Implementing Rowhammer Memory Corruption in the gem5 Simulator

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

Implementing Rowhammer Memory Corruption in the gem5 Simulator

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

***Abstract*—Modern computer memories have shown to have reliability issues. The main memory is the target of a security threat called Rowhammer, which causes bit flips in adjacent victim cells of aggressor rows. Numerous countermeasures have been proposed, some of the most efficient ones relying on memory controller modifications, which make them non-integrable in existing systems. These solutions have to be effective against attacks on current and future architectures and technology nodes. In order to prove the efficiency of such mitigation techniques, we have to use simulation platforms. Unfortunately, existing architecture simulators do not provide any implementation of unintended memory modifications like bit-flips. Integrating memory corruption into architecture simulators would allow the construction of attacks and mitigations for current and future computers, using feedback from the simulator. In this paper, we propose an implementation of the Rowhammer effect in the gem5 architecture simulator, demonstrate its capabilities and state its limitations.**

I. Introduction

Memory is a key component in modern computers. It is divided in multiple categories: programs and files are stored in a persistent memory, and data used during run-time is stored in a Random-Access Memory (RAM). Multiple layers of cache memories are placed between the RAM and the CPU(s) to speed up access to the most frequently used data. In recent computers, Flash memory is used as the persistent memory, while Dynamic Random-Access Memory (DRAM) is used to store run-time data.

In the past decades, CMOS process technology became more efficient and smaller. RAM manufacturers have been able to put more memory cells in much smaller spaces, resulting in better performance and lower cost [1]. However, making DRAM smaller resulted in higher vulnerability to what Kim et al. [2] described as disturbance errors: activating a DRAM row slightly disturbs adjacent rows due to electromagnetic coupling between adjacent wordlines. Repeated activation of neighbors of a victim row can imply a loss of charge for capacitors in said victim row, effectively deleting the stored data. This became an important security threat as attacks exploiting this vulnerability appeared. These attacks are known as Rowhammer (RH) attacks. Using precisely targeted bit-flips, this type of attack is able to perform privilege escalation [3] or to retrieve sensitive information [4]. Recently, RH became even more important as state-of-the-art attacks were successfully mounted in JavaScript from a web browser sandbox [5] [6], or event without malicious code running on a victim server, using only network requests [7].

Therefore, this problem is being widely studied in academia and industry at present. In order to counter the RH attack, multiple mitigation techniques have been proposed [2] [8] [9] [10] [11] [12] [13]. A lot of them require the introduction of modifications into the memory controller and/or a new hardware component in the architecture. However, testing architecture modifications for RH defenses can be difficult. FPGA emulated system will be too slow to generate the number of DRAM requests to mount the RH attack with the same efficiency as on a modern computer Furthermore, mitigation techniques need to be efficient not only for the current technology node, but also for future, smaller technology nodes, which will be even more vulnerable to disturbance errors.

In order to evaluate the efficiency of their mitigation techniques, researchers could rely on testing environments that are able to simulate the considered vulnerabilities. However, no existing platform provides consideration for memory corruption to-date.

In this paper, we propose to create a module to introduce the memory corruption to system simulations in gem5, with Ramulator for the main memory simulation. This will allow researchers to design attacks and countermeasures implying modifications of the system architecture, or test the efficiency of their countermeasures on systems with different vulnerabilities to the RH attack.

II. Related work

In this section we distinguish our work from related works on RH simulation.

Although the RH attack has been an important issue since its discovery and its mitigation has been widely studied, there has been limited work on its simulation.

Tatar et al. presented in 2018 Hammertime [14], a tool to check the efficiency of RH attacks on current DRAM modules. It uses a memory configuration file containing information about the memory controller, address routing, DRAM geometry and on-chip remapping; and a profile consisting of a list of pairs of aggressor addresses and bit-flip locations. If needed, Hammertime can generate the profile and memory configuration files of the computer it runs on. Using these two files, it can evaluate the efficiency of RH exploits using models of RH attacks. An attack model is represented by a function

that checks its interest for each bit-flip from the provided profile. Hammertime evaluates the efficiency of attacks without simulating the attacks, using models of memories and attacks. while this is a very fast solution to test the efficiency of attacks on different systems, the simplicity of the attack and memory models does not allow researchers to test new mitigations or attack algorithms.

Kim et al. recently published a survey about DRAM devices vulnerability and RH mitigations [15]. They extended Ramulator [16] to simulate the execution of workloads from the SPEC benchmark suite to evaluate the impact of mitigation techniques on the system performance when no attack is running. The evaluation solution used by Kim et al. is useful to determine whether a mitigation affects the system during standard usage, but it cannot be used to test mitigations against attacks.

In these two publications, the memory corruption was not simulated. When designing RH attacks and countermeasures, it is important to evaluate the efficiency on a complete system with a modern architecture and a running operating system. Unfortunately, none of the referenced prior work permits such an evaluation.

III. Background

Our goal is to create a simulator that integrates the memory corruption from RH attacks. A good understanding of how computer memory works and how RH attacks manages to corrupt the memory is necessary to introduce it in a simulator.

A. DRAM architecture and behavior

Modern computers contain one or multiple processors, connected to the main memory through multiple levels of cache memories. Processors generally have their own first level caches, divided into instructions cache and data cache. All the processors are then connected to global cache levels. The last level cache (LLC) is connected to the main memory.

When the processor needs to read or write data, it sends a request to one of the first level caches. There are two scenarios:

1) Cache hit: the data is already stored in this cache, and sent back;
2) Cache miss: the data is not stored in this cache. The request is transferred to the next cache level.

If a cache miss occurs in the LLC, the request is sent to the main memory. After a cache miss, when the next memory level (cache or main memory) returns the data, the current cache memory stores the data to speed up future accesses to the same data, discarding previously-stored data if needed to make space for the new data.

In modern computers, the main memory uses the DRAM technology. Figure 1 presents the communication between the processor and the DRAM, and its architecture. Addressing a data in the main memory is done using multiple levels: channel, rank, chip, bank, then row and column. The CPUs communicate with the DRAM modules through different channels. These modules are represented by ranks (Fig. 1.a). A DRAM chip contains multiple banks (Fig. 1.b). A bank is made of multiple memory arrays (mats, Fig. 1.a) containing a matrix of memory cells (Fig. 1.d). A memory cell stores one bit of data using the charge of a capacitor. An access transistor acts as a relay controlled by the wordline (WL) to connect this capacitor to the bitline (BL) (Fig. 1.e). All memory cells of one row share the same WL and all memory cells of one column share the same BL.

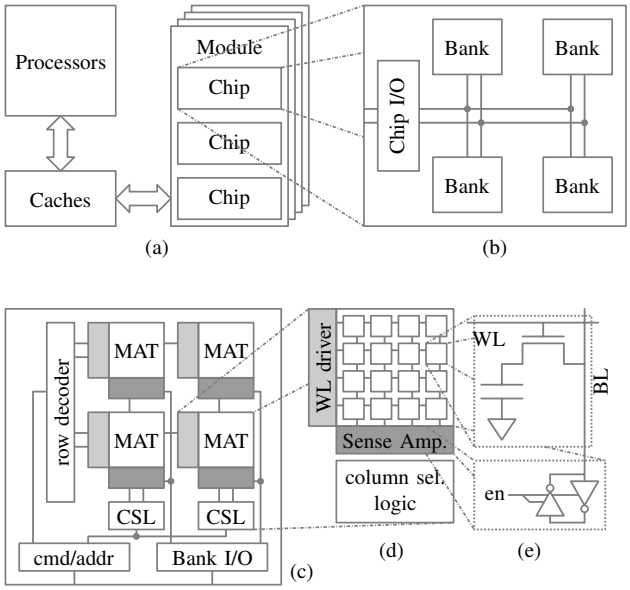

Fig. 1. Computer memory architecture and DRAM architecture [17]. (a) Top-level view of the communication between the processors and the memory; (b) representation of the content of a DRAM chip; (c) Block schematic of a bank; (d) memory array (mat) content, with a memory cells matrix, a WL driver and a sense amplifier; (e) memory cell (top) and bitline sense amplifier with enable pin (bottom) schematics.

In addition to the memory cells matrix, a mat has a WL driver that selects the row to access, and a sense amplifier (SA) that interacts with the BL to read and write the data in the memory cells, and transfer the read data to the bank's row buffer (Fig. 1.e). This row buffer acts as a cache for the last accessed row of the bank.

When the main memory receives a read or write request from the LLC, it first redirects the request to the appropriate bank. Again, there are two scenarios:

1) Row hit: the row buffer already contains the requested data. If the request is a write, the buffer is modified, otherwise it is sent back to the LLC.
2) Row miss: the row buffer does not contain the requested data. If a previous row was stored in the buffer, it is written back in the memory cells. Then, the requested data is fetched from the appropriate row to the row buffer, and finally the buffer is modified or its value is sent back to the LLC.

The process to read or write data to a row in the memory cells is as follow. V_{dd} being the power voltage, The BLs are set to $V_{dd}/2$, and the WL associated to the address is set to V_{dd}. All the access transistors controlled by the WL are opened, connecting every bitline to one memory capacitor each. For

each BL, if the storage capacitor was charged, its charge is partially emptied into the bitline, raising said BL's voltage to $V_{dd} + \delta$. On the contrary, if the capacitor was already empty, it draws charge from the BL, lowering the voltage to $V_{dd} - \delta$. The SA then compares the voltage of each BL, and invert it to the ground voltage (GND) or V_{dd} to transfer it to the row buffer. The value is then written back to the capacitors by raising the BL voltage to V_{dd} or lowering it to GND. Finally, the WL is set to GND to close all access transistors.

As capacitors are not perfect insulators, the charge leaks over time. Capacitors with the fastest leakage in a memory will lose their charge in less than 1s [18]. To maintain the data during long periods of time, all the memory cells need to be frequently refreshed by reading and writing back all the data. This process is done automatically by the memory controller. Typically, all rows are refreshed every 64 ms.

B. Rowhammer

As DRAM technology became denser, the WL got closer to other WL. This proximity made them more vulnerable to electromagnetic coupling: raising the voltage of one WL slightly raises the voltage of neighboring WL. As access transistors are not perfect switches, the small voltage raise on unselected WL slightly opens the access transistors of unselected memory cells. The capacitors, connected to the BL through weakly opened transistors, leak a little bit of their charge through these transistors in addition to the natural leakage of capacitors. Thus, accessing a row repeatedly a large number of times without accessing the neighboring rows can empty the capacitors of unselected memory cells, effectively deleting the stored data.

To exploit this error, aggressors need to avoid cache hits and row hits, which do not lead to rows activations. On x86 systems, an aggressor can perform this attack by running the simple assembly loop presented in listing 1 [2]. In this assembly loop, X and Y are two addresses for two different rows in the same DRAM bank. The first two instructions (`mov`) read the data stored at addresses X and Y, activating the two rows. the next two `clflush` instructions are used to evict the data for addresses X and Y from the caches. `mfence` indicates to the processor that all memory-related instructions must be finished before moving to the next instruction. Finally, `jmp` restarts the loop. The memory under attack by this program is illustrated Figure 2.

Listing 1. Rowhammer loop on x86 system

```
loop:
    mov (X), %eax
    mov (Y), %eax
    clflush (X)
    clflush (Y)
    mfence
    jmp loop
```

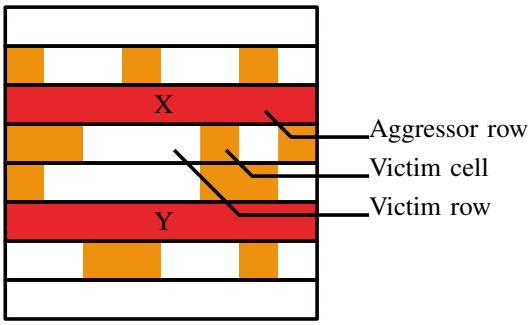

Fig. 2. DRAM bank under attack by program from listing 1

IV. A cycle-accurate Rowhammer simulator

In this section, we discuss different approaches to model RH.

The design of countermeasures requires a proper evaluation of the mitigation effectiveness and associated cost overheads. For this purpose, we need a platform that is capable of executing RH attacks, on which countermeasures can be integrated and which is capable of simulating memory corruption without disturbing the timing of the system.

RH countermeasures can be divided into two categories: some solutions protect the system by detecting the attack and prevent memory accesses that could damage the memory [10] [12] [13], and some other solutions let the attack access the memory, but protect the victim rows from suffering from the attack [2] [8] [11]. The first category mitigations generally don't need major modifications to the architecture. Some rely only on a software solution and use system-generated traces with existing architecture components to retrieve information from the hardware [9].Others use a new component to detect dangerous memory accesses and warn the system to make it stop or slow down the dangerous processes [12] [13]. The second category mitigations require the integration of a component in the memory controller to perform additional refreshes on victims [2] [11]. This modification is necessary to distinguish row hits and row misses, and to know the internal layout of the DRAM, *i.e.*, which rows are adjacent to aggressor rows.

Because of the required modifications in the controller, we cannot test these countermeasures with commodity DRAM modules. Furthermore, future memory technologies like DDR5 or even different technologies like STT-MRAM [19] or PC-RAM [20] have to be taken into account, which makes it impossible to use existing memory modules.

There are multiple solutions to evaluate countermeasures against RH attacks.

Hardware solutions such as a custom chip or a Field Programmable Gate Array (FPGA) are not adapted for RH simulation. A custom chip would be very close to real systems, but it needs to emulate the DRAM to introduce the corruption, and include the countermeasure to test. The cost would be very high and it would require multiple iterations for exploration or optimisation of the mitigation. An FPGA could

offer more flexibility. All the components would be easily modifiable with no additional cost, and the simulation would be close to real time. However, the frequency limit when using a CPU in the programmable logic is very low (a few hundred MHz for the most powerful FPGAs). Additionally, the memory would have to be emulated to add corruption capability and notify refresh events to the mitigation. Finally, emulating a memory big enough to run an Operating System and complex programs such as web browsers would require too many resources on the FPGA. This type of solution is more useful for architecture exploration.

Alternatively, we can use software computer architecture simulation. This type of solutions has the potential to be very flexible and to model technologies that are not physically available. The available simulators have different abstraction levels and we need to specify the basic needs of RH simulation in order to select the best option. The simulation needs to be accurate enough for the main memory to behave just as in a real system, with row hits and row misses. It also needs to be able to simulate a complex program running on the system with a reasonable simulation time. We do not need to simulate electrical details of the system architecture. Electrical simulators, such as SPICE, that could be very accurate, are too heavy and slow for a system emulation. Additionally, it would not be relevant to simulate the corruption as the physical effects causing the disturbance error are still not well understood. As long as the simulated system can run complex programs while maintaining cycle-accurate timing simulation, we can settle for a higher level of abstraction with architecture simulators. The goal of these simulators is to reproduce the behavior of computing devices to generate metrics while programs are running on the simulated device. It is primarily meant for design space exploration, validation of architectures for future hardware and instrumentation.

There are two different categories of simulators: functional simulators and timing simulators. Functional simulators (sometimes called Instruction Set Simulators) are meant to reproduce the architecture from a programmer point of view. They reproduce the functionality without considering internal components timings. Timing simulators, on the contrary, implement the microarchitecture (i.e., the system internals) more precisely. It considers the communication between components and the time needed for their operation.

System simulators from both categories can run two types of simulations: trace- and execution-driven simulations. Trace-driven simulations work by reading a trace of instructions captured by a previous simulation or execution on a simulator or a real device. This has the advantage of being a fast solution to compare multiple architectures for the same program execution, but programs that interact with the system to collect data from the memory to take decisions will not behave correctly from the execution trace. Alternatively, execution-driven simulations work by running the program on the simulated system. Slower than trace-driven simulations, they allow running programs on the simulated architecture without having to run it on another system before. This approach can simulate programs that interact with a changing system or an external user.

For the simulation of memory corruption by RH attacks, we need to use Execution-driven Timing simulators. Functional simulators are not a good fit for memory corruption because they do not simulate timings properly. As some attacks need to witness memory corruption to plan bit-flips on relevant data, trace-driven timing simulators are not applicable either. The simulator needs to be modular so that we can integrate the corruption of the memory and microarchitecture modifications required by countermeasures. To be able to run a large variety of attacks, it must be compatible with the two most common architectures, such as x86 and ARM, and be compatible with full-system simulation. Among all simulators presented by Ayaz Akram et al. in their survey in 2019 [21], only gem5 [22] fits all these requirements.

gem5 is a modular computer architecture simulator widely used in academia and industry. It is used to define a custom system architecture by configuring and connecting cores and memories, and simulate this architecture running programs and operating systems. gem5 integrates the DRAMSIM2 [23] main memory simulator. However gem5 can also be configured to use other memory simulators. In our case, we chose Ramulator [16] as our main memory simulator but the extensions proposed in this paper could easily be implemented in DRAMSIM2. Ramulator is a fast and extensible cycle-accurate DRAM simulator that brings timing accuracy and flexibility to the main memory simulation. However, it still does not model memory corruptions.

In this paper, we will integrate the memory corruption mechanism in a gem5 + Ramulator simulation in order to create a platform to design RH attacks and countermeasures for current and future computer architectures.

V. Memory corruption simulation

In order to simulate the memory corruption in gem5, we created a Memory-Corruption (M-C) module. It is responsible for measuring the disturbance of every row in the memory from activation of neighboring rows, and modifying the stored data to simulate corruption. The integration was made with gem5 version 19.

A. Integration in gem5 and Ramulator

This module needs to be notified for all row activations in the main memory, and have access to the simulation host memory to perform the corruption.

We use Ramulator to assure an accurate simulation of the main memory, we can concentrate our effort on Ramulator and its wrapper in gem5. Ramulator only brings timing consideration to gem5. Data is always read or written by the memory module on gem5. To be able to access the location of the data on the host memory, we need to integrate the module in gem5, not only in Ramulator.

The Ramulator wrapper in gem5 is implemented as a main memory module. When it receives a read or a write packet from other modules of gem5 , it first performs the functional

operation of reading or writing the data from/to the host memory, and then sends a request to the DRAM simulator of Ramulator to simulate the timing of the request. Ramulator first calculates the address vector to find the rank, bank group, bank, row and column for the accessed address. If the memory is busy with parallel accesses or a refresh operation, the request is delayed. Then, Ramulator checks if the request results in a row hit or a row miss, virtually activates the row if necessary to change the row "stored" in the row buffer. Finally, Ramulator stores the address vector in the request and sends it back to the wrapper in gem5 using its callback function. At regular intervals (typically 7.8µs), a refresh request is sent by Ramulator to itself in order to simulate the periodic refreshes of the DRAM. It has no direct effect, but makes the memory busy for a short period of time. As refresh events are not part of any memory access from the rest of the system, they are not notified to gem5.

DRAM row activations are the cause of disturbance in neighbor rows. Thus, all activations must be acknowledged by the M-C module. Row activations can be caused by row misses during memory accesses, and refresh events. Memory accesses are already notified to gem5 through the request callback, but without the important information of whether the request ended on a row hit or a row miss. We simply add this information to the request callback to let the M-C module know when a memory access has led to a row activation. The address vector is specified in the request callback. It is used by the module to determine which row got activated and which rows were disturbed.

The refresh events are not notified to gem5. We fix this by adding a method `set_refresh_callback(callback)` to the C++ Memory base class of Ramulator. When used, this method adds the callback function to all subsequent refresh requests. We make sure to call this function once at the start of the simulation in the M-C module. However, refresh events do not specify the range of refreshed rows. We have to wait for the full refresh cycle to end to consider the refresh on all rows at once. In reality, all row refreshes are distributed in scheduled refresh intervals (typically 7.8µs) across the whole refresh cycle (typically 64ms) [24].

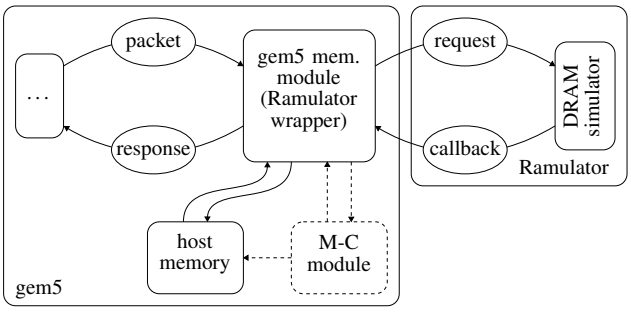

Fig. 3. Integration of the M-C module in gem5

B. Memory-Corruption module behavior

The behavior of the M-C module can be summarized by the algorithm presented in listing 1. In this pseudo-code, `C` is a map associating a row placement to a counter of how many times neighbor rows were activated since the last activation of this row. `T` is the minimum corruption threshold, at which the row starts to get corrupted, and `ref_count` is a counter that decrements at every refresh. When it reaches 0, `C` is cleared, effectively resetting all counters.

Listing 1. Memory Corruption module behavior

```
param T : int ← 50k     # corruption threshold
C : Map (int → int)     # counters map
ref_count : int ← 8192 # refresh counter

function on_ACT(addr_vec) :
    p ← physical_placement(addr_vec)
    if p ∈ C :
        C.delete(p)
    if is_placement_valid(p-1) :
        disturb(p-1)
    if is_placement_valid(p+1) :
        disturb(p+1)

function disturb(p) :
    if p ∈ C :
        C[p] ← C[p] + 1
        if C[p] ≥ T :
            corrupt(p, C[p] - T)
    else :
        C[p] ← 1

function on_REF() :
    ref_count ← ref_count - 1
    if ref_count = 0 :
        C.clear()
        ref_count ← 8192
```

In DRAM banks, consecutive row numbers are not necessarily adjacent. The function `physical_placement` transforms the address into an identifier for the physical placement of the row in the memory. This function can be configured to move the rows inside the memory. More details are discussed in Section VI.

The `corrupt` function is responsible for the modification of the host memory. As the modification happens directly in the host's memory, it does not create any delay for the simulated system. The corruption can be configured to follow a probabilistic polynomial law, detailed in Section VI.

VI. Evaluation

In this section, we will discuss the usage of the created module and its limitations.

A. Parameters

In order to make the created module as versatile as possible, we need to be able to adapt it to various cases and to let it easily evolve with new technologies.

Between two DRAM modules, the internal mapping of rows in the memory chip can change. Two adjacent rows in one chip may not be adjacent on another chip [25]. The M-C module takes a DRAM layout file as an optional parameter, containing the position of each rows in a bank. An example is presented

in Table I. In this example, rows 0, 1, 3 and 6 are respectively at positions 0, 1, 3 and 6. Row 2 is placed at position 4, row 4 is at position 5 and row 5 is at position 2. That means that with this configuration, row 2 is adjacent to row 3 and 4.

TABLE I
DRAM LAYOUT CONFIGURATION EXAMPLE.

logical row	0	1	2	3	4	5	6
physical row	0	1	4	3	5	2	6

The corruption threshold depends on the technology. For example, the corruption threshold is around 139k for DDR3 DRAM and 50k for DDR4 DRAM [2]. We let the user change the corruption threshold with a parameter.

Finally, the corruption is not instantaneous on all bits of a row. There is a few sensitive cells that get corrupted faster than others. The more a row is disturbed from its neighbors, the more cells get corrupted. To model this behavior, the corruption function is defined as presented in listing 2. In this pseudo-code, the parameter `seed_offset` is used to initialise the random function. The parameter `probability_polynomial` defines the polynomial function to determine the flip probability of cells, given the neighbors activation count minus the threshold. By default, all bits are reset when the threshold is reached.

Listing 2. memory corruption function

```
param seed_offset ← 0
param probability_polynomial ← f : x ↦1

function corrupt(position, count) :
    row_addr ← get_row_addr(position)
    random_seed ← seed_offset + row_addr
    probability ← probability_polynomial(count)
    if probability ≥ 1 :
        clear_row(row_addr)
    else if probability > 0 :
        for bit in row(row_addr) :
            if random([0..1]) ≤ probability :
                bit ← 0
```

B. Impact on the simulation

We checked the impact of the M-C module on the simulation time and memory usage for a simple RH attack, the STREAM [26] benchmark and the boot of a Linux system. The simulation runs on a server with an Intel® Xeon® CPU E5-2697 v3 @ 2.60GHz and 252 GB memory. The simulation time is measured using the `time` command, and the peak memory usage using the `valgrind` command. The gem5 system configuration we use, includes one TimingSimpleCPU running at 1 GHz; two 32KB L1 caches: one instruction cache and one data cache; one 512KB L2 cache; and finally a DDR4 DRAM at 2400 MHz as main memory, with a storage limit of 4 GB handled by Ramulator. Table II presents the experiments results. The total simulation time is presented in columns 2 and 3, and the peak memory usage is presented in columns 4 and 5. The M-C module is enabled for columns 2 and 4, and disabled for columns 3 and 5. We measured no noticeable difference for the total simulation time, and no difference in peak memory usage with our module enabled or disabled. For all tested benchmarks, the measured noise is higher than the difference that appears when we integrate the M-C module. The experiments show that our module has no negative impact on the simulation performance.

C. limitations

The simulation of memory corruption is limited on some parameters such as temperature and technology variations. The physical phenomenon behind the RH attack, at the time of writing, is still not entirely understood. Even among the known parameters that affect the corruption of the memory, the created module does not integrate all of them. First, parameters such as temperature, capacitors and transistors technologies have a direct impact on the global corruption threshold and are therefore not integrated as separate parameters. Users will have to change the threshold parameter if they need to change the temperature or capacitor size. At the time of writing, it is not possible to specify vulnerable bits. When specifying the dram bank layout, all banks have the same internal layout. Additionally, the data pattern dependence of the RH effect, which has been measured by J.S. Kim et al. in 2014 [2] is not taken into account for the simulation. Finally, the corruption of near rows that are not directly adjacent (i.e., a blast radius higher than one), which was recently demonstrated by S. Qazi [27], is not yet implemented.

D. Countermeasure design with the M-C module

The purpose of the M-C module is to design and improve attacks and countermeasures, especially countermeasures that require the modification of the memory controller. The module provides all the necessary information for the design of such countermeasures: all DRAM row activations are notified to the module, and it is capable of determining the addresses of adjacent rows. Therefore, this module allows researchers to easily design and improve countermeasures that react when rows are activated, and refresh neighboring rows before the corruption threshold is met. Countermeasures like Graphene [11] and PARA [2] could easily be integrated in a system simulation using the M-C module. The integration of PARA is presented in listing 3. In this code, the `Mitigation` class is the base class for countermeasures integration with the M-C module. The `neighbor` function is used to get one of the neighbors of a row, and returns `true` if the neighbor exists.

Listing 3. PARA [2] implementation in gem5 with the M-C module.

```
class PARA : public Mitigation {
  float p; //!< refresh probability
public:
  PARA(float p) : Mitigation(), p(p) {}
  void onACT(const std::vector<int>& addr) {
    float r = ranged_random(0,1);
    std::vector<int> adj;
    if (r < p/2) {
        if (neighbor(addr, -1, adj))
            refresh(adj);
    } else if (r < p && neighbor(addr, 1, adj))
            refresh(adj);
} } }
```

TABLE II
IMPACT OF THE M-C MODULE ON SIMULATION PERFORMANCE

Benchmark	**time** (avg $\pm 3\sigma$) M-C enable	**time** (avg $\pm 3\sigma$) M-C disabled	peak **memory** usage M-C enable	peak **memory** usage M-C disabled
STREAM	6m32s $\pm$ 7% (8 samples)	6m30s $\pm$ 4% (8 samples)	x86: 885.2MB	x86! 885.2MB
Rowhammer attack	11.47s $\pm$ 10.5% (10 samples)	11.63s $\pm$ 10.4% (10 samples)	x86: 890.44MB ARM: 941.13MB	x86: 890.44MB ARM: 938.14MB
Linux boot and shutdown	31m10s $\pm$ 23% (5 samples)	30m58s $\pm$ 15.8% (5 samples)	not measured	not measured

VII. CONCLUSION

In this paper, we presented a gem5 module to simulate the memory corruption in DRAM caused by Rowhammer attacks [28]. We showed that this module is configurable and can efficiently simulate memory corruption in the memory. It will allow researchers to integrate the architecture simulation when designing attacks to improve the efficiency, and when creating and improving countermeasures. This module will be integrated in the simulator of the ARCHI-SEC ANR project [29]. In future work, we plan to integrate the module on other memory technologies, in particular emerging non-volatile memories such as STT-MRAM, which has been demonstrated to be affected by a variation of the RH attack [30], [31].

ACKNOWLEDGMENT

The authors acknowledge the support of the French Agence Nationale de la Recherche (ANR), under grant ANR-19-CE39-0008 (project ARCHI-SEC). They also acknowledge the French Ministère des Armées – Agence de l'innovation de défense (AID) under grant ID-UM-2019 65 0036.

REFERENCES

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

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

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

[4] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *Proceedings of IEEE SP*, 2020.

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

[6] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "{SMASH}: Synchronized many-sided rowhammer attacks from javascript," in *Proceedings of {USENIX} Security*, 2021.

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

[8] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-based protection against next-generation rowhammer attacks," *ACM SIGPLAN Notices*, 2016.

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

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

[11] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn, and J. W. Lee, "Graphene: Strong yet lightweight row hammer protection," in *Proceedings of IEEE/ACM MICRO*, 2020.

[12] A. G. Yağlikçi, M. Patel, J. S. Kim, R. Azizi, A. Olgun, L. Orosa, H. Hassan, J. Park, K. Kanellopoulos, T. Shahroodi *et al.*, "Blockhammer: Preventing rowhammer at low cost by blacklisting rapidly-accessed dram rows," in *Proceedings of IEEE HPCA*, 2021.

[13] L. France, M. Mushtaq, F. Bruguier, D. Novo, and P. Benoit, "Vulnerability assessment of the rowhammer attack using machine learning and the gem5 simulator-work in progress," in *Proceedings of ACM SaT-CPS*, 2021.

[14] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi, "Defeating software mitigations against rowhammer: a surgical precision hammer," in *Proceedings of RAID*, 2018.

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

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

[17] V. Seshadri and O. Mutlu, "In-dram bulk bitwise execution engine," *arXiv e-prints*, 2019.

[18] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "Raidr: Retention-aware intelligent dram refresh," *ACM SIGARCH Computer Architecture News*, 2012.

[19] T. Ohsawa, H. Koike, S. Miura, H. Honjo, K. Tokutome, S. Ikeda, T. Hanyu, H. Ohno, and T. Endoh, "1mb 4t-2mtj nonvolatile stt-ram for embedded memories using 32b fine-grained power gating technique with 1.0 ns/200ps wake-up/power-off times," in *Proceedings of VLSIC*, 2012.

[20] A. Pirovano, A. L. Lacaita, A. Benvenuti, F. Pellizzer, and R. Bez, "Electronic switching in phase-change memories," *IEEE Transactions on Electron Devices*, 2004.

[21] A. Akram and L. Sawalha, "A survey of computer architecture simulation techniques and tools," *IEEE Access*, 2019.

[22] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, 2011.

[23] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE computer architecture letters*, 2011.

[24] DDR4 SDRAM STANDARD, "JESD79-4," *Joint Electron Device Engineering Council*, 2012.

[25] L. Cojocar, J. Kim, M. Patel, L. Tsai, S. Saroiu, A. Wolman, and O. Mutlu, "Are we susceptible to rowhammer? an end-to-end methodology for cloud providers," in *Proceedings of IEEE SP*, 2020.

[26] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE TCCA Newsletter*, 1995.

[27] S. Qazi, Y. Kim, N. Boichat, E. Shiu, and M. Nissler, ""Half-Double": Next-row-over assisted rowhammer," 2021. [Online]. Available: https://github.com/google/hammer-kit/blob/main/20210525_half_double.pdf

[28] gem5-rowhammer, https://gite.lirmm.fr/adac/gem5-rowhammer, [Accessed October-2021].

[29] ARCHI-SEC ANR Project, https://archi-sec.telecom-paristech.fr/, [Accessed July-2021].

[30] S. Agarwal, H. Dixit, D. Datta, M. Tran, D. Houssameddine, D. Shum, and F. Benistant, "Rowhammer for spin torque based memory: Problem or not?" in *Proceedings of INTERMAG*, 2018.

[31] M. N. I. Khan and S. Ghosh, "Analysis of row hammer attack on sttram," in *Proceedings of ICCD*, 2018.