Hammulator: Simulate Now – Exploit Later

Fabian Thomas, Lukas Gerlach, and Michael Schwarz
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
firstname.lastname@cispa.de

***Abstract*—Rowhammer, first considered a reliability issue, turned out to be a significant threat to the security of systems. Hence, several mitigation techniques have been proposed to prevent the exploitation of the Rowhammer effect. Consequently, attackers developed more sophisticated hammering and exploitation techniques to circumvent mitigations. Still, the development and testing of Rowhammer exploits can be a tedious process, taking multiple hours to get the bit flip at the correct location.**

In this paper, we propose Hammulator, an open-source rapid-prototyping framework for Rowhammer exploits. We simulate the Rowhammer effect using the gem5 simulator and DRAMsim3 model, with a parameterizable implementation that allows researchers to simulate various types of systems. Hammulator enables faster and more deterministic bit flips, facilitating the development of Rowhammer proof-of-concept exploits and defenses. We evaluate our simulator by reproducing 2 open-source Rowhammer exploits. We also evaluate 2 previously proposed mitigations, PARA and TRR, in our simulator. Additionally, our micro- and macrobenchmarks show that our simulator has a small average overhead in the range of 6.96 % to 10.21 %. Our results show that Hammulator can be used to compare Rowhammer exploits objectively by providing a consistent testing environment. Hammulator and all experiments and evaluations are open source, hoping to ease the research on Rowhammer.

I. Introduction

The Rowhammer effect was first shown by Kim et al. [23] in 2014. While initially considered mainly a reliability problem, it turned out to be a significant security problem. Several works [18], [40], [13], [25], [17], [37] showed how the bit flips caused by the Rowhammer effect can be exploited to violate the confidentiality and integrity of data. Hence, a lot of research has been conducted to find effective and efficient techniques to prevent exploitation [23], [5], [3], [15], [1], [21], [43], [44]. Still, even DDR4 and LPDDR4 are affected by Rowhammer, allowing attackers to exploit systems [22]. Although mitigations have been implemented [13], more sophisticated hammering techniques successfully circumvented these mitigations [17], [7], [13], [20], [25].

In addition to hammering techniques, also the Rowhammer exploits evolved. The first exploits still relied on random "blind" hammering [37]. Since then, very sophisticated exploits have been shown using various techniques, including the templating and massaging of memory [17], [26], [34]. Even though attackers can only flip single bits without knowing the location of the bit flips a priori, these techniques led to powerful end-to-end attacks. However, developing and testing such exploits is a tedious process, as it can take multiple hours to obtain the necessary bit flips in hardware [17].

In this paper, we propose Hammulator, a rapid-prototyping framework for Rowhammer exploits. To faithfully emulate Rowhammer bit flips, we simulate the Rowhammer effect in the gem5 simulator [4] using the DRAMsim3 DRAM simulator [28]. We implement a parameterizable Rowhammer simulator to simulate various types of systems. These parameters include the minimum number of row activations and the expected number of bit flips per row. Hammulator allows researchers to quickly change the behavior from a system with many deterministic bit flips to a real-world-like system. Consequently, researchers can develop exploits in a setup that resembles a natural system but still prototype the exploits quicker by enabling faster and more deterministic bit flips. Hence, in line with such prototyping frameworks for other microarchitectural attacks [10], we hope to accelerate the development process of Rowhammer proof-of-concept exploits.

To evaluate Hammulator, we successfully reproduce 2 open-source Rowhammer exploits [34], [37]. We demonstrate that Hammulator supports Rowhammer exploits targeting page tables [37], as well as user-space-only exploits [34]. These exploits work out of the box with our framework, showing that the simulated bit flips resemble real Rowhammer bit flips. To evaluate the performance, we rely on the STREAM [30] benchmark and additional microbenchmarks [16]. We observe a small performance overhead in the range of $6.96\,\%$ to $10.21\,\%$. Moreover, the benchmarks show that our modifications do not introduce spurious bit flips on benign programs.

While the main focus of Hammulator is on rapid prototyping, it can also be used to compare Rowhammer exploits objectively. Currently, such exploits are tested on vastly different setups. Thus, runtime values depend on the underlying hardware, making them hard to compare. With Hammulator, all exploits can be tested with the same setup, resulting in comparable runtime and success numbers. Hence, Hammulator could become a standard for comparing Rowhammer exploits, avoiding flawed benchmarks and comparisons [41]. Moreover, Hammulator allows testing various Rowhammer defenses, allowing for tests in a wide range of scenarios. To show the capability of Hammulator to incorporate Rowhammer defenses, we implement two different defense mechanisms, PARA [23] and a TRR variant.

Contributions. The contributions of this paper are:

1) We present Hammulator[1], a parameterizable Rowhammer simulator.
2) We reproduce and evaluate 2 open-source Rowhammer exploits and 2 defenses in Hammulator.
3) We show that Hammulator has a slight performance overhead and does not suffer from spurious bit flips.

[1]Open source at https://github.com/cispa/hammulator.

Outline. Section II provides background. Section III introduces the simulation goals of Hammulator. Section IV describes the architecture employed by Hammulator. Section V discusses the performance optimizations employed in our simulator. Section VI evaluates the performance and capability of emulating open-source Rowhammer exploits and defenses. Section VII-A introduces related work. We conclude in Section VIII.

II. Background

In this section, we introduce Rowhammer and the simulation frameworks used to build Hammulator.

A. Rowhammer

Since first being studied in 2014 [23], the Rowhammer problem received a significant amount of attention in both academia and industry [37], [22], [25], [17], [13]. Rowhammer sidesteps memory isolation by inducing *disturbance errors* in adjacent memory cells in the DRAM. This effect is strengthened by the constant shrinking of chips, making them more susceptible to *disturbances* [22], [32]. More specifically, the Rowhammer problem is triggered when a DRAM row is repeatedly *hammered* by opening and closing it within the DRAM refresh interval. Multiple works have analyzed the Rowhammer effect on a large scale [23], [22]. In addition, new techniques such as one-location [23] and many-sided hammering [13] and new effects such as half-double [25] have been discovered that intensify the disturbances on the victim DRAM, hindering defenses.

Attacks: In addition to various ways to trigger Rowhammer-induced bit flips [13], [23], [25], their exploitation has also been studied in depth [42], [37], [17]. The first proof of concept showing that Rowhammer can be effectively exploited showed that a user-space process with code execution can gain kernel-level privileges using Rowhammer [37]. Starting from this initial work, multiple other Rowhammer-based exploits were shown to be practical [17], [25], [42], [34]. Rowhammer has been shown on mobile devices [42], in constrained environments such as browsers [18] and virtual machines [34], and by hammering with other devices such as GPUs or network cards [12], [40], [29].

Defenses: Multiple defenses against Rowhammer have been proposed [23], [5], [3], [15], [1], [21], [43], [44]. While initial defenses such as using ECC DRAM or doubling the refresh rate have proven inefficient [7], [6], [33], other academic proposals emerged. Most defenses rely on selectively refreshing DRAM rows on an indication of a Rowhammer attack [13]. Despite the progress in Rowhammer defenses, it remains an open research field with the emergence of novel Rowhammer attacks [22], [20], [19], [9].

B. DRAMsim3

DRAMsim3 [28] is a cycle-accurate DRAM simulator that supports most modern DRAM standards. DRAMsim3 can work as a stand-alone tool or as a component of system simulators such as gem5. Originally, DRAMsim3 was intended for the performance and temperature simulation of DRAM modules. DRAMsim3 is easily extensible and offers good simulation performance [28], making it a good candidate for simulating DRAM timings for Rowhammer attacks.

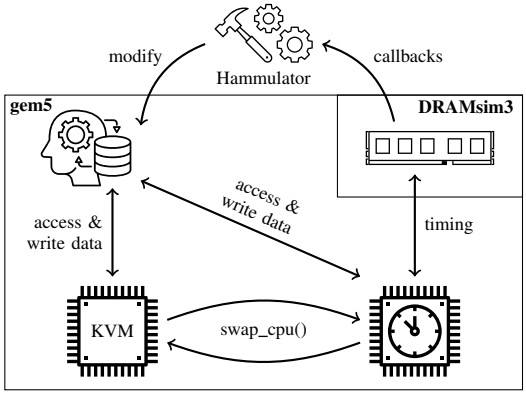

Fig. 1: Outline of the Hammulator architecture. DRAMsim3 is used for cycle-accurate DRAM simulation and provides callbacks to Hammulator. Hammulator modifies the flipped rows in the host memory of gem5.

C. gem5

The gem5 simulation framework provides utilities for system- and processor-level simulation. This simulator is widely used for the evaluation of microarchitecture designs [14], [35]. In addition, hardware fixes for microarchitectural vulnerabilities have been evaluated using gem5 [2], [45]. Two significant features used in this paper are *CPU swapping* and *checkpointing*.

gem5 supports swapping between CPU models during runtime, providing a tradeoff between accuracy and performance. The most accurate is the `TimingSimpleCPU` model, incorporating latencies and timings induced by the memory hierarchy. The fastest is the `KvmCPU` CPU model, using the Linux kernel KVM feature to run on bare hardware at nearly native speed [36]. Using the model can speed up a simulation by a factor of up to 19 000 [36]. The gem5 simulator also supports taking so-called checkpoints of the simulation state. These checkpoints store the simulation state so that execution can be restored and resumed from a previous state.

III. Simulation Goals

In this section, we introduce the overall architecture of Hammulator. We use a combination of the gem5 simulator and DRAMsim3 as a cycle-accurate DRAM model as outlined in Figure 1. The overall target of Hammulator is to simulate the Rowhammer effect so that fast prototyping and engineering of Rowhammer exploits are possible. We define the following goals for Hammulator:

- $\mathcal{G}1$ **Faithful Emulation**: Hammulator should simulate the observable effects of Rowhammer as closely as practically feasible. That is, the observable state should match that of physical DRAM.
- $\mathcal{G}2$ **Performance**: The overhead should be small such that running exploits is fast.
- $\mathcal{G}3$ **Compatibility and Extensibility**: Hammulator should support arbitrary Rowhammer exploits with minimal changes. Additionally, Hammulator should be extensible for specific use cases and prototyping mitigations.

To achieve the goals, we use the gem5 full-system emulator [4] and DRAMsim3 [28], a cycle-accurate DRAM simulator. In the following sections, we describe how Hammulator achieves goals $\mathcal{G}1$-$\mathcal{G}3$.

IV. Simulator Architecture

In this section, we introduce the architecture of Hammulator to achieve $\mathcal{G}1$, i.e., faithful emulation. To match the Rowhammer effects on physical DRAM, we rely on results from several works [25], [23], [13], [20], [17], [22]. As a result, the following behaviors are supported by Hammulator:

1) **Determinism**: The Rowhammer effects are mostly deterministic [23], [22]. Hence, Hammulator always flips the same bits in the same order.
2) **Increased flips over time**: As observed by Kim et al. [22], the number of bit flips grows linearly with the number of accesses per row in a refresh interval. Further, flips occur only after a specific number of accesses. Hammulator emulates these effects.
3) **Flip mask**: Some exploits require bit flips in specific bits of a quadword [37], [17]. Hammulator supports a specific pattern of flips, which we call *flip mask*.
4) **Flips per quadword**: Kim et al. [23], [22] report that most DRAM modules produce in between 1 to 4 bit flips per quadword. Therefore, Hammulator supports specifying a distribution of bit flips in a quadword.
5) **Blast radius**: Since bit flips can occur at distance two and further away from the aggressor row [23], [22], Hammulator supports a blast radius.

Hammulator does not aim to simulate the physical effects connected to Rowhammer but rather emulates the observable Rowhammer effects of a vulnerable DRAM module. Therefore, the design of Hammulator leverages this property to gain performance whenever possible.

We design Hammulator based on the results of Kim et al. [22] and use their terminology. They refer to the number of accesses of an aggressor row as Hammer Count (HC). Similarly, they call the minimum number of accesses to an aggressor row so that flips occur, i.e., the threshold, HC_{first}. We design Hammulator so that it keeps track of HC for every row. If HC exceeds HC_{first}, i.e., the threshold, bit flips start to occur.

Further, Kim et al. [22] find that the log of the number of bit flips in a row scales linearly with the log of HC. While there are infinitely many functions that satisfy this property, we choose to only model the simplest case where we have a direct linear relationship. More complex functions can be modeled by modifying one line in the source code of Hammulator. We add two new parameters to Hammulator to model this linear relation. We extend the terminology by HC_{last}, the upper threshold up until new flips may occur, and by FR_{last}, the bit flip rate at that point, i.e., the maximum bit flip rate. Note that the bit flip rate (FR) is the number of bit flips over the number of bits in a row, as defined by Kim et al. [22]. Further, note that in our model $FR_{first} = 0$, since no flips occur at HC_{first}. The two new parameters HC_{last} and FR_{last} together with the previously known HC_{first} parameter, allow modeling any of the tested DRAM modules used by Kim et al. [22]. Deciding if a flip should occur in a bit of a given victim row is done by

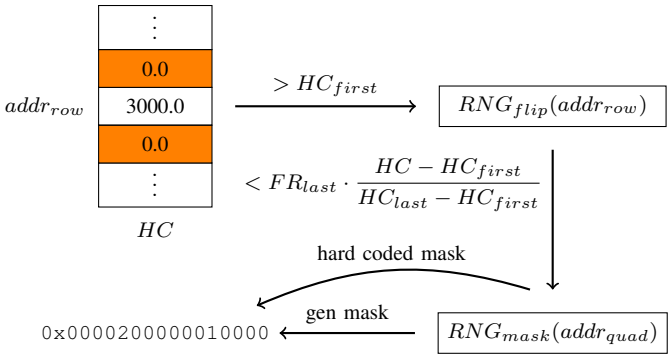

Fig. 2: Overview of the seeded random-number generators used to flip bits deterministically. The value in parentheses specifies which seed is used. If a row exceeds HC_{first}, RNG_{flip} generates a random number that is compared to the flip probability calculated from HC, HC_{first}, HC_{last}, and FR_{last}. If the row is decided to get flipped, a mask is generated with RNG_{mask}, but only if no mask is hard coded.

generating a random number in the range $[0, 1]$. If the random number is bigger than the current bit-flip rate, calculated by $FR_{last} \cdot \frac{HC - HC_{first}}{HC_{last} - HC_{first}}$, a bit flip occurs, as illustrated in Figure 2.

To support flip masks and specify how many flips should occur in a quadword, we divide a victim row into quadwords and decide per quadword if a flip should occur. Note that the flip rate we calculated before must be multiplied by 64 since we decide per full quadword (64 bit) instead of per bit. The upper half of Figure 2 shows how Hammulator decides if a flip should occur in a quadword. The current HC of a row is compared against HC_{first}. Once HC exceeds HC_{first}, a random number is generated for each quadword in the victim row and compared to the flip probability of that row.

The last step is to generate a flip mask that determines which of the quadwords present in the victim row should be flipped. This flip mask is applied to the selected quadwords by a xor operation. As outlined in Figure 2, another random number generator (RNG) is used to infer the bit mask for each quadword. Hammulator supports hard-coding this flip mask such that no RNG is used to generate a flip mask.

Row Information: To implement Hammulator according to goal $\mathcal{G}_1$, it requires access to the internal mapping of the DRAM model to access read, write, and refresh events. DRAMsim3 [28] provides callbacks only for writing and reading. Thus, we extend these callbacks by a refresh callback. Further, we add a function that translates a given row back to a physical address to easily find adjacent rows given the index of an aggressor row.

Hammer Count: As discussed before, we keep track of HC per row. On a read callback, we determine if the read originates from the row buffer or a row. In case the read was served from a row, we increment HC for the adjacent, i.e., the victim, rows. Once a refresh callback is issued, the number of accesses, i.e., the hammer count, to the DRAM rows is reset. Hammulator offers additional flexibility compared to the

simulation by France et al. [11] by allowing refreshes on a per-bank basis.

Blast Radius: To support the blast radius phenomenon [23], [22], Hammulator supports incrementing HC for adjacent rows of up to a distance of 5, in line with the work by Kim et al. [22]. As Kim et al. report a varying correlation between distance and the number of bit flips [22], we model the blast radius of Hammulator to be fully configurable. For each distance up to 5, we add a config-defined variable specifying how much a row's HC is incremented by distance. For a simple DRAM module with no flips further away than the adjacent row of an aggressor row, we set the increment to 1.0 for distance one and to 0.0 for distance two to 5.

Determinism: One of the main goals of Hammulator is to be deterministic. Bit flips should always occur in the same bits and the same order relative to HC. We reach this goal by seeding the used RNGs with the address of the victim row and quadword, as outlined in Figure 2. Seeding with the address ensures that we always get the same random numbers, independent of when and how the victim row is hammered.

Flip Mask Generation: The process of generating a flip mask for a quadword is illustrated in Figure 2. The flip mask generation is skipped if a hard-coded flip mask is specified. Otherwise, a flip mask is generated in two steps. RNG_{mask} determines how many bits should get flipped, and a random mask with the according hamming weight is generated. This procedure allows precisely configuring the number of induced bit flips.

V. Performance Optimizations

In this section, we discuss the implementation details of Hammulator, that ensure that it is practically usable. Although the Hammulator architecture discussed in Section IV is functional and achieves $\mathcal{G}1$, the performance overhead is impractical. The main reason is that full-system emulation with the `TimingSimpleCPU` model runs nowhere near the speed of a native CPU [36]. Therefore, we propose to use checkpointing and CPU swapping as performance optimizations to enable fast prototyping. Checkpointing makes it possible to restore a saved state to speed up the initial setup of a test run, while CPU swapping increases emulation performance by switching to a faster CPU for parts without hammering. With further minor optimizations, we achieve $\mathcal{G}2$, i.e., the overhead of Hammulator is small enough to run full-system exploits [37].

Checkpointing: Some exploits, such as the page-table exploit by Google Project Zero [37], require full-system emulation since they use kernel APIs or attack other processes. Therefore, frictionless full-system emulation is required for Hammulator to be effective in developing exploits. Since the bootup of a full-blown GNU/Linux system can take multiple seconds, even when the KVM-based CPU is used, we use the checkpointing feature of gem5 to bring up a testing environment for exploits quickly. Using checkpointing, even if Hammulator crashes because, e.g., a page-table entry is flipped to an inconsistent state, the system can be up and running again within seconds. This can be used further to debug exploits by taking checkpoints while running the exploit. Once the exploit crashes, Hammulator can be started again with a previous checkpoint.

CPU Swapping: The gem5 simulator supports swapping out the CPU during the simulation. Since running on the KVM-based CPU [36] is faster by a large margin[2], we use this feature to speed up the simulation. It is sufficient to swap to the `TimingSimpleCPU` model only for the parts where hammering is done. This especially speeds up the enumeration and mapping of large chunks of memory. Figure 1 illustrates the CPU-swapping process. The CPU is only swapped for the function that performs the hammering. To benefit from this optimization, the exploit code must only use the Hammulator-provided function `swap_cpu`.

VI. Evaluation

In this section, we evaluate 2 attacks (Section VI-A) and 2 defenses (Section VI-B) in Hammulator, demonstrating $\mathcal{G}3$. Additionally, we measure the performance overhead of Hammulator (Section VI-C).

A. Attacks

We evaluate two different attack types using Hammulator. First, we show that a page-table-based exploit [37], requiring full emulation of the Linux operating system, is possible. Second, we show two exploits on RSA [34] that can be mounted without full-system emulation. These vastly different targets nicely illustrate the flexibility of Hammulator.

1) Page-Table Exploit: To show that exploits, depending on full-system emulation, work, we test the page-table-based exploit by Seaborn et al. [37]. Since the exploit needs to fill the entire physical memory with page tables, the exploit maps a large chunk of memory, e.g., around 3 GB for 4 GB of physical memory. This increases the runtime of the exploit and can be optimized in the development stage by booting the operating system with less physical memory, e.g., 200 MB. Further, CPU swapping can be used to enable the `TimingSimpleCPU` model only in the hammering parts of the exploit.

We reproduce the exploit in Hammulator, i.e., we gain access to our page table. To evaluate the reliability of the page-table exploit in Hammulator, we run the code 10 times in a checkpointed GNU/Linux environment with 200 MB physical RAM. The exploit succeeded in 4 of the 10 tries, i.e., has a reliability of 40 %. In all other cases, the exploit can be restarted. On successful runs, the exploit takes around 1 min. Note that this includes searching for vulnerable rows. In contrast, Jattke et al. [20] reported that finding exploitable bit flips can take up to 1 h on real machines. In addition, some modules might just not induce bit flips. Hammulator can be used on any machine and can produce bit flips in under 1 s.

2) RSA Attacks: We evaluate two attacks on RSA encryption. First, we evaluate an attack from Razavi et al. [34], which is based on bit flips in the public modulus N. By inducing a bit flip in the modulus $N = p * q$, it is reasonably likely that the corrupted N' decomposes into a product of smaller primes instead of the two big prime numbers p and q. This leads to an attack where N' can be directly factored, compromising the RSA encryption. Similarly to Razavi et al. [34], we first

[2]On our machine, booting up GNU/Linux with the KVM-based CPU takes under 10 s, while booting up with the `TimingSimpleCPU` model takes over 30 min, i.e., a speed up by a factor of at least 180. Sandberg et al. report even higher overhead when using a more detailed CPU model [36].

find small factors using Pollard's rho algorithm (and Brent's extension) [31]. Bigger factors are then found using Lenstra's Elliptic Curve Factorization Method (ECM) [27]. In our case study, we flip 8 bits of the modulus and try to factor it afterward. Our results show that for a key size of 512 bits, the RSA modulus can be factored in minutes to hours depending on the position of the bit flips. The attack is dominated by the time to factor the resulting altered modulus, as simulating the required bit flips takes only seconds.

In addition, we present a second attack on RSA using a Rowhammer-induced bit flip in the encryption exponent $e = 65537$. We target a bit that leads to $e' = 1$ or $e' = e - 1$. For $e' = 1$, RSA is trivially broken as the encryption $pt^{e'} = ct$ turns into the identity operation, directly yielding the plaintext. For $e' = e - 1$, we need a second encryption of the same plaintext without the flip. We compute the modular inverse of $pt^{e'} = pt^{e-1}$ and compute $pt^e \cdot (pt^{e'})^{-1} = pt^e \cdot pt^{(-e+1)} = pt$ yielding the plaintext again. While the bit flips for this attack need to be more targeted than the ones for the previously described attack [34] on the RSA modulus, we observe that given the proper simulator parameters, the attack works after, on average, 150 tries or 2 min in real-time. A benefit of this attack is that it reduces the complexity of exploitation on a successful flip to a minimum, as no time intensive-computations are required.

B. Defenses

We evaluate the capability of Hammulator to model defenses by implementing 2 Rowhammer defenses. We implement the probabilistic PARA (Probabilistic Adjacent Row Activation) [23] and a version of TRR (Target Row Refresh) that is a widely deployed mitigation against Rowhammer [13]. For both defenses, we evaluate whether we still observe bit flips when hammering the memory.

1) PARA: PARA randomly refreshes adjacent DRAM rows on activation, reducing the chance of bit flips introduced by Rowhammer. By adding a probabilistic element to the memory access patterns, PARA helps to increase the system's robustness against Rowhammer attacks. Still, as it is stateless, it minimizes the impact on system performance. We implement PARA in the read callbacks of Hammulator. On each access, we generate a random number for each adjacent row up to distance 5, and compare it to config-defined probability p. If a row is chosen for a refresh, we reset the hammer count HC for that row. We observe no bit flips after adding PARA to Hammulator when using a refresh probability of 0.001, in line with Kim et al. [23].

2) TRR: We implement a Panopticon-like [3] countermeasure that refreshes DRAM rows next to frequently-accessed ones. The goal is to mitigate Rowhammer by refreshing the rows with the highest risk of bit flips early, avoiding flips. Similar to the PARA implementation, we extend the read callback. On each access, we compare HC for each of the adjacent rows, again up to a distance of 5, to a config-defined threshold. This threshold should be picked below HC_{first} to ensure no flips occur while using the mitigation. We verify that Hammulator produces no flips when using a threshold of 8000, while $HC_{first} = 10000$.

These experiments illustrate the capability of our framework to not only evaluate and prototype attacks but also include countermeasures, allowing us to evaluate attacks specifically targeting countermeasures.

C. Performance

To evaluate the performance overhead of Hammulator, we use the STREAM [30] benchmark and additional microbenchmarks [16]. We evaluate the baseline gem5 simulator v22.1.0.0 in combination with the DRAMsim3 framework 1.0.0 against our modified version. For our benchmarks, we use an Intel i9-12900K processor with 128 GB of DRAM. For the STREAM benchmark, the execution time in Hammulator is 70.64 min. The baseline, measured in gem5, is 66.04 min, leading to an overhead of 6.96 %. For the microbenchmarks, the execution time is 270.47 s. The baseline, again in gem5, is 245.41 s, resulting in an overhead of 10.21 %. Overall, the overhead of Hammulator is small, especially when considering the additional speedup that can be gained by using checkpointing and CPU swapping.

VII. Discussion

In this section we discuss related work and the limitations of Hammulator.

A. Related Work

While multiple simulators for Rowhammer exist, we distinguish our work from related works. Most closely related to our work is the work of France et al. [11] that introduces a gem5-based Rowhammer simulator based on Ramulator. Unlike this previous work, we focus on directly evaluating open-source exploits and evaluating the performance overhead of Hammulator, showing that it can be used in practice. We also use the more recent and advanced DRAMsim3 [28] simulator and provide advanced features such as CPU swapping and checkpointing to enable rapid prototyping. In addition, we model blast radius and provide a stochastic model triggering bit flips.

Tatar et al. [39] introduced Hammertime [38], a tool to profile and test the susceptibility of a system to the Rowhammer effect. Hammertime provides utilities to check the feasibility of a specific Rowhammer exploit on the profiled system. The key difference between our work and Hammertime is that Hammertime statically analyzes whether a system is vulnerable. While more efficient, this static analysis loses the timing information present when performing full-system emulation. Additionally, Hammertime does not allow easy testing of Rowhammer mitigations.

Mitigations for Rowhammer attacks have been simulated using gem5 and the Ramulator [24] DRAM simulator [44]. The evaluation focussed on the performance overhead introduced when running the SPEC [8] benchmark suite under different mitigations. However, this work focuses only on the simulation of defenses and can, without adaptions, not be used to evaluate new Rowhammer attacks.

B. Limitations

While we try to model several important effects of Rowhammer, we do not claim to emulate the effect in a physically correct way. However, Hammulator provides a great framework for testing the initial feasibility of an exploit and can be used in future work to study mitigations in more detail. In the following we discuss the main deviations from real DRAM.

Determinism: Kim et al. [23] report that, while Rowhammer is mostly deterministic, some cells only flip in rare cases. For exploitation, unreliable bit flips are of little importance since an attacker would use cells that induce bit flips deterministically to increase the reliability of their exploit. As a result and in favor of easing rapid prototyping in our simulator, we choose to not model this behavior and treat Rowhammer bit flips as fully deterministic. Note that the simulator can easily be extended to support these random infrequent bit flips by overlaying the deterministic flips with random bit flips.

Data Dependence: Previous work has shown that Rowhammer bit flips are data dependent [23], [22]. We decide to not model this data dependence to simplify the simulator and increase performance. However, future work could extend Hammulator to model data dependence to increase faithfulness.

Blast Radius: We support the simulation of a simple blast radius, independent of the hammering pattern. However, Half-Double [25] has shown that accesses to immediate neighbor rows of the victim row are necessary to induce bit flips. While Hammulator does not stay true to this effect in all configurations, the framework can be configured to enforce this behavior by carefully aligning the flip threshold HC_{first} and the increment steps per distance.

VIII. Conclusion

In this paper, we proposed Hammulator, an open-source rapid-prototyping framework for Rowhammer exploits. We simulated the Rowhammer effect using the gem5 simulator and DRAMsim3 model, with a parameterizable implementation that allows researchers to simulate various types of systems. Hammulator enables faster and more deterministic bit flips, facilitating the development of Rowhammer proof-of-concept exploits. We evaluated our framework by reproducing 2 open-source Rowhammer exploits, showing that they work with Hammulator. Additionally, benchmarks demonstrate a small performance overhead in the range of $6.96\,\%$ to $10.21\,\%$. Our results show that Hammulator can be used to compare Rowhammer exploits objectively, as it provides a consistent testing environment.

Acknowledgment

We thank Lorenz Hetterich for insightful discussions and recommendations during our early phase of prototyping. Further, we thank the reviewers for their valuable feedback.

References

[1] 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.

[2] P. Ayoub and C. Maurice, "Reproducing spectre attack with gem5: How to do it right?" in *EuroSec*, 2021.

[3] T. Bennett, S. Saroiu, A. Wolman, and L. Cojocar, "Panopticon: A complete in-dram rowhammer mitigation," in *Workshop on DRAM Security (DRAMSec)*, 2021.

[4] 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.

[5] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "CAn't touch this: Software-only mitigation against Rowhammer attacks targeting kernel memory," in *USENIX Security Symposium*, 2017.

[6] 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 *IEEE Symposium on Security and Privacy (SP)*, 2020.

[7] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks," in *S&P*, 2019.

[8] S. P. E. Corporation, "SPEC CPU 2017," 2017. [Online]. Available: https://www.spec.org/cpu2017/

[9] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "Smash: Synchronized many-sided rowhammer attacks from javascript." in *USENIX*, 2021.

[10] C. Easdon, M. Schwarz, M. Schwarzl, and D. Gruss, "Rapid Prototyping for Microarchitectural Attacks," in *USENIX Security*, 2022.

[11] L. France, F. Bruguier, M. Mushtaq, D. Novo, and P. Benoit, "Implementing rowhammer memory corruption in the gem5 simulator," in *RSP*. IEEE, 2021.

[12] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *S&P*, 2018.

[13] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *S&P*, 2020.

[14] J. Fustos, F. Farshchi, and H. Yun, "SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks," in *DAC*, 2019.

[15] M. Ghasempour, M. Lujan, and J. Garside, "ARMOR: A Runtime Memory Hot-Row Detector," 2015. [Online]. Available: http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer

[16] V. R. Group, "Microbenchamarks," 2022. [Online]. Available: https://github.com/VerticalResearchGroup/microbench

[17] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another Flip in the Wall of Rowhammer Defenses," in *S&P*, 2018.

[18] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA*, 2016.

[19] H. Hassan, Y. Can Tuğrul, J. S. Kim, V. Van der Veen, K. Razavi, and O. Mutlu, "Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications," in *IEE MICRO*, 2021, extended classification tree and PoCs at https://transient.fail/.

[20] P. Jattke, V. van der Veen, P. Frigo, S. Gunter, and K. Razavi, "Blacksmith: Scalable rowhammering in the frequency domain," in *S&P*, 2022.

[21] B. K. Joardar, T. K. Bletsch, and K. Chakrabarty, "Learning to mitigate rowhammer attacks," in *DATE*, 2022.

[22] J. S. Kim, M. Patel, A. G. Yaglikçi, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques," *ISCA*, 2020.

[23] 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 *ISCA*, 2014.

[24] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," *IEEE Comput. Archit.*, 2015.

[25] A. Kogler, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu, M. Nissler, and D. Gruss, "Half-Double: Hammering From the Next Row Over," in *USENIX Security Symposium*, 2022.

[26] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading Bits in Memory Without Accessing Them," in *S&P*, 2020.

[27] H. W. Lenstra Jr, "Factoring integers with elliptic curves," *Annals of mathematics*, 1987.

[28] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator," *IEEE Comput. Archit. Lett.*, 2020.

[29] M. Lipp, M. T. Aga, M. Schwarz, D. Gruss, C. Maurice, L. Raab, and L. Lamster, "Nethammer: Inducing Rowhammer Faults through Network Requests," in *SILM Workshop*, 2020.

[30] J. D. McCalpin, "Stream benchmark," 1995. [Online]. Available: www.cs.virginia.edu/stream/ref.html

[31] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 2018.

[32] O. Mutlu, "The RowHammer problem and other issues we may face as memory becomes denser," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[33] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[34] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *USENIX Security Symposium*, 2016.

[35] G. Saileshwar and M. Qureshi, "MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design," in *USENIX Security Symposium*, 2021.

[36] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer, "Full speed ahead: Detailed architectural simulation at near-native speed," in *IEEE*. IEEE, 2015.

[37] M. Seaborn, "Exploiting the DRAM rowhammer bug to gain kernel privileges," March 2015, retrieved on June 26, 2015. [Online]. Available: http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html

[38] A. Tatar, "Hammertime: a software suite for testing, profiling and simulating the rowhammer dram defect," 2018. [Online]. Available: https://github.com/vusec/hammertime

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

[40] A. Tatar, R. Krishnan, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer Attacks over the Network and Defenses," in *USENIX ATC*, 2018.

[41] E. van der Kouwe, G. Heiser, D. Andriesse, H. Bos, and C. Giuffrida, "Sok: Benchmarking flaws in systems security," in *EuroS&P*, 2019.

[42] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *CCS*, 2016.

[43] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM," in *DIMVA*, 2018.

[44] A. G. Yağlıkçı, M. Patel, J. S. Kim, R. Azizibarzoki, A. Olgun, L. Orosa, H. Hassan, J. Park, K. Kanellopoullos, T. Shahroodi, S. Ghose, and O. Mutlu, "BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows," in *HPCA*, 2021.

[45] T. Zhang, F. Liu, S. Chen, and R. B. Lee, "Side channel vulnerability metrics: the promise and the pitfalls," in *HASP*, 2013.