ECC.fail: Mounting Rowhammer Attacks on DDR4 Servers with ECC Memory

Nureddin Kamadan and Walter Wang, *Georgia Tech;* Stephan van Schaik, *University of Michigan;* Christina Garman, *Purdue University;* Daniel Genkin, *Georgia Tech;* Yuval Yarom, *Ruhr University Bochum*

https://www.usenix.org/conference/usenixsecurity25/presentation/kamadan

This paper is included in the Proceedings of the
34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

ECC.fail: Mounting Rowhammer Attacks on DDR4 Servers with ECC Memory

Nureddin Kamadan*
Georgia Tech
kamadan@gatech.edu

Walter Wang*
Georgia Tech
walwan@gatech.edu

Stephan van Schaik
University of Michigan
stephvs@umich.edu

Christina Garman
Purdue University
clg@cs.purdue.edu

Daniel Genkin
Georgia Tech
genkin@gatech.edu

Yuval Yarom
Ruhr University Bochum
yuval.yarom@rub.de

Abstract

Rowhammer is a hardware vulnerability present in nearly all computer memory, allowing attackers to modify bits in memory without directly accessing them. While Rowhammer has been extensively studied on client and even mobile platforms, no successful Rowhammer attack has been demonstrated on server platforms using DDR4 ECC memory.

Tackling this challenge, in this paper we demonstrate the first end-to-end Rowhammer technique effective against Intel servers using Hynix DDR4 ECC memory. To that aim, we first characterize the Hynix implementation of Target Row Refresh (TRR) on server parts, demonstrating effective hammering patterns on both FPGA and Intel-based testing platforms with ECC disabled. We then reverse engineer Intel's ECC implementation on Skylake and Cascade Lake servers. We find that it has a coding distance of four, which often allows triggering incorrect ECC correction with just two bit flips.

Combining the two observations, we present an end-to-end Rowhammer attack which can flip bits on Intel servers, without causing crashes. Finally, we demonstrate the effectiveness of our attack by hammering RSA public keys loaded into memory, causing the server to accept messages not signed by the original key.

1 Introduction

Rowhammer is a security vulnerability present in nearly all modern computer memory [35], allowing attackers to modify the code and data of a victim program without directly accessing it. At a high level, by executing a specific sequence of memory accesses to their own address space, an attacker can drain the charge stored in memory cells located in geometrically adjacent rows, thereby causing bit flips across isolation boundaries. Known as Rowhammering, this cross-row disturbance effect has been used effectively by adversaries to violate numerous security properties [4, 14, 15, 16, 17, 20, 32, 39, 43, 52, 54, 62, 64], such obtaining root privileges [15], breaking cryptographic implementations [16, 39, 54], and even mounting browser-based exploits [4, 14, 17, 20, 32].

Starting from a humble origin of being a reliability issue in DDR3 laptop memory [35], Rowhammer attacks have been demonstrated against a large variety of computing hardware, including standard DDR3 DIMMs [17, 19, 39] on PCs as well as mobile LPDDR3 memory [66]. The advent of Targeted Row Refresh (TRR) countermeasures in DDR4 memory has led to the development of fuzzer-assisted TRR bypasses [18, 27, 28], with Rowhammer-induced bit flips demonstrated on DDR4 DIMMs as well as on mobile devices equipped with LPDDR4 memory [36]. Finally, Rowhammer has been recently demonstrated on some Samsung DDR5 memory modules, albeit without a systematic characterization [28].

While Rowhammer attacks have been extensively studied on PC and mobile platforms, much less is known about the vulnerability of server-grade memory to Rowhammer-induced bit flips. Indeed, server DIMMs contain an additional layer of bit flip protection, in the form of dedicated chips storing check bits for Error Correcting Codes (ECC), which aim to detect and correct memory errors. With Rowhammer attacks on ECC-equipped server memory only described on outdated DDR3-based platforms without TRR-based mitigations [11, 39], in this paper we ask the following questions:

Are DDR4-based server platforms vulnerable to Rowhammer attacks? What would it take to craft such attacks and how can we best defend against them?

1.1 Our Contributions

In this paper we demonstrate to the best of our knowledge the first end-to-end Rowhammer induced bit flips on DDR4 servers in default BIOS settings, with all ECC mechanisms enabled. We then use our Rowhammer technique to mount fault attacks on RSA public keys used for signature verification, resulting in the server accepting RSA signatures that are not signed by the original key.

Characterizing Rowhammer on DDR4 ECC DIMMs. We begin our investigation by characterizing the vulnerability of

*Both authors contributed equally to this work.

DDR4 Registered DIMMs with ECC memory to Rowhammer. We perform Rowhammer analysis across 30 server-grade Hynix ECC RDIMMs made between 2017 and 2022, using both an FPGA-based memory controller implementation and an Intel server equipped with a custom UEFI BIOS, which disables ECC and data scrambling. While the precise location of flippy bits varies across DIMMs and cannot be controlled by the attacker (thus requiring individual DIMM-level characterization), we find that we can induce bit flips in all 30 DIMMs using FPGAs, with 24 of them being vulnerable to hammering on server platforms.

Reverse Engineering Intel's ECC and Scrambling Implementation. Next, we proceed to reverse engineer the error correcting and data scrambling mechanisms of Intel platforms. To that aim, we use a Logic Analyzer (LA) to snoop traffic on the DDR4 bus. Here, we document for the first time Intel's ECC implementation, supporting Single-Device Data Correction (SDDC) capabilities. We find that Intel elected to use a distance-4 ECC code on Skylake and Cascade Lake platforms. In such codes, a four bit flip is enough for an undetectable ECC bypass, whereas two bit flips are sufficient to trigger incorrect ECC correction. Both these cases result in observable Rowhammer-induced bit flips on the target machine. Finally, we find that Intel's data scrambling implementation appears to be deterministic, XORing the same scrambling string into the data across boots and even machines.

End-to-End Rowhammer Attacks on Server Platforms. Finally, we combine our hammering technique with our knowledge of Intel's ECC and scrambling implementation, presenting the first end-to-end Rowhammer attack on Intel servers in default BIOS configurations with their ECC and scrambling mechanisms enabled. As Intel machines crash when an uncorrectable bit flip is detected, we develop a novel approach for profiling memory for bits flippable via Rowhammer without inducing crashes. Combining our templating technique with improved hammering patterns designed to target Hynix's TRR implementation, we achieve architecturally visible bit flips without inducing crashes within about 2.5 hours. Finally, we use our ability to induce bit flips to hammer RSA signature public keys stored in the server's memory, resulting in the server accepting signatures not signed by the original key.

Summary of Contribution. We contribute the following:

- We characterize Rowhammer across 30 Hynix ECC server DIMMs using both FPGA and server platforms (Section 4).
- We reverse engineer Intel's ECC and data scrambling mechanisms in Skylake and Cascade Lake platforms (Section 5).
- We present end-to-end Rowhammer flips on Intel servers using default settings, without inducing crashes (Section 6).

1.2 Responsible Disclosure

We shared our findings with the Intel Security Team in January 2025 and subsequently discussed our findings over email and video calls. We have also notified ASRock regarding the ability to disable ECC via custom BIOS firmware in April 2025. Both vendors considered the issue to be out of scope.

2 Background

2.1 Memory Organization and Addressing

DRAM Organization. Modern DRAM systems consist of several hierarchical structures: channels, DIMMs, ranks, chips and banks, aiming to maximize the system's memory level parallelism. A DRAM channel is the set of traces between the memory controller and the DIMM physically connected to one of the channel's slots in the motherboard. Different channels can be accessed in parallel, allowing for simultaneous access to DIMMs placed in different channel slots. Within a DIMM, ranks are used to allow parallel access to different chips in the same module. There can be 1 or 2 ranks, each consisting multiple chips. See Figure 1 (left).

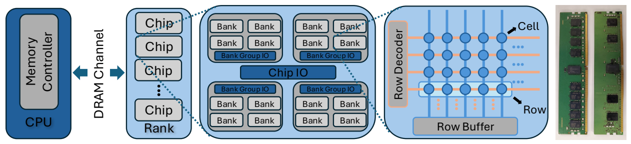


Figure 1: (left) Typical Memory System Organization. (right) Server RDIMMs with x4 Chips and x8 Chips.

Registered DIMM and Server Platform. DDR4 Registered DIMMs (RDIMM) are commonly used in server platforms, where the data bus between the memory controller and DIMM is expanded from 64-bits to 72-bits for transmitting checkbits. To align with the memory controller's data bus interface, each rank includes a sufficient number of chips for a 72-bit wide data bus, e.g., eighteen x4 chips, or nine x8 chips. Figure 1 (right) shows two RDIMMs with x4 and x8 chips.

Chip Layout. In the case of DDR4, each chip is organized into 4 bank groups, which are in turn composed of rows, typically 8KB in size. These rows are made up of individual cells, which are the smallest units in DRAM. Each DRAM cell consists of a single transistor and a capacitor, which together store a single bit of data. When a row is to be accessed in a bank, the memory controller issues an activate command to activate the wordline of the row which allows sense amplifiers to be connected with the cells in the row. This process loads the accessed row to the row buffer which is a per bank and cache like structure in DRAM to enable fast access to an activated row.

DRAM Addressing. To map the physical address space to DRAM locations, the CPU uses proprietary addressing functions. Next, as Rowhammer is a spatial attack between neighboring rows, before mounting Rowhammer, an attacker must recover the platform's DRAM addressing functions. Previous works [28, 52] have tackled the problem on both Intel and AMD's consumer platforms. Specifically, there are three parts that need recovering: bank addressing functions, row bits, and column bits. For bank addressing functions, previous works either use physical probing or bank conflict timing side-channels. On Intel's consumer platforms, bank functions are linear functions of XORing physical address bits while row and column bits are directly mapped from top bits and bottom bits, respectively, of physical addresses.

For Skylake and Cascade Lake servers we have found that Intel provided the addressing functions in the Linux's Error Detection and Correction (EDAC) source code [42], allowing the kernel to translate physical addresses to DRAM locations for accurate error logging and reporting. Figure 2 gives an example of addressing functions on Skylake and Cascade Lake CPUs with single rank DIMMs. Compared to recovered addressing functions on Intel's consumer platforms [28, 52], the bits used in DRAM row addressing are not in order, but mixed. Specifically, the first four row bits (R3...R0) are taken from physical address bits 20, 16, 15, 14 in order, while the next row bit R4 is not taken from physical address bit 21, but from bit 28. DRAM bank and column functions of both consumer and server platforms are alike.

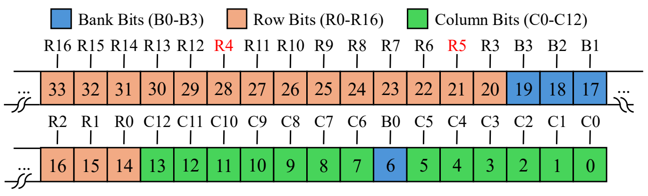


Figure 2: DRAM Addressing Functions on Intel Skylake and Cascade Lake CPUs with Single Rank DIMMs

2.2 Rowhammer

As the technology node size shrinks, cells in DRAM have become more prone to parasitic electrical interaction and passing gate effects from cells in nearby rows. Kim et al. [35] termed this electrical phenomena Rowhammer and showed that by performing a rapid series of activations to two aggressor rows, the bits stored in a victim DRAM row sandwiched between them can be flipped without accessing it. Weaponizing this effect, Rowhammer has been used for numerous attacks on computing systems, including client machines [15, 16, 32, 39, 43, 52, 54, 62, 64], web browsers [4, 14, 17, 20, 32], mobile [36, 66] and even DDR3-based servers [11].

TRR Mitigation. To mitigate Rowhammer on DDR4 memory, DRAM manufacturers [34, 41, 47] and CPU vendors [61] introduced Target Row Refresh (TRR) (either in DRAM or in the memory controller), which makes use of DRAM refresh commands to preventatively refresh potential victim rows before flips occur. Specifically, the DDR4 standard requires the memory controller to issue refresh commands every 7.8μs to let the DRAM refresh a certain number of rows and allow TRR to recharge potential victim rows. At a high level, a TRR mechanism can be abstracted into two phases: tracking phase and mitigating phase. In the tracking phase, TRR logic keeps track of potential Rowhammer aggressor rows by monitoring row activation commands. When the DRAM receives a refresh command, it switches to mitigating phase, allowing the TRR mechanism to decide whether to mitigatively refresh potential victim rows. However, these mitigations neither solve the root cause of the problem nor prevent exploitation by carefully crafted Rowhammer attack patterns [18, 27] or by a recently discovered electrical phenomenon known as Rowpress [45].

TRR Analysis with SoftMC. With the presence of TRR mitigation, Rowhammer attacks on DDR4 need to create access patterns that can successfully bypass its protection. While researchers have shown it is possible to observe and bypass the TRR mechanism on real systems [18, 27, 28, 32], recent work [23] introduced methods of directly reverse engineering in-DRAM TRR implementations in all major DRAM vendors. This is built on an FPGA-based software memory controller (SoftMC) [49], which provides the capability of precise control of commands sent to the memory module. Recovering vendor proprietary TRR implementations enables attackers to craft custom Rowhammer patterns that either bypass previously secure TRR implementations or produce more bit flips on TRR implementations with known effective patterns.

Rowhammer Attacks on ECC RAM. ECCPloit [11] first successfully attacked ECC implemented on DDR3 servers by reverse engineering ECC through various fault injection methods and cold boot attacks. This was then followed up by RAMBleed [39], which demonstrated how Rowhammer can be used for reading data from victim memory, including bits corrected by DDR3 ECC. While end-to-end attacks on servers using DDR4 ECC memory have not yet been demonstrated, some prior work was able to achieve bit flips on DDR4 RDIMMs by disabling refreshes using custom DIMM fault injectors [10] or on Broadwell servers [60], both with ECC disabled.

2.3 ECC and Data Scrambling in DRAM

ECC Implementations. Error correcting codes (ECC) are used to address memory errors caused by many factors ranging from cosmic rays to disturbance errors. Implementation of the ECC differs as the system requirements change. For low-end systems, ECC is implemented in-band, referring to the fact that the generated ECC bits are stored within the same memory as the data. This incurs a performance penalty as operations to read or write ECC bits to the same memory space do not happen in parallel. On higher-end systems (e.g., servers and workstations), ECC is implemented in a side-band manner, meaning that ECC bits are written to and read from a dedicated chip on the DIMM intended solely for ECC use. ECC computation and checks are done in parallel with normal data reads, with the help of additional data lines on the memory bus reserved for ECC data transfer. Thus, when the CPU wants to write data to the DRAM, its memory controller calculates the parity bits according to the underlying ECC algorithm and writes the resulting codeword to the DRAM.

A final method is on-die, which is implemented by memory vendors due to DRAM scaling challenges and is completely encapsulated within the DRAM chip [51]. This type of ECC is common on LPDDR4 and DDR5 memory, enhancing production yield by silently correcting errors.

Data Scrambling. Rapid changes in current can cause voltage fluctuations due to inductance present on bus circuitry. The memory bus, being high speed and handling non-uniform traffic often containing long sequences of 0s and 1s, is particularly susceptible to this, resulting in excessive voltage and current fluctuations. To mitigate this, memory controllers often implement data scrambling features, which disrupt long consecutive sequences of 0s and 1s by XORing pseudorandom patterns to data writes (and XORing them again during reads) [12].

3 Threat Model

In this paper we focus on Intel server platforms and Hynix DDR4 ECC RDIMM modules. We assume a typical Rowhammer model where the attacker has unprivileged code execution on the target machine. Next, for the attacks described in Section 6, we assume that the machine runs Linux, with all side-channel countermeasures both in BIOS and in the OS left in their default state. In particular, this includes the machine's ECC settings, which will crash the machine in the case where an uncorrectable bit flip is detected.

4 Observing Rowhammer on ECC DIMMs

In this section we characterize the vulnerability of DDR4 ECC DIMMs to Rowhammer bit flips, using both an FPGA-based memory controller and a Cascade Lake server.

For FPGA-based Rowhammer testing, we use a Xilinx Alveo U200 FPGA card, running the SoftMC memory controller implementation [49]. Next, for mounting Rowhammer on server platforms, we use a machine equipped with an ASRock EPC621D8A motherboard and an Intel Xeon W-3235 (Cascade Lake) CPU (unless stated otherwise). For memory, we use Hynix Registered DDR4 ECC memory, with exact part numbers specified in Table 1. Finally, our motherboard uses BMC controller firmware 01.60.00, and BIOS version 2.10, which we modify as outlined in Section 4.2.

4.1 FPGA-Based TRR Analysis

To reverse engineer TRR implementations across different memory modules, we follow the methodology of U-TRR [23].

Selecting Rowhammer Rows. We recall that the DDR4 TRR mechanism prevents Rowhammer-induced bit flips by deliberately recharging potential victim rows during DRAM refresh commands. Thus, by observing whether TRR is triggered in each refresh command and which rows are targeted for refreshing, we are able to recover the DIMM's TRR implementation.

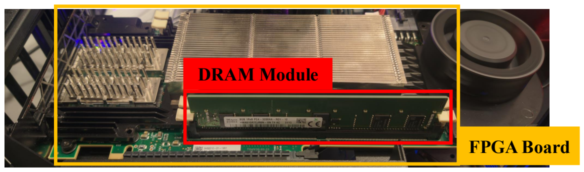


Figure 3: FPGA-based SoftMC Setup

At a high level, we use the following procedure for TRR reverse engineering. First, we modify the FPGA-based memory controller to not issue any DIMM refresh commands. We then test each row for its data retention time, i.e., the minimal time needed for it to have data errors due to lack of refreshes. After finding rows whose retention time is stable and does not vary across multiple tests, we use those rows as Rowhammer victims and nearby rows as Rowhammer aggressors.

TRR Reverse Engineering. We can now test if a given Rowhammer pattern triggers the DIMM's TRR implementation to refresh the victim. First, we write data to the victim rows and wait for half of its retention time. Then, we access the aggressor and dummy rows as per the pattern being tested, followed by a single DIMM refresh command. Finally, we wait for the rest of the retention time of the victim's row, and subsequently check it for bit flips. If the victim row does not have bit flips, we can deduce that TRR happened in the refresh commands issued during the pattern and targeted the victim rows. On the other hand, if the victim row does exhibit bit flips due to retention error, we are able to deduce that the DIMM's TRR mechanism decided not to issue any mitigative refreshes at the refresh command, or the access pattern confused TRR to refresh another row thereby successfully bypassing it.

Experimental Results. Using the method described above, we reverse engineer mechanisms of both the tracking phase and mitigating phase in different Hynix DIMMs. Figure 3 shows the hardware setup of the FPGA-based SoftMC platform, with Table 1 summarizing our findings.

High-Level TRR Implementation. Our experiments show that the DIMMs tested use 3 different TRR implementations with several high-level characteristics in common. The aggressor detection method describes how TRR logic keeps track of potential Rowhammer aggressor rows. Prior work has proposed three models to describe different detection methods: counter-based, sampling-based, and the mix of the two [23]. Counter-based detection keeps track of activation counts of a certain number of rows in a counter table and uses the counter values to select mitigation targets. Sampling-based detection probabilistically makes a decision on whether to track the row in every activation command, and thus usually tracks only one aggressor row. Mixed detection uses both counter tables and probabilistic sampling to track aggressor rows.

We found that *all* Hynix DIMMs tested use *sampling-based* TRR mechanisms, with each DRAM bank having its own individual TRR sampler. Moreover, Hynix's TRR sampler only seems to have the capacity to store a single aggressor, which is sampled during the Rowhammer access pattern.

Tested DIMMs			TRR Analysis Results				FPGA Rowhammer Results					Server RH
Module	Part Number	Date Code	TRR Ratio	Nbr. Ref.	F-to-N Ref. Ratio	Biased Sampl.	Row Ref. Period	Max Act. Per Ref.	Eff. Patt.	Avg. Flips Per Row	Avg. HC_{first}	P_{BS} Avg. Flips / Row
H0–H2	HMA451R7AFR8N-UH TD AC	17-21	1/4	2	-	✓	3640	168	$P1$	183.7	66K	1.39
H3–H4	HMA451R7AFR8N-UH TD BC	18-01	1/4	2	-	✓	3640	168	$P1$	227.6	62K	0.01
H5–H6	HMA451R7AFR8N-VK T3 AC	19-02	1/2	4	1/31	✓	4352	168	$P1$	873.9	35K	12.3
H7–H8	HMA42GR7BJR4N-UH TD AC	18-27	1/2	4	1/31	✓	4096	168	$P1$	1294.9	30K	20.8
H9–H10	HMA42GR7BJR4N-UH TD AA	18-29	1/2	4	1/31	✓	4096	168	$P1$	1246.6	31K	4.59
H11–H13	HMA41GR7BJR4N-UH TD AC	19-09	1/2	4	1/31	✓	4096	168	$P1$	1150.1	28K	2.24
H14–H16	HMA81GR7CJR8N-XN TG AC	19-31	1/2	4	1/31	✓	4352	170	$P1$	1029.3	34K	14.5
H17–H21	HMA81GR7CJR8N-XN TG AD	19-31	1/2	4	1/31	✓	4352	170	$P1$	1139.8	32K	19.8
H22	HMA81GR7CJR8N-XN TG AD	19-32	1/2	4	1/31	✓	4352	170	$P1$	1278.8	33K	32.4
H23	HMA81GR7CJR8N-XN TG AC	22-17	1/2	4	1/31	✓	4352	170	$P1$	945.4	40K	0.68
H24–H25	HMA81GR7CJR8N-XN TG AD	20-24	1/2	4	1/7	✗	3290	170	$Q1$	31.4	18K	-
H26–H28	HMA81GR7CJR8N-XN TG AD	20-49	1/2	4	1/7	✗	3288	170	$Q1$	27.9	19K	-
H29	HMA81GR7CJR8N-XN T4 AC	21-13	1/2	4	1/7	✗	3288	170	$Q1$	25.3	19K	-

Table 1: TRR Analysis and Rowhammer Results. All tested DIMMs used a per-bank sampling TRR implementation, with a capacity for a single aggressor.

TRR-to-REF Ratio. TRR prevents bit flips by deliberately refreshing potential Rowhammer victims during DIMM refresh commands. However, as TRR utilizes existing refresh commands, the DIMM must still preserve the original command's functionality of refreshing each row at least once every 64ms. Since the DIMM can only refresh a limited number of rows during each command, not all refreshes are TRR-capable, i.e., able to refresh the victim row identified by the bank's TRR.

On our Hynix DIMMs we observe that TRR-capable refreshes occur deterministically once every fixed amount of non-TRR refreshes. In Table 1 we refer to this as TRR Ratio, with 1/4 (1/2) indicating that every 4 (2) refresh commands there is one TRR-capable refresh command, respectively.

Number of Neighbor Rows Refreshed and Far-to-Near Refresh Ratio. When the TRR mechanism identifies a DRAM row as a potential aggressor, it needs to decide which victim rows need to be refreshed to avoid Rowhammer-induced flips.

In our DIMMs, except for H0–H4, most TRR versions refresh $A\pm2$ rows on each side of a sampled aggressor row A, yielding a total of 4 neighboring rows refreshed. We further analyze the behavior of refreshing near victims rows ($A-1$ and $A+1$)—or "near-victim refreshes"—and far victim rows ($A-2$ and $A+2$)—or "far-victim refreshes". Here, we observe that a single TRR-capable refresh command refreshes either near or far victim rows, but never targets a mix of the two. Moreover, we notice a fixed ratio between near-victim refreshes and far-victim refreshes. For example, for DIMMs H24–H29, one far-victim refresh happens after 7 near-victim refreshes, yielding a far-to-near refresh ratio of 1/7.

Biased Sampling. Moving to analyzing how aggressor rows are identified, we recall from above that Hynix's sampling-based TRR only keeps track of one single aggressor. Hence, to have a high probability of correct aggressor detection, it is crucial that the sampler samples the aggressor row address uniformly during row activations. Here, biased sampling occurs when rows in certain activations are more likely to be identified as potential aggressors compared to others.

On our DIMMs, we observe that the aggressor samplers are often biased towards the rows activated before a refresh command, commonly identifying those rows as potential aggressors. This makes bypassing the sampling-based TRR simpler, as the attacker can always hide the real aggressor row accesses at the start of the Rowhammer pattern, far away from the refresh commands. This has the effect of making the DIMM's TRR implementation incorrectly identify and refresh fodder rows, while missing flips in the pattern's targeted victim rows.

Regular Refresh Period. The DDR4 standard requires that each row be refreshed at least once every 64 ms, i.e., once every 8K refresh commands. With prior work [23] documenting that DRAM chips internally refresh rows at a higher rate than the DDR4 standard, we report on our DIMMs' refresh period in Table 1, likewise observing refreshes at a much higher rate.

Effective Rowhammer Attack Patterns. To obtain a bit flip, an attacker must cause a sufficient amount of aggressor row activations between two refresh commands targeting the victim row (e.g., within the internal refresh period) while simultaneously using dummy rows to prevent the DIMM's TRR mechanism from correctly detecting the aggressors and refreshing the victim row. As discussed above, the TRR implementations on our Hynix DIMMs probabilistically sample one single aggressor from all incoming activations. For TRR versions with biased sampling, to maximize the possibility of the sampler detecting a dummy row, we hammer aggressor rows at the start of the period between two refresh commands, before transitioning to accessing the dummy rows in anticipation of the refresh command and its associated aggressor sampling. As each row access (hammering) requires a row activation command, in Table 1 we also report the number of activations between consecutive refresh commands for our DIMMs, which we compute by dividing time of refresh interval (tREFI) by row cycle time (tRC) for each DIMM module.

Thus, our first Rowhammer pattern ($P1$) consists of two aggressor rows sandwiching a victim row and one dummy row to bypass TRR. The pattern repeats at every refresh command, regardless of whether it is TRR-capable or not, and lasts for at least twice the DIMM's regular refresh period. Between two consecutive refresh commands, the number of accesses to aggressor rows followed by dummy accesses is dependent on the individual DIMM being tested. However, it is typically the case that about 130–160 aggressor accesses followed by 5–30 dummy accesses, totaling to 170 memory accesses, are sufficient for obtaining a considerable number of bit flips.

Next, as P1 heavily exploits Hynix's biased TRR implementation, it does not yield any bit flips on DIMMs with a non-biased TRR sampler (H24–H29). Instead, for these DIMMs we exploit the fact that there is one far-victim refresh after every 7 near-victim refreshes and that they never happen simultaneously in a single TRR-capable refresh command. As such, we define the pattern $Q1$ by only hammering aggressors in the two "safe" refresh intervals right before every far-victim refresh and hammering a dummy row in other time.

Average Amount of Flips and Hammer Count. Finally, using the above hammering patterns, in Table 1 we summarize the average amount of flips per 8KB row across all of our tested DIMMs. Moreover, we note that the DIMM's row refresh period multiplied by the number of accesses between two consecutive refresh commands sets an upper bound to the number of activations an attacker can cause to aggressor rows. Thus, in Table 1 for each DIMM we also report on its Hammer Count (HC), which is the minimal number of row activations required to trigger the first bit flip. As can be seen, the HC required for inducing Rowhammer ($\sim$50K activations) is substantially lower than the DIMM's internal refreshing period ($>$500K activations), providing ample opportunities for mounting Rowhammer attacks.

4.2 Disabling ECC and Data Scrambling

As we move from Rowhammer on FPGAs to attacking real systems, we begin with disabling the ECC checking and data scrambling mechanisms on these server machines. However, we could not locate a motherboard for Intel Skylake or Cascade Lake platforms where UEFI would let us control these settings. Instead, we note a discrepancy between the UEFI settings presented in the setup GUI of these motherboards, and the actual settings available in the UEFI image. Exploiting this fact, we developed a tool that parses a UEFI image file, allowing us to inspect and modify settings not typically presented in the board's setup GUI. We then flash these modified images directly to the board's SPI NOR flash chip, obtaining greater control over the board's configuration, allowing us to enable/disable ECC and data scrambling. See Appendix A for more details. Overall, we were able to obtain eight platforms from three types (EVGA, ASRock, and Asus) with disabled ECC and data scrambling, allowing us to evaluate the susceptibility of ECC DDR4 DIMMs to Rowhammer attacks.

4.3 Improving Rowhammer for Server RAM

In this section, we move on to launching Rowhammer attacks on our test platforms. We begin by testing existing DDR4 Rowhammer attacks and reporting their (in)effectiveness in causing bit flips on server memory. We then analyze the possible issues causing this, and proceed to present and analyze our own improved Rowhammer techniques. For the rest of this section, we use a test platform based on the ASRock EPC621D8A motherboard and an Intel Xeon W-3235 (Cascade Lake) CPU, with ECC and data scrambling disabled.

Testing Existing DDR4 Rowhammer Techniques. We begin our analysis by testing existing state of the art Rowhammer techniques on (regular) DDR4, namely TRRespass [18] and Blacksmith [27]. As these have targeted client devices, we adapt them to use the server addressing functions from Section 2.1 while leaving default values for other parameters. We run TRRespass's many sided hammering strategies, as well as Blacksmith's more general fuzzing tests on a single DIMM for each part number reported in Table 1.

We observe that TRRespass did not cause any bit flips on all DIMMs tested, even when using up to 32 sided hammering. While Blacksmith fuzzing is able to find effective patterns generating bit flips on DIMMs with the first two TRR implementations (H5–H23), the most effective hammering pattern it finds can only cause 32.4 bit flips per row. This is significantly lower than our FPGA-based analysis in Table 1 and is unlikely to be sufficient to bypass the DIMMs' ECC protections once enabled.

We make a few observations to explain these discrepancies. First, we look at Blacksmith's most effective pattern, P_{BS}. Similar to our pattern $P1$, it exploits the biased sampling and also accesses dummy rows towards the end of refresh intervals. We conjecture that TRRespass' multi-sided hammering approach is ineffective at bypassing Hynix's TRR due to the lack of refresh synchronization and its associated dummy accesses, as both P_{BS} and $P1$ are effective and agree on these techniques. We also note that $P1$ on the FPGA platform is able to cause considerably more bit flips (200-1000 per row) compared to P_{BS} (under 50 per row, rightmost column in Table 1) on a server system. Given the seeming need for using dummy accesses and refresh synchronization, we now proceed to benchmark P_{BS} for row activation throughput.

Row Activation Throughput Benchmarking. In addition to executing specific Rowhammer patterns designed to bypass TRR, an attacker also must ensure a high rate of row activations, causing charge to drain from nearby rows faster than they can be refreshed. Here, unlike prior Rowhammer attacks [18, 35] which compiled manually optimized hammering code, Blacksmith uses Just in Time (JIT) compilation to dynamically generate it. To test the effect of JIT compilation on row activation throughput we manually implement P_{manual}, a pattern similar to P_{BS}, using the code framework of [45]. We run both the JIT-compiled P_{BS} and the manual pattern P_{manual} on 3 DIMMs from Table 1, one for each speed. Table 2 presents a summary of our findings, containing the results of our row activation throughput benchmark, showing obtained activation counts in one refresh interval ($7.8\mu s$) and standard row refresh period ($64ms$).

DIMM Tested	Speed (MT/s)	JITed Pattern # Act. $7.8\mu s$	JITed Pattern # Act. $64ms$	Manual Pattern # Act. $7.8\mu s$	Manual Pattern # Act. $64ms$
H0	2400	57.0	468k	72.0	590k
H5	2666	67.0	550k	72.2	592k
H14	2933	67.6	554k	71.9	590k

Table 2: Average Activation Counts of JITed Hammering Pattern P_{BS} and Manually Optimized Hammering Pattern P_{manual}.

Overall, it is clear that manually optimized patterns achieve a 10.7% higher activation count than the JIT-compiled pattern. Thus, in our setup, we conclude that manual pattern optimization is still required to achieve a maximal row activation account. Therefore, we manually implemented our FPGA-based $P1$ pattern using native C++ code.

The Need for Refresh Synchronization. We recall from Section 4.1 that our $P1$ pattern requires refresh synchronization, so that aggressor rows are accessed only after refresh commands followed by accesses to dummy rows just before the next refresh is issued. While achieving such tight synchronization is trivial on an FPGA-based memory controller which is also responsible for issuing refreshes, on server platforms our attack needs to properly synchronize accesses to aggressor and dummy rows with refresh commands issued by the CPU's memory controller.

Obtaining Refresh Synchronization. Prior works [9, 27, 28] achieved refresh synchronization by detecting refreshes through memory access latency, where spikes indicate that the DRAM access is delayed due to refreshes. Applying such a strategy for $P1$ would involve timing memory accesses using a high-precision `rdtscp` timer until a refresh command is detected. Then, we hammer the aggressor rows followed by accessing dummy rows for a duration of a single refresh interval (about 70 accesses on our machines). Finally, we perform additional memory accesses while measuring their latency, re-synchronizing with the next refresh interval.

However, when attempting to apply the above strategy to $P1$, we notice that accesses to aggressor rows take up 64 out of the 70 memory accesses available between two consecutive refreshes. Thus, with nearly all accesses used for hammering aggressors, there is little room left for accesses to TRR-misleading dummy rows or accesses needed for refresh synchronization. Moreover, spending little time on dummy accesses and refresh synchronization increases the probability of failure, as the nondeterminism introduced by the machine's execution might result in us not performing sufficient dummy accesses, or missing the next refresh command altogether. Either of these events will lead the DIMM's TRR to correctly detect our aggressor rows, issuing victim refreshes.

Improving Refresh Synchronization. Tackling this challenge, we make two modifications to $P1$. First, we notice that there is no difference between accessing dummy rows for bypassing TRR and row accesses used for refresh synchronization. In fact, we can combine these accesses, as long as they do not overlap with aggressor addresses in the same bank. Thus, we define the pattern $P2$ by removing the accesses to dummy rows at the end of $P1$, and directly start refresh synchronization after 64 aggressor accesses. This is preferable as we now have about 6 memory accesses which simultaneously perform both refresh synchronization and dummy activation, giving $P2$ a higher probability in both bypassing TRR and detecting the next refresh command.

Pattern Refining. To find the best balance between aggressor row activations and room for TRR bypass and refresh synchronization, we test different variations of $P2$ using our H21 DIMM with different aggressor access counts. We noticed that the number of Rowhammer bit flips reaches maximum with 46 accesses to aggressor rows, with the remaining 26 accesses simultaneously dedicated to TRR bypassing and refresh synchronization. As such, we define this refined pattern as $P3$, which we use going forward across our experiments.

4.4 Analyzing Data Patterns for Rowhammer

With our Rowhammer pattern $P3$ in hand, we now proceed to evaluate its effectiveness using different data patterns. While prior work [11, 18, 35, 39] has already observed that Rowhammer is most effective when the aggressor and victim rows have opposing data patterns (i.e., 1-0-1 arranged vertically), in this section we proceed to evaluate our hammering pattern across all eight possible 3-bit data patterns. To that aim, we hammered 1024 rows on the H21 DIMM using $P3$, following the methodology from Section 4.3.

Data Pattern	1 0 1	0 0 1	1 0 0	0 1 0	0 1 1	0 0 0	1 1 0	1 1 1
Avg. Flips Per Row	136	68	66	49	31	27	25	1.5

Table 3: Number of Bit Flips Observed With Different Data Pattern Hammerings (Aggressor 1 Data - Victim Data - Aggressor 2 Data). All DRAM timing parameters set to their default values.

As shown in Table 3, the most effective pattern is the one widely recognized in prior works where both aggressor rows are filled with 1s and the victim row with 0s, resulting in a $0 \to 1$ flip in the victim. However, we also observe Rowhammer-induced bit flips using "single sided" patterns where different bit values are used in the two aggressor rows, including both $0 \to 1$ and $1 \to 0$ flips. Finally, we also notice Rowhammer induced bit flips in the 0-0-0 and 1-1-1 pattern, where the victim and both aggressors share the same bit values.

4.5 Observing Rowhammer on Server RAM

Having obtained an optimized Rowhammer pattern $P3$ and an optimal data pattern, in this section we use our ASRock server platform to survey the DIMMs from Table 1 for their Rowhammer susceptibility. We sweep each DIMM for a region of 1GB, corresponding to 8192 rows across 16 banks for single rank DIMMs, or 4096 rows across 16 banks over 2 ranks for dual rank DIMMs. See Table 4.

DIMM Tested	Avg. Flips Per Row	DIMM Tested	Avg. Flips Per Row
H0–H2	1.73	H11–H13	100.3
H3–H4	1.78	H14–H16	100.4
H5–H6	84.8	H17–H21	131.7
H7–H8	130.3	H22	125.4
H9–H10	139.5	H23	94.4

Table 4: Rowhammer Survey Results. We leave DRAM timing parameters set to their default values (tREFI $\approx 7.8\mu s$).

As shown, we are able to achieve more than 100 bit flips per row on most DIMMs with our optimized Rowhammer pattern. This corresponds to $\times 5 - \times 10$ improvement compared to the patterns found by Blacksmith. Finally, we recall that experiments in this section were conducted with the machine's ECC and scrambling features artificially disabled via a custom BIOS image, which we investigate in the next section.

5 Reverse Engineering Server ECC

Having characterized the susceptibility of DDR4 ECC DIMMs to flips, we now reverse-engineer the error correcting and data scrambling mechanisms present on Cascade Lake servers. We first provide basic coding theory background.

5.1 Coding Theory Background

In coding theory, block codes refer to codes which encode data pieces of some fixed bit block length k. During encoding, data is divided into blocks that are encoded individually into a codeword of length n, providing error detection and correction redundancy with extra $r = n - k$ parity bits. Such a code C is denoted as an (n,k) code and has a granularity of a single bit. One important property is the minimum code distance $d_{\min}$, which is the minimum Hamming Distances (HD) between any two codewords. A well-known example of linear block codes is the Hamming (7,4) code [21] which has a code distance $d_{\min} = 3$, meaning at least 3 bits need to be flipped inside a codeword to get another codeword.

Encoding and Decoding in Linear Block Codes. For an (n,k) block code operating over $\mathbb{F}_2$, its encoding function $E: \mathbb{F}_2^k \to \mathbb{F}_2^n$ takes a k-bit row vector u as input and maps it to an n-bit row vector v as codeword output. Moreover, for linear codes, the encoding operation can be defined as matrix multiplication using the code's generator matrix $\mathbf{G}$ of size $k \times n$, i.e., $E(u) = u \cdot \mathbf{G}$. Finally, the set of all valid codewords C is defined by the image of the generator matrix $\mathbf{G}$, $C = \{uG : u \in \mathbb{F}_2^k\}$.

The decoding operation can be viewed as a three-step process: parity checking, correction, and data decoding. For the first step, a message m of n-bits is checked by the code's parity checking function $P: \mathbb{F}_2^n \to \mathbb{F}_2^r$, yielding an r-bit vector s, usually called the syndrome. Similar to the encoding process, for linear codes the parity checking function can be represented by $F_P(m) = \mathbf{H} \cdot m^T = s^T$ where $\mathbf{H}$ is the parity checking matrix of size $r \times n$. The parity checking matrix guarantees that if $s = 0^r$ then the received message $m \in C$, meaning that m is a valid codeword with no errors. Alternatively, a non-zero syndrome s means that there is error in the message m, requiring us to invoke the code's correction algorithm, resulting in an error-free codeword $\widehat{m}$. In general, a code with distance $d_{\min}$ can detect $d_{\min} - 1$ errors and correct $\lfloor (d_{\min} - 1)/2 \rfloor$ errors. Finally, the message is decoded back to its corresponding data vector u, using the inverse function E^{-1} of E, that is $E^{-1}(m) = \mathbf{G}^{-1} \cdot m^T = u^T$.

Systematic vs. Non-Systematic Codes. A linear (n, k) code is systematic if for any input k-bit vector u, the vector formed by the first k bits of its corresponding codeword v is equal to u. For systematic linear codes, this implies that the generator matrix has a standard form $\mathbf{G} = [I_k | P]$, where I_k is the $k \times k$ identity matrix and P is a matrix of size $k \times r$. Correspondingly, the code's parity check matrix and data coding matrices are $\mathbf{H} = \left[P^T | I_r\right]$ and $\mathbf{G}^{-1} = [I_k | \mathbf{0}]$, respectively. For systematic linear codes, a codeword $v = (u,p)$ can be viewed as two parts: a data part u and a parity check part p. This simplifies the data decoding procedure as taking the first k bits of a codeword, allowing for parity checking (i.e., syndrome computation) to be done in parallel.

Concatenated Codes. Concatenated codes form a class of error correction codes which are constructed by applying an inner (n,k) code C_{in} on the outputs of an outer (N,K) code C_{out} [21]. Thus, to encode a vector u with the concatenated code $C_{\text{out}} \circ C_{\text{in}}$, the outer code C_{out} is first used to encode u to a codeword v. Next, each symbol v_i is encoded using C_{in}, with their concatenation $(C_{\text{in}}(v_1), C_{\text{in}}(v_2), \ldots, C_{\text{in}}(v_m))$ being the output of the concatenated code $C_{\text{out}} \circ C_{\text{in}}$ on the input u. Finally, if C_{out} has minimum code distance $D_{\min}$ and C_{in} has $d_{\min}$, the concatenated code is a (nN,kK) code with minimum distance $d_{\min} \cdot D_{\min}$ [21].

ECC in Server DRAM. On server and workstation systems ECC is implemented in a side-band manner, involving modifications to both the CPU and memory hardware. From the CPU side, the memory controller is responsible for encoding and decoding the data during DRAM read/write requests. On the memory side, ECC capable memory modules typically contain an extra chip for storing codewords (as opposed to data values). Finally, the bus connecting CPU and memory is also expanded to accommodate the extra data, growing from 64-bit bursts for regular DDR4 DIMMs to 72-bits for ECC-equipped DDR4 memory. The resilience guarantee of a system's error correction mechanism stems from the underlying error correction code, which is in turn designed based on certain data corruption models. The most commonly known error correction and detection capability is Single Error Correction and Double Error Detection (SEC-DED) [56]. Here, the memory controller will be able to correct and report 1-bit errors and forcefully "crash" the system when encountering 2 or more errors. Another common DRAM failure mode is single-chip failure, where a single DRAM chip has multiple bit errors across bursts or even completely fails. For high-availability systems, vendors [26] have designed advanced ECC technologies that enhance the memory system against single-chip failure, which is beyond the capability of SEC-DED. Collectively known as Single-Device Data Correction (SDDC), a common example is Chipkill, which provides device failure protections.

5.2 Recovering Intel's ECC Implementation

Focusing on DDR3-based servers, ECCPloit [11] was able to recover the code's generator matrix using the code's syndromes reported by the system's BIOS or by using the system's data sheet. However, in our case we were unable to find a DDR4 Intel system which reports syndromes used for error correction or documentation regarding Intel's ECC implementation. Thus, we used a logic analyzer setup to directly observe and recover the code's generator matrix, which we now describe.

Logic Analyzer Setup. Logic analyzers allow one to capture logic signals from a digital circuit. We use a Nexus Technology DDR4INTR288-BB-XL-A01 memory interposer connected to a pair of Tektronix TLA7BB4 logic analyzer modules, each capable of state speed of 1.4 GHz and equipped with 64MB of RAM. The modules themselves are housed in a TLA7016 chassis, which uses an interface module to connect to a host PC via Ethernet. See Figure 4. With DDR4 transmitting data on both rising and falling clock edges, this setup is capable of observing DDR4 transactions up to 2800 MT/s. Finally, our setup is based on equipment acquired on the secondary market, totaling at around $3000.

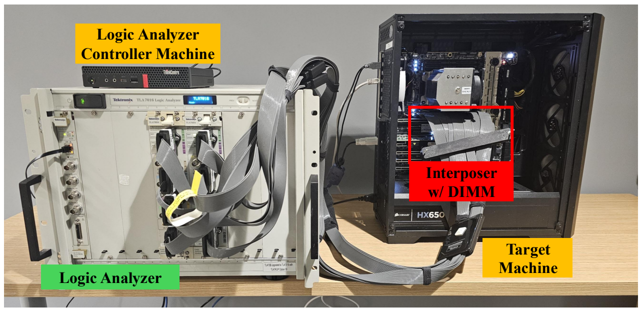
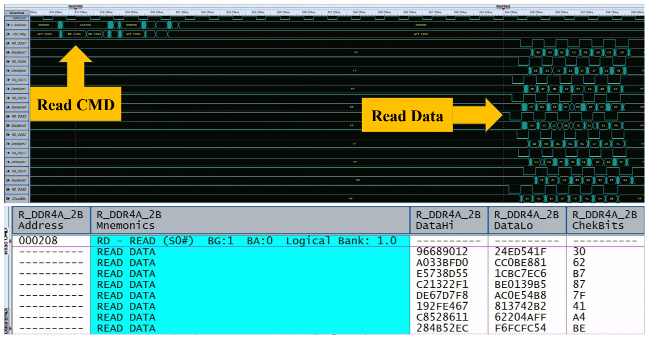


Figure 4: Logic Analyzer and DDR4 Interposer Setup

Target Setup. We use a target machine equipped with an EVGA SR-3 Dark motherboard and an Intel Xeon W-3235 (Cascade Lake) CPU. The motherboard was flashed with a modified BIOS image (version 2.10) that disables DRAM data scrambling, while leaving the ECC functionality enabled. We limited the machine to 2666 MT/s, to stay below the maximal acquisition speed supported by the logic analyzer. Finally, our target machine was used with a Hynix HMA81GR7CJR8N–XN TG AD DIMM placed inside the interposer.

Observing Bus Transactions. Figure 5 (top) shows an example of acquired memory read transactions, as captured by our logic analyzer, which is then parsed into a listing in Figure 5 (bottom). The logic analyzer allows us to observe a significant amount of information regarding the CPU's memory activity, including memory commands, addresses, read/write data, and, crucially, ECC check bits not accessible via software.



R_DDR4A_2B Address	R_DDR4A_2B Mnemonics	R_DDR4A_2B DataHi	R_DDR4A_2B DataLo	R_DDR4A_2B ChekBits
000208	RD - READ (S0#) BG:1 BA:0 Logical Bank: 1.0	----------	----------	----------
----------	READ DATA	96689012	24ED541F	30
----------	READ DATA	A033BFD0	CC0BE881	62
----------	READ DATA	E5738D55	1CBC7EC6	B7
----------	READ DATA	C21322F1	BE0139B5	87
----------	READ DATA	DE67D7F8	AC0E54B8	7F
----------	READ DATA	192FE467	813742B2	41
----------	READ DATA	C8528611	62204AFF	A4
----------	READ DATA	284B52EC	F6FCFC54	BE

Figure 5: Waveform View of Acquisition Data (top) and Parsed Listing View of Acquisition Data (bottom)

Recovering the Code's Size. We now proceed to find out the size of the error correction code on the Intel Cascade Lake platform. The DDR4 standard specifies the ratio of normal data bits to redundancy bits as 8-to-1. Thus, recovering the input data block size k is effectively recovering the code size by $n = k + r = k + k/8$. Next, as each data block is encoded individually and does not affect the checkbits of other blocks, we can recover the value of k by changing the data written to the cache line and examine consequent changes in ECC checkbits via the logic analyzer. Using this method, we observe that on DDR4 systems the ECC operates on the entire cache line (i.e., $k = 512$ bits) with the code's output being $n = 576$ bits long. Finally, we note that DDR4 ECC systems seem to be using much wider codes compared to their DDR3 counterparts, with prior works investigating ECC on DDR3 systems [11, 39] reporting $k = 64$ and $n = 72$ bits, respectively.

Recovering the Code's Generator and Parity Check Matrix. Next, using our logic analyzer we can also recover the code's generator matrix **G** of size 512×576 bits. We do this by performing 512 individual memory write operations, where the ith operation has all data bits set to 0, with the ith bit set to 1. As ECC encoding consists of multiplying the message m by the generator matrix **G**, writing these data patterns has the effect of setting the ECC bits during the ith write transaction to be the ith row of **G**.

We use this method to recover the generator matrix **G** used on Intel's Skylake and Cascade Lake platforms. Finally, we also recover the code's parity check matrix **H** using Section 5.1, and release both artifacts in Section 9.

5.3 Bypassing Error Correction Mechanisms

With both the generator and parity check matrices recovered, we now analyze both the minimum code distance $d_{\min}$ of Intel's ECC implementation and its data correction capabilities.

Obtaining Minimal Code Distance and Rowhammer Implications. We begin by recalling that for linear codes, the code distance $d_{\min}$ is equal to the minimum weight of its non-zero codewords [21]. Moreover, a message of n bits is a valid codeword if and only if it gives a zero syndrome, namely the product of the parity check matrix $\mathbf{H}$ and its transpose is a zero vector. Combining the above two facts, we calculate all 576 bits data with all bits being zero and only i bits set to one, for $i = 1, 2, \cdots$, stopping as soon as a valid code word is found. While this process requires a non-polynomial $\binom{576}{i}$ checks during its ith iteration, we were able to find 1186 codewords with $i = 4$, which also sets the minimal distance of Intel's ECC implementation.

From the perspective of Rowhammer attacks, let $m\mathbf{G}$ be a 576-bit code word corresponding to a 512-bit message m. If an attacker is able to inject an error vector e into the machine's memory $m\mathbf{G} + e$, the attack will remain undetected when the value is read back by the processor if e is a valid code word generated by $\mathbf{G}$.[1] Thus, we call the set of 1186 codewords of weight 4 a *4-bit bypass template*, as simultaneously flipping these indices in a cache line via Rowhammer will result in an attack undetectable to the machine's ECC implementation.

Investigating Intel's SDDC Capability. Beyond error detection, we notice that Intel explicitly advertises that the Cascade Lake Platform has x4 SDDC functionality, namely the ability to correct errors resulting from the failure of a single x4 DRAM chip. To investigate this functionality, we first determine which bits in a cache line lie in the same x4 chip via the DDR4 RDIMM standard [29].

Then, we proceed to confirm the existence of x4 SDDC on the Cascade Lake Platform. To that aim, we temporarily disable the machine's ECC (see Section 4.2) and profile DIMM H21 for 64-byte cache lines which contain more than 1 bit flip in the same x4 chip. We then enable ECC and observe the machine's behavior with only those bits flipped using Rowhammer. With bit flips confirmed via the logic analyzer, we find that the error correction mechanism can correct more than one bit flip (tested up to 5) in a single x4 chip and will crash if there are flips across two or more chips. Therefore, we conjecture that the ECC code on Cascade Lake platforms is a concatenated code with an outer code for detecting faulty chips and an inner code for error-correction. Overall, the resulting code can correct all bit flips within a single x4 chip and will crash if bit flips span across two or more chips. Finally, we note that H21 used for testing in fact uses eight x8 chips. However, while H21 appears as a x8 DIMM in the system's BIOS, we empirically confirm that Intel's ECC implementation appears to ignore this data, treating H21 as a x4 DIMM composed of 16 chips.

Exploiting Error Correction for Bypassing ECC. We now exploit the x4 SDDC correction capability to further loosen the requirements of a successful Rowhammer bit flipping attack. We first analyze every 4-bit bypass template by counting the number of chips where its bits lie. We find that there are 334 4-bit bypass templates which span across two chips, with 275 and 577 spanning three and four chips (respectively). Focusing on 4-bit bypass templates that span *three* chips, each template has two bits that lie in one chip and another two bits that lie in two other different chips. See Figure 6 (left).

Next, by flipping the latter two bits (in different chips, Figure 6 (middle)), Intel's SDDC mechanism will mistakenly identify the *former* chip *without* flips as faulty, *architecturally* "correcting" the value of the entire code word to include the flips corresponding to the first chip from the considered 4-bit bypass template (Figure 6 (right)). Finally, we empirically verify this behavior across multiple 4-bit bypass templates via our logic analyzer (see Appendix B for one example).

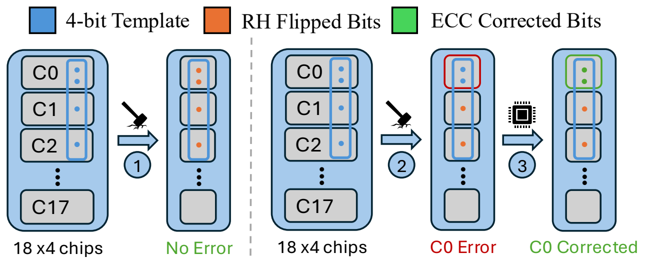


Figure 6: 4-bit Bypass Template (left) and 2-bit Correctable Bypass Template (right)

Overall, while simultaneously flipping indices from a 4-bit bypass template results in a Rowhammer attack undetectable to the machine's ECC implementation, flipping just two bits in different chips from a 4-bit bypass template spanning 3 chips results in an attack that is detectable by the ECC, which is subsequently incorrectly corrected by it. Going forward, we refer to these as 2-bit correctable bypass templates.

Number of Flippy Cache Lines and Exploitable Templates. With our analysis of Intel's ECC implementation in hand, we now present a breakdown of cache lines within a 1GB region based on the number of Rowhammer-induced bit flips in them. To that aim, we hammer with the machine's ECC mechanisms disabled, averaging our results across DIMMs H0–H23. Figure 7 presents a summary of our findings, showing a considerable number of cache lines exhibiting more than two Rowhammer-induced bit flips.

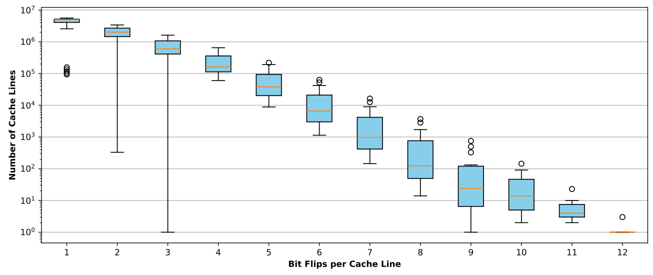


Figure 7: Number of Rowhammer Induced Bit Flips Per Cache Line

[1] i.e., there exists e' such that $e = e' \cdot G$, implying that $m\mathbf{G} + e = (m + e')\mathbf{G}$ with the CPU reading back $m + e'$ after decoding.

Next, we also estimate the average number of 4-bit bypass templates and 2-bit correctable bypass templates across DIMMs H0–H23. As can be seen in Table 5, while the number of 4-bit bypass templates appears to be very limited, most of the DIMMs tested have several thousand 2-bit correctable bypass templates in a 1GB region.

DIMM Tested	Avg. # 4-bit Templates	Avg. # 2-bit Templates	DIMM Tested	Avg. # 4-bit Templates	Avg. # 2-bit Templates
H0–H2	0	0.33	H11–H13	0	4958.7
H3–H4	0	1	H14–H16	0.33	4946
H5–H6	0	3095.5	H17–H21	0.2	8549
H7–H8	0	6779	H22	0	8408
H9–H10	0.5	8493	H23	0	4497

Table 5: Number of 4-bit Bypass and 2-bit Correctable Bypass Templates

5.4 Investigating Intel's Data Scrambling

Until now, we have recovered and analyzed Intel's ECC implementation with the data scrambling functionality disabled. In realistic Rowhammer scenarios, it is necessary to consider the effects of data scrambling on actual data stored in memory. Thus, we now proceed to enable data scrambling and investigate its functionality on our Cascade Lake server.

Observing Data Scrambling. We begin our investigation by determining whether data scrambling happens before ECC encoding or after ECC encoding. To that aim, we write a test program that writes an all zero cache line u_0 to a fixed memory location, observing the scrambled 576-bits data s_0 via the logic analyzer. Using our recovered parity check matrix, we verified that s_0 is *not* a valid code word, which implies that data scrambling happens after ECC encoding. Finally, we observe that the check bits part in s_0, which is all zero after encoding, is also scrambled, which means Intel's scrambling mechanism operates on strings of 576 bits.

Predicting Scrambling Values. As outlined in Section 2.3, memory controllers disrupt long consecutive sequences of 0s and 1s by XORing pseudorandom patterns to data writes (and XORing them again during reads). To recover Intel's scrambling implementation, we use our program to write the same all zero data to multiple different memory locations, across different banks, rows, and cache lines in each row. Comparing the resultant scrambling strings via the logic analyzer, we find that our server generates different scrambling string across banks and cache lines in each row, while the string for corresponding cache lines across rows in a bank stays the same. We also observed that scrambling string generation is not randomized across system boots, or even machines, appearing to be hard-coded. As our setup contains 16 banks and 128 cache lines per row, this results in a total of 2048 different scrambling strings which we recover via the logic analyzer.[2] Finally, as Intel's scrambling implementation appears to be a deterministic XORing of fixed masks into the data stream, it (obviously) does not prevent the generation of data patterns required for Rowhammer attacks.

[2]This can be found as an artifact in Section 9.

6 Rowhammer Attacks with ECC Enabled

Having recovered the hammering templates required to bypass ECC checks on Intel architectures, we now construct Rowhammer attacks with ECC and data scrambling enabled.

Experimental Setup. We mirror the experimental setup of Section 4, using a machine equipped with an ASRock EPC621D8A motherboard and an Intel Xeon W-3235 (Cascade Lake) CPU. For memory, we use Hynix Registered DDR4 ECC memory, part number HMA81GR7CJR8N-XN TG AD. Next, our motherboard uses BMC controller firmware revision 1.60, and BIOS version P2.10. For software, our target machine runs Linux Debian with kernel version 4.9.0-19-amd64. Finally, unlike in previous sections, going forward we leave all BIOS and Linux settings in their default state, yielding a DRAM refresh rate of about 7.8μs, and ECC and data scrambling both enabled.

6.1 Observing Bit Flips with ECC Enabled

When ECC is enabled, any Rowhammer-induced single bit flips are likely to be corrected, making it impossible for us to infer whether a specific index is affected by Rowhammer via software memory accesses. Tackling this issue, prior work on DDR3 [11, 39] observed significantly higher access latency (up to 5 orders of magnitude) to cache lines containing bit flips, due to synchronous software operations caused by System Management Interrupts (SMI) and Corrected Machine Check Interrupts (CMCI) [13], which notify the system of memory errors and attempt to correct them. Tracked under CVE 2018-18904, we did not observe this interrupt-based side channel on any of our machines, presumably due to mitigations present on modern DDR4 systems.

Reliably Observing Bit Flips Under ECC. To investigate the system's behavior when encountering a correctable bit flip, we first templated a DIMM with ECC disabled (see Section 4.2), collecting information about the locations of flippy cache lines. Next, we enabled the system's ECC functionality and hammered multiple flippy and non-flippy cache lines, measuring their access latencies after each hammering round. See Figure 8. As can be seen, despite our system BIOS default settings being set to not issue SMI and CMCI interrupts on individual bit flips, memory accesses to cache lines containing a correctable bit flip take about x3-4 times longer compared to normal access latency. Finally, we note that accessing a cache line may rarely coincide with a refresh operation occurring in the same bank, causing a spike in access latency that could be misinterpreted as a false positive. Thus, by hammering and accessing the tested location multiple times, we are able to reliably observe correctable bit flips via access latencies.

Investigating the Root Cause of Correction Spikes. Investigating further why these latency spikes occur, we observed the system's behavior while accessing a flippy cache line via the logic analyzer (see Appendix C for examples).

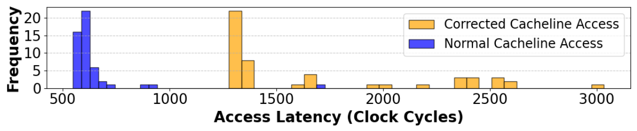


Figure 8: Access Latencies For Normal and Corrected Cache Lines

Here, we noticed that when an ECC error is detected, the memory controller issues three additional read requests to the same location. This behavior resembles the request replay feature described in Intel and AMD documentation [1, 25], which is designed to address transient memory bus errors. After detecting the same error in all three replayed requests, the controller corrects the error, writing back the verified data using a different ECC. Finally, we observed that the memory controller reverts to using the original ECC after a write operation to the corrected cache line.

6.2 DIMM Templating

Having identified 2-bit correctable bypass templates and a timing side channel allowing us to reliably observe bit flips corrected by the machine's ECC implementation, we proceed to template the DIMM for locations of bits that can be exploited using our 2-bit correctable bypass templates. To that aim, we proceed to test the bits of every 64-byte codeword individually, using the most effective 1-0-1 data pattern identified in Section 4.4. Thus, we set all the bits in the aggressor and victim cache lines to 1, except the (single) bit being tested in the victim cache line which we set to zero.

Unfortunately, even when the data values in the aggressor and victim cache lines differ by only a single bit (the bit being currently tested) the ECC bits corresponding to a victim cache line sandwiched between two aggressor lines create many unintentional effective hammering data patterns. This in turn increases the likelihood of uncorrectable errors due to flips in the code's check bits, typically resulting in system crashes.

ECC Aware Hammering. To minimize hammering the effective patterns in ECC bits, we use the layout shown in Figure 9.

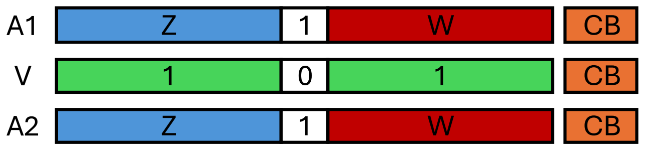


Figure 9: Hammering Layout for Minimizing ECC Bits Hammering

Here, we begin by filling the victim cache line V (middle) with 1s (green), except the bit under test which is set to zero (white). We then use the code's generator matrix **G** to compute the corresponding value CB of the code's check bits (orange). Next, we would like to find values Z and W for the aggressor cache lines (A1 and A2), such that $Z|1|W$ produces the same check bit values CB as the victim cache line V. While this results in only 1-1-1 and 0-0-0 hammering patterns appearing in the check bit values (orange), we would also like to minimize the amount of bit locations that do not form a 1-1-1 pattern in the Z and W parts of Figure 9 in order to prevent these bits from being hammered.

Mathematically, this problem is equivalent to finding two values Z and W such that $A_1 \oplus V$ have zero-valued check bits, i.e., that $Z|1|W|CB \oplus 1\cdots1|0|1\cdots1|CB$ is a valid code word generated by **G**. Moreover, we would like $\overline{Z}|1|\overline{W}|0$ to have as few non-zero bits as possible, minimizing the amount of effective hammering patterns outside the bit being tested.

To accomplish this task, we utilized the generator matrix **G** to search for "light" values of $\overline{Z}$ and $\overline{W}$ that satisfy the above condition. Here, we begin with Z = W = 1, and test different pairs of Z and W, testing all $\binom{512}{i}$ values that have at most i bits set to 1, before proceeding to $i+1$. With **G**'s code distance being only four, we were able to generate values of Z and W for all 512 bits in a cache line, containing only 3 non-zero bits. For a bit index j, we refer to the corresponding values Z_j and W_j as its testing values.

Blacklisting Weak Cache Lines. Having successfully generated testing values for all bits in a victim cache line, we are still left with 0-0-0 and 1-1-1 hammering patterns in the check bits part of the aggressor cache lines (A1 and A2) and the victim V. Observing Table 3, such patterns (especially the 0-0-0 pattern) alone are sufficient to cause system crashes. In order to mitigate this, we avoid hammering cache lines whose ECC bits are vulnerable to such patterns. Thus, before testing any cache lines, we must first test the flipping characteristic of its ECC bits under the 0-0-0 and 1-1-1 patterns. In case a cache line is found to be susceptible to such flips, we exclude it from the hammering list. We refer to this process as blacklisting.

Performance Degradation During Blacklisting. Unfortunately, the blacklisting phase can itself cause system crashes when testing for cache lines containing bits susceptible to the 0-0-0 and 1-1-1 patterns, resulting in crashes in case two such bits are present in a 512-bit cache line. To address this, instead of starting the blacklisting phase with our optimized pattern $P3$, we hammer in stages of increasing effectiveness.

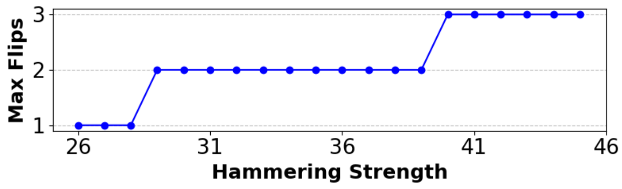


Figure 10: Rowhammer Effectiveness as a Function of Row Activations. Notice the Bit-by-Bit Increase in Cache Line Flippiness.

Figure 10 demonstrates this effect, where hammering effectiveness is directly proportional to the amount of row activations done within a refresh period. This allows us to test bits susceptible to 0-0-0 and 1-1-1 patterns one at a time, with any flips being corrected by the machine's ECC mechanism and the affected cache line added to the blacklist.

6.3 Hammering For ECC Bypass

After identifying the 2-bit bypass tuples, discovering a timing side channel to reliably observe individual bit flips corrected by the machine's ECC, as well as implementing bit-testing data values and blacklisting to prevent crashes, in this section we now perform end-to-end Rowhammer attacks with the machine's ECC mechanisms fully enabled. To that aim, we begin by allocating 2MB huge pages, which we will profile for containing Rowhammer-induced bit flips. For each row under test, we first perform the blacklisting phase, aiming to eliminate cache lines whose ECC bits are vulnerable to unwanted patterns (e.g., 0-0-0). After blacklisting, we systematically test each 2-bit bypass template from Section 4.4 for every non-blacklisted cache line inside the row before proceeding to the next 8KB row in the huge page.

Evaluation. We evaluate our end-to-end attack under both idling conditions and while executing the `500.perlbench_r` SPEC2017 [7] benchmark in parallel to our hammering code. During each evaluation scenario we performed 10 hammering experiments starting from randomly selected rows. Here, we measured the time taken by each of our attack phases, end-to-end hammering time, as well as the number of rows needed to be hammered til the first architectural bit flip is observed. Table 6 summarizes our findings. As can be seen, for an idle system, obtaining a Rowhammer-induced bit flip can be done within about 2.5 hours on average, despite the machine's ECC mechanisms being fully enabled. Moreover, our hammering technique did not result in any crashes or machine check exceptions, with the machine being fully responsive during our hammering process. Next, hammering in parallel to running the SPEC2017 benchmark, the time to obtain a bit flip increases to an average of 10 hours, with five out of ten experiments resulting in system crashes. Finally, we note that in both scenarios our attack did generate correctable errors (in addition to the architectural bit flip), which were reported and logged by both IPMI and OS kernel EDAC.

Cond.	Blcklst Time Per Row	# Blcklsted CL Per Row	Time to Test a Template	Total Row RH Time	# Rows Til Bit Flip	Total Time
Idle	20m	23.8	8.4s	15.1m	4.1	2.5h
SPEC	20m	27.9	8.4s	15.1m	26.8	10.4h

Table 6: End-to-end statistics for obtaining visible bit flips via Rowhammer. The numbers reported are averages across all ten runs.

6.4 Rowhammer Attacks on RSA Signatures

Having established the feasibility of Rowhammer-induced bit flips on Intel server platforms with ECC mechanisms enabled, we now demonstrate end-to-end Rowhammer exploits on RSA signatures, reproducing the public key flipping attack of [11].

RSA Overview. RSA is a public key cryptographic algorithm that relies on the hardness of factoring to ensure security [55]. We focus here on its digital signature variant, which allows one to verify the authenticity and integrity of a message. RSA signatures work by generating two primes p,q of roughly equally size and then multiplying them together to form a composite modulus $n = pq$. This modulus, along with an exponent e (typically 3 or 65537) then comprises the public key (e,n) that is used to verify signatures. The private key d is computed such that $ed \pmod{\phi(n)} \equiv 1$. Signatures are computed as $\sigma = m^d \pmod{n}$, and verified with the public key by checking that $m = \sigma^e \pmod{n}$. As factoring is considered to be a hard problem for sufficiently large (and properly generated) n, it should be hard to recover the private key if only given the public key, preventing an attacker from forging signatures without obtaining the private key. However, as noted in [54], even a handful of Rowhammer-induced bit flips can be used by attackers for RSA signature forgery. At a high level, once the location of flippy bits has been identified, the attacker performs a memory massaging step, which is a series of memory deallocations designed to trick the allocator into placing a copy of the public key on the flippy bits. Next, by flipping bits in the RSA modulus via Rowhammer, we can turn it into a product of multiple smaller primes, making it substantially easier to factor. Next, by factoring the modulus and thus recovering a corresponding private key, we enable a variety of attacks, such as those against OpenSSH discussed in [54].

RSA Rowhammering. We now discuss how we can utilize our attack to gain such advantageous bit flips in an RSA modulus to enable practical factoring. Using standard OpenSSL 3.2.2 functionality, we generate 1024-bit RSA keys and utilize a program that acts as a signature verification oracle, taking in signatures via a socket to verify using the public key. By flipping bits in this public verification key via Rowhammer on our server platform, our goal is to create a new modulus n' that we can easily factor to recover the corresponding private key d' (using similar methodology to [11, 54]). With this, we can then generate signatures that the oracle will correctly verify using the modified keypair.

We run our factoring code on a system with two Intel Xeon Platinum 8352Y CPUs and 1.5TB of RAM[3], and use our standard test platform to run the oracle. After executing our bit flipping attack, we recover the modified public key, which we then attempt to factor. We use a Sage implementation of ECM for factoring n' [63]. We note that the location of our bit flips within the key is crucial to the (timely) success of our attack, as these locations dictate how the modulus is modified and thus how easy it is to factor. For example, we are able to factor certain 1024 bit moduli in under 100 seconds, while others took at most 55 minutes.

7 Countermeasures and Future Work

Our attack relies on multiple properties of the target machine for its success, leaving many avenues for countermeasures.

[3]Though our code is single-threaded, and we only parallelize different trials.

7.1 Hardware Mitigations

Existing hardware mitigations for Rowhammer only protect the memory system from certain Rowhammer access patterns and cannot provide rigorous security guarantees. As a result, past works have repeatedly demonstrated new access patterns that overcome existing hardware defenses [14, 18, 27, 32, 36].

In response, researchers have proposed systematic hardware mechanisms for protecting against Rowhammer [5, 8, 31, 40, 48, 50, 53, 57, 58, 67, 68, 69]. New designs [5, 31, 40, 50, 53, 58] are proposed for counter-based or probabilistic in-DRAM trackers with low performance overhead, guaranteeing Rowhammer resilience even in the case of low activation tolerance of DRAM cells. These tracker designs can be mathematically proven to provide Rowhammer protection across different row activation count tolerances. Randomized row swapping [57, 68] or shuffling [67] protects DRAM rows from Rowhammer bit flips by breaking temporal aggressor and victim row adjacency in a randomized fashion. BreakHammer [8] and BlockHammer [69] attempt to improve these defenses by slowing down a potential attacker when detecting activity consistent with Rowhammer attacks. This slow down potentially prevents Rowhammer attacks without the need to identify the row that the attacker targets. DDR5 has introduced a new command for Rowhammer protection, referred to as refresh management (RFM), with several works [33, 46] exploring measures for Rowhammer protection using RFM.

An alternative to monitoring behaviors that are consistent with an attack is to improve the error detection and correction code and make it more robust. CSI:Rowhammer [30] replaces the ECC code with a cryptographic hash. The approach promises correction of every single bit error and probabilistic detection of any number of bit flips.

7.2 Software Mitigations

A common limitation of all hardware-based approaches is that these cannot be applied to already deployed systems, requiring hardware redesigns. On the other hand, existing systems can be protected with software-based Rowhammer mitigations [2, 3, 6, 37, 44, 62, 70]. ANVIL [2] and SoftTRR [70] take the approach of detecting potential attack memory access patterns and refreshing victim memory preventatively. ANVIL uses kernel performance counters to monitor cache miss rate and track possible Rowhammer aggressor rows, because Rowhammer attacks typically incur frequent cache misses. However, it suffers from false positives and attack escape by carefully limiting hammering rate. SoftTRR focuses on protecting kernel page tables from malicious bit flips, which takes less performance penalty at the cost of weaker security guarantees. Another common Rowhammer mitigation approach is memory isolation based on security domains [3, 6, 37, 44, 62]. Various proposals have explored memory isolation at different fine-grain levels or focusing on different applications: between kernel memory and user space memory [6], between virtual machines [44], between applications [3, 62], or between every DRAM row [37].

7.3 Future Work

Our attack is specific to the hardware we used, including Intel machines and Hynix DIMMs. We leave the task of extending and generalizing the attack to future work. One promising candidate for extension is investigating other memory vendors (Samsung and Micron), who likely use completely different approaches for TRR, thereby necessitating new approaches for overcoming the defense. Similarly, very little research has been done on attacking DDR5 memory. Another direction is investigating other CPU models or vendors, such as AMD or Apple. In particular, we expect the ECC algorithms and the memory scrambling functions to vary significantly between vendors. Expanding the scope of the investigation will not only improve our understanding of the scope of the issue, but also allow for developing more generic techniques for carrying out the necessary steps for developing attacks.

Finally, modern high-availability servers are equipped with subsystems for reliability, availability, and serviceability (RAS). Our Rowhammer attack generates correctable errors, which are reported and logged by both the motherboard management system (IPMI) and OS kernel EDAC. Thus, our attack is detectable if system logs are being monitored by system administrators. While we leave the task of creating an undetectable Rowhammer attack against server hardware to future work, as a stop gap measure we note that a higher ECC error logging level and lower bit flip reporting threshold will increase the probability of attack detection until a fundamental solution to Rowhammer is developed and deployed.

Acknowledgments

This research was supported by the Alfred P Sloan Research Fellowship; an ARC Discovery Project number DP210102670; the Defense Advanced Research Projects Agency (DARPA) under contract numbers W912CG-23-C-0022; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972; and a gift from Qualcomm.

8 Ethics Considerations

We own all devices used in our experiments, and these devices are free of any sensitive user data or personal information. These devices are only accessible to lab members, and are not exposed to unauthorized users. All keys and cryptographic secrets were generated solely for our experiments.

9 Open science

We list all artifacts supporting this paper below. We have published all of them at https://doi.org/10.5281/zenodo.15579424.

1. Source code, instructions, scripts, and/or files for TRR profiling and testing Rowhammer patterns using an FPGA memory controller, testing Rowhammer on a test platform and the tool that parses and modifies a UEFI image file (cf. Section 4)
2. Source code and instructions for testing Intel's ECC implementation, data correction capabilities, and data scrambling, and the ECC generator matrix and parity check matrix for Intel's Skylake and Cascade Lake platforms (cf. Section 5)
3. Source code, instructions, and testing scripts for DDR4 Rowhammer with ECC and data scrambling enabled and for RSA (cf. Section 6)

References

[1] AMD. 5th gen amd epyc processor architecture white paper. Technical report, Advanced Micro Devices, 2023. Accessed: 2025-01-19.

[2] 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. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 743–755, New York, NY, USA, 2016. Association for Computing Machinery.

[3] Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahamd-Reza Sadeghi. Rip-rh: Preventing rowhammer-based inter-process attacks. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, page 561–572, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 987–1004, 2016.

[5] F. Nisa Bostanci, Ismail Emir Yüksel, Ataberk Olgun, Konstantinos Kanellopoulos, Yahya Can Tugrul, A. Giray Yaglikçi, Mohammad Sadrosadati, and Onur Mutlu. CoMeT: Count-min-sketch-based row tracking to mitigate RowHammer at low cost. In *HPCA*, pages 593–612, 2024.

[6] 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 *26th USENIX Security Symposium (USENIX Security 17)*, pages 117–130, Vancouver, BC, August 2017. USENIX Association.

[7] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery.

[8] Oguzhan Canpolat, A. Giray Yaglikçi, Ataberk Olgun, Ismail Emir Yuksel, Yahya Can Tugrul, Konstantinos Kanellopoulos, Oguz Ergin, and Onur Mutlu. BreakHammer: Enhancing RowHammer mitigations by carefully throttling suspect threads. In *MICRO*, pages 915–934, 2024.

[9] Yaakov Cohen, Kevin Sam Tharayil, Arie Haenel, Daniel Genkin, Angelos D. Keromytis, Yossi Oren, and Yuval Yarom. HammerScope: Observing DRAM power consumption using Rowhammer. In *CCS*, pages 547–561, 2022.

[10] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are we susceptible to rowhammer? an end-to-end methodology for cloud providers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 712–728, 2020.

[11] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 55–71. IEEE, 2019.

[12] Intel Corporation. *12th Generation Intel Core Processors Datasheet, Volume 1 of 2*. Intel Corporation, Santa Clara, California, 2021. Section: Data Scrambling.

[13] Harish Dattatraya Dixit, Fan (Fred) Lin, Bill Holland, Matt Beadon, Zhengyu Yang, and Sriram Sankar. Optimizing interrupt handling performance for memory failures in large scale data centers. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 193–201, New York, NY, USA, 2020. Association for Computing Machinery.

[14] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized many-sided rowhammer attacks from JavaScript. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1001–1018. USENIX Association, August 2021.

[15] Matthew Dempsky and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges.

[16] Michael Fahr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray Perlner, Arkady Yerukhimovich, and Daniel Apon. When frodo flips: End-to-end key recovery on frodokem via rowhammer. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 979–993, New York, NY, USA, 2022. Association for Computing Machinery.

[17] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P*, May 2018. Pwnie Award Nomination for Most Innovative Research, DCSR Paper Award Runner-up.

[18] Pietro Frigo, Emanuele Vannacc, 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 *2020 IEEE Symposium on Security and Privacy (SP)*, pages 747–762, 2020.

[19] 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 *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261, 2018.

[20] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, page 300–321, Berlin, Heidelberg, 2016. Springer-Verlag.

[21] Venkatesan Guruswami, Atri Rudra, and Madhu Sudan. Essential coding theory. https://cse.buffalo.edu/faculty/atri/courses/coding-theory/book/web-coding-book.pdf.

[22] Harding, Franklin and FitzPatrick, Joe. Tigard. https://github.com/tigard-tools/tigard, 2020.

[23] 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-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1198–1213, New York, NY, USA, 2021. Association for Computing Machinery.

[24] Intel. Intel debug technology. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/intel-debug-technology.html, 2021.

[25] Intel Corporation. Reliability, availability, and serviceability (ras) in 4th gen intel xeon processors. Technical report, Intel Corporation, 2023. Accessed: 2025-01-19.

[26] Intel Corporation. Reliability, availability, and serviceability with xeon processors. Technical paper, Intel Corporation, 2023. Accessed: 2025-01-22.

[27] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. BLACKSMITH: Scalable Rowhammering in the Frequency Domain. In *IEEE S&P '22*, 2022-05. https://comsec.ethz.ch/wp-content/files/blacksmith_sp22.pdf.

[28] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. ZenHammer: Rowhammer attacks on AMD zen-based platforms. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1615–1633, Philadelphia, PA, August 2024. USENIX Association.

[29] JEDEC Solid State Technology Association. Jedec committee jc-45. JEDEC Website, 2023. Accessed: 2025-01-22.

[30] Jonas Juffinger, Lukas Lamster, Andreas Kogler, Maria Eichlseder, Moritz Lipp, and Daniel Gruss. CSI:Rowhammer - cryptographic security and integrity against Rowhammer. In *SP*, pages 1702–1718, 2023.

[31] Ingab Kang, Eojin Lee, and Jung Ho Ahn. CAT-TWO: counter-based adaptive tree, time window optimized for DRAM row-hammer prevention. *IEEE Access*, 8:17366–17377, 2020.

[32] Ingab Kang, Walter Wang, Jason Kim, Stephan van Schaik, Youssef Tobah, Daniel Genkin, Andrew Kwong, and Yuval Yarom. SledgeHammer: Amplifying rowhammer via bank-level parallelism. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1597–1614, Philadelphia, PA, August 2024. USENIX Association.

[33] Michael Jaemin Kim, Jaehyun Park, Yeonhong Park, Wanju Doh, Namhoon Kim, Tae Jun Ham, Jae W. Lee, and Jung Ho Ahn. Mithril: Cooperative row hammer protection on commodity DRAM leveraging managed refresh. In *HPCA*, pages 1156–1169, 2022.

[34] Woongrae Kim. Apparatus and method for performing target refresh operation, August 2024.

[35] 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 *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014.

[36] 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 *31st USENIX Security Symposium (USENIX Security 22)*, pages 3807–3824, Boston, MA, August 2022. USENIX Association.

[37] 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 697–710, Carlsbad, CA, October 2018. USENIX Association.

[38] Xeno Kovah and Corey Kallenberg. Advanced x86: BIOS and system management mode internals flash descriptor. `https://opensecuritytraining.info/IntroBIOS_files/Day2_02_Advanced%20x86%20-%20BIOS%20and%20SMM%20Internals%20-%20Flash%20Descriptor.pdf`, 2023.

[39] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 695–711, 2020.

[40] Eojin Lee, Ingab Kang, Sukhan Lee, G. Edward Suh, and Jung Ho Ahn. TWiCe: preventing row-hammering by exploiting time window counters. In *ISCA*, pages 385–396, 2019.

[41] Jung-Bae Lee. Green memory solution. `https://aod.teletogether.com/sec/20140519/SAMSUNG_Investors_Forum_2014_session_1.pdf#page=15`.

[42] Linux Kernel Developers. Linux kernel source code. Available at `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/edac/skx_base.c?h=v6.3.1`, version 6.3.1, `drivers/edac/skx_base.c`.

[43] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. Nethammer: Inducing rowhammer faults through network requests. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 710–719, 2020.

[44] Kevin Loughlin, Jonah Rosenblum, Stefan Saroiu, Alec Wolman, Dimitrios Skarlatos, and Baris Kasikci. Siloz: Leveraging dram isolation domains to prevent inter-vm rowhammer. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 417–433, New York, NY, USA, 2023. Association for Computing Machinery.

[45] Haocong Luo, Ataberk Olgun, Abdullah Giray Yağlıkçı, Yahya Can Tuğrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. Rowpress: Amplifying read disturbance in modern dram chips. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[46] Michele Marazzi, Flavien Solt, Patrick Jattke, Kubo Takashi, and Kaveh Razavi. REGA: scalable Rowhammer mitigation with refresh-generating activations. In *SP*, pages 1684–1701, 2023.

[47] Micron Inc. DDR4 SDRAM, 4Gb: x4, x8, x16 DDR4 SDRAM Features, 2017.

[48] Onur Mutlu, Ataberk Olgun, and Abdullah Giray Yaglikçi. Fundamentally understanding and solving RowHammer. In *ASP-DAC*, pages 461–468, 2023.

[49] Ataberk Olgun, Hasan Hassan, A. Giray Yağlıkçı, Yahya Can Tuğrul, Lois Orosa, Haocong Luo, Minesh Patel, Oğuz Ergin, and Onur Mutlu. Dram bender: An extensible and versatile fpga-based infrastructure to easily test state-of-the-art dram chips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(12):5098–5112, 2023.

[50] 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 *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1182–1197, New York, NY, USA, 2021. Association for Computing Machinery.

[51] Minesh Patel, Jeremie S. Kim, Taha Shahroodi, Hasan Hassan, and Onur Mutlu. Bit-exact ecc recovery (beer): Determining dram on-die ecc functions by exploiting dram data retention characteristics, 2020.

[52] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, Austin, TX, August 2016. USENIX Association.

[53] Moinuddin Qureshi, Salman Qazi, and Aamer Jaleel. Mint: Securely mitigating rowhammer with a minimalist in-dram tracker. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 899–914, 2024.

[54] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: hammering a needle in the software stack. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 1–18, USA, 2016. USENIX Association.

[55] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[56] William Ryan and Shu Lin. Channel codes classical and modern. *Channel Codes: Classical and Modern*, 01 2009.

[57] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J. Nair. Randomized row-swap: mitigating row hammer by breaking spatial correlation between aggressor and victim rows. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 1056–1069, New York, NY, USA, 2022. Association for Computing Machinery.

[58] Seyed Mohammad Seyedzadeh, Alex K. Jones, and Rami G. Melhem. Mitigating wordline crosstalk using adaptive trees of counters. In *ISCA*, pages 612–623, 2018.

[59] Nikolai Shley. NVRAM device in UEFI-compatible firmware, part four. https://habr.com/en/articles/281901/, Apr 2016.

[60] Chihun Song, Michael Jaemin Kim, Tianchen Wang, Houxiang Ji, Jinghan Huang, Ipoom Jeong, Jaehyun Park, Hwayong Nam, Minbok Wi, Jung Ho Ahn, et al. Tarot: A cxl smartnic-based defense against multi-bit errors by row-hammer attacks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 981–998, 2024.

[61] Cisco Systems. Mitigations available for the dram row hammer vulnerability. Cisco Blogs, 2020. Accessed: 2025-01-22.

[62] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 213–226, Boston, MA, July 2018. USENIX Association.

[63] The Sage Development Team. The Elliptic Curve Method for Integer Factorization (ECM), 2025.

[64] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G. Shin. Go go gadget hammer: Flipping nested pointers for arbitrary data leakage. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1635–1650, Philadelphia, PA, August 2024. USENIX Association.

[65] UEFI Forum, Inc. Unified extensible firmware interface (UEFI) specification. https://uefi.org/sites/default/files/resources/UEFI_Spec_2_10_Aug29.pdf, Aug 2022.

[66] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1675–1689, New York, NY, USA, 2016. Association for Computing Machinery.

[67] Minbok Wi, Jaehyun Park, Seoyoung Ko, Michael Jaemin Kim, Nam Sung Kim, Eojin Lee, and Jung Ho Ahn. Shadow: Preventing row hammer in dram with intra-subarray row shuffling. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 333–346, 2023.

[68] Jeonghyun Woo, Gururaj Saileshwar, and Prashant J. Nair. Scalable and secure row-swap: Efficient and safe row hammer mitigation in memory systems. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 374–389, 2023.

[69] A. Giray Yağlıkçi, 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 *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 345–358, 2021.

[70] Zhi Zhang, Yueqiang Cheng, Minghua Wang, Wei He, Wenhao Wang, Surya Nepal, Yansong Gao, Kang Li, Zhe Wang, and Chenggang Wu. SoftTRR: Protect page tables against rowhammer attacks using software-only target row refresh. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 399–414, Carlsbad, CA, July 2022. USENIX Association.

A BIOS Modification Details

We now describe our BIOS modification process and tool in more detail.

Visualizing Hidden Settings. First, to introspect any hidden settings, we have to understand the UEFI image format itself. On Intel platforms, this image starts with an Intel Flash Descriptor (IFD) followed by one or more partitions, called firmware volumes [38]. Some of these volumes are formatted with the Firmware File System (FFS) which stores a hierarchy of actual drivers and executables to perform platform initialization during boot. Furthermore, some of these executables contain Human Interface Infrastructure (HII) packages containing strings, images, but more importantly Internal Form Representation (IFR) [65], which not only describes the user interface that the UEFI setup should render on screen, but also what options are available for each setting, and how to correctly adjust the variables in the underlying non-volatile RAM (NVRAM) storage.

We implemented a number of parsers that can locate these IFR descriptions and visualize these in a terminal user interface to easily introspect what settings are available. Using our tool, we observe that the SocketSetup executable in the UEFI images for Skylake and Cascade Lake motherboards often contains a Memory Dfx Configuration menu with an ECC Checking setting that allows us to disable the ECC checking mechanism. We conjecture that this menu is actually intended for debugging purposes, as Dfx indicates design for debug, test, manufacturing and/or validation [24]. Similarly, we find that the SocketSetup executable also contains a Memory Configuration menu with the Data Scrambling for DDR4 setting.

Adjusting the NVRAM. In addition to these user interfaces, we also have to persist any of the changed settings to the underlying NVRAM storage, responsible for holding the UEFI settings, on the flash chip. Typically, these settings are either stored in their own firmware volume or a file in one of the firmware file systems. While there are several UEFI implementations available, each with their own NVRAM storage format, we note that most platforms implement AMI Aptio's NVAR format, as AMI Aptio is the most prevalent for desktop and server motherboards [59]. Thus, we have also implemented a parser that allows us to parse AMI's NVAR format to read the current settings and a serializer to modify them. This results in a UEFI image file, which we must flash to the board's flash chip.

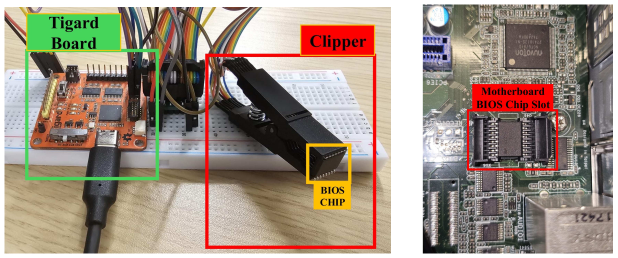


Figure 11: The setup of the Tigard flash programmer hooked up to the flash chip (left). The location of the socket containing the BIOS flash chip on the ASRock Rack EPC621D8A motherboard (right).

Flashing the UEFI Image. While most motherboards allow users to update the UEFI image via the BIOS update process, we have discovered that motherboards typically refuse to flash images edited via our tools, due to failing checksum and signature checks, originally intended to prevent users from flashing corrupted images. Thus, rather than reverse engineering these safety mechanisms, we have instead opted for simpler methods to flash the board's flash chip, bypassing them altogether.

First, many ASRock Rack motherboards typically have a socketed flash chip, allowing end users to directly swap these chips in case of a flashing failure or version upgrade. For such boards, we have successfully used external flash programmers, such as the Tigard multiprotocol board [22], to directly flash these chips with our UEFI software, subsequently installing them into the board's socket. See Figure 11.

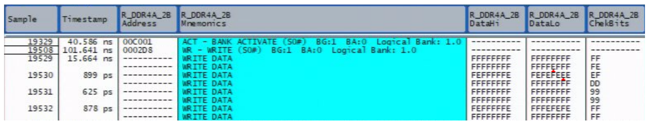
Moreover, high-end gaming motherboards supporting Xeon processors (such as Asus ROG Dominus Extreme or the EVGA SR-3 dark) often allow users to update the system's BIOS without the CPU present, aiming to resolve compatibility issues across Intel's CPU ecosystem. This process involves connecting a drive containing the required UEFI image into a dedicated USB port, followed by pressing a specific button sequence on the motherboard with the CPU removed. Empirically attempting this process we have discovered that both Asus and EVGA seem to skip all checksum and signature checks, directly flashing the image into the system's flash chip.

B 2-bit Correctable Bypass Template Example

We now give an example of obtaining Rowhammer bit flips using a 2-bit correctable bypass template with logic analyzer traces. We choose the 4-bit bypass template: bits 76, 116, 132, 180 as an example. The four bits in this template lie in chips 3, 13, 1, 13 (respectively). Thus, bit 76 and bit 132 form a 2-bit correctable bypass template. The experiment has the following steps.

We first write to the target cache line with all bits set to 1 except for bits 76 and 132 (indicated with a red triangle), which are set to 0 (Figure 12).

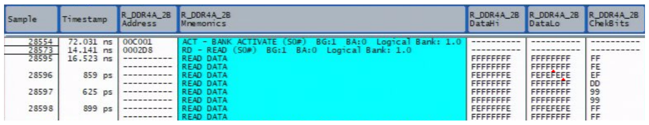
Next, we hammer the cache line to flip both bits (76



```
Sample  Timestamp    R_DDR4A_2B  R_DDR4A_2B                                          R_DDR4A_2B  R_DDR4A_2B  R_DDR4A_2B
                     Address     Mnemonics                                           DataHi      DataLo      ChekBits
19329    40.586 ns   00C001      ACT - BANK ACTIVATE (S0#)  BG:1  BA:0  Logical Bank: 1.0  ----------  ----------  ----------
19508   101.641 ns   0002D8      WR - WRITE (S0#)  BG:1  BA:0  Logical Bank: 1.0       ----------  ----------  ----------
19529    15.664 ns   ----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    FF
                     ----------  WRITE DATA                                          FFFFFFFF    FFFFEFFF    FE
19530       899 ps   ----------  WRITE DATA                                          FEFFFFFE    FEFEFEEE    EF
                     ----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    DD
19531       625 ps   ----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    99
                     ----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    99
19532       878 ps   ----------  WRITE DATA                                          FEFFFFFE    FFFEFEFE    FF
                     ----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    FF
```

Figure 12: Logic Analyzer Trace of Writing Unflipped Data into Cache Line

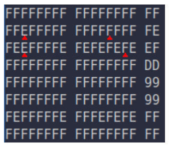
and 132) from 0 to 1. Figure 13 presents the logic analyzer trace of reading flipped data from memory. The half bytes containing bits 76 and 132 turns from `E` to `F` with all other bytes unchanged, indicating only bits 76 and 132 flipped.



```
Sample  Timestamp    R_DDR4A_2B  R_DDR4A_2B                                          R_DDR4A_2B  R_DDR4A_2B  R_DDR4A_2B
                     Address     Mnemonics                                           DataHi      DataLo      ChekBits
28554    72.031 ns   00C001      ACT - BANK ACTIVATE (S0#)  BG:1  BA:0  Logical Bank: 1.0  ----------  ----------  ----------
28573    14.141 ns   0002D8      RD - READ (S0#)  BG:1  BA:0  Logical Bank: 1.0        ----------  ----------  ----------
28595    16.523 ns   ----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
                     ----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FE
28596       859 ps   ----------  READ DATA                                           FEFFFFFE    FEFEFEFE    EF
                     ----------  READ DATA                                           FFFFFFFF    FFFFFFFF    DD
28597       625 ps   ----------  READ DATA                                           FFFFFFFF    FFFFFFFF    99
                     ----------  READ DATA                                           FFFFFFFF    FFFFFFFF    99
28598       899 ps   ----------  READ DATA                                           FEFFFFFE    FFFEFEFE    FF
                     ----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
```

Figure 13: Logic Analyzer Trace of Reading Flipped Data from Cache Line

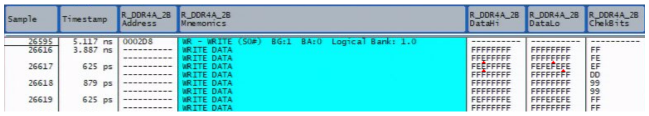
Next, Figure 14 presents the value our Rowhammer code observes as returned from the machine's memory. Comparing the cache line data between Figure 12 and Figure 14, fours bits are different: bits 76 and 132, which are flipped from 0 to 1 by Rowhammer, and bits 116 and 180 (also indicated with a red triangle), which are "flipped" from 1 to 0 by ECC correction.



```
FFFFFFFF FFFFFFFF FF
FFEFFFFF FFFFFFFF FE
FEEFFFFE FEFEFEFE EF
FFFFFFFF FFFFFFFF DD
FFFFFFFF FFFFFFFF 99
FFFFFFFF FFFFFFFF 99
FEFFFFFE FFFEFEFE FF
FFFFFFFF FFFFFFFF FF
```

Figure 14: Cache Line Data Seen by Rowhammer Attack Code

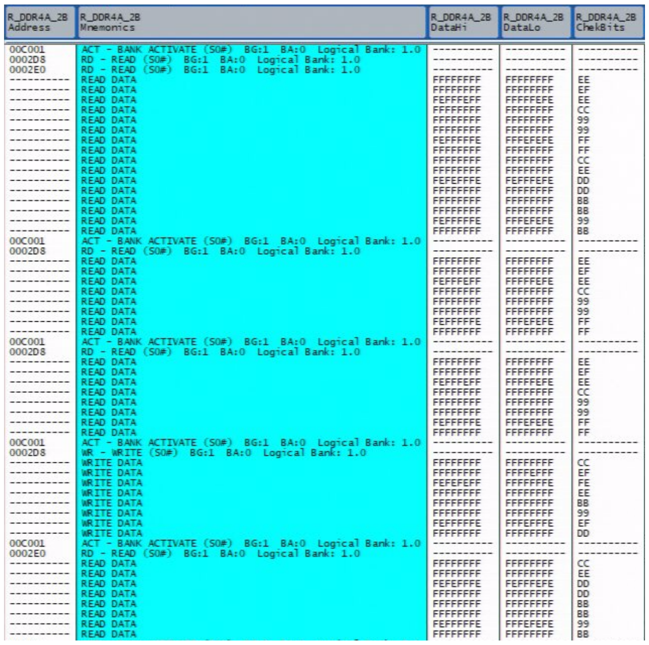
Finally, we confirm that these bit flips are permanent by writing back the cache line with data read in the last step and observe that the bit flips still exist, as shown in Figure 15.



```
Sample  Timestamp    R_DDR4A_2B  R_DDR4A_2B                                          R_DDR4A_2B  R_DDR4A_2B  R_DDR4A_2B
                     Address     Mnemonics                                           DataHi      DataLo      ChekBits
26595     5.117 ns   0002D8      WR - WRITE (S0#)  BG:1  BA:0  Logical Bank: 1.0       ----------  ----------  ----------
26616     3.887 ns   ----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    FF
                     ----------  WRITE DATA                                          FFEFFFFF    FFFFFFFF    FE
26617       625 ps   ----------  WRITE DATA                                          FEEFFFFE    FEFEFEFE    EF
                     ----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    DD
26618       879 ps   ----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    99
                     ----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    99
26619       625 ps   ----------  WRITE DATA                                          FEFFFFFE    FFFEFEFE    FF
                     ----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    FF
```

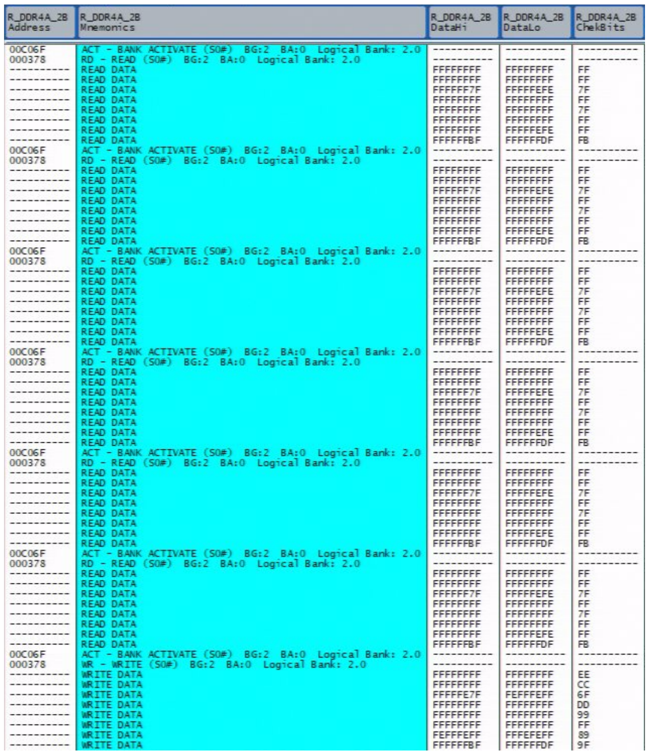
Figure 15: Logic Analyzer Trace of Writing Flipped Data into Cache Line

C Logic Analyzer Output when Accessing Flippy Cache Lines



```
R_DDR4A_2B  R_DDR4A_2B                                          R_DDR4A_2B  R_DDR4A_2B  R_DDR4A_2B
Address     Mnemonics                                           DataHi      DataLo      ChekBits
00C001      ACT - BANK ACTIVATE (S0#)  BG:1  BA:0  Logical Bank: 1.0  ----------  ----------  ----------
0002D8      RD - READ (S0#)  BG:1  BA:0  Logical Bank: 1.0        ----------  ----------  ----------
0002E0      RD - READ (S0#)  BG:1  BA:0  Logical Bank: 1.0        ----------  ----------  ----------
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    EE
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    EF
----------  READ DATA                                           FEFFFEFF    FFFFFEFE    EE
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    CC
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    99
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    99
----------  READ DATA                                           FEFFFFFE    FFFEFEFE    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    CC
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    EE
----------  READ DATA                                           FEFEFFFE    FEFFFEFE    DD
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    DD
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    BB
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    BB
----------  READ DATA                                           FEFFFFFE    FFFEFEFE    99
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    BB
00C001      ACT - BANK ACTIVATE (S0#)  BG:1  BA:0  Logical Bank: 1.0  ----------  ----------  ----------
0002D8      RD - READ (S0#)  BG:1  BA:0  Logical Bank: 1.0        ----------  ----------  ----------
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    EE
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    EF
----------  READ DATA                                           FEFFFEFF    FFFFFEFE    EE
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    CC
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    99
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    99
----------  READ DATA                                           FEFFFFFE    FFFEFEFE    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
00C001      ACT - BANK ACTIVATE (S0#)  BG:1  BA:0  Logical Bank: 1.0  ----------  ----------  ----------
0002D8      RD - READ (S0#)  BG:1  BA:0  Logical Bank: 1.0        ----------  ----------  ----------
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    EE
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    EF
----------  READ DATA                                           FEFFFEFF    FFFFFEFE    EE
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    CC
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    99
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    99
----------  READ DATA                                           FEFFFFFE    FFFEFEFE    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
00C001      ACT - BANK ACTIVATE (S0#)  BG:1  BA:0  Logical Bank: 1.0  ----------  ----------  ----------
0002D8      WR - WRITE (S0#)  BG:1  BA:0  Logical Bank: 1.0       ----------  ----------  ----------
----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    CC
----------  WRITE DATA                                          FFFFFFFF    FFFFEFFF    EF
----------  WRITE DATA                                          FEFEFEFF    FFFFFFFE    FE
----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    EE
----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    BB
----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    99
----------  WRITE DATA                                          FEFFFFFE    FFFEFFFE    EF
----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    DD
00C001      ACT - BANK ACTIVATE (S0#)  BG:1  BA:0  Logical Bank: 1.0  ----------  ----------  ----------
0002E0      RD - READ (S0#)  BG:1  BA:0  Logical Bank: 1.0        ----------  ----------  ----------
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    CC
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    EE
----------  READ DATA                                           FEFEFFFE    FEFFFEFE    DD
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    DD
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    BB
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    BB
----------  READ DATA                                           FEFFFFFE    FFFEFEFE    99
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    BB
```

Figure 16: Correction Replay Requests Pattern



```
R_DDR4A_2B  R_DDR4A_2B                                          R_DDR4A_2B  R_DDR4A_2B  R_DDR4A_2B
Address     Mnemonics                                           DataHi      DataLo      ChekBits
00C06F      ACT - BANK ACTIVATE (S0#)  BG:2  BA:0  Logical Bank: 2.0  ----------  ----------  ----------
000378      RD - READ (S0#)  BG:2  BA:0  Logical Bank: 2.0        ----------  ----------  ----------
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFF7F    FFFFFEFE    7F
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    7F
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFEFE    FF
----------  READ DATA                                           FFFFFFBF    FFFFFFDF    FB
00C06F      ACT - BANK ACTIVATE (S0#)  BG:2  BA:0  Logical Bank: 2.0  ----------  ----------  ----------
000378      RD - READ (S0#)  BG:2  BA:0  Logical Bank: 2.0        ----------  ----------  ----------
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFF7F    FFFFFEFE    7F
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    7F
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFEFE    FF
----------  READ DATA                                           FFFFFFBF    FFFFFFDF    FB
00C06F      ACT - BANK ACTIVATE (S0#)  BG:2  BA:0  Logical Bank: 2.0  ----------  ----------  ----------
000378      RD - READ (S0#)  BG:2  BA:0  Logical Bank: 2.0        ----------  ----------  ----------
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFF7F    FFFFFEFE    7F
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    7F
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFEFE    FF
----------  READ DATA                                           FFFFFFBF    FFFFFFDF    FB
00C06F      ACT - BANK ACTIVATE (S0#)  BG:2  BA:0  Logical Bank: 2.0  ----------  ----------  ----------
000378      RD - READ (S0#)  BG:2  BA:0  Logical Bank: 2.0        ----------  ----------  ----------
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFF7F    FFFFFEFE    7F
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    7F
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFEFE    FF
----------  READ DATA                                           FFFFFFBF    FFFFFFDF    FB
00C06F      ACT - BANK ACTIVATE (S0#)  BG:2  BA:0  Logical Bank: 2.0  ----------  ----------  ----------
000378      RD - READ (S0#)  BG:2  BA:0  Logical Bank: 2.0        ----------  ----------  ----------
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFF7F    FFFFFEFE    7F
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    7F
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFEFE    FF
----------  READ DATA                                           FFFFFFBF    FFFFFFDF    FB
00C06F      ACT - BANK ACTIVATE (S0#)  BG:2  BA:0  Logical Bank: 2.0  ----------  ----------  ----------
000378      RD - READ (S0#)  BG:2  BA:0  Logical Bank: 2.0        ----------  ----------  ----------
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFF7F    FFFFFEFE    7F
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    7F
----------  READ DATA                                           FFFFFFFF    FFFFFFFF    FF
----------  READ DATA                                           FFFFFFFF    FFFFFEFE    FF
----------  READ DATA                                           FFFFFFBF    FFFFFFDF    FB
00C06F      ACT - BANK ACTIVATE (S0#)  BG:2  BA:0  Logical Bank: 2.0  ----------  ----------  ----------
000378      WR - WRITE (S0#)  BG:2  BA:0  Logical Bank: 2.0       ----------  ----------  ----------
----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    EE
----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    CC
----------  WRITE DATA                                          FFFFFE7F    FEFFFEFF    6F
----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    DD
----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    99
----------  WRITE DATA                                          FFFFFFFF    FFFFFFFF    FF
----------  WRITE DATA                                          FEFFFEFF    FFFEFEFF    89
----------  WRITE DATA                                          FFFFFFBF    FFFFFFDF    9F
```

Figure 17: Crash Replay Requests Pattern