LightRoAD
Lightweight Rowhammer Attack Detector

Taouil, Mottaqiallah ; Reinbrecht, Cezar; Hamdioui, Said ; Sepulveda, Johanna

DOI
10.1109/ISVLSI51109.2021.00072

Publication date
2021

Document Version
Accepted author manuscript

Published in
Proceedings - 2021 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2021

LightRoAD: Lightweight Rowhammer Attack Detector

Mottaqiallah Taouil, Cezar Reinbrecht, Said Hamdioui
Delft University of Technology – EEMCS Faculty
Delft, The Netherlands
{m.taouil, c.r.wedigreinbrecht, s.hamdioui}@tudelft.nl

Johanna Sepúlveda
Airbus Defense and Space
Munich, Germany
johanna.sepulveda@airbus.com

Abstract—**Dynamic Random Access Memory (DRAM)-based systems are widely used in mobile and portable applications where low-cost and high-storage memory capability are required. However, such systems are prone to attacks. A latent threat to DRAM-based system security is the so-called Rowhammer attacks. By repeatedly accessing memory, an attacker is able to perform unauthorized data modifications into physically adjacent memory locations. As a consequence, powerful privilege-escalation attacks can be achieved. Although most of the known countermeasures are based on refresh strategies or intensive address monitoring, their efficient and low-cost realization is still a challenge. In this work, we present LightRoad, a lightweight and flexible hardware detector for Rowhammer attacks. Additionally, we propose two variants that further extend the LightRoad security, namely LightRoAD+Sec and LightRoAD+PARA. Our experiments show that LightRoad and its variants are very efficient and effective to detect attacks while having an affordable cost that varies according to the desired security level.**

I. Introduction

Technology scaling has contributed to the development of faster memories, but simultaneously made them more vulnerable [1]. Consequently, Rowhammer attacks emerged as one of the critical threats of today's computer systems [2]. It affects DRAM memories which are used as central storage unit for a wide variety of Systems-on-Chip (SoCs), from servers to Internet-of-Things (IoT) devices. DRAM memories manufactured with technologies from 2014 and onwards suffer from a specific vulnerability that causes bit flips in the cells when rows are repeatedly accessed [3]. Such continuous accesses are known as hammering. A successful Rowhammer attack allows adversaries to perform privilege escalation (thus taking control of the system) or to retrieve sensitive information [4, 5]. Besides, recent studies have also shown that Flash memories are vulnerable to Rowhammer attacks, which further increases the attack scope of this threat [3]. Therefore, it is imperative to protect computer systems by developing methods and techniques to prevent and detect such attacks.

Different strategies have been proposed to mitigate the Rowhammer threat. They consist of higher DRAMs refresh rates [1] and isolation of memory regions to store sensitive information [6, 7]. Although they mitigate against Rowhammer attacks, such approaches have some disadvantages. The refresh technique is only successful for older DRAM technologies, degrades performance, increases power and it is not scalable. Isolating memory regions limits the available memory and reduces the memory efficiency, as only some regions will be effectively used at a time. Recent methods alleviate such disadvantages by using (non-)deterministic approaches to only refresh specific rows when needed. Deterministic approaches use high-performance counters (HPC) [1, 8] or dedicated monitors [9, 10]. For example, proactive throttling [11, 12] is a deterministic approach that identifies rows that are continuously accessed and subsequently limits the access to them. Although efficient, this strategy requires a detailed understanding of the memory architecture to reduce false-positives. These events are critical as it strongly degrades the system performance. Non-deterministic approaches use probabilities in the protection mechanism, as is for example the case in PARA (Probabilistic Adjacent Row Activation) [1]. Each time a row is accessed in PARA, its neighbouring row is also accessed with a certain probability. To protect older DRAM memories, it is sufficient to use a low probability. However, PARA degrades the performance drop and does not guarantee complete security, especially for newer DRAM memories. An efficient and effective monitoring scheme is key for detecting and stopping Rowhammer attacks.

In this paper we propose an efficient and effective hardware Lightweight Rowhammer Attack Detector called LightRoAD. It detects malicious attempts accurately by monitoring properties extracted from Rowhammer attacks. Such properties are derived from our attack model. Additionally, we propose two variants to further improve the security level. Our main contributions can be summarized as follows:

- A lightweight Rowhammer attack detector.
- A formal model to describe Rowhammer attacks.
- An evaluation of the proposed detector under different attack patterns (false-positives).
- An evaluation of performance penalty and area overhead.
- An evaluation of two variants of lightRoad: LightRoAD+Sec and LightRoAD+PARA

The paper is organized as follows. Section II describes the target platform and threat model, and introduces Rowhammer attacks. Section III presents the attack model and its properties. Section IV introduces the proposed detector. Section V presents our experiments and results. Finally, Section VI discusses our results and concludes the paper.

II. Background on RowHammer Attacks

The continuous evolution of DRAM memories has led to a critical vulnerability. As DRAM cells are getting smaller due to technology scaling, they become more susceptible to be affected by internal or external disturbances. These disturbances may cause bit-flips [13]. In 2014, the authors in [14] presented a methodology to corrupt the DRAM memory. By accessing certain rows repeatedly within a short time, bit-flips in the adjacent (i.e., target) row were performed. This vulnerability was already known before to the test community as row hammering [15].

The success of a Rowhammer attack is highly dependent on the DRAM access frequency of the attacker. Consequently, the attack requires efficient methods to bypass the cache hierarchy between processor and DRAM. There are three ways to accomplish such a bypass: i) cache eviction [16, 17], which relies on accessing different DRAM addresses that map on the same cache line. By alternating accesses between such addresses, cache eviction takes place and the DRAM is frequently accessed; ii) cache flush [1], where flush instructions allow the processor to invalidate used cache lines. Hence, the next time the processor accesses such an address it causes a new request to the DRAM memory; and iii) uncached accesses [18], which bypasses the cache hierarchy completely when accessing the DRAM by disabling the cache and/or using Direct-Memory-Access (DMA). Functions like *memcopy* use DMA to optimize the transfer of high volume data between different DRAM sections.

The attacker can access the victim's row in many different ways. In general they are divided into two classes: single-sided and double-sided attacks. In single-sided attacks, frequent memory accesses (hammering) are applied to a single row which is adjacent to the target row. In contrast, in double-sided hammering two memory rows are frequently accessed, one on each side of the target row. As the two hammered rows must be on different sides of the target row, double-sided hammering generally requires partial knowledge of virtual-to-physical mappings. Since Rowhammer attacks have been successfully demonstrated, they have been exploited to create more complex attacks such as privilege escalation [19], sandbox escapes [19], and cryptographic keys exploitation [20].

III. Rowhammer Attack Model

This paper presents a novel attack model that describes the methodology behind Rowhammer attacks in a systematic way. The model is inspired by the framework proposed in [3]. However, in contrast with this approach, our model describes the attack actions in terms of the involved hardware operations. Such an approach contributes to enhance design-for-security of SoCs. For example, new security verification methods can be built based on this attack model.

A. Attack Stages

The proposed attack model is divided into three stages: i) *setup*: ii) *manipulate*; and iii) *access*. The *setup* stage, which aims to define the target attack location in the DRAM memory. This stage consists of the following setup **operations**:

- S_F: The attacker fills random places of the memory until an address near the target location is found.
- S_P: The attacker adds content to the end of the target location, which is called padding.
- S_R: The attacker uses Operating System services to reallocate the victim's data.
- S_T: The attacker uses a trial and error approach to select the victim's address. Once the attack is completed, the attacker verifies if the target location has been altered. If not successful, the attacker tries another address.

Next, the *manipulate* stage describes the action performed to force accesses to the DRAM memory. This stage consists of the following manipulation **actions**:

- M_F: The attacker accesses the target DRAM row frequently indirectly through the cache memory. Prior to accessing the DRAM row, specific cache lines corresponding to the target DRAM row are first flushed.
- M_E: Similarly as in the previous case, the attacker accesses the target row through the cache memory. Instead of using a flush operation to evict the data in the cache, the attacker accesses another DRAM address that will map on the same cache line.
- M_U: The attacker uses special addresses that can bypass the cache hierarchy in order to access the main memory. This is typically achieved through DMA.

Finally, the *access* stage defines the access patterns. The *manipulate* and *access* stages are frequently repeated in a short time period. The *access* stage consists of the following access patterns:

- A_S: A single-side access pattern which aims to access one or more random locations either above or below the target row.
- A_D: A double-side access pattern which aims to access one or more random locations above and below target row.
- A_O: A one-location access pattern which aims to access only one specific location near the target row.

In our model, the exploitation of a successful attack is not considered as a separate stage since it is application and target specific. Instead, our model defines a successful attack as any action that results in one or more bit-flips in the DRAM, This is independent from the attacker's intention. As a result, even non-malicious applications are considered threats when they meet all the requirements that lead to bit-flips. Note that a security-oblivious application may unintentionally cause a serious security incident.

B. Attack Formula

Each Rowhammer attack in our model can be described by a specific formula which consists of operations/actions selected from the three main stages. In general, a Rowhammer attack formula can be described by Equation 1.

$$S_X \longrightarrow M_X \longrightarrow A_X \tag{1}$$

Where S_X, M_X, and A_X refer to an operation/action in the setup, manipulate and access stages, respectively. Table I maps different Rowhammer attacks described in previous works to

TABLE I

MAPPING STATE-OF-THE-ART ROWHAMMER ATTACKS ON PROPOSED ATTACK MODEL.

ID	Formula	Attacks	ID	Formula	Attacks	ID	Formula	Attacks	ID	Formula	Attacks
1	$S_F \longrightarrow M_F \longrightarrow A_S$	[19] [21]	10	$S_P \longrightarrow M_F \longrightarrow A_S$	[22]	19	$S_R \longrightarrow M_F \longrightarrow A_S$		28	$S_T \longrightarrow M_F \longrightarrow A_S$	[5]
2	$S_F \longrightarrow M_F \longrightarrow A_D$		11	$S_P \longrightarrow M_F \longrightarrow A_D$		20	$S_R \longrightarrow M_F \longrightarrow A_D$	[23] [20]	29	$S_T \longrightarrow M_F \longrightarrow A_D$	[24] [25]
3	$S_F \longrightarrow M_F \longrightarrow A_O$		12	$S_P \longrightarrow M_F \longrightarrow A_O$		21	$S_R \longrightarrow M_F \longrightarrow A_O$		30	$S_T \longrightarrow M_F \longrightarrow A_O$	[4] [26]
4	$S_F \longrightarrow M_E \longrightarrow A_S$	[27] [28]	13	$S_P \longrightarrow M_E \longrightarrow A_S$	[29]	22	$S_R \longrightarrow M_E \longrightarrow A_S$		31	$S_T \longrightarrow M_E \longrightarrow A_S$	[5]
5	$S_F \longrightarrow M_E \longrightarrow A_D$		14	$S_P \longrightarrow M_E \longrightarrow A_D$	[30]	23	$S_R \longrightarrow M_E \longrightarrow A_D$		32	$S_T \longrightarrow M_E \longrightarrow A_D$	
6	$S_F \longrightarrow M_E \longrightarrow A_O$		15	$S_P \longrightarrow M_E \longrightarrow A_O$		24	$S_R \longrightarrow M_E \longrightarrow A_O$		33	$S_T \longrightarrow M_E \longrightarrow A_O$	[26]
7	$S_F \longrightarrow M_U \longrightarrow A_S$		16	$S_P \longrightarrow M_U \longrightarrow A_S$		25	$S_R \longrightarrow M_U \longrightarrow A_S$		34	$S_T \longrightarrow M_U \longrightarrow A_S$	
8	$S_F \longrightarrow M_U \longrightarrow A_D$		17	$S_P \longrightarrow M_U \longrightarrow A_D$	[18] [31] [7]	26	$S_R \longrightarrow M_U \longrightarrow A_D$		35	$S_T \longrightarrow M_U \longrightarrow A_D$	
9	$S_F \longrightarrow M_U \longrightarrow A_O$		18	$S_P \longrightarrow M_U \longrightarrow A_O$		27	$S_R \longrightarrow M_U \longrightarrow A_O$		36	$S_T \longrightarrow M_U \longrightarrow A_O$	[26]

our attack model. They are expressed by the attack formula. Understanding the stages of the attack and their associated operations, give a systematic insight of how the attacks work.

For example, by using our model it becomes evident that the attacks in [19, 21] are similar. Both can be described by $S_F \longrightarrow M_F \longrightarrow A_S$. Their setup stage fills the main memory randomly, a process also known as *spraying*. Thereafter, flush instructions are used to force DRAM accesses. Two random locations near the target row are alternately accessed in order to cause a bit-flip. The difference between these attacks is that [21] uses a non-temporal instruction to flush the cache, while [19] uses the common *clflush* instruction.

IV. LIGHTROAD

In this section we describe the concept, design and architecture of LightRoAD, and preliminary security analyses.

A. Concept

Our model presented in Section III showed that Rowhammer attacks can be represented by three stages. When analyzing state-of-the-art countermeasures, we observe that most hardware-driven solutions focus mostly on the third stage, which is related to the location of the DRAM accesses. They verify which addresses are accessed and count them to determine if an attack is taking place. Consequently, this requires multiple counters and control mechanisms which come with a high hardware cost. From our attack model, we observe that the attacks could also be mitigated by focusing mostly on the second stage, i.e., the *manipulation stage*. The second stage defines which method is employed to ensure accesses to the DRAM memory. It is possible to detect potential attacks by identifying when the second stage of an attack takes place through system monitors. An additional benefit is that with this strategy the detector can evaluate the DRAM access in the third stage with a coarse granularity i.e., multiple rows can be evaluated simultaneously by only looking at the most significant bits of the physical address passed to the DRAM controller. Note that when the detector understands that the system is continuously creating conditions that forces accesses to the main memory, the detector only needs to verify if the accesses target the same region (i.e., a region here is defined by a group of contiguous physical addresses). Analyzing the access based on regions is important, as the work in [2] has shown that addresses in a range of 16 rows can successfully cause bit-flips in the victim's row. As a result of monitoring the manipulative actions, it is possible to create a detector residing fully on the processor chip and hence not modifying the DRAM. As a result, we end up with a very low resource requirement, face minimal integration issues and have high detection efficiency. In addition, by monitoring the manipulation actions, the detector can trigger different types of alarm flags, e.g., alert the system which component and process is responsible for exploiting the vulnerability. The alarms can be triggered by cache misses, cache flushes, or direct DRAM accesses through DMA. LightRoad is based on all these concepts.

B. Design and Architecture

LightRoAD is designed as a standalone component inside the MPSoC. It monitors internal signals to identify which operations are responsible for main memory accesses. LightRoAD counts the number of memory accesses to the target region. An alarm is triggered when a certain threshold is reached. This threshold is defined based on the minimum amount of accesses that are required to cause a bit-flip. This number is highly dependent on the DRAM technology and architecture [32]. The counters are reset when the DRAM is refreshed; a timeout counter is used for this purpose (usually set to 64 ms by most DRAM manufactures [32]).

Figure 1 shows the architecture of LightRoAD. It contains a timeout counter, a Last Level Cache (LLC) miss counter (in our case L2), a flush counter and a DMA counter. When a manipulation action takes place in the system, its respective counter is incremented. When the sum of all counters reaches the threshold value, the alarm signal is raised. Since there are specific counters for each manipulation method, LightRoAD can provide the root cause of the attack as well. As a result, more efficient countermeasures can be put in place. For example, when Rowhammering takes place via DMA, the system could decide to disable the DMA for a certain time. When it is via LLC, the system manager (e.g., OS) could reallocate the victim's data in the main memory.

LightRoAD monitors a single memory region per time. A region is defined as a group of contiguous physical addresses. These regions are identified by the MSB address bits. When a manipulation action takes place, the corresponding counter is only incremented if the address falls into the same region of the previously accessed address. Adjacent regions are considered part of the target region, and therefore, the counter is also incremented. However, when there is an access to a region far away from the previous accesses, all counters are reset and the target region is updated.

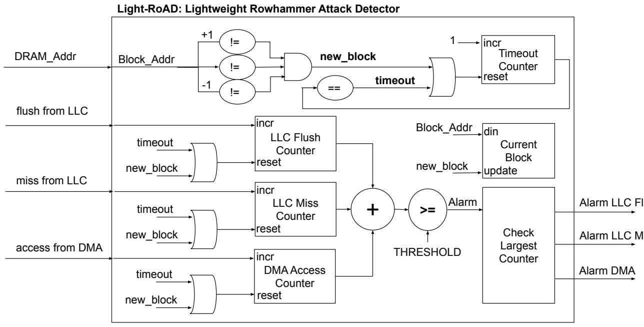

Fig. 1. LightRoAD Architecture

C. Security Analysis and Optimizations

LightRoAD targets one region (i.e., group of contiguous physical addresses) per time when evaluating potential threats. It assumes that the attacker is unaware of the defense mechanism. However, in a scenario where the attacker is able to understand how this detector works (i.e., white-box scenario), the attacker can try to bypass the security by using two regions (one real and one dummy distant region) in the DRAM. Consequently, alternately accessing both regions would reset the counters with each alternate access. Although in such a scenario the attacker reduces the efficiency to access a specific region of DRAM by half (due to real and dummy interleaving accesses), the total amount of accesses might still be sufficient to cause a bit-flip. Therefore, to overcome this issue, we propose two alternatives: LightRoAD+Sec and LightRoAD+PARA.

LightRoAD+Sec monitors the DRAM addresses of multiple regions simultaneously. This means that the current detector is duplicated multiple times, and an arbiter is added to define the region each detector monitors. For example, a memory of 4 GB with regions of 64 row addresses (each address containing 128 bits) would in the worst case require 4096 dedicated detectors to monitor all possible regions. However, each new dedicated detector added to LightRoAD+Sec reduces the attack's efficiency. This means only a limited number of detectors are needed. For example, monitoring the last four target regions simultaneously forces the attacker to insert 4 dummy DRAM operations all the time to reset all detector's counters, i.e., only 1 out of 5 accesses for the attacker are desired. Let's consider for example a CPU running at 1 GHz with a penalty of 100 cycles to access the DRAM, a DRAM refresh rate of 64 ms and a minimum need of 85000 accesses to perform the Rowhammer attack [32]. In such a scenario, only $\lceil \frac{64 \cdot 10^{-3}}{100 \cdot 10^{-9} \cdot 85000} \rceil$ =8 detectors would be required. Note that the minimum of 85k accesses is design and technology depended. In summary, LightRoAD+Sec can trade-off area to match the desired security.

LightRoAD+PARA uses a different approach and takes advantage of the DRAM refresh unit. PARA (probabilistic adjacent row activation), introduced by Kim et al [1], refreshes neighbor rows of accessed addresses with a very low random probability. This means that after a high number of accesses, there is a high chance that a victim address gets refreshed in time. PARA by itself is not sufficient to protect newer memories as it will require a very high probability adjacent row activation. For example, a DRAM memory that can be vulnerable to only 4.8k accesses (as presented in [32] for LPDDR4) and having the PARA configured for a probability $p = 0.001$ would have a probability of bit-flip of $(1 - \frac{0.001}{2})^{4800} = 9.06\%$ (this equation is provided in [1]). According to the authors of PARA, the flip probability can be considered negligible at a value around 1.9^{-22}; hence, a probability of 9.06% (or 0.0906) is far from the desired scenario. However, by combining LightRoAD and PARA much better results are achieved. LightRoAD by default increases the required accesses to accomplish an attack, and as seen in LightRoAD+Sec, as more detectors are included more accesses are needed. Taking this into consideration, the PARA equation can be rewritten to Equation 2, where N refers to the minimum amount of accesses to cause a bit-flip.

$$Flip\_Probability = (1 - \frac{0.001}{2})^{(N*(num\_detectors+1))} \quad (2)$$

Solving this equation for N=4800 and a flip probability of $1.9 * 10^{-22}$, it follows that only 20 detectors are needed. The same scenario would require 134 detectors for LightRoAD+Sec. Therefore, LightRoAD+PARA shows that it can leverage the protection of LightRoAD without increasing significantly the area overhead like LightRoAD+Sec.

V. Experimental Results

This section presents the experimental results. We evaluate LightRoAD's security, performance penalty and area overhead.

A. Setup

LightRoAD detector was implemented using the Verilog hardware description language. To simulate, synthesize and verify the functionality, Xilinx Vivado 2019.2 was used. The detector was integrated into the CVA6 SoC platform [33] (formerly Ariane SoC). CVA6 SoC is a public platform able to run Linux OS on a 64-bits RISC-V processor. All experiments were conducted through hardware simulations, while the area results have been taken from FPGA synthesis.

Security Evaluation: The effectiveness of LightRoad is evaluated by applying all nine access patterns derived from the attack model.

Performance Evaluation: The false-positive detection rate of non-malicious applications is evaluated using seven applications. Five of them are benchmarks taken from the riscv-tests repository [34]. The other two applications are based on the STREAM benchmark [35]. STREAM is the de facto industry standard benchmark for measuring sustained memory bandwidth. We modify it to use DMA to move the blocks of data. We name them as DMA V1 and DMA V2. A short description of each application is provided next. The **Median** benchmark performs a basic three-element 1D median filter over a 400 element input array. The **Multiply** benchmark multiplies two 100-input sized arrays element wise through a shift-and-add algorithm. The **Qsort** benchmark implements the quicksort algorithm on a 2048-input array. The **Towers**

TABLE II
DETECTION EFFICIENCY OF LIGHTROAD

Attack Pattern	Unprotected	Detector 1x	Detector 10x
	# of Access per 64 ms	# of Access per 64 ms	# of Access per 64 ms
$S_X \longrightarrow M_F \longrightarrow A_S$	156500	8400	84000
$S_X \longrightarrow M_F \longrightarrow A_D$	156500		
$S_X \longrightarrow M_F \longrightarrow A_O$	333400		
$S_X \longrightarrow M_E \longrightarrow A_S$	132500		
$S_X \longrightarrow M_E \longrightarrow A_D$	132500		
$S_X \longrightarrow M_E \longrightarrow A_O$	267000		
$S_X \longrightarrow M_U \longrightarrow A_S$	201000		
$S_X \longrightarrow M_U \longrightarrow A_D$	201000		
$S_X \longrightarrow M_U \longrightarrow A_O$	465000		

TABLE III
FALSE-POSITIVE RATE OF LIGHTROAD

Benchmarks	Unprotected	Detector 1x	Detector 10x
	# of Access per 64 ms	False Positives	False Positives
Median	15650	3%	0%
Multiply	15650	0.5%	
QSort	9334	0%	
Towers	13250	0%	
Vvadd	10250	0%	
DMA V1	26700	0%	
DMA V2	102100	0%	

TABLE IV
IMPACT OF LIGHTROAD AREA IN THE CVA6 SOC

Design	LUTs	REGs	Overhead (LUTs + REGs)
CVA6 SoC	99000	75000	N/A (baseline)
Light-RoAD	340	167	0.29%
LightRoAD+Sec	2640	1336	2.28%
Light-RoAD+PARA	340	167	0.28%

benchmark is a purely arithmetic intensive algorithm with a marginally sized dataset. It simulates a round of the Towers of Hanoi puzzle. The **Vvadd** benchmark adds two 300-input sized arrays element wise. The **DMA V1** application copies a range of data with the same size of the region protected by LightRoAD. The **DMA V2** application copies a range of data with 5 times the size of the region protected by LightRoAD.

Hardware Overhead Evaluation: LightRoAD was synthesized for the the Genesys 2 board [36] along with the CVA6 SoC platform to determine the area overhead and evaluate timing constraints.

B. Security Evaluation

Table II provides the results of the security evaluation. The table shows the maximum amount of DRAM accesses that could be applied for each of the nine attack patterns within a period of 64 ms for the unprotected and protected cases. In this experiment, 85000 accesses are consider the minimum amount of accesses required to successfully attack the DRAM (a new DDR3 according to [32]). Hence, all nine patterns can successfully perform Rowhammer in our platform. Note that pattern $S_X \longrightarrow M_U \longrightarrow A_O$ can access the DRAM much more frequently in the same time period. However, A_O is known as a very effective hammering pattern [3], which brings the pattern $S_X \longrightarrow M_U \longrightarrow A_D$ as the most dangerous one in our results. When the system employs LightRoAD (see column detector 1x), it can detect and mitigate any attack with much less accesses (as the threshold equals 8400), meaning that the detector can identify threats at an early stages. A different configuration was also tested, which set the threshold to 84000, which is close to the attack limit of 85000. A higher *threshold* decreases the possibility of false-positives when running non-malicious applications, reducing performance penalties. Note that also with this ten times higher threshold the number of DRAM accesses was not sufficient to perform an attack. Depending on the target memory, a different *threshold* should be applied.

C. Performance Evaluation

Table III shows LightRoad's false-positive rate, caused when non-malicious applications are executed on the target platform while performing legitimate DRAM accesses. The results show that only two applications (Media and Multiply) had some false-positive results. These false-positive cases only happened at cold start, where a high amount of cache misses occur in the beginning of the execution. However, after the cache contains most of the used data, the number of accesses to the main memory is reduced and no false alarms are triggered anymore. Although the applications have a low complexity, they are very suitable for this evaluation since they contain few computations and many cache/memory accesses. In contrast, the DMA-based applications raised no false alarms. The main reason is that DMA functions run over contiguous blocks of memory, and such special operations are not frequently repeated. As a result, even when multiple accesses are applied to the same region, the total number of accesses is not sufficient to generate false positives. In the case where the threshold is set ten times higher, no false positive alarms have been observed. The performance experiment has shown that any non-malicious application would not trigger false positives as specific access patterns would be required to do so. Even when a non-malicious applications triggers the detector, there is a real possibility that a DRAM fault might happen. Hence, even false-positives are important as they prevents bit flips.

D. Hardware Overhead

Table IV shows the synthesis results of the SoC and LightRoAD. We included the LightRoAD+Sec and LightRoAD+PARA configured with 8 and 1 dedicated detectors, respectively. These configurations guarantee full protection when 85k accesses are needed to create a bit flip. Results show that LightRoAD requires very few hardware resources. Even LightRoAD+Sec can be considered a lightweight solution as our target platform uses a very low-profile processor. However, LightRoAD+Sec cannot be considered lightweight anymore for large vulnerable memories.

VI. DISCUSSION AND CONCLUSION

In this paper we demonstrated a low cost detector for Rowhammer attacks. From the results, we conclude:

LightRoAD and variants: The proposed countermeasure reaches a high efficiency while requiring only limited hard-

TABLE V
IMPACT IN CVA6 SOC AREA FOR DIFFERENT SCENARIOS.

	Accesses	LightRoAD+Sec	LightRoAD+PARA
DDR3-old	69.2k	3.0% (10 detectors)	0.29% (1 detector)
DDR3-new	85k	2.28% (8 detectors)	0.29% (1 detector)
DDR4-old	17.5k	10.8% (37 detectors)	1.4% (5 detectors)
DDR4-new	10k	18.6% (64 detectors)	2.62% (9 detectors)
LPDDR4-1x	43.2k	4.37% (15 detectors)	0.58% (2 detectors)
LPDDR4-1y	4.8k	38.2% (134 detectors)	5.8% (20 detector)

TABLE VI
COMPARISON WITH RELATED WORKS.

	Full Protection	Performance Drop	Deterministic	Costs
Refresh Rate [1]	No	High	Yes	+
CATT [6]	No	Medium	Yes	+
GuardION [7]	No	Medium	Yes	+
ANVIL [8]	No	Small	Yes	++
Twice [10]	Yes	Small	Yes	++++
PARA [1]	No	Small	No	+
Blockhammer [12]	Yes	Medium	Yes	+++++
LightRoAD	No	Small	Yes	+
LightRoAD+Sec	Yes	Small	Yes	+++
LightRoAD+PARA	Yes	Small	No	++

ware resources. An important aspect is the fact the detector was tailored based on a reliable attack model. The variants LightRoAD+Sec and LightRoAD+PARA have shown that the level of protection can be improved.

Area Overhead: All LightRoAD solutions may have a different cost depending on the target memory. Older memories require high number of accesses to cause a bit flip while new ones only few accesses are enough. For that reason, we present in Table V a comparison based on the required resources for different type of memories. The number of accesses to create vulnerabilities (see column Accesses) were obtained from [32]. The table shows that for more vulnerable memories a limited number of detector units is required especially when LightRoAD is combined with PARA.

Flexible: In this paper we target the protection of DRAMs. However, LightRoAD is implemented on the processor chip interfacing with the memory controller, which means that it can be used for other memories as well. For instance, Rowhammer attacks have been successfully applied to Flash as well [3], and new memory technologies like STT-RAM or RRAM are expected to be vulnerable as well [37].

Related Work: Table VI provides a comparison with existing solutions using four metrics. For solutions with a small performance drop, results show that most of techniques require medium or high area overhead, with exception of PARA. However, PARA has the drawback of not being feasible for new memory technologies. As a complete protection with low performance drop and small are overhead, LightRoAD+PARA is an interesting choice. Note that LightRoAD+Sec is one of the most powerful solution. However, the area overhead can only be afforded for high-end processors.

REFERENCES

[1] Y. Kim *et al.*, “Flipping bits in memory without accessing them: An experimental study of dram disturbance errors,” in *ISCA*, 2014.

[2] L. Cojocar *et al.*, “Are we susceptible to rowhammer? an end-to-end methodology for cloud providers,” 2020.

[4] D. Gruss *et al.*, “Another flip in the wall of rowhammer defenses,” *CoRR*, 2017.

[3] X. Lou *et al.*, “Understanding rowhammer attacks through the lens of a unified reference framework,” 2019.

[5] S. Bhattacharya and D. Mukhopadhyay, “Curious case of rowhammer: Flipping secret exponent bits using timing analysis,” in *CHES*, B. Gierlichs and A. Y. Poschmann, Eds., 2016.

[6] F. Brasser *et al.*, “Can't touch this: Practical and generic software-only defenses against rowhammer attacks,” *CoRR*, 2016.

[7] V. van der Veen *et al.*, “Guardion: Practical mitigation of dma-based rowhammer attacks on arm,” in *DIMVA*, 2018.

[8] Z. B. Aweke *et al.*, “Anvil: Software-based protection against next-generation rowhammer attacks,” in *ASPLOS*, 2016.

[9] J. Corbet, “Defending against rowhammer in the kernel,” 2018.

[10] E. Lee *et al.*, “Twice: Preventing row-hammering by exploiting time window counters,” in *ISCA*, 2019.

[11] Z. Greenfield and T. Levy, “Throttling support for row-hammer counters (us patent).”

[12] A. G. Yağlıkçı *et al.*, “Blockhammer: Preventing rowhammer at low cost by blacklisting rapidly-accessed dram rows,” 2021.

[13] O. Mutlu, “The rowhammer problem and other issues we may face as memory becomes denser,” in *DATE*, 2017.

[14] D. Kim *et al.*, “Architectural support for mitigating row hammering in dram memories,” *IEEE Computer Architecture Letters*, 2015.

[15] R. Huang *et al.*, “Alternate hammering test for application-specific drams and an industrial case study,” in *DAC*, 2012.

[16] M. T. Aga *et al.*, “When good protections go bad: Exploiting anti-dos measures to accelerate rowhammer attacks,” in *2017 IEEE HOST*, 2017.

[17] D. Gruss *et al.*, “Rowhammer.js: A remote software-induced fault attack in javascript,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero *et al.*, Eds., 2016.

[18] V. van der Veen *et al.*, “Drammer: Deterministic rowhammer attacks on mobile platforms,” in *ACM SIGSAC CCS*, 2016.

[19] M. Seaborn and T. Dullien, “Exploiting the dram rowhammer bug to gain kernel privileges,” 2017.

[20] K. Razavi *et al.*, “Flip feng shui: Hammering a needle in the software stack,” in *25th USENIX*, 2016.

[21] R. Qiao and M. Seaborn, “A new approach for rowhammer attacks,” in *IEEE HOST*, 2016.

[22] Y. Cheng *et al.*, “Still hammerable and exploitable: on the effectiveness of software-only physical kernel isolation,” *arXiv: Cryptography and Security*, 2018.

[23] Y. Xiao *et al.*, “One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation,” in *25th USENIX*, 2016.

[24] Y. Jang *et al.*, “Sgx-bomb: Locking down the processor via rowhammer attack,” in *SysTEX*, 2017.

[25] F. Zhang *et al.*, “Persistent fault analysis on block ciphers,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018.

[26] M. Lipp *et al.*, “Nethammer: Inducing rowhammer faults through network requests,” *CoRR*, 2018.

[27] D. Gruss *et al.*, “Rowhammer.js: A remote software-induced fault attack in javascript,” in *13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016.

[28] M. T. Aga *et al.*, “When good protections go bad: Exploiting anti-dos measures to accelerate rowhammer attacks,” in *HOST*, 2017.

[29] E. Bosman *et al.*, “Dedup est machina: Memory deduplication as an advanced exploitation vector,” in *IEEE SP*, 2016.

[30] P. Frigo *et al.*, “Grand pwning unit: Accelerating microarchitectural attacks with the gpu,” in *IEEE SP*, 2018.

[31] A. Tatar *et al.*, “Throwhammer: Rowhammer attacks over the network and defenses,” in *USENIX*, 2018.

[32] J. S. Kim *et al.*, “Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques,” in *ISCA*, 2020.

[33] F. Zaruba and L. Benini, “The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology,” *IEEE TVLSI*, 2019.

[34] riscv-tests benchmark repository. [Online]. Available: https://github.com/riscv/riscv-tests

[35] J. McCalpin. (1995) Sustainable memory bandwidth in current high performance computers. [Online]. Available: http://www.cs.virginia.edu/ mccalpin/papers/bandwidth/bandwidth.html

[36] *Genesys 2 Board Reference Manual*, Digilent, 2021. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/genesys-2/reference-manual

[37] O. Mutlu and J. S. Kim, “Rowhammer: A retrospective,” *CoRR*, 2019. [Online]. Available: http://arxiv.org/abs/1904.09724