2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179390

CSI:Rowhammer – Cryptographic Security and Integrity against Rowhammer

Jonas Juffinger*†, Lukas Lamster†, Andreas Kogler†, Maria Eichlseder†, Moritz Lipp‡, Daniel Gruss*†

*Lamarr Security Research, †Graz University of Technology, ‡Amazon Web Services

***Abstract*—In this paper, we present CSI:Rowhammer, a principled hardware-software co-design Rowhammer mitigation with cryptographic security and integrity guarantees, that does not focus on any specific properties of Rowhammer. We design a new memory error detection mechanism based on a low-latency cryptographic MAC and an exception mechanism initiating a software-level correction routine. The exception handler uses a novel instruction-set extension for the error correction and resumes execution afterward. In contrast to regular ECC-DRAM that remains exploitable if more than 2 bits are flipped, CSI:Rowhammer maintains the security level of the cryptographic MAC. We evaluate CSI:Rowhammer in a gem5 proof-of-concept implementation. Under normal conditions, we see latency overheads below $0.75\,\%$ and no memory overhead compared to off-the-shelf ECC-DRAM. While the average latency to correct a single bitflip is below $20\,\text{ns}$ (compared to a range from a few nanoseconds to several milliseconds for state-of-the-art ECC memory), CSI:Rowhammer can detect any number of bitflips with overwhelming probability and correct at least 8 bitflips in practical time constraints.**

1. Introduction

Rowhammer is a widespread DRAM issue, where cells lose charge faster upon accesses to rows in physical proximity [40]. By repeatedly accessing a row, an attacker can corrupt data in adjacent rows to undermine system security, *i.e.*, to gain kernel privilege from unprivileged applications [60], [10], [23], [66], [68], [41] and web browsers [24], escape web browser sandboxes [60], [21], [19], hammer from inside secure enclaves [23], [33], escape virtual machines [72], attack over the network [67], [47], and read memory to extract encryption keys [43]. These attacks motivated a long list of research on Rowhammer mitigations.

Rowhammer mitigations focus on *detection*, *neutralization*, or *elimination* [23] in software or hardware. *Detection*-based mitigations use static code analysis, performance counters from software, and the analysis of memory access patterns in hardware. However, they can be circumvented [23], [41]. *Neutralization*-based mitigations tolerate bitflips but restrict exploitation by physically distancing *aggressor* and *victim* rows [14], [42]. However, some hammer patterns can still succeed [40], [41]. *Elimination*-based mitigations are mainly hardware solutions, e.g., doubling the refresh rate and error correction methods like ECC DRAM and Chipkill, with limited effect on Rowhammer [40], [16]. Target Row Refresh (TRR) is designed specifically against Rowhammer. It can, however, be bypassed by exhausting the number of counters [22], [19], [34] or by using more advanced access patterns, e.g., half-double Rowhammer [41]. Thus, modern devices are still vulnerable [41], [34], highlighting the need for effective Rowhammer mitigations.

In this paper, we propose CSI:Rowhammer, a novel Rowhammer mitigation. Instead of focusing on specific properties of Rowhammer that can later turn out to be incomplete, e.g., flips happen only in directly neighboring rows [42], [41], [14], or at least two rows in a bank must be accessed [8], [23], the idea of CSI:Rowhammer is to provide more principled security guarantees. CSI:Rowhammer roots its security in a cryptographic message authentication code (MAC) for cryptographic security and integrity. It combines MAC-based error detection in hardware and flexible error correction in hard- and software. CSI:Rowhammer stores its MAC next to the data on the DRAM similarly to ECC memory and has, therefore, the same memory overhead.

With CSI:Rowhammer, the MAC is computed and compared in the memory controller upon every DRAM access. If data was corrupted and the comparison fails, the memory controller tries to correct a single bitflip. If unsuccessful, it raises a new exception which the kernel handles by trying to correct the data using different strategies. For this purpose, the MAC computation is also implemented as a new CPU instruction. A simple strategy, the universal fallback, is our parity-guided search, which outperforms a direct brute-force search that can become inefficient for higher numbers of bitflips. With this approach, we can correct 5 bitflips in a 256-bit data word in less than $300\,\text{ms}$. Beyond this, CSI:Rowhammer also enables more advanced strategies. It will, for instance, not correct errors in unmodified file-backed pages and reloads them from the disk instead, enabling the correction of any number of bitflips practically instantly. Another advantage of the correction in software is that CSI:Rowhammer enables OS vendors to implement an interface that allows admins to define reliability levels per process and page. For example, the maximum number of bitflips searched before killing a very important process in contrast to a process that can easily be restarted.

We analyze the detection and correction abilities of CSI:Rowhammer and show that it achieves significantly higher security and integrity than state-of-the-art error correction. In contrast to ECC memory, CSI:Rowhammer does not have a strict upper bound or constraints on the number and location of detectable bitflips. To break CSI:Rowham-

© 2023, Jonas Juffinger. Under license to IEEE.
DOI 10.1109/SP46215.2023.00014

mer, an attacker needs to forge a cryptographic MAC, which is infeasible with the means of Rowhammer bitflips.

We evaluate the performance of CSI:Rowhammer based on a proof-of-concept implementation in a CPU and memory controller in gem5, running a modified Linux kernel. We show that our hardware-software co-design allows for low-latency, high-throughput integrity checking and, therefore, a performance impact of less than $0.75\,\%$ under normal operation. While the average latency to correct a single bitflip is less than $20\,\mathrm{ns}$, CSI:Rowhammer can detect any number of bitflips and correct up to 8 bitflips in 256 bits of data within a practical time frame. CSI:Rowhammer has less than $0.01\,\%$ overhead on the CPU area. This underlines that CSI:Rowhammer is a practical mitigation that should be deployed to fully and permanently mitigate Rowhammer.

Contributions. We make the following contributions:

1) We propose CSI:Rowhammer, a hardware-software co-design to fully mitigate Rowhammer attacks using a cryptographic MAC for error detection in hardware.
2) We propose a novel software-level correction mechanism using a parity-guided bit-correction search as well as sophisticated correction strategies.
3) We evaluate CSI:Rowhammer with a proof-of-concept[1] implementation in gem5. We show that overheads are minimal, e.g., the performance overhead is below $0.75\,\%$ while the security level is raised to cryptographic levels.

Outline. First, we provide background in Section 2 and an overview of CSI:Rowhammer in Section 3. Hardware and software changes are discussed in Section 4 and Section 5. We evaluate the security and performance of CSI:Rowhammer in Section 6 and discuss related work in Section 7. In Section 8, we discuss the compatibility to other technologies and possible improvements and conclude in Section 9.

2. Background

In this section, we discuss DRAM, Rowhammer and its properties, as well as lightweight cryptography.

2.1. DRAM

The DRAM has multiple levels, allowing parallel access to the last level, the banks, to maximize throughput. Banks are divided into *rows* of *cells*, *i.e.*, the transistors and capacitors storing the data. A DRAM cell's charge depletes over time, requiring refreshes every $32\,\mathrm{ms}$ to $64\,\mathrm{ms}$ [35] to prevent data loss. These refresh commands are typically interleaved with normal data accesses by the memory controller, which is responsible to refresh each row within this time window.

Error Correction Code (ECC) and Chipkill memory have an additional memory chip to store data with correction codes [16], 8 redundancy bits per 64 bits (DDR4) or 32 bits (DDR5) of data. These are transferred over a wider data bus of 72 bits and 40 bits, respectively. Chipkill is a DRAM error correction method, which, using the same additional memory, can correct up to 8 bitflips coming from a failure of a single DRAM chip [20]. ECC and Chipkill can correct errors from a single faulty DIMM contact. We compare ECC and Chipkill to CSI:Rowhammer in Section 7.2.

1. Proof of Concept URL: https://github.com/IAIK/CSIRowhammer.

DRAM Errors. In large-scale systems, DRAM errors from various physical causes like cosmic rays [63] or random occurrences, increasing with aging cells and temperature [59] are well studied. Schroeder et al. [59] monitored the majority of Google's server fleet for 2.5 years starting in 2006. They observed failure rates of up to $70\,000$ per billion device hours (FIT) per Mbit of DRAM capacity. On the worst motherboard and memory configuration, $0.03\,\%$ of DIMMs saw an uncorrectable error in one year. Hwang et al. [29] studied DRAM errors on four systems and observed similar failure rates. Out of the $40\,960$ nodes monitoring Chipkill errors where bitflip recovery failed, $1.34\,\%$ had at least one Chipkill error in 583 days. Bautista-Gomez et al. [9] studied the frequency of multi-bit errors, focusing on independent errors, *i.e.*, they only count errors from the same word once. In total, they observed $55\,000$ independent memory errors in over a year, of which 85 ($0.15\,\%$) were not correctable.

2.2. Rowhammer

In 2014, Kim et al. [40] discovered the *Rowhammer effect*, *i.e.*, bitflips can be induced in DRAM through disturbance errors triggered by frequent memory accesses. For a more detailed background on Rowhammer, we refer the reader to this work or [50] for an overview. We discuss and compare mitigations in Section 7.

Data Dependency. Rowhammer causes a bit flip by discharging the capacitor in a neighboring cell. For this to happen the neighboring capacitor in the aggressor row must be discharged [40], [69]. Therefore, there is a correlation between the data in the aggressor rows and the flips, usually the victim adopts the values of the aggressor row. Kwong et al. [43] used this dependency to leak encryption keys with RAMBleed. If the orientation of the cells (true-cell / anticell) is inverted between the aggressors and victim row, the inverted values of the aggressor row are adopted [69].

Rowhammer Bitflips. Kim et al. [40] examined the number of multi-bit errors caused by Rowhammer per single 64-bit word. The worst tested module had $1.9\,\%$ uncorrectable errors, with $0.002\,\%$ being undetectable by ECC, and Rowhammer is getting worse with every new DRAM generation and increasing density. LPDDR4 requires less than a 10th of the activations than DDR3 memory to hammer [53], [36].

Cojocar et al. [16] used a public Rowhammer bitflip database with 14 DIMMs [65] to craft the first Rowhammer attack on ECC memory. Their AMD CPU raises a machine check exception when detecting an uncorrectable bitflip, while the Intel CPU *ignores* it. On AMD, on average, $7.42\,\%$ of bitflips were undetectable; on Intel, $10.89\,\%$ were uncorrectable and, therefore, usable for an attack. They also found that many ECC implementations use more complex symbol-based correction methods [16], capable of *detecting* up to 4 bitflips if they are in different symbols but not able correct multi-bitflips. Hassan et al. [26] flipped up to 7 bits in a single 64-bit data word. This is beyond the detection and correction capability of every ECC implementation.

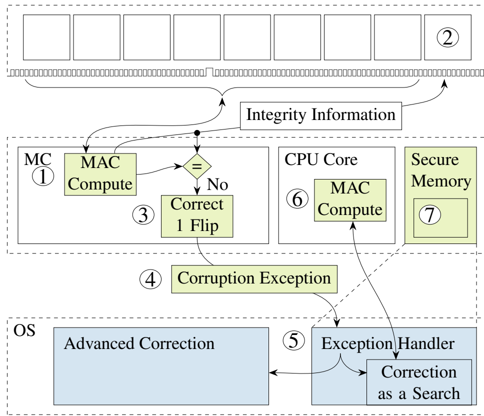


Figure 1: High-level overview of the hardware and software components of CSI:Rowhammer.

2.3. Lightweight Cryptography

Modern cryptography is often not designed for constrained devices that face various limitations but for high-performance devices. Lightweight cryptography offers algorithms designed to provide security and privacy guarantees while enabling their implementation on devices constrained in their energy consumption, die area, or latency [52].

Lightweight Message Authentication Codes (MAC). A message authentication code (MAC) is an authentication tag computed from a message input and a secret key [51]. The key protects the integrity and authenticity of the message: Only the owner of the key can compute and verify the MAC of a message. The attacker cannot forge a valid MAC of a new or modified message. Most widely-deployed MACs are based either on hash functions (like HMAC [51]) or on block cipher chaining (like CMAC), both of which are inherently sequential and thus have a high latency. A lightweight design for short inputs is SipHash [5], but its ARX design is more suited for software and not optimized for latency in hardware. More recently, tweakable block ciphers (TBCs) [48] have emerged as a promising lightweight primitive. TBCs are similar to block ciphers, but have an additional public tweak input that selects the encryption permutation together with the key. QARMA is a TBC with low latency [6].

3. CSI:Rowhammer

The idea of CSI:Rowhammer is to protect the integrity of all data in the DRAM with a cryptographic MAC to mitigate software-based DRAM fault attacks like Rowhammer but also randomly occurring bitflips. It is designed to work on a large variety of systems, from smartphones to large-scale virtualized server environments. As illustrated in Figure 1, CSI:Rowhammer is a hardware-software co-design.

On every access to main memory, CSI:Rowhammer uses a cryptographic MAC to detect if data has been corrupted. The computation and comparison of these MACs are performed in the memory controller ① and stored with the data ② to enable low-latency accesses. The MACs are stored on the same memory chip next to each other to make

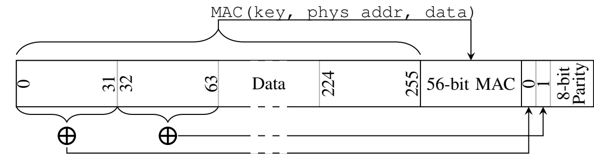


Figure 2: Partitioning of the data, MAC, and parity bits. The data is 256 bits (DDR5) or 512 bits (DDR4).

hammering them difficult. MACs cannot directly correct data. The memory controller tries to correct a single flip directly in hardware ③ by correction as a search. If this fails, this task is handed over to the operating system ⑤ by raising a data corruption exception ④. The exception handler uses different correction procedures to reconstruct the correct data. Using a new dedicated instruction set extension ⑥, the data correction is performed efficiently. However, corruption could also occur in the exception handler. Hence, it is stored in a dedicated secure on-die memory ⑦ that is not vulnerable to Rowhammer, similar to a CPU cache.

Detecting Bitflips. CSI:Rowhammer uses ECC memory for its additional space to store the integrity information (MAC and parity), as shown in Figure 2. The usage of a MAC allows CSI:Rowhammer to detect all memory corruptions, unaffected by their number, location, or distribution, which is enough to mitigate all Rowhammer attacks other than denial of service. We use a MAC based on the QARMA block cipher [6], designed for low latency, power, and area and used on ARM for pointer authentication [4]. With this basis, CSI:Rowhammer can be integrated into CPUs from smartphones to servers with negligible performance impact.

Typical ECC DDR4 memory stores 8 redundancy bits for 64 bits of data, ECC DDR5 for 32-bits of data. CSI:Rowhammer uses a 56-bit MAC and 8 parity bits, summing up to 64 bits in total. Thus, we protect 512 bits of data on DDR4 or 256 bits on DDR5. ECC data words can also be larger than 64 bits [16], making efficient correction and secure detection impossible. A trade-off can be a reduced correction capability in favor of an equally strong detection.

The MAC's key is randomly generated on each boot by the CPU internally. The memory controller and CPU cores have access to it, but it is not exposed to the software level. The memory controller computes the integrity information for every memory access. On a write operation, it computes the new MAC and parity and stores them in the DRAM. On a read, it computes and compares the MAC and parity with the stored one. The MAC computation is also implemented in every CPU core as a new instruction to enable quick and efficient multi-bit error correction from software.

Correcting Bitflips. The memory controller corrects single bitflip errors caused by corrupted data, MAC, or parity and permanent errors caused by a faulty contact as-a-search.

If it fails because there is more than one bitflip, it raises an exception for the operating system. For instance, the advanced error correction algorithm can reload disk-backed data directly. Other data errors with no additional information are corrected by searching for the bitflips in the data. Our instruction set extension allows to access raw

data and compute the MAC efficiently in hardware without software access to the key. Our search algorithm is guided by parity bits and brute forces possible corrections within each parity block. Thus, the search time increases exponentially with the number of bitflips. Depending on the importance of the process with the corrupted data or instruction, the operating system can decide whether to continue the search or kill and restart the process, or even halt the system. More generally, these decisions are up to the system vendor or administrator. OS vendors can decide on correction steps and to which extent they are tweakable by administrators, e.g., the maximum number of flip corrections for the kernel or user space. In case of a successful correction, the operating system can remember the necessary information for future corrections of the same physical location. If the correction is impossible, CSI:Rowhammer still mitigates Rowhammer exploits, as the detection is sufficient to stop an attack.

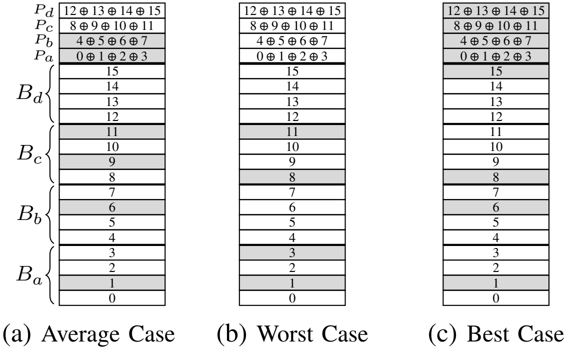
Protecting the Correction Mechanism. The exception handler for the data corruption exception must not be corrupted itself. Otherwise, a correction is not possible anymore, and the system must halt. For CSI:Rowhammer, we propose a small secure on-die memory, similar to a cache but mapped to a physical address range. The operating system puts all code required for correction into this secure memory.

Limitations. The goal of CSI:Rowhammer is to provide a negligible performance impact while fully mitigating Rowhammer. However, it does not protect against *all* data integrity attacks. In our threat model, the attacker can flip bits with Rowhammer, which is difficult to do precisely and allows only a limited number of flips. Replay, relocation, or substitution attacks, which require physical access to the memory bus, are out of scope. CSI:Rowhammer does not give any correction guarantees for a high number of bitflips, it can correct typical numbers of bitflips caused naturally or by Rowhammer. The primary goal is the detection of memory corruption. CSI:Rowhammer detects *all* bitflips except the negligible portion resulting in MAC collisions due to the cryptographic design (see Section 6.7).

3.1. Error Correction as a Search

ECC memory uses symbol-based error-correction codes to detect a certain number of bitflips and usually correct one bitflip [16]. A MAC can only detect bitflips in the data but cannot correct them or give information about their location. To find the correction where no other strategy can be used, we have to flip bits and compute and compare the MAC successively. We find it when the MACs match. We start with flipping one bit and if the correction is not found, continue with all permutations with two flips and so on. Huang et al. [28] defined this search for bitflips in corrupted data as *Error Correction as a Search*. While our approach follows the same principle, the implementation differs.

Parity Bits. To improve the correction time, we use parity bits to find locations of flips. The 8 parity bits are computed by XORing 8 blocks of the data, as shown in Figure 2. The chosen partitioning allows for an efficient correction of permanent errors, e.g., on a transmission line or contact (see Section 4.4). We use Figure 3 to show how to get bitflip



(a) Average Case (b) Worst Case (c) Best Case

Figure 3: Bitflip locations and parity bits. Gray data cells show a bitflip, and gray parity bits a corresponding parity mismatch. In the worst case, blocks (B_a - B_d) contain only an even number of flips. In the best case, every parity bit corresponds to one or zero flips.

locations from the parity bits. In Figure 3a, the non-matching parity bits indicate an odd number of flips in B_a and B_b. The matching parities indicate an even number or no bitflips in B_c and B_d. In 3b, there is a mismatching MAC. The parity bits indicate an even number of flips in at least one block. In 3c, there is an odd number of bitflips in all blocks.

Correction Algorithm. We first consider only flips in the data and explain bitflips in the MAC and parity next. The correction algorithm pseudocode is attached in Appendix A.

In Figure 3a, there are at least 2 bitflips, one in B_a and B_b. We try all permutations of these two flips. Because of the bitflips in 9 and 11, the MACs never match, and the correction continues. We add a double flip into our permutations that can be in any of the 4 blocks. While testing these permutations, we have find the correct data.

MAC Corruption. Error correction as a search in the data assumes that the MAC is uncorrupted. However, as the MAC is stored next to the data on the DRAM, it is also vulnerable. In summary, there are four possible scenarios:

- No corruption, the MACs are equal.
- MAC corruption, the resulting MAC and the one in the DRAM only differ by a few bits.
- Data corruption, the resulting MAC differs significantly.
- Data and MAC corruption, the resulting MAC and the one in the DRAM differ by few bits when data is corrected.

CSI:Rowhammer performs error correction as a search and does not check the MAC for equality but approximate equality. If the data is correct, bitflips in the MAC can be found quickly, by checking for approximately equality. The limitations and security of approximate MAC equality are evaluated in Section 6.6. The single MAC computation for this check is done prior to the bitflip search on the data.

Parity Corruption. CSI:Rowhammer can correct data with a maximum of one flip in the parity bits. This is a limit we choose for a good compromise between correction time and capability. The parity bits make up only $2.5\,\%$ (DRR5) or $1.4\,\%$ (DRR4) of the data and integrity information, and 2 flips are, therefore, a reasonable limit. Error detection is not compromised by the chosen limit of one parity flip.

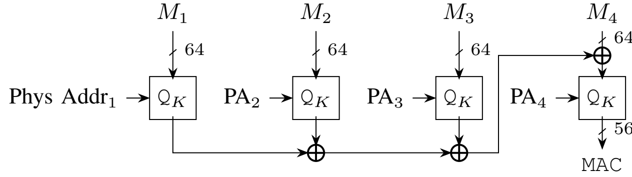


Figure 4: CSI:Rowhammer PMAC for 256-bit data [57]. The output of the fourth QARMA block is truncated to $56\,\mathrm{bit}$. The physical addresses used as tweaks are shifted right by $\log_2(64) = 6\,\mathrm{bit}$. They are unique for every message block.

To take a parity bitflip into account, we add one additional step to the correction algorithm. We again take Figure 3a and assume P_c to be flipped. In the first step, the algorithm will try the correction with 3 bitflips corresponding to the mismatching parity bits. This attempt fails, and the additional step is performed. Every parity bit is flipped successively, and the new expected number of bitflips is calculated. If P_d is flipped, the new expected number of bitflips is 4. If P_c is flipped, it is 2. The first step is repeated with all parity flips that decrease the number of expected bitflips. The correction with this additional step fails, so a double flip is assumed without flipping parity bits, which also fails. The additional step is added again. During this step, P_c is flipped, and the correction is attempted with the 2 single bitflips from P_a and P_b and one double flip. This correction succeeds, similarly to the first example.

Limitations In theory, it is possible to correct any number of bitflips this way. However, there are two upper bounds: First, the maximum number of flip permutations must not exceed the security boundaries of the MAC (see Section 6.6). Second, the number of permutations and computation time increases exponentially with the number of flips, limiting this search method to lower numbers of flips.

3.2. Cryptographic Design of the MAC

For a low-overhead mitigation, we require a MAC with very low latency, low power, and low hardware area requirements. QARMA is a lightweight tweakable block cipher design [6] fulfilling these requirements, and is notably already used in practice by ARM for pointer authentication [4]. QARMA allows pipeline stage boundaries between any two rounds [6] and supports block sizes of 64 or 128 bits.

CSI:Rowhammer MAC. We need to map 256-bit or 512-bit inputs to 56-bit tags. We use a parallel MAC (PMAC) construction [13], [57], [49] with 64-bit message blocks, where the physical address of each block is used as tweak for QARMA (see Figure 4). The resulting 64-bit tag is truncated to 56 bits to compute the final MAC. The security of the MAC is evaluated in Section 6.6. Table 1 shows latencies of the QARMA cipher variants [6], different MAC computation methods, and CPU memory accesses for comparison. The CSI:Rowhammer MAC is built using one QARMA cipher with 6 pipeline stages, resulting in the following latency for a 256-bit input (see Figure 4 and Table 1):

$$2.20\,\mathrm{ns} + 2.20\,\mathrm{ns} \cdot \frac{1}{6} \cdot 2 + 2.20\,\mathrm{ns} = 5.13\,\mathrm{ns}$$

TABLE 1: Latency comparison of a single QARMA invocation, simulated for a $7\,\mathrm{nm}$ node similar to modern CPUs [6], the PMAC construction using pipelined QARMA$_5$-64-σ_0, and CPU memory accesses [18]. The throughput defines how many calls fit in the pipeline of a single QARMA block.

Operation	Latency	Throughput
QARMA$_5$-64-σ_0	2.20 ns	6
MAC$_{\text{PMAC}-256}$	5.13 ns	$\frac{\text{6 pipeline stages}}{\text{4 invocations}} = 1.5$
MAC$_{\text{PMAC}-512}$	6.60 ns	0.75
L2 cache access	3.86 ns	–
L3 cache access	10.27 ns	–
DRAM access	113 ns	–

4. Hardware Modifications

The generic design of CSI:Rowhammer can be integrated into different architectures. However, for our PoC (see Section 6.1) and this section, we focus on the changes for x86.

CSI:Rowhammer implements the MAC computation in hardware to minimize the overhead for DRAM accesses and allow the OS to compute it quickly during a correction. The MAC key is randomly generated. The memory controller computes and compares the MAC, and corrects single-bit errors and errors from faulty contacts. It provides a small secure memory to store the code required to handle a correction exception. Finally, the CPU cores must provide instructions to access corrupted data and compute the MAC.

4.1. MAC Key Generation

CSI:Rowhammer uses a MAC with a CPU-internal key generated using a cryptographically secure random number generator at boot [30], [1], [3]. At runtime, this key is available to the memory controller and all CPU cores in a software-inaccessible register. Multi-processor systems share the DRAM and, therefore, the key, over the processor interconnect. Attacks with physical access are out of scope.

4.2. Integrity Check

The memory controller is responsible for detecting data integrity failures in DRAM. It computes the integrity information for all memory accesses. On writes, the memory controller computes and stores the integrity information on the DRAM. On reads, it retrieves the integrity information and compares it with the computed one. A mismatch indicates a corruption of the data or integrity information.

4.3. Single-Bit Correction

If a corruption is detected, the memory controller starts the correction by performing a correction as a search for a single bitflip in hardware. A corruption in the MAC is found during the integrity check verification by checking for approximate equality. If the corruption is in the data, one parity bit provides the location of the affected data block. In the worst case, the correction requires 32 or 64 MAC computations for 256 and 512 bits of data, respectively, half of that on average. If the integrity check reveals errors in the parity bits, the parity bits are recalculated.

During the search, only one of the 4 or 8 input blocks changes. The other blocks can be computed once and reused. Thus, the subsequent MAC computations are two QARMA calls with a total latency of $4.40\,\mathrm{ns}$ and 12 pipeline stages. This results in an average single bitflip correction time of:

$$5.13\,\mathrm{ns} + 16 \cdot \frac{4.40\,\mathrm{ns}}{12} = 11\,\mathrm{ns}$$

For DDR4 memory, it is $18.3\,\mathrm{ns}$. If the parity bits indicate two or more flips, the correction for a faulty contact is tried.

4.4. Permanent Errors from a Faulty Contact

Permanent faults from, e.g., a faulty DIMM contacts or transmission lines make up a considerable part of DRAM errors [62]. They must be corrected in hardware, as they cannot be fixed by writing back correct data [63]. Similar to ECC and Chipkill, CSI:Rowhammer is able to correct the errors caused by a singly faulty contact. The correction is performed before raising a corruption exception, if the MAC and more than one parity bit mismatch. For this explanation, we assume that a faulty contact causes bits to *stick-at-0*.

The memory controller first computes the logical `OR` over blocks with the size of the DRAM data bus width. The `OR` result shows the stuck contact offset as these bits are always ‘0’. The parity bits mismatch for the blocks where the stuck contact had an effect. Together, both block and bit offset are identified. With this, the correction requires as many MAC computations as there are ‘0’s in the `OR` result. With random data, there is a probability of $11.8\,\%$ (32-bit bus) or $22.2\,\%$ (64-bit bus) that there is an additional ‘0’, not caused by a faulty contact, resulting in a correction time of $5.74\,\mathrm{ns}$ and $8.05\,\mathrm{ns}$, respectively. This the CSI:Rowhammer performance impact, but keeps the system usable.

For stuck integrity information contacts, it works similarly. It affects 7 bits in the MAC and 1 in the parity bits. Correct data can be recognized if all bits at an offset are stuck and the rest of the MAC is correct. This reduces the effective MAC size to $56 - \frac{7}{2} = 52.5\,\mathrm{bit}$. We leave the design and analysis of a faulty contact correction with bitflips in the DRAM for future work. CSI:Rowhammer cannot correct both at the same time, similar to ECC and Chipkill.

4.5. Data Corruption Exception

CSI:Rowhammer adds one new exception to inform the operating system about a data corruption not correctable in hardware. We call it *data corruption exception* and use one of the reserved vectors 21-31 [31]. The data corruption exception is designed as a fault, allowing to continue at the faulted instruction after the correction. The handler receives the physical address of the corrupted data or instruction from an architecture-dependent register. We propose the same register used for page faults, the `CR2` on x86 [31]. It contains the physical address to not rely on page tables. In virtualized environments, the host receives the exception, translates the host physical address (HPA) to the guest physical address (GPA), and forwards it to the guest. This mechanism allows transparent correction for guests not supporting CSI:Rowhammer via the host (see Section 8.1).

TABLE 2: The three instructions of CSI:Rowhammer used for efficient and secure correction in software.

Instruction	OPs	Description
`csi_mac`	ZMM rx rx	Calculates the MAC of the data (OP1) for the physical address (OP2) and writes it into OP3.
`csi_load`	rx ZMM rx	Reads from physical address (OP1) the raw corrupted data (OP2) and integrity information (OP3).
`csi_xchg`	rx ZMM rx ZMM	Writes new data (OP4) into the physical address (OP1) if the old data (OP2) and MAC (OP3) are unchanged at physical address (Listing 4 in Appendix B).

4.6. Secure On-Die Memory

A corruption must not happen in any parts of the OS required to correct corruptions, otherwise, the system must halt. To guarantee no corruption for the exception handler code, IDT, and GDT, the CPU provides a small on-die random access memory.[2] It is part of the physical address range and the OS can map it into virtual memory with non-evictable TLB entries. For systems without an MMU, the memory is at a fixed physical address. It is 4 pages large to fit our prototype implementation of the correction as a search algorithm (0x1500 bytes), the IDT, and GDT.

In virtualized environments, the secure memory is simulated for the guests in the DRAM. In case of a corruption in the simulated secure memory, the host transparently corrects it. This guarantees the incorruptibility of the secure memory for all guests without requiring a larger SRAM.

4.7. CPU Cores

The operating system corrects data with more than one bitflip. The CSI:Rowhammer hardware modifications support the operating system with instructions to compute the MAC and access the raw corrupted data and integrity information.
Instruction Set Extension CSI:Rowhammer adds three privileged instructions, shown in Table 2, to allow for correction in software. `csi_mac` takes the 256-bit or 512-bit data in an AVX register and the physical address, *i.e.*, the tweak, and writes the computed MAC back to a general-purpose 64-bit register. Implementing the MAC computation in hardware allows the MAC key to stay secret.

Additionally, two new instructions access the raw data and integrity information by its physical address without raising an exception. The `csi_load` instruction loads the corrupted data into a ZMM register and the integrity information into a 64-bit register. The `csi_xchg` instruction writes the corrected data from a ZMM register back into the memory if it did not change in the memory while the correction was running. Section 8.4 describes the purpose of `csi_xchg`, *i.e.*, prevent correction race conditions. Listing 4 in Appendix B shows its pseudo-code.

2. On-die area is not the primary limitation today anymore. Some Intel processors utilized unused die area already with the integration of, e.g., $128\,\mathrm{MB}$ DRAM chips, the Apple M1 quadrupled the L1 size with marginal thermal and efficiency changes, as they are using 4 times larger pages.

Within a virtualized guest, the instructions transparently translate the given GPA internally to an HPA to match the MAC's tweak and the actual physical location. The last translation is cached for quick succeeding executions.

Nesting Bit We need to detect nesting of the corruption exception handler (see Section 5.2). For this, we need one bit in a register that is unique for every thread. We propose using the highest bit of the CR3 register, as it is unused, and its value is already unique for every task. The only hardware change is to allow writing this bit without flushing the TLB.

Non-evictable TLB Entries To ensure compatibility with the IDT and the CPU's frontend, the secure memory is addressable with virtual addresses. The paging structures to map the secure memory to a virtual address also need to be protected against Rowhammer. We propose reusing existing mechanisms and allowing the locking of secure memory pages in the TLB [2]. This allows the CPU to resolve addresses directly from the TLB. We propose using an MSR interface to configure these virtual to physical mappings.

5. Software Modifications

The memory controller detects data corruption and tries to correct single bitflips and errors from faulty contacts. If it fails, the operating system takes over to try the correction and limit the impact of an uncorrectable corruption.

5.1. Exception Handling

The operating system must register a new handler for the data corruption exception. The handler and the correction-as-a-search code are secured against corruption by residing in the secure on-die memory provided by the CPU. The advanced correction procedure uses various kernel code and structures and is, therefore, not protected against bitflips. If kernel code or data is corrupted and accessed during a correction, another exception is raised and handled in a nested handler. The nested handler detects the nesting (see Section 5.2) and attempts a correction as a search without accessing any data outside the secure memory.

The kernel stack, used by the exception, is unique for every task, and can, therefore, not be in the secure memory, and is vulnerable to bitflips. To deal with corruption in the kernel stack, we forward the kernel stack pointer 64 bytes before accessing it. In case of corruption, the following exception handler moves the kernel stack pointer out of the corrupted data block. It can then correct the kernel stack and continue with the previous exception handler.

5.2. Nesting Detection

Detecting the nesting of our exception handler is important as it means that there was a corruption during an advanced correction attempt. If that happens, the exception handler must not run the advanced correction and can only correct by search. Listing 1 shows how the nesting detection is implemented using the highest bit in the CR3 register. To detect nesting, we use one bit, called the *nesting bit*. If the current Linux task is in the corruption exception handler it is 1 and if not, it is 0. It must fulfill two conditions:

```
1  corruption_exception_handler() {
2    if (has_nested_bit_set(CR3)) {
3      enable_interrupts();
4      error_correction_as_a_search(corruption_address);
5    } else {
6      set_nested_bit(CR3);
7      enable_interrupts();
8      advanced_error_correction(corruption_address);
9      clear_nested_bit(CR3);
10   }
11 }
```

Listing 1: Pseudocode of the corruption exception handler with nesting detection. The advanced error correction is guarded by the nested flip being one. If a second exception is raised the error correction as a search is executed.

```
1  error_correction_as_a_search(corruption_address) {
2    // get the raw corrupted data
3    data, integrity = csi_load(corruption_address);
4    data_copy = data;
5    mac = integrity & 0x0FFFFFFFFFFFFFFF;
6    parity = (integrity & 0xF000000000000000) >> 56;
7
8    correction_as_a_search(parity, [&](flip_mask) {
9      data ^= flip_mask;
10     // compute the mac of the data
11     computed_mac = csi_mac(data, corruption address);
12     if (popcount(mac ^ computed_mac) < 4) {
13       goto found_correction;
14     }
15     data ^= flip_mask;
16   });
17   // no correction found, kill the process or panic
18   panic();
19 found_correction:
20   // write the correct data back if it did not change
21   csi_xchg(corruption_address, data_copy, mac, data);
22   reti;
23 }
```

Listing 2: Pseudocode showing the usage of the `csi_load`, `csi_mac` and `csi_xchg` instructions.

- When the corruption exception happens, the nesting bit must already be stored in a CPU register as retrieving it from memory could cause another corruption exception.
- The bit is either set or not set for each task, *i.e.*, process or thread. Scheduling is enabled during correction, but two independent tasks can have two (independent) exceptions.

5.3. Locking

We use a lock in the advanced correction preventing two tasks from correcting the same data. If a task accesses data already being corrected, it waits, and the CPU core can run other tasks. This lock does not exist in a nested handler as it cannot access kernel resources. In that case, the `csi_xchg` instruction prevents the unintended overwriting of new data with old corrected data (see Section 8.4). The locking of shared kernel resources allows for a theoretically unlimited number of simultaneously running advanced corrections.

5.4. Correction Procedure

The correction procedure uses the great flexibility that can be achieved with the knowledge available to the operating system. File-backed pages like memory-mapped files, page cache pages, or the kernel itself can be reloaded from the

disk, correcting any number of bitflips. Taking into account that Rowhammer bitflips are reproducible [40], the kernel can remember the locations of previously corrected bitflips and use them for future corrections. If the page cannot be reloaded from the disk and is not in the cache, the correction is performed as a search. If successful, the page is reallocated to minimize the chance of a corruption in the near future. If the correction as a search was unsuccessful, the operating system can behave differently. OS vendors can enable to define different reliability levels, for example, the maximum duration of the search based on the importance of a process or the impact of a restart of a process, e.g., a database server vs. a service logging system stats. If the uncorrectable corruption is in the kernel, it can only try to shutdown and restart as gracefully as possible. Figure 8 in Appendix E shows the correction in a flow chart.

Listing 2 shows the usage of the `csi_load`, `csi_mac` and `csi_xchg` instructions. First, `csi_load` is used to get the data and integrity bits from the physical `corruption_address`. The data is duplicated for the `csi_xchg`. During the correction as a search, the `csi_mac` instruction is used to compute the MAC. If data was found with an approximately equal MAC, it is written back into the DRAM with the `csi_xchg` instruction. It checks if the data was modified in the meantime and only writes it if it is unmodified.

6. Evaluation

To test the performance, correct functionality, and security of CSI:Rowhammer, we evaluate it using different techniques. We build a prototype with gem5 running a modified Linux, simulate corrections to test our algorithm, and calculate the security boundaries of our MAC under different scenarios.

6.1. Prototype Implementation with gem5

We implement a prototype of CSI:Rowhammer's hardware modifications in gem5 [12] and the required software changes in the Linux kernel. We use gem5 in the full-system emulation mode running an up-to-date Debian buster with our modified Linux kernel 4.19, as shown in Table 3.

Memory. There is no ECC memory in gem5. Therefore, we first modify the memory controller to extend its memory bus to 72-bit to simulate ECC DIMMs. We add the integrity information computation and check to the memory controller with the latency for the MAC computation and the official QARMA software implementation from the NIST submission [7]. The memory model of gem5 does not simulate any memory flaws like spontaneous bitflips or Rowhammer. To test CSI:Rowhammer we add an interface to cause corruptions in the DRAM. Similar to related work [58], [15] we do not use a cycle accurate DRAM model like dramsim3 [46] or dramsys4 [64], because CSI:Rowhammer does not change the number, frequency or ordering of DRAM accesses.

CPU. We add a data corruption CPU exception that is raised by the memory controller if a MAC mismatch is detected, and the three new instructions for the correction. As gem5 does not support AVX, we use code by Wang et al. [70] that adds basic support, sufficient for our use.

TABLE 3: The gem5 system used for our evaluation.

Architecture	x86-64
CPU	Single Core, 3 GHz, O3 (Out-Of-Order) CPU
DRAM	2 Channel, 4 GB DDR4_2400_8x8
Cache	64 kB L1I, 32 kB L1D, 2 MB LLC
gem5 Mode	Full System
Operating System	Debian Buster with Linux 4.19

Linux. A modified Linux kernel is required to support the new exception and perform the correction. We add the exception handler that performs the correction as a search. As we introduce the new `csi_` CPU instructions, there is no compiler support. Hence, we call them using inline assembly to insert raw bytes into the machine code directly; for an example, see Listing 5 in Appendix C.

Limitations. As the secure memory has no impact on the performance or correction, we do not implement it. The secure memory is only required to prevent corruption in the correction code which cannot occur in our simulation.

6.2. Performance Evaluation

To evaluate the performance of CSI:Rowhammer, we first measure the performance impact without memory corruptions using the PARSEC [11] and SPLASH2x [56] benchmarks. Second, we estimate the duration of the data correction in software, depending on the number of bitflips.

Runtime Performance Overhead We evaluate the performance impact of CSI:Rowhammer during normal operation without memory errors for the PARSEC [11] and SPLASH-2x [56] benchmarks, which cover a wide variety of memory intensive and non-intensive workloads. We ran the entire benchmarks and compare the overall time they took with or without the delay added by CSI:Rowhammer. The performance degradation is caused by the MAC computation performed on every memory access. We use the MAC computation durations for 256-bit data ($5.13\,\mathrm{ns}$) and 512-bit data ($6.60\,\mathrm{ns}$) from Table 1. The overhead for the different applications is shown in Figure 5. The geometric mean overhead over all benchmarks is $0.67\,\%$ (n = 45, $\sigma_{\bar{x}}$ = 0.12) on the system with a $5.13\,\mathrm{ns}$ delay and $0.74\,\%$ (n = 25, $\sigma_{\bar{x}}$ = 0.17) with a $6.6\,\mathrm{ns}$ delay. The worst overhead is $3.28\,\%$ (n = 55, $\sigma_{\bar{x}}$ = 0.013) and $4.24\,\%$ (n = 30, $\sigma_{\bar{x}}$ = 0.017), respectively. This overhead is small in comparison to the security that CSI:Rowhammer can provide. Figure 7 in Appendix D shows the memory accesses of every benchmark. Benchmarks with a high number of memory accesses also experience a worse performance impact, confirming that the slowdown comes from the MAC computation overhead.

Simulated Correction Overhead Evaluation We evaluate the duration of the correction as a search. For data that can be corrected through other means, e.g., reloading from disk, any number of bitflips can be corrected in a very short time. The gem5 simulator is well suited for relative performance comparisons but not for absolute run time numbers. To give realistic numbers for the average correction duration on a modern CPU that implements CSI:Rowhammer, we simulate the correction algorithm on three physical CPUs.

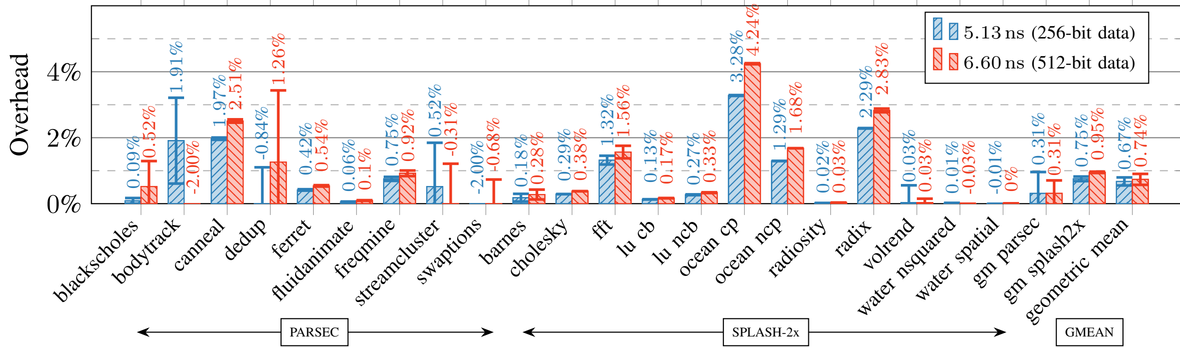


Figure 5: PARSEC and SPLASH-2x benchmarks run in our modified gem5 with two different integrity check delays. The worst overhead of CSI:Rowhammer is 4.24 % (n = 30, $\sigma_{\bar{x}}$ = 0.017), the geometric mean is 0.67 % (n = 45, $\sigma_{\bar{x}}$ = 0.12) for 256-bit data and 0.74 % (n = 25, $\sigma_{\bar{x}}$ = 0.17) for 512-bit data.

TABLE 4: The number of MAC computations in the best, average, and worst case, based on the flip distribution, see Figure 3. Followed by the theoretical MAC duration, and the permutation loop durations for three CPUs. The higher MAC or loop duration is a good estimate for the respective CPU. 256 Data Bits, BC = Broken Connection, *Correction in Hardware

Errors	**# MAC Computations**			**MAC Duration**	**Simulated Bit Permutation Loop Duration**		
	Best Case	Average Case	Worst Case	Average Case	Apple M1	Intel i7-1165G7	AMD 5800X
1*	17	17	17	11 ns	-	-	-
2	528	711	1985	2.43 µs	952 ns	3.68 µs	3.85 µs
3	17 440	33 800	63 520	115 µs	40.2 µs	124 µs	126 µs
4	576 608	1.51×10^6	3.94×10^6	5.17 ms	1.74 ms	6.65 ms	7.49 ms
5	1.91×10^7	6.91×10^7	1.26×10^8	236 ms	78.9 ms	261 ms	293 ms
6	6.32×10^8	3.07×10^9	5.87×10^9	10.5 s	3.51 s	12.8 s	14.0 s
7	2.10×10^{10}	1.21×10^{11}	1.88×10^{11}	6.87 min	2.30 min	9.11 min	10.0 min
8	6.97×10^{11}	5.72×10^{12}	5.82×10^{12}	5.44 h	1.70 h	6.11 h	6.74 h
BC*	1	1.125	32	5.77 ns	-	-	-

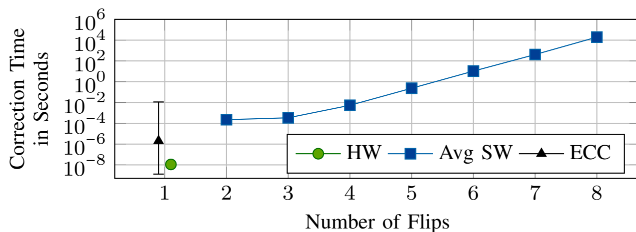


Figure 6: Correction time from Table 4 - MAC Duration. One bitflip is corrected in hardware, the exception overhead for the software correction is 223 µs. The ECC correction time is dependent on the CPU and varies greatly. MAC and parity bitflips are corrected with one MAC computation and therefore not shown.

TABLE 5: MAC strength with approximate equality for 256 bit. p is the number of bit permutations tried to correct the data. d is the number of flips allowed to be different in the MAC for the approximate equality. The MAC strength is a result of d. CSI:Rowhammer is secure because the strength is always higher than $\log_2(p)$.

Data Flips	$\log_2(p)$	d	MAC Strength
5	26.0	3	41.2
6	31.5	2	45.4
7	38.8	1	50.2
8	42.4	0	56.0

First, we implement our correction as a search algorithm with a `memcmp` instead of the MAC comparison; we call it *bit permutation loop*. Then we run a Monte Carlo simulation 10 000 times to get the average number of MAC computations per number of bitflips and the duration of the *bit permutation loop* on all three CPUs.

Table 4 shows the results. The MAC computation duration is calculated by multiplying the number of MAC computations with the duration of a single MAC computation divided by the throughput (see Table 1). Under the assumption that the MAC computation is performed on an AVX floating-point execution unit, the MAC computation can run in parallel to the *bit permutation loop* that uses different execution units, and their times do not add up.

If the simulated *bit permutation loop* without the MAC computation is faster than the combined MAC computation, we use the MAC computation duration as a estimate for the correction time, otherwise, *bit permutation loop* duration. The Apple M1 could, because of its large number of ALUs, keep more than one QARMA cipher block busy, as it is limited by the MAC computation duration. The other CPUs can almost keep the QARMA cipher block busy. For 512-bit

data, correcting 7 bitflips requires 1.41×10^{13} MAC computations and takes $35\,\mathrm{hours}$ and for 6 bitflips $28\,\mathrm{minutes}$.

We use our gem5 prototype to measure the overhead caused by the exception, which is $223\,\mu\mathrm{s}$. In Figure 6, we add this overhead to the durations from Table 4. The correction time of ECC memory is highly dependent on the CPU. Kwong et al. [43] measured access times 5 orders of magnitude larger on an error. Cojocar et al. [16] measured accesses 500 times as well as only 1.01 times larger. To compare these numbers with our durations, we multiplied them with a typical DRAM access time of $113\,\mathrm{ns}$ [18].

6.3. DRAM Organization & Energy Cost

Unlike other memory integrity protection methods like IVEC [28] or Synergy [58], CSI:Rowhammer requires only a single memory access to get the data and integrity information. Memory organization in the form of DRAM channels, therefore, has no impact on the performance. The QARMA authors did not analyze the energy cost, reasoning that the single memory access dominates the energy consumption and that a few thousand gates are nearly negligible in modern CPUs [6]. Similar to Qualcomm's pointer authentication, we also use the QARMA cipher in combination with a memory access and can apply the same argument. The added SRAM has a fraction of the cache capacity on a typical modern CPU and, therefore, negligible energy cost as well.

6.4. CPU and DRAM Area Estimation

In this section, we evaluate the area requirements in the CPU and DRAM to implement CSI:Rowhammer.

CPU Die Area. The components required are a MAC circuit in the memory controller, one per core, and a secure SRAM of 4 pages ($16\,\mathrm{kB}$). The SRAM is not a hard requirement, and it can be omitted with a slight decrease in system reliability without compromising security.

We estimate the SRAM size for TSMC's $7\,\mathrm{nm}$ process [71], because the area of QARMA is also provided for a $7\,\mathrm{nm}$ process [6]. One TSMC 7nm SRAM cell has $0.027\,\mu\mathrm{m}^2$ [71], resulting in an area for 4 pages of $0.027\,\mu\mathrm{m}^2 \cdot 8 \cdot 4096 \cdot 4 = 3539\,\mu\mathrm{m}^2$. With the area of one latency optimized QARMA$_5$-64-σ_0 [6] block of $1238.1\,\mu\mathrm{m}^2$, the overall area requirement for a 4-core CPU is $3539\,\mu\mathrm{m}^2 + 5 \cdot 1238\,\mu\mathrm{m}^2 = 9729\,\mu\mathrm{m}^2$. This is approximately $0.0067\,\%$ of the Tiger Lake CPU die with comparable feature size and an area of $146.1\,\mathrm{mm}^2$ [17], a lot less than the $0.5\,\%$ and $0.6\,\%$ of Blockhammer [73] and Graphene [55].

DRAM Area. On systems already equipped with ECC memory, CSI:Rowhammer does not add any DRAM area. For systems without ECC memory, CSI:Rowhammer adds $12.5\,\%$ (DDR4) or $25\,\%$ (DDR5) area.

6.5. Cost vs. Device Vulnerability

Previous works report that similar devices show high variance in Rowhammer susceptibility [26], [34], [40], [44]. Kim et al. [40] used DDR3 memory and observed a very low number of multi-bit errors ($1.9\,\%$), CSI:Rowhammer corrections would have a minor performance impact. More recently, Kim et al. [36] found that first-generation DDR4 DRAM is four times as vulnerable. The higher rate of multi-bit errors resulting from it can cause longer correction times, but the chance of uncorrectable errors is still low.

Recent generations of (LP)DDR4 memory show a significantly higher susceptibility to Rowhammer. Hassan et al. [26] found up to 7 flips in a single 64-bit data word. The worst cells Kim et al. [36] found, fail after less than 5000 hammers; this memory requires TRR, see Section 8.1.

6.6. MAC Security Evaluation

We now evaluate the security and correctness of the MAC.

Construction. The PAM construction with tweakable block ciphers is proven to be secure [57]. The usage of the physical address as the tweak for the MAC makes targeted hammering of the data and MAC infeasible (see Section 6.7).

Approximate Equality. We can precisely determine the security loss due to the approximate equality check. If we consider an n-bit MAC and allow approximate equality with up to d errors, then the success probability of a single forgery attempt increases from 2^{-n} to $2^{-n}\sum_{k=0}^{d}\binom{n}{k}$, corresponding to a security loss of $\log_2(\sum_{k=0}^{d}\binom{n}{k})$ bits. As an example, consider $n = 56$ and $d = 3$, where the security degrades from 56 to $56 - 14.8 = 41.2\,\mathrm{bits}$. d is reduced with a higher number of data bitflips, as shown in Table 5, and is never higher than $d = 3$. For 512-bit data, $d = 3$ for 4 flips in the data and reduced to $d = 0$ for 7 flips, the maximum.

MAC Leakage. Bitflips in the MAC are corrected with approximate equality, so there is no relation between the correction time and flipped MAC bits. A timing side channel can leak the location of a flipped bit in the data. A RAMBleed-like attack could, in theory, be used to leak the MAC and virtual machines can compute the MAC. However, the dependency of the MAC on the physical address makes an attacke, even then, infeasible, see Section 6.7.

6.7. Detection and Correction Security Evaluation

In this section, we evaluate the security of CSI:Rowhammer in regards to data corruption, protection against a Rowhammer attack, erroneous corrections, and uncorrectable errors.

Bitflip Location. The location of Rowhammer bitflips and memory errors within the data or integrity information has no impact on the error detection capabilities of CSI:Rowhammer. Therefore, CSI:Rowhammer can guarantee the detection of all Rowhammer attacks and memory errors with a very high chance and mitigate all attacks, as shown in the following paragraphs. If no advanced correction method is applicable, the correction as a search can correct up to 8 bitflips in 256-bit data if there are no bitflips in the integrity information and up to 5 bitflips in the data with 3 bitflips in the MAC and 1 flip in the parity bits (see Section 6.6).

Single Bit Data Corruption. We use the MAC and the parity bits to verify the integrity. Therefore, 1 bitflip can never lead to a silent data corruption, if it creates a second preimage with the same MAC, the parity bit check still fails.

Silent Data Corruption. Every system is exposed to bitflips. We evaluate the frequency of silent data corruptions

caused by naturally occurring bitflips that cause a second preimage. There must be at least two flips in 256 or 512 bits of data for the possibility of a silent corruption. With a secure MAC, the probability of producing a second preimage is independent of the number of bitflips. Previous studies only report multi-bit errors per 64-bit ECC data word. However, multi-bit errors are relatively rare, so there is a low chance of them piling up in a 256 or 512-bit data block. Therefore, we can take the frequency of multi-bit errors observed in other studies also for this evaluation. Additionally, we take the worst reported FIT by Schroeder et al. [59] of $70\,000$ and distribute them uniformly over the memory to get the probability of additional multi-bit errors in the data. On a double error, 3969 MAC computations are performed in the worst case. The overall Silent Data Corruption rate of CSI:Rowhammer is less than once per 10^9 billion years.

Random Rowhammer Attack. We evaluate the probability of an attacker finding a second preimage by randomly hammering data. As a single bitflip is always detected, an attacker must flip at least two bits in 256 or 512 bits. If flips did not produce a second preimage, the correction is run by the OS, which takes time exponentially corresponding to the number of flips. An attacker flipping two bits is the best case for finding a second preimage as quickly as possible. In a very optimistic scenario for the attacker, two bits always flip after hammering a page for two refresh intervals or $128\,\mathrm{ms}$. The attacker can either hammer a new location containing different data or change the victim data after every hammer. In this scenario, the probability of finding a second preimage after hammering for one year straight is $9.75 \cdot 10^{-5}\%$.

Targeted Rowhammer Attack from a Virtual Machine. We evaluate if it is possible for an attacker to create valid data and integrity information in a victim row with Rowhammer. For an attacker VM it would be possible to compute the MAC for victim data, free the row and hammer it after a victim VM or the host allocated it. The usage of the physical address as a tweak makes this attack infeasible:

In short, an attacker cannot construct an aggressor row that has both, matching data and checksum to the intended victim row data, which is a requirement stemming from the data dependency of Rowhammer (see Section 2.2).

In more detail, there are two options for an attacker:

1) Flip the victim data to a second preimage without changing the MAC: The attacker must find an exploitable second preimage of the victim row data and a second preimage of the aggressor row data that does change the victim data but not the victim MAC.
2) Flip both data and MAC: The attacker has to find aggressor data that changes the victim data to be exploitable and has a MAC that changes the MAC in the victim row to correspond to the new victim data.

Finding the required second preimages for the victim and/or aggressor rows is computationally infeasible. Additionally, most modern DRAM requires double-sided hammering to flip bits, doubling the number of data and MAC combinations for the aggressor rows an attacker must find [22], [34]. For half-double Rowhammer the near and far aggressors must contain the correct data for it to succeed, quadrupling the computational work [41].

Finally, for a second preimage to exist, on average 41.2 bits (see Table 5) must be flippable in the victim data. This makes the attack rather hypothetical, as it is way beyond the amount of bit flips we observe in DRAM today and would render the DRAM module likely unusable for most tasks. Additionally, these on average 41.2 changed bits must still be valid data that is useful for the exploit, so that the attacked application or kernel does not crash and behaves in the way anticipated by the attacker.

Highly Vulnerable DRAM. The number of bitflips does not change the detection guarantees of CSI:Rowhammer. A high number of flips can cause a denial of service, but cannot be exploited. The same is true if the memory chip that stores the integrity information is very vulnerable to Rowhammer. It increases the chance of uncorrectable errors and cases where many flips in the MAC look like uncorrectable data errors, but the error detection is always guaranteed.

Erroneous Correction. During the correction, the operating system computes many MACs with different data, each of which could be a second preimage. In the worst possible case, 8 bits flip in the last tested locations of the same last parity bit. This leads to the maximum number of flip permutations: the worst-case number in Table 4 times 2, *i.e.*, $1.164{\times}10^{13}$, and $\log_2(1.164{\times}10^{13}) = 43.4$ is lower than the width of our MAC $n = 56$, see Table 5. The probability p of finding a second preimage is $0.0161\,\%$. This correction takes approximately 6 hours. To get a chance of $50\,\%$, the attack takes $\frac{\log(0.5)}{\log(1-0.000161)} \cdot 6\,\mathrm{h} = 1076\,\mathrm{days}$. The outcome of this correction is impossible to predict for the attacker, and thus of only very limited use. With 512-bit data, only 6 flips are correctable in reasonable time; 7 bits take approximately 92 hours in the worst case. The number of permutations for 7 flips, in this case, is 5.035×10^{13}. The probability p of finding a second preimage is $0.0699\,\%$.

Uncorrectable DRAM Errors. In Section 2.1, we summarize the findings of three studies on DRAM errors in large-scale systems. From these numbers, it is possible to make predictions about the correction capabilities. Bautista-Gomez et al. [9] give insight into the number of flips in multi-bit errors. Less than $0.002\,\%$ of multi-bit errors have more than 8 bits flipped and are therefore not correctable as a search. According to Hwang et al. [29], $1.34\,\%$ of nodes in one system they observed had at least one Chipkill error in 583 days. These errors are not correctable as a search but, e.g., by reloading pages from disk and always detected.

7. Related Work

There are many proposals for Rowhammer mitigations and data integrity protection but with little overlap. All presented Rowhammer mitigations were already broken by follow-up work. Data integrity methods are reliable at detecting bitflips, but either have a significant performance impact or cannot correct bitflips caused by Rowhammer.

7.1. Rowhammer Mitigations

Rowhammer mitigations hinder specifically the flipping of bits or the exploitation of Rowhammer bitflips. Gruss et al. [23] described Rowhammer mitigations in three categories:
Elimination. TRR is a Rowhammer mitigation in recent DRAM modules. It counts the accesses to rows and refresh adjacent rows if the accesses exceed a pre-configured value. TRR can be bypassed in two ways: by multi-sided hammering [22], [34] and by half-double hammering [41]. It is also possible to detect Rowhammer attacks with frequent element analysis in the DRAM data streams [55], [39]. They have a chip area overhead that is dependent on the number of DRAM ranks and is higher than that of CSI:Rowhammer.
Neutralization. While not preventing bitflips, some mitigations prevent Rowhammer attacks by physically distancing rows. ZebRAM [42] isolates all rows by making only every second row accessible to programs. The rows in between are used as integrity-checked swap space. ZebRAM can be circumvented with half-double hammering [41]. Brasser et al. [14] physically distance kernel and userspace. CSI:Rowhammer neither introduces such physical distancing nor can it be bypassed by known or future hammering patterns.
Detection. Irazoqui et al. [32] presented MASCAT, a static code analysis tool to detect Rowhammer attack code. Others used performance counters to detect microarchitectural attacks (e.g., Rowhammer) [27], [25], [75]. However, they are limited to specific assumptions about Rowhammer and have detection rates far below those of CSI:Rowhammer.

7.2. Data Integrity Protection

Data integrity protection is a very long-studied topic. However, no solution specifically focuses on Rowhammer [28], [58], [15], [74], [54], [38], [37]. Instead, they focus primarily on sophisticated attacks requiring physical access to a device like cold-boot attacks or attacks on the memory bus, enabling replay, relocation, and data substitution and often include encryption. These protections are required, e.g., in secure enclaves like Intel SGX or ARM TrustZone, where security is the primary aspect. They come, however, with a significant performance impact on memory-intensive workloads [58]. CSI:Rowhammer manages the balance between security and performance better than any other DRAM integrity verification method by focusing on data modifications caused by single-event upsets and Rowhammer.
ECC and Chipkill. ECC and Chipkill can only detect a certain number of flips, making them unsuitable as a Rowhammer countermeasure. Chipkill significantly impacts DRAM performance or memory overhead depending on the used protection scheme [20]. All of the schemes described by Dell et al. [20] are additionally not able to correct multiple flips if they come from different chips, which is common for Rowhammer. CSI:Rowhammer can correct errors from different chips, but not from a faulty chip. All three can correct errors from a faulty contact.

The correction time of typical single-bit errors and faulty contact errors is similar between CSI:Rowhammer, ECC, and Chipkill. For systems that currently have ECC or Chipkill deployed, performance does not change with CSI:Rowhammer when these errors happen.
IVEC (Integrity Verification with Error Correction). Huang et al. [28] proposed IVEC, a memory integrity verification method that includes error correction as a search similar to CSI:Rowhammer. IVEC uses an integrity tree and a GMAC with split counters to protect against data modification and replay, relocation, or substitution attacks. The integrity tree requires multiple memory accesses for every integrity verification, which has a significant performance impact. IVEC does not use ECC DRAM to store the MAC.
Synergy. Synergy by Saileshwar et al. [58] is similar to IVEC but uses ECC DRAM to reduce memory accesses. Parity information is stored at another location for error correction but is only retrieved on corruption. Encrypting the data also protects against replay, relocation, or substitution attacks. The overall performance overhead is significantly higher than CSI:Rowhammer's. Synergy can correct 8 bits in 64-bit data if they come from the same memory chip but only detect them if spread over multiple chips.
MemGuard. With MemGuard, Chen et al. [15] verify the data integrity by computing a write-log hash and a read-log hash with all accesses. The operating system checks the integrity of the whole DRAM periodically by comparing the write-log hash with the read-log hash. If they do not match, the OS resets the memory state to a prior checkpoint causing a delay of seconds. Frequently induced bitflips would halt the system as it could not progress to the next checkpoint.

8. Discussion

In this section, we discuss additional performance improvements, compatibility with existing technologies, and possible race conditions that can occur during the data correction.

8.1. Compatibility with other Technologies

CSI:Rowhammer and TRR TRR cannot provide any security guarantees and was already broken in multiple ways [22], [41]. Most recently, Jattke et al. [34] were able to flip bits in all 40 of their recently-purchased DDR4 DIMMs with TRR. TRR can therefore not be considered even remotely a secure Rowhammer mitigation. When used in combination with CSI:Rowhammer, it provides all secure guarantees, while TRR can provide a performance gain and reduce the risk of DoS by reducing the number of bitflips.
Compatibility with Memory Encryption. If used together with memory encryption, the `csi_load` instruction returns the encrypted data. The correction as a search is then performed on the encrypted data and written back with the `csi_xchg` instruction. If the corrupted data could be reconstructed by, e.g., reading it from disk, the operating system writes the data back with a fourth instruction similarly to `csi_xchg` that encrypts the data in the memory controller.
Backward Compatibility and Upgrade Path. An additional benefit of CSI:Rowhammer is its backward compatibility. It could be controlled by an MSR, with a fall-back to ECC, facilitating hardware upgrades, as software can follow slower and is not required to support CSI:Rowhammer at all.

Compatibility with Virtualized Environments. CSI:Rowhammer can be used in virtualized environments as described in Sections 4 and 5. Virtual machines do not have to support CSI:Rowhammer. The MSR to activate it is simulated by the host that, therefore, knows which VM wants to use or supports CSI:Rowhammer. For VMs that did not activate CSI:Rowhammer or where security is of uttermost importance, the host corrects all flips as a search. For VMs that did activate it, the host forwards the exception to the VM, allowing the VM's OS to perform all possible operations in the advanced correction, like reloading data from disk. Not requiring every VM to support CSI:Rowhammer greatly eases the upgrade path of virtual environments, because only the host must be upgraded in the first step.

8.2. Possible Improvements

Memory Scrubbing. During phases with low memory bus load, the memory controller performs memory scrubbing by checking the MACs of data. Memory Scrubbing enables the detection and correction of data corruptions prior to the access, increasing the overall system performance. It also prevents the build-up of bitflips that exceed the maximum number of correctable flips, increasing the system reliability.
Speculative Data Forwarding. When accessing data from the DRAM, the memory controller first reads the bytes required by the CPU, and afterward, the missing bytes to fill the cache line. However, the integrity check requires the entire cache line to compute the MAC. The CPU could continue execution after receiving the first word transiently until the integrity is checked. If the integrity check passes, CSI:Rowhammer has no impact on performance. Transiently executing unverified code or data could impact the security of the system. Shi et al. [61] and Lehman et al. [45] already suggested solutions for safe speculative execution.

8.3. CSI:Rowhammer and DRAM Scaling

We believe that because of the continuously increasing susceptibility of DRAM to Rowhammer [53], [36], it is crucial to have a solution like CSI:Rowhammer that translates DRAM error rates to performance without sacrificing security. DRAM is optimized for the optimal ratio between density and performance. If a solution like CSI:Rowhammer was widely used today, increasing the density too much would degrade the performance substantially, because of very frequent bitflips, before the density becomes so high that the bitflips can be exploited. DRAM cannot be optimized anymore without taking security into consideration.

8.4. Race Conditions during Data Correction

CSI:Rowhammer allows other tasks and multiple corrections to run simultaneously. We discuss the possible scenarios that could arise and how we prevent any race condition.
Read While Correction. If one task is correcting a corruption and another is accessing the same memory location, a lock in the exception handler prevents the concurrent correction of the same flip. In the rare case that both handlers resort to the nested exception handler, concurrent correction cannot be prevented. If the first handler finishes the correction and modifies the data, the `csi_xchg` instruction prevents the second handler from overwriting the new data with old data.
Write While Correction. If a task uses streaming or non-temporal stores, an uncached memory location can be written without loading into the cache. This can overwrite corrupted data in the DRAM that is currently corrected. When the correction is finished, `csi_xchg` detects that the data is already modified and does not overwrite it.
Corruption While Correction. Data may become more corrupted while a correction of this data is running. In that case, the correction finishes but does not write the corrected data back because the `csi_xchg` detects a change. After returning from the exception, another corruption exception is raised. This is a disadvantage but outweighed by the advantages the `csi_xchg` has in the other cases. A corruption can also happen in the advanced correction code while it is de-scheduled. This causes a corruption exception when it is scheduled again. The error is corrected as a search from the secure memory, and the initial correction is continued.
NMI During Corruption Exception. The exception mechanism follows a very similar semantic to the page fault mechanism and can be paused and unscheduled without a problem. The NMI can cause another corruption exception due to flips in the NMI handler. The nesting of corruption handlers is detected with the nesting bit, and a correction as a search is performed. Frequently recurring NMIs, where the corrupted handler causes a corruption exception, could hang the system, similar to combinations of PFs and NMIs.

9. Conclusion

CSI:Rowhammer is a principled hardware-software co-design against Rowhammer. Our low-latency lightweight-cryptographic MAC brings cryptography-grade integrity protection, detecting any number of bitflips up to the strength of the cryptographic MAC. Our software-level correction routine takes less than 1 second to correct 5 bitflips and adds great flexibility to the correction. We implemented CSI:Rowhammer as a proof-of-concept in gem5 and Linux. We observe a $0.74\,\%$ latency overhead and no memory overhead over off-the-shelf ECC-DRAM, showing the practicality of our approach under normal conditions. Even under attack, CSI:Rowhammer maintains a low latency, with the ability to correct a single bitflip in less than $20\,\mathrm{ns}$, compared to up to $63\,\mu\mathrm{s}$ for regular ECC-DRAM as well as errors from faulty DIMM contacts. This shows that CSI:Rowhammer should be deployed in future processors.

10. Acknowledgements

We thank the anonymous reviewers, especially our shepherd, for their guidance, comments and suggestions. We also thank Yoongu Kim, Martin Schwarzl, Stefan Gast and Sarah Rinderer for their valuable input. Part of the funding was provided by a generous gift from Google. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] AMD. AMD Random Number Generator, 2017.

[2] ARM. ARM Cortex-A9 Technical Reference Manual, 2012.

[3] ARM. Arm True Random Number Generator (TRNG) Technical Reference Manual, 2020.

[4] ARM Connected blog. Armv8-A architecture: 2016 additions, 2016. URL: https://www.community.arm.com/processors/b/blog/posts/armv8-a-architecture-2016-additions.

[5] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A fast short-input PRF. In *INDOCRYPT*, 2012.

[6] Roberto Avanzi. The QARMA block cipher family: Almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency S-boxes. *IACR Transactions on Symmetric Cryptology*, 2017(1):4–44, 2017.

[7] Roberto Avanzi, Subhadeep Banik, Andrey Bogdanov, Orr Dunkelman, Senyang Huang, and Francesco Regazzoni. Qameleon v.1.0 - A Submission to the NIST Lightweight Cryptography Standardization Process, 2019. URL: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/submissions/qameleon.zip.

[8] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-based protection against next-generation Rowhammer attacks. *ACM SIGPLAN Notices*, 2016.

[9] Leonardo Bautista-Gomez, Ferad Zyulkyarov, Osman Unsal, and Simon McIntosh-Smith. Unprotected Computing: A Large-Scale Study of DRAM Raw Error Rate on a Supercomputer. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.

[10] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In *CHES*, 2016.

[11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 2011.

[13] John Black and Phillip Rogaway. A block-cipher mode of operation for parallelizable message authentication. In *EUROCRYPT*, 2002.

[14] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't touch this: Software-only mitigation against Rowhammer attacks targeting kernel memory. In *USENIX Security Symposium*, 2017.

[15] Long Chen and Zhao Zhang. MemGuard: A low cost and energy efficient design to support and enhance memory system reliability. In *ISCA*, 2014.

[16] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *S&P*, 2019.

[17] Ian Cutress. I Ran Off with Intel's Tiger Lake Wafer. Who Wants a Die Shot?, January 2020. URL: https://www.anandtech.com/show/15380/i-ran-off-with-intels-tiger-lake-wafer-who-wants-a-die-shot.

[18] Johan De Gelas. AMD Rome Second Generation EPYC Review: 2x 64-core Benchmarked, 2019. URL: https://www.anandtech.com/show/14694/amd-rome-epyc-2nd-gen/.

[19] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript. In *USENIX Security Symposium*, 2021.

[20] Timothy J. Dell. A white paper on the benefits of chipkill-correct ecc for pc server main memory. Technical report, IBM Microelectronics, 1997.

[21] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P*, 2018.

[22] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P*, 2020.

[23] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P*, 2018.

[24] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*, 2016.

[25] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[26] Hasan Hassan, Yahya Can Tugrul, Jeremie S. Kim, Victor van der Veen, Kaveh Razavi, and Onur Mutlu. Uncovering In-DRAM RowHammer Protection Mechanisms:A New Methodology, Custom RowHammer Patterns, and Implications. In *MICRO*, 2021.

[27] Nishad Herath and Anders Fogh. These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In *Black Hat Briefings*, 2015.

[28] Ruirui Huang and G. Edward Suh. IVEC: Off-Chip Memory Integrity Protection for Both Security and Reliability. In *ISCA*, 2010.

[29] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *ASPLOS*, 2012.

[30] Intel. Intel Digital Random Number Generator (DRNG) - Software Implementation Guide, 2012.

[31] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2019.

[32] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Mascat: Preventing microarchitectural attacks before distribution. In *CODASPY*, 2018.

[33] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX*, 2017.

[34] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. BLACKSMITH: Rowhammering in the Frequency Domain. In *S&P*, November 2021.

[35] Jedec Solid State Technology Association. Low Power Double Data Rate 3, 2013. URL: http://www.jedec.org/standards-documents/docs/jesd209-4a.

[36] Jeremie S. Kim, Minesh Patel, A. Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques. In *ISCA*, 2020.

[37] Jungrae Kim, Michael Sullivan, and Mattan Erez. Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory. In *HPCA*, 2015.

[38] Jungrae Kim, Michael Sullivan, Seong-Lyong Gong, and Mattan Erez. Frugal ECC: efficient and versatile memory error protection through fine-grained compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[39] Kwangrae Kim, Jeonghyun Woo, Junsu Kim, and Ki-Seok Chung. HammerFilter: Robust Protection and Low Hardware Overhead for RowHammer. In *International Conference on Computer Design (ICCD)*, 2021.

[40] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*, 2014.

[41] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-Double: Hammering from the next row over. In *USENIX Security Symposium*, 2022.

[42] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In *USENIX OSDI*, 2018.

[43] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *S&P*, 2020.

[44] Mark Lanteigne. How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware, 2016. URL: http://www.thirdio.com/rowhammer.pdf.

[45] Tamara Silbergleit Lehman, Andrew D. Hilton, and Benjamin C. Lee. PoisonIvy: Safe speculation for secure memory. In *MICRO*, 2016.

[46] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *IEEE Computer Architecture Letters*, 2020.

[47] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing Rowhammer Faults through Network Requests. *arXiv:1711.08002*, 2017.

[48] Moses D. Liskov, Ronald L. Rivest, and David A. Wagner. Tweakable block ciphers. In *CRYPTO*, 2002.

[49] Eik List and Mridul Nandi. Revisiting full-PRF-secure PMAC and using it for beyond-birthday authenticated encryption. In *CT-RSA*, 2017.

[50] Onur Mutlu and Jeremie S. Kim. RowHammer: A Retrospective. *IEEE TCAD*, 2020.

[51] National Institute of Standards and Technology. FIPS PUB 198-1: The keyed-hash message authentication code (HMAC), 2008. URL: http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf.

[52] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the lightweight cryptography standardization process, 2018. URL: https://csrc.nist.gov/projects/lightweight-cryptography.

[53] Lois Orosa, Abdullah Giray Yaglikci, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S. Kim, and Onur Mutlu. A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses. In *Proceedings of the Annual International Symposium on Microarchitecture*, 2021.

[54] David J. Palframan, Nam Sung Kim, and Mikko H. Lipasti. COP: To compress and protect main memory. In *ISCA*, 2015.

[55] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. Graphene: Strong yet Lightweight Row Hammer Protection. In *MICRO*, 2020.

[56] PARSEC Group. A Memo on Exploration of SPLASH-2 Input Sets, June 2011. URL: http://parsec.cs.princeton.edu.

[57] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In *ASIACRYPT*, 2004.

[58] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, and Moinuddin K Qureshi. Synergy: Rethinking secure-memory design for error-correcting memories. In *HPCA*, 2018.

[59] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: A large-scale field study. *Commun. ACM*, 2011.

[60] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat Briefings*, 2015.

[61] Weidong Shi and Hsien-Hsin S. Lee. Authentication Control Point and Its Implications For Secure Processor Design. In *MICRO*, 2006.

[62] Taniya Siddiqua, Athanasios E. Papathanasiou, Arijit Biswas, Sudhanva Gurumurthi, Intel Corp, and Teradata Aster. Analysis and modeling of memory errors from large-scale field data collection. In *SELSE*, 2013.

[63] Vilas Sridharan and Dean Liberty. A study of DRAM failures in the field. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[64] Lukas Steiner, Matthias Jung, Felipe S. Prado, Kirill Bykov, and Norbert Wehn. DRAMSys4.0: A Fast and Cycle-Accurate SystemC/TLM-Based DRAM Simulator. In *Springer LNCS International Conference on Embedded Computer Systems Architectures Modeling and Simulation (SAMOS)*, 2020.

[65] Andrei Tatar. Hammertime: a software suite for testing, profiling and simulating the Rowhammer DRAM defect, 2018. URL: https://github.com/vusec/hammertime.

[66] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against rowhammer: a surgical precision hammer. In *RAID*, 2018.

[67] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX ATC*, 2018.

[68] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS*, 2016.

[69] Andrew J Walker, Sungkwon Lee, and Dafna Beery. On dram rowhammer and the physics of insecurity. *IEEE Transactions on Electron Devices*, 2021.

[70] Zhengrong Wang. Bring AVX Support to Gem5, 2020. URL: https://seanzw.github.io/posts/gem5-avx/.

[71] Shien-Yang Wu, C.Y. Lin, M.C. Chiang, J.J. Liaw, J.Y. Cheng, S.H. Yang, C.H. Tsai, P.N. Chen, T. Miyashita, C.H. Chang, V.S. Chang, K.H. Pan, J.H. Chen, Y.S. Mor, K.T. Lai, C.S. Liang, H.F. Chen, S.Y. Chang, C.J. Lin, C.H. Hsieh, R.F. Tsui, C.H. Yao, C.C. Chen, R. Chen, C.H. Lee, H.J. Lin, C.W. Chang, K.W. Chen, M.H. Tsai, K.S. Chen, Y. Ku, and S. M. Jang. A 7nm cmos platform technology featuring 4th generation finfet transistors with a 0.027um^2 high density 6-t sram cell for mobile soc applications. In *IEEE International Electron Devices Meeting*, 2016.

[72] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *USENIX Security Symposium*, 2016.

[73] A. Giray Yaglikci, Minesh Patel, Jeremie S. Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In *HPCA*, 2021.

[74] Doe Hyun Yoon and Mattan Erez. Virtualized ECC: Flexible Reliability in Main Memory. *IEEE Micro*, 2011.

[75] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In *RAID*, 2016.

Appendix A.
Correction as a Search Pseudocode

Pseudocode of parts of the correction as a search algorithm required to correct the flips in Figure 3a. The comments are based on this correction attempt.

```
bool correction_as_a_search(data, paddr, MAC, parity)
{
  // maybe the error was transient
  computed_mac = csi_mac(data, paddr);
  if (popcount(MAC ^ computed_mac) < 4) {
    return 1;
  }

  // we have two mismatching parity bits
  mmpb =
    compute_mismatching_parity_bits(data, parity);
  mmpb_count = count(mmpb);
  min_flips = mpb_count;

  if (min_flips == 0) {
    min_flips += 2; // one double flip
  }

  if (min_flips == 1) {
    ...
  }

  if (min_flips == 2) {
    if (mpb_count == 0) {
      // one double flip
      ...
    }

    if (mpb_count == 2) {
      // our first guess are two single flips
      if (perm_two_single_flip(data, paddr, mpb))
        return 1;
    }

    // We do not find the correction
    // Add a double flip to our guess
    min_flips += 2;
  }

  if (min_flips == 3) {
    ...
  }

  if (min_flips == 4) {
    if (mpb_count == 0) {
      // two double flips
      ...
    }

    if (mpb_count == 2) {
      // two single flips, one double flip

      // try all possible double flip permutations
      // for all four parity bit blocks
      for (p = 0; p < 4; p++) {
        for (i = p * 4; i < (p + 1) * 4; ++i) {
          flip_bit(data, i);

          for (j = p * 4; j < i; ++j) {
            flip_bit(data, j);

            // include all single flip permutations
            // from the first guess
            if (
              perm_two_single_flip(data, paddr, mpb))
              return 1;

            flip_bit(data, j);
          }
          flip_bit(data, i);
        }
      }
    }

    if (mpb_count == 4) {
      // four single flips
      ...
    }
  }

  return 0;
}

bool perm_two_single_flip(data, paddr, mpb) {
  for (i = mpb[0] * 4; i < (mpb[0] + 1) * 4; ++i)
  {
    // flip bit at index i in data
    flip_bit(data, i);

    for (j = mpb[1] * 4; j < (mpb[1] + 1) * 4; ++j)
    {
      flip_bit(data, j);

      // Check if we found the correction
      computed_mac = csi_mac(data, paddr);
      if (popcount(MAC ^ computed_mac) < 4) {
        return 1;
      }

      flip_bit(data, j);
    }
    flip_bit(data, i);
  }

  return 0;
}
```

Listing 3: Correction as a search pseudocode.

Appendix B.
`csi_xchg` Pseudocode

The `csi_xchg` instruction is used to write the corrected data back into the memory without overwriting changed or already corrected data (see Section 4.7).

```
int csi_xchg(rx phys_addr, ZMM old_data, rx old_mac,
             ZMM new_data) {

  ZMM curr_data, rx curr_mac = csi_load(phys_addr);

  if (curr_data == old_data && curr_mac == old_mac) {
    *phys_addr = new_data;
    return 1;
  }
  return 0;
}
```

Listing 4: Pseudecode of the `csi_xchg` instruction.

Appendix C. Inserting Raw Bytes into Code

Listing 5 shows how to invoke our instruction set extension without the need for a compiler change by directly placing the byte sequence in the binary.

```
1  asm volatile(
2    "vmovdqu64 %1,%%zmm0                    \n\t"
3    "mov %2,%%rax                           \n\t"
4    ".byte 0x62, 0xf1, 0xfe, 0x48, 0x6e, 0xc1 \n\t"
5    : "=m"(mac)
6    : "m"(*data), "m"(address)
7    : );
```

Listing 5: Calling the custom `csi_mac` instruction. It takes the data and the physical address as an argument and returns the computed MAC.

Appendix D. Benchmark DRAM Accesses

Figure 7 shows the number of DRAM accesses for the gem5 benchmarks of Figure 5. We observe a direct correlation between the performance overhead and the number of DRAM accesses due to the added latency of the MAC.

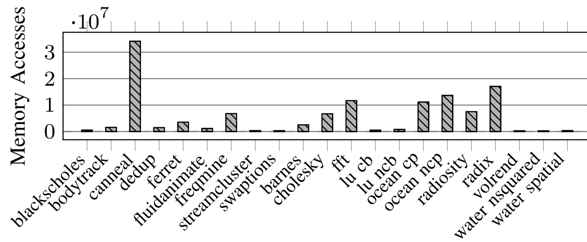


Figure 7: The number of memory accesses that miss the caches. The performance impact is dependent on this number of memory accesses, but also on how well out-of-order execution prevents stalling while waiting for a value or instruction from memory for the different instruction streams of the benchmarks.

Appendix E. Correction Procedure Flowchart

Figure 8 shows the correction procedure from Section 5.4 as a flow chart.

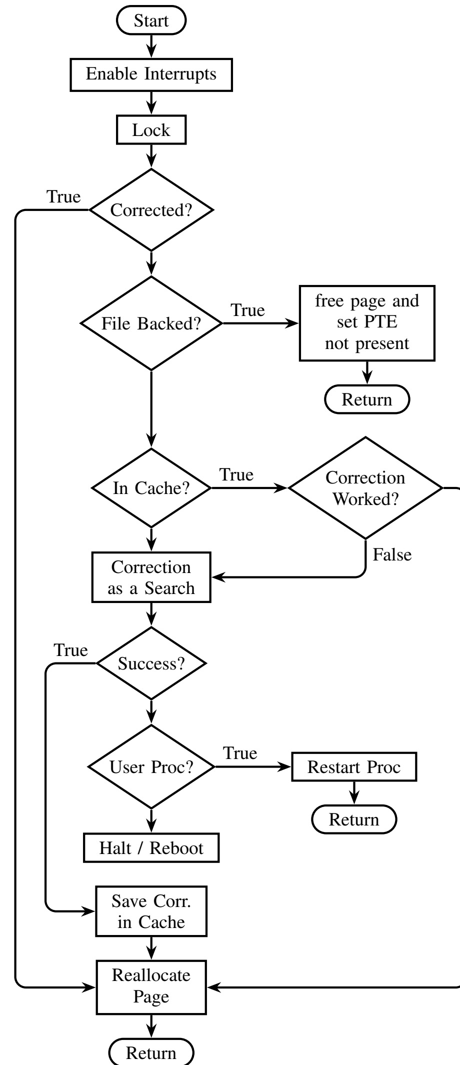


Figure 8: A flow chart depicting the advanced correction procedure that is run if no nesting of the corruption exception handler is detected. It is not in the secure memory because it accesses several other kernel resources.