we expect that they will be harder to detect with simply HPCs. Experimental evaluations should then be performed on our ability to detect other kinds of attacks with only HPCs and other signals should be proposed.

References

- [1] Correlation coefficient: Simple definition, formula, easy steps. <https://www.statisticshowto.com/probability-and-statistics/correlation-coefficient-formula/>. Accessed: 2022-01-25.
- [2] Kanad Basu, Prashanth Krishnamurthy, Farshad Khorrani, and Ramesh Karri. A theoretical study of hardware performance counters-based malware detection. *IEEE Transactions on Information Forensics and Security*, 15:512–525, 2019.

- [3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.
- [4] Marcus Botacin and André Grégio. Why we need a theory of maliciousness: Hardware performance counters in security. In Willy Susilo, Xiaofeng Chen, Fuchun Guo, Yudi Zhang, and Rolly Intan, editors, *Information Security*, pages 381–389, Cham, 2022. Springer International Publishing.
- [5] Gorgovan Cosmin. Poc code implementing variant 3a of the meltdown attack for aarch64. https://github.com/lgeek/spec_poc_arm, 2018.
- [6] cryptax. Attempt of implementation of spectre for armv7a. <https://github.com/cryptax/spectre-armv7>, 2018.
- [7] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Computer Architecture News*, 41(3):559–570, 2013.
- [8] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 300–321. Springer, 2016.
- [9] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.
- [10] Berk Gulmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Fortuneteller: Predicting microarchitectural attacks via unsupervised deep learning. *arXiv preprint arXiv:1907.03651*, 2019.
- [11] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard 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)*, pages 3–14. IEEE, 2001.
- [12] Zecheng He, Aswin Raghavan, Guangyuan Hu, Sek Chai, and Ruby Lee. Power-grid controller anomaly detection with enhanced temporal deep learning. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 160–167. IEEE, 2019.
- [13] Brian Kenji Iwana and Seiichi Uchida. An empirical survey of data augmentation for time series classification with neural networks. *Plos one*, 16(7):e0254841, 2021.
- [14] Park Jinbum. Cscs (crypto side channel attack). <https://github.com/jinb-park/crypto-side-channel-attack>, 2018.
- [15] Kangseok Kim. Gan based augmentation for improving anomaly detection accuracy in host-based intrusion detection systems. *International journal of engineering research and technology*, 13:3987, 2020.
- [16] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.
- [17] Prashanth Krishnamurthy, Ramesh Karri, and Farshad Khorrami. Anomaly detection in real-time multi-threaded processes using hardware performance counters. *IEEE Transactions on Information Forensics and Security*, 15:666–680, 2019.
- [18] Abraham Peedikayil Kuruvila, Xingyu Meng, Shamik Kundu, Gaurav Pandey, and Kanad Basu. Explainable machine learning for intrusion detection via hardware performance counters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.
- [19] Erik G Learned-Miller. Entropy and mutual information. *Department of Computer Science, University of Massachusetts, Amherst*, 2013.
- [20] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.
- [21] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 eighth IEEE international conference on data mining*, pages 413–422. IEEE, 2008.
- [22] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Vianney Lapotre, and Guy Gogniat. Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2018.
- [23] Maria Mushtaq, Jeremy Bricq, Muhammad Khurram Bhatti, Ayaz Akram, Vianney Lapotre, Guy Gogniat, and Pascal Benoit. Whisper: A tool for run-time detection of side-channel attacks. *IEEE Access*, 8:83871–83900, 2020.
- [24] Institute of Applied Information Processing and Communications (IAIK). Armageddon: Cache attacks on mobile devices. <https://github.com/IAIK/armageddon>, 2017.
- [25] Institute of Applied Information Processing and Communications (IAIK). Flush + flush. https://github.com/IAIK/flush_flush, 2021.
- [26] Nikolaos-Foivos Polychronou, Pierre-Henri Thevenon, Maxime Puys, and Vincent Beroulle. A comprehensive survey of attacks without physical access targeting hardware vulnerabilities in iot/iiot devices, and their detection mechanisms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 27(1):1–35, 2021.
- [27] Nikolaos Foivos Polychronou, Pierre-Henri Thevenon, Maxime Puys, and Vincent Beroulle. Madman: Detection of software attacks targeting hardware vulnerabilities. In *2021 24th Euromicro Conference on Digital System Design (DSD)*, pages 355–362. IEEE, 2021.
- [28] Oleksandr I Provotar, Yaroslav M Linder, and Maksym M Veres. Unsupervised anomaly detection in time series using lstm-based autoencoders. In *2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT)*, pages 513–517. IEEE, 2019.
- [29] Kishwar Sadaf and Jabeen Sultana. Intrusion detection based on autoencoder and isolation forest in fog computing. *IEEE Access*, 8:167059–167068, 2020.
- [30] SecLab. rowhammer_armv8. https://github.com/0x5ec1ab/rowhammer_armv8, 2019.
- [31] Pierre-Henri Thevenon, Sébastien Riou, Duc-Minh Tran, Maxime Puys, Nikolaos Foivos Polychronou, Mustapha El-Majjidi, and Camille Sivelle. iMRC: Integrated monitoring & recovery component, a solution to guarantee the security of embedded systems. *J. Internet Serv. Inf. Secur.*, 12(2):70–94, 2022.
- [32] Han Wang, Hossein Sayadi, Sai Manoj Pudukotai Dinakarrao, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. Enabling micro ai for securing edge devices at hardware level. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 11(4):803–815, 2021.
- [33] Xueyang Wang, Sek Chai, Michael Isnardi, Sehoon Lim, and Ramesh Karri. Hardware performance counter-based malware identification and detection with adaptive compressive sensing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):1–23, 2016.
- [34] Shijia Wei, Aydin Aysu, Michael Orshansky, Andreas Gerstlauer, and Mohit Tiwari. Using power-anomalies to counter evasive micro-architectural attacks in embedded systems. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 111–120. IEEE, 2019.
- [35] Yarom Yuval. Mastik: A micro-architectural side-channel toolkit. <https://github.com/0xADE1A1DE/Mastik>, 2022.
- [36] Xinchuan Zeng and Tony R Martinez. Distribution-balanced stratified cross-validation for accuracy estimation. *Journal of Experimental & Theoretical Artificial Intelligence*, 12(1):1–12, 2000.