# Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks

Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, Herbert Bos
Vrije Universiteit Amsterdam

***Abstract*—Given the increasing impact of Rowhammer, and the dearth of adequate other hardware defenses, many in the security community have pinned their hopes on error-correcting code (ECC) memory as one of the few practical defenses against Rowhammer attacks. Specifically, the expectation is that the ECC algorithm will correct or detect any bits they manage to flip in memory in real-world settings. However, the extent to which ECC really protects against Rowhammer is an open research question, due to two key challenges. First, the details of the ECC implementations in commodity systems are not known. Second, existing Rowhammer exploitation techniques cannot yield reliable attacks in presence of ECC memory.**

**In this paper, we address both challenges and provide concrete evidence of the susceptibility of ECC memory to Rowhammer attacks. To address the first challenge, we describe a novel approach that combines a custom-made hardware probe, Rowhammer bit flips, and a cold-boot attack to reverse engineer ECC functions on commodity AMD and Intel processors. To address the second challenge, we present *ECCploit*, a new Rowhammer attack based on composable, data-controlled bit flips and a novel side channel in the ECC memory controller. We show that, while ECC memory does reduce the attack surface for Rowhammer, *ECCploit* still allows an attacker to mount reliable Rowhammer attacks against vulnerable ECC memory on a variety of systems and configurations. In addition, we show that, despite the non-trivial constraints imposed by ECC, *ECCploit* can still be powerful in practice and mimic the behavior of prior Rowhammer exploits.**

## I. Introduction

Originally designed to handle accidental and rare occurrences of data corruption in DRAM chips due to cosmic rays or electrical interference [1]–[4], Error-Correcting Code (ECC) memory is also perceived as one of the few effective bulwarks against Rowhammer attacks [5]. These attacks exploit a vulnerability in DRAM hardware that allows attackers to flip bits in memory that should not be accessible to them [6]. Since the discovery of the Rowhammer vulnerability in 2014, the security community has devised ever more worrying exploitation techniques. Starting with fairly simple, probabilistic corruption of page tables from native x86 code [6], researchers have extended the Rowhammer attack surface across all sorts of computing systems (including PCs [6]–[8], clouds [9], [10], and mobile devices [11], [12]), launching exploits from different environments (such as native C binaries [6] and browser-based JavaScript [7], [8], [12]), using a variety of processors (notably x86 [6], ARM [11], and GPU [12]), against a variety of targets (page tables [6], [11], encryption keys [10], object pointers [7], repository URLs [10], and opcodes [13]), in different types of memory (DDR3 [6] and DDR4 [11]). As a result, Rowhammer has grown into a major security concern in real-world settings.

Not surprisingly, there has been much speculation on the effectiveness of ECC memory in deterring real-world Rowhammer attacks [5], [6], [10], [11], [13], often hypothesizing ECC memory would reduce Rowhammer to a denial-of-service vulnerability [6], [13]. As a result, practical Rowhammer exploits have thus far only targeted non-ECC-equipped platforms. However, once the uncommon case, ECC-equipped platforms are now on the rise, from large cloud providers (e.g., Amazon EC2 [14]) to high-end consumer platforms [15]. In addition, ECC memory is increasingly deployed on low-power platforms such as mobile and IoT devices to drop the DRAM refresh rate below "*safe*" values and save power [16], [17]. It has therefore become important to quantitatively assess the effectiveness of ECC memory as a Rowhammer mitigation.

ECC is able to correct $n$ bit errors (with $n \geq 1$) and detect cases where more than $n$ bits have flipped, up to some maximum. For this purpose, ECC adds redundant ECC bits to every data word that "check" the other bits. The combination of the data bits and the ECC bits is known as a code word. ECC ensures that if any bit in a valid code word changes, it is no longer a valid code word. Thus, in a chipset with ECC memory, attackers may still use Rowhammer to cause a bit flip in physical memory, but the ECC mechanism immediately catches it on the first subsequent access, and flips it back. Since the probability of flipping exactly the right set of bits to turn one valid code word into a new valid code word using Rowhammer is extremely low, state-of-the-art Rowhammer attacks either fail, or trigger uncorrectable errors, leading to denial of service. Better still, modern processors apply additional memory reliability measures such as data masking (scrambling) to turn the data that the CPU really writes to main memory into pseudo-random patterns—making it even harder for an attacker to flip the right bits. The research question in this paper is whether the assumption is true that Rowhammer attacks are really not practical on ECC memory. In particular, we examine the strength of ECC in several modern chipsets and show that this is not the case: reliable attacks in real-world settings are harder, but still possible.

To determine the exact protection offered by ECC, we must know the details of the ECC algorithms. Unfortunately, vendors such as Intel and AMD do not release these details. Moreover, to the best of our knowledge, no prior work has managed to reverse engineer the ECC functions. Important contributions of this paper are therefore the recovered ECC computation for popular chipsets and a detailed description of the *techniques* to reverse engineer other ECC algorithms.

A major challenge in examining a DRAM's susceptibility to

Rowhammer on ECC memory, both for us and for attackers, is detecting the bit flips in the first place. How do we even know that we flipped a bit using Rowhammer, if the hardware automatically flips it back when we try to read it? Phrased differently, *observing* ECC errors is hard, precisely because the hardware is designed to hide them. To solve this problem, we describe a novel side channel that allows us to observe bit flips even when the error correction functionality flips them back when we read the corresponding memory location.

Armed with the ability to detect (correctable) bit flips and knowledge of a fully reverse engineered ECC algorithm, another challenge towards reliable attacks is to surgically trigger the "*right*" combination of bit flips in a single code word to bypass ECC. An invalid combination may be corrected or, worse, trigger uncorrectable errors and crash the system. To address this challenge, we develop a new Rowhammer attack technique based on composable, data-controlled bit flips. The key insight is that Rowhammer bit flips are data-dependent and, if we study how specific data patterns determine the triggering of individual bit flips, we can then reliably isolate/-compose multiple bit flips by placing the "*right*" data patterns in memory. Our attack, termed *ECCploit*, relies on such insight to incrementally find an exploitable combination of bit flips in a code word and bypass ECC memory.

Given the need to bypass ECC checks, such exploits are more constrained compared to existing Rowhammer attacks. For this reason, we reproduce known end-to-end exploits on ECC memory and analyze the attack surface, that is the probability of finding the bit flip patterns that bypass the ECC checks for these exploits. While we do find that ECC checks significantly reduce the Rowhammer attack surface, we show *ECCploit* can still be used to successfully mount Rowhammer exploits in practical settings. In addition, while we evaluate *ECCploit* in an ideal scenario where the system is configured properly to handle ECC errors (i.e., the worst case for attackers), we find that in many systems this is not the case. For example, while we expect a crash in case of uncorrectable errors, sometimes the system does not immediately crash, allowing for much simpler exploitation with ECC memory.

**Contributions.** Our main contribution is showing that ECC memory, even when combined with data scrambling, does not offer adequate protection against Rowhammer. We do so by:

- Describing a novel reverse engineering technique for recovering ECC implementations on commodity hardware.
- Identifying the ECC implementation on several popular chipsets and investigating how commodity systems respond to ECC exceptions.
- Presenting *ECCploit*, a new reliable Rowhammer attack that leverages undocumented ECC implementation details, a novel side channel in the memory controller, and composable, data-controlled bit flips. We show *ECCploit* can be used for practical privilege escalation attacks by reproducing existing exploits on ECC-based systems.

## II. Background

In the following, we provide a high-level description of the DRAM architecture, the Rowhammer vulnerability, and ECC properties we rely on for our *ECCploit* attack.

### A. DRAM Organization

**Architecture.** DRAM uses one of the last parallel buses in modern systems. In a common setup, 64 lines connect a Dual Inline Memory Module (DIMM) to the CPU forming a 64-bit wide data bus. Multiple chips inside a DIMM form the 64 bits of data every time DRAM is accessed. For example, with 8-bit wide chips (i.e., 8x), eight chips are involved in each DRAM read or write operation. Each chip consists of multiple banks. Multiple rows of DRAM cells are stacked together to form each of these banks. Cells are the smallest unit of storage in DRAM and are built using a capacitor and an access transistor. The amount of charge stored in the capacitors denotes the value of one or zero depending on the charge level.

**Accessing DRAM.** The smallest unit of access inside DRAM is a row. To access DRAM, the same bank is selected in all chips and the data from the selected row is moved to a cache called *row buffer* before being transmitted on the bus (i.e., row activation). Subsequent accesses to addresses that map to the same row will be served from the row buffer (i.e., row hit) and addresses that map to a different row require writing the contents of the row buffer back to the cells and moving the target row into the row buffer (i.e., row miss).

**Refresh.** Given that DRAM cells are built from capacitors, they lose charge and hence their value over time. To restore the charge, the cells need to be recharged, a process called *DRAM refreshing*. This process is orchestrated by the memory controller, which is responsible for periodically refreshing individual DRAM cells at a predetermined *refresh rate*. The refresh rate is determined based on the expected amount of charge leakage (e.g., dependent on the manufacturing process), and the implementation constraints (e.g., presence of ECC).

**Supporting ECC.** Cosmic rays and other external events can cause corruption in DRAM cells by changing the charge levels in the capacitors [1]–[4]. To address this problem, ECC memory stores extra parity bits (also known as control bits) next to the data bits to correct these corruptions. DRAMs with ECC support come with additional chips. The memory bus is then enlarged with eight additional lines (i.e., 72-bit wide bus) to transfer the control bits next to their data bits [18]–[21].

### B. Rowhammer

As transistors become smaller, their reliability starts to suffer. Kim et al. [5] showed that frequent activations of the same row cause bits to flip in adjacent rows *without accessing them*. The reason is the increased amount of charge leakage from DRAM cell capacitors (built from transistors) due to parasitic coupling and passing gate effects. Termed the Rowhammer vulnerability, soon a plethora of attacks abused a single bit flip to compromise desktops, laptops, and mobile phones [6]–[8], [10]–[13]. Such attacks come in

different variants, double-sided, single-sided, or one-location Rowhammer [13]—depending on the *aggressor* row(s) used by the attacker to corrupt the *victim* row—and exploit the fact that Rowhammer bit flips are *observable* and *reproducible*.

All of these attacks have been executed on systems without ECC and, while there has been speculation on the possibility of bypassing simple ECC functions since the original Rowhammer paper [5], an end-to-end Rowhammer attack on ECC memory on a real system has never been attempted for two main reasons. First, ECC implementations on modern systems are often undocumented and go beyond the simple SECDED ECC which we describe shortly. Second, it is challenging to trigger Rowhammer corruptions without triggering corrections or crashes on a system protected by ECC. Before further discussing these challenges, we need to understand how ECC is currently implemented on modern commodity systems.

### C. ECC in DRAM

In current designs, the only ECC-aware unit inside the processor is its memory controller. Assuming the CPU wants to write a message of $k$ bits, the memory controller appends $r$ bits of redundant information for error correction and detection and stores a *codeword* of $n = k+r$ bits in DRAM. In practice, CPU vendors choose $k$ to be a multiple of a memory word (64 bits) and $r = \frac{k}{8}$. In fact, the ratio of redundant to data bits (1-to-8) is embedded in the current Double Data Rate (DDR) standards (DDR3 [18] and DDR4 [19]), memory bus standards with 8 control bits and 64 data bits. For manufacturing simplicity, the same type of memory chips is used to store both the data bits and as well the control bits. Concretely, one can identify DIMMs that provide ECC by counting the number of memory chips on the module.

**Block codes.** DRAM ECC uses *linear block codes* for calculating the $r$ bits [22]. Differences in the size of $r$ bits and their actual value provide different trade-offs in terms of reliability and performance. There are two types of linear block error correcting codes, *binary* and *non-binary* codes. A binary code is denoted as $(n, k)$ and has a granularity of a single bit while non-binary codes treat multiple bits as a single *symbol*. A particular case of binary code, the $(7, 4)$ code, was first studied and generalized by Richard Hamming [23] and represents an improvement from the simple parity checking as it offers error correcting capabilities with 3 parity bits for 4 bits of data.

**SECDED.** The Hamming Distance (HD) between any codeword ($d_{min}$) of the $(7, 4)$ code is at least 3, meaning that it can detect up to 2 bit errors and correct a single detectable error. However, distinguishing between a message that has a corruption of one bit and a message that has a corruption of two bits is not possible. The implication is that some 2-bit faulty messages will falsely be "corrected". An *extended* Hamming code adds an extra parity bit to solve this problem and serves as the basis of the design of ECC used in modern memory systems as it provides single error correction and double-bit error detection (SECDED) [22].

**Chipkill.** High-available systems need to detect multiple adjacent bit errors. This requirement of the error correcting capabilities is known as the *chipkill* [21] functionality. BCH codes [24] have the desired property of precise control of the error guarantees. The Reed–Solomon (RS) codes [25] are a class of effective and easy-to-construct non-binary codes which can be viewed as particular BCH codes. The commonly deployed Chipkill implementation, based on BCH/RS codes [26], provides double-chip error detect and single-chip error correct (SCDCD). Note that Chipkill can correct bit errors up to the size of the symbol, which is often chosen to be the number of bits in a chip. As a result, even if the system loses an entire chip, it can still continue operation.

More generally, a linear block error detecting and correcting code with a $d_{min}$, can *detect* $d_{min} - 1$ errors and *correct* $\lfloor (d_{min} - 1)/2 \rfloor$ errors. Similarly, an RS code that can correct $t$ symbols has a HD of $2t + 1$ and uses $2t$ redundant error correcting symbols. As we shall see in Section V-F, our setups use a version of RS codes.

**ECC functions.** For simplicity and compatibility with non-ECC DIMMs [22], it is desirable for the memory controller to store the control bits and the data in distinct memory chips. From a theoretical perspective, this requirement maps over the *systematic encoding procedure*, in which the message is always a prefix in the codeword.

To encode a message $d = (d_1, d_2, \cdots, d_k)$, where $d_i$ represents a symbol from the alphabet (e.g., a bit), the encoder performs a multiplication with a generator matrix $G$, i.e. $v = d \cdot G$, where $v$ is the encoded message (data). For the practical systematic encoding procedure, $G = [I_k|P]$, where $I_k$ is the identity matrix of size $k$, and $P$ is the *parity check matrix* which has $k$ rows and $r$ columns:

$$\begin{aligned} v &= d \cdot G \\ &= d \cdot [I_k|P] \\ &= d \cdot ([I_k|0_{k,r}] + [0_{k,k}|P]) \text{ where } 0_{m,n} \text{ is a zero matrix} \\ &= d \cdot [I_k|0_{k,r}] + d \cdot [0_{k,k}|P] \end{aligned} \tag{1}$$

Let $ECC(d)$ be the last $r$ bits from the $d \cdot [0_{k,k}|P]$ product, which we loosely call *the ECC bits for data* $d$. Using the Kronecker function ($\delta_{i,j} = 1$ if $i = j$ and $\delta_{i,j} = 0$ if $i \neq j$), we can rewrite the ECC bits as:

$$\begin{aligned} ECC(d) &= \left( \sum_{i=1}^{k} d_i \cdot [\delta_{1,i}, \delta_{2,i}, \cdots, \delta_{k,i}] \right) \cdot P \\ ECC(d) &= \sum_{i=1}^{k} d_i \cdot [P_{i,1}, P_{i,2}, \cdots, P_{i,r}] \end{aligned} \tag{2}$$

where $P_{i,j}$ represents the value (0 or 1) from the parity check matrix with coordinates row $i$ and column $j$. Each row of the parity check matrix can be expressed as an $r$ bit number called *parity value*. Parity check matrices are not disclosed by processor manufacturers. We devise techniques for obtaining this information on various systems in Section V. Once we have the parity check matrix, we can predict ECC values for arbitrary data. On top of ECC, some systems further scramble data before sending them on the memory bus, complicating the reverse engineering of parity check matrices.

TABLE I: Target systems.

| ID | Manufacturer | CPU model | Microarchitecture |
|---|---|---|---|
| AMD-1 | AMD | Opteron 6376 | Bulldozer (15h) |
| Intel-1 | Intel | Xeon E3-1270 v3 | Haswell |
| Intel-2 | Intel | Xeon E5-2650 v1 | Sandy Bridge |
| Intel-3 | Intel | Xeon E5-2620 v1 | Sandy Bridge |

## III. Threat model

We assume computer systems protected with ECC memory where bit flips are detected and/or corrected in the memory controller. This is common in clouds, high-end workstations, and low-power devices. We further assume the memory chips to be affected by the Rowhammer vulnerability [5]. In addition, we assume that the attacker does not have access to ECC exceptions as these are often exposed to privileged software. Thus the attack can be carried by a non-privileged local user. We assume that the attacker can learn the CPU model and the memory technology. This is trivial to satisfy as access to `/proc/cpuinfo` is unrestricted and cloud providers' public documentation usually contains a description of the underlying hardware [27], [28]. Similar to existing Rowhammer attacks, the attackers' aim is to reliably compromise co-located virtual machines [10], [29] or escalate their privilege by executing unprivileged and/or sandboxed code on the target machine [6]–[8], [11]–[13].

## IV. Summary of Challenges

To exploit a system protected with ECC memory using Rowhammer, the attacker first needs to find the ECC algorithm implemented in the memory controller of the target system's processor. Given the knowledge of the ECC function, the attacker then needs to safely compose enough bit flips to trigger a Rowhammer corruption that is not detected (and corrected) by the ECC algorithm—without triggering uncorrectable errors that may crash the system. These corruptions are different than normal Rowhammer corruptions given that they flip multiple bits at the same time. Because the probability of bits to be in the "flips-from" state *decreases* as the number of bits that flip *increases*, it becomes challenging to exploit such constrained bit flips to compromise a system. In summary, to achieve successful and reliable end-to-end exploitation, we need to address the following challenges:

**[$\mathcal{C}_1$]** How to reverse engineer unknown ECC functions on commodity processors?

**[$\mathcal{C}_2$]** How to trigger Rowhammer corruptions on ECC memory without crashing the system?

**[$\mathcal{C}_3$]** How to exploit the system given that Rowhammer-based ECC corruptions corrupt multiple bits at the same time?

We address **[$\mathcal{C}_1$]** in Section V, **[$\mathcal{C}_2$]** in Section VI, and **[$\mathcal{C}_3$]** in Section VI-B and in Section VII.

## V. Challenge $\mathcal{C}_1$: Reverse Engineering ECC

To get a rough idea of the ECC functions used by CPU manufacturers, we first consulted their patents and the CPUs' public documentation. Unfortunately, these were neither complete nor fully accurate, so additional techniques were necessary. As we shall see, the coding theory behind our attacks is quite involved, so we first provide the intuition.

Whenever an ECC system writes a value in memory, it will also write some ECC bits. For instance, some ChipKill implementations write 4 ECC nibbles (for a total of 16 bits) for every 128 bits of data. The exact calculation of the ECC nibbles is not important at this point, but the first ECC nibble will use one set of data nibbles, the second one a slightly different set, and so on. Upon accessing this value in memory at a later stage, it will calculate the ECC nibbles again and XOR them with the ECC nibbles in memory. The result is known as a *syndrome*. If the syndrome is non zero, there must have been an error. By looking at which syndromes indicate an error, ChipKill can locate the faulty nibble and correct it.

As we shall see, the calculation of the syndromes in mathematical terms involves a fairly complicated multiplication of the transposed and extended parity check matrix with the error pattern, but in practice the multiplication matrix is precomputed and stored as a table, while the multiplications and additions are simply AND and XOR operations (as shown above). The point is that if we have the syndromes for known error patterns, we can also perform the inverse operation and obtain the parity check matrix—and hence the ECC function.

To this end, we artificially injected single bit errors in memory to see what happens and deduce what the syndrome must have been, and also performed cold boot attacks to recover the ECC bits as generated by one machine on another machine. We detail these techniques after providing a theoretical foundation for the attacks. To our knowledge, we are the first to reverse engineer the ECC functions of common CPUs (Table I).

### A. Theoretical foundation

Both Hamming and BCH codes are *polynomial codes*. Polynomial codes can use *exclusive-or* instead of addition and *and* instead of multiplication in the Galois Field (GF), simplifying their implementation in hardware.

**Proposition 1.** *We can recover the complete ECC function by finding the ECC value for every ECC-word with exactly one data bit asserted.*

Each row of the parity check matrix, can be expressed as a $r$ bit number called *parity value*. Considering Equation 2, the ECC value for a data word ($d$) that has bits asserted on positions $s$, can be expressed as an exclusive-or operation between the parity value of each data word ($d'$) with a single $d'_i$ asserted ($\forall i \in \{s|d_s = 1\}$). ■

To decode and correct errors of a received codeword $v' = (v_1, v_2, \cdots, v_{k+r})$, linear codes use an efficient technique called *syndrome decoding*. The syndrome is computed as $S(v') = v' \cdot H^T$ where $H = [-P^T|I_{k+r}]$ for the systematic encoding and $S$ has dimensions $(1, r)$. When no error occur in the transmission ($v' = v$) then $S(v') = d \cdot [I_k|P] \cdot [-P^T|I_{k+r}]^T \Rightarrow S(v') = 0$.

**Proposition 2.** *The ECC value of a data word with a single bit asserted on a specific position is equal to the syndrome obtained when that specific bit is faulted.*

In the presence of an error $e = (e_1, e_2, \cdots, e_{k+r})$ with $e \neq 0_{1,k+r}$, $v' = v + e$, and because $S(v) = 0$, we can rewrite the syndrome as:

$$\begin{aligned} S(v+e) &= (v+e) \cdot H^T \\ &= v \cdot H^T + e \cdot H^T \\ &= S(v) + e \cdot H^T \\ &= e \cdot H^T \\ &= e \cdot ([-P^T | I_r]^T) \\ &= e \cdot ([-P^T | 0_{r,r}]^T + [0^T_{k,r} | I_r]^T) \end{aligned} \tag{3}$$

We use the notation $SYND(v') = (e_1, e_2, \cdots, e_k) \cdot -P$, to refer to the *syndrome obtained when errors are inserted only in the data bits*. Using the Kronecker function we can rewrite the syndrome obtained under faults as:

$$\begin{aligned} SYND(v') &= -\left( \textstyle\sum_{i=1}^{k} e_i \cdot [\delta_{1,i}, \delta_{2,i}, \cdots, \delta_{k,i}] \right) \cdot P \\ SYND(v') &= -\textstyle\sum_{i=1}^{k} e_i \cdot [P_{i,1}, P_{i,2}, \cdots, P_{i,r}] \end{aligned} \tag{4}$$

As the operations are performed on a binary GF and the code is cyclic, the "$-$" sign has no meaning. Therefore by choosing $e_i = d_i$ in Equation 2 and 4, we obtain the proof below. For simplicity, we choose $e_i$ such that at most one bit is flipped.

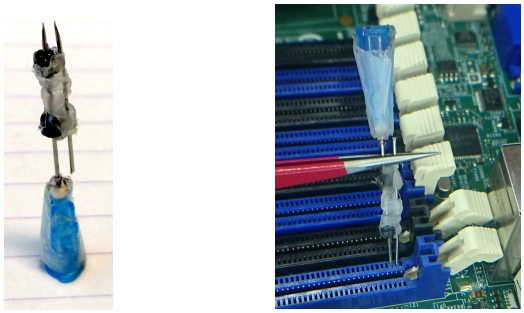
$$\forall v : ECC(v) = SYND(v). \blacksquare \tag{5}$$

Assuming the attacker has access to the same machine as the victim, we show how an attacker can use Proposition 1 and 2 to inject faults and perform cold boot attacks to reverse engineer the contents of the parity matrix and the order in which the output data is mapped to the DRAM bus lines. Note that the attacker needs to perform this process only once and reuse the recovered information when attacking victim machines that use the same CPU model. The CPU model information on the victim machine is available through sources such as `cpuid`.

### B. Fault Injection

In this section, we describe how to obtain all syndromes (and thus the ECC function) by observing only the syndromes for specific errors that we inject ourselves in a controlled way, where exactly one bit is flipped. For now, we assume that when the ECC engine corrects an error, the attacker can also read the syndrome for that specific error. We will show how we relax this assumption later. The crux of our attack is that if we repeatedly flip a single bit at every possible bit position of an ECC word, and obtain all the corresponding syndromes, the recovery of the ECC function is trivial (Equation 5). For example, the ECC value of an ECC word where bit $i$ and $j$ are asserted is the result of the `XOR` operation between the syndrome when a 1-to-0 bit is flipped in the $i$ position and the syndrome when the bit is flipped in the $j$ position. To recover the syndromes, we flip bits at the desired bit positions using one of the following three fault injection mechanisms: 1) a custom built shunt probe. 2) facilities provided by some memory controllers. 3) Rowhammer bit flips. We describe these mechanisms next.

Fig. 1: DDR3 socket pin-out. $DQ_x$ (▬), $V_{SS}$ (▬) and other signals (▬).



(a) A custom shunt probe. (b) Tweezers short-circuiting $DQ_0$ and $V_{SS}$.

Fig. 2: Fault injection with the help of syringe needles.

**Error injection with a shunt probe.** To reduce noise and cross-talk between high-speed signals, data pins of the DDR DIMM ($DQ_x$) are physically placed next to a ground ($V_{SS}$) signal. As the ground plane ($V_{SS}$) has a very low impedance compared to the data signal and because the signal driver is (pseudo) open drain, short-circuiting the $V_{SS}$ and $DQ_x$ signals will pull $DQ_x$ from its high voltage level to "0". Depending on the encoding of the high voltage, this short-circuiting results in a 1-to-0 or 0-to-1 bit flip on a given $DQ_x$ line.

Figure 1 displays the locations of the important signals and shows that a $DQ_x$ signal is always adjacent to a $V_{SS}$ signal. Therefore, to inject a single correctable bit error, while the system exercises the memory by writing and reading all ones, we have to short-circuit a $DQ_x$ signal with $V_{SS}$. We can achieve the short-circuiting effect with the help of a custom-built shunt probe using syringe needles (Figure 2a). We insert the probe in the holes of the DIMM socket as shown in Figure 2b. For clarity, we omit the memory module from the picture. We then use tweezers to control when the error is injected by shorts-circuiting the two needles and thus the targeted $DQ_x$ and nearby $V_{SS}$ signal. This method, while simple (and cheap), is effective in the case of a memory controller that computes ECCs in a single memory transaction (ECC word size is 64 bits) and can be used instead of expensive ad-hoc equipment [30], [31].

On some systems (e.g., configuration AMD-1) data is retrieved in two memory transactions and then interleaved. Because of the low temporal accuracy of the shunt probe method, an error inserted on memory line $DQ_k$ ($0 \leq k < 64$) that appears on data bit $2 * k$ will also "reflect" on data bit $2*k+1$ inside the 128 bit ECC word. In this case the syndrome corresponds to two bit errors and contradicts Proposition 1. To ensure single bit errors, once the interleaved mechanism is understood, the exercising data can be constructed such that the reflected positions contain only bits that are encoded to low voltage, essentially masking the reflections.

**Error injection with memory controller.** Some server-grade processors incorporate memory controllers that provide the functionality for artificially injecting errors in memory. This mechanism is useful when testing the error-reporting functionality of the software stack. The error injection facility is exposed as PCI registers, but the OEM can choose to lock these resources from the firmware. Furthermore, the way to specify where the error and what type of error is injected varies across platforms. For example, on some systems the error is injected on the next uncached memory access (e.g., AMD-1) while on others the error is injected on an address that is explicitly specified (e.g., Intel-1).

**Error injection with Rowhammer.** It is also possible to use Rowhammer to trigger bit flips when support for error injection in the memory controller is lacking. Note that this Rowhammer "attack" is merely intended to detect the syndromes and not (yet) to bypass ECC. When a vulnerable aggressor-victim row is detected (either by observing ECC error counters or by using the side-channel introduced in Section VI-A), the position of the bit flip is still unknown to the attacker. However, as we show in Section VI-B, we can overwrite the value of the vulnerable bit with the value to which it flips, to stop the bit from flipping under Rowhammer. Therefore, no error is observed when the bit is masked. We can then leverage this property to perform a binary search for the position of the bit flip. The main problem with this method is the need to find bit flips on every possible position within ECC-word size. On the other hand, once attackers own a set of such vulnerable DIMM(s), they can use these DIMMs to reverse engineer any target.

### C. Dealing with lack of syndromes

On some systems, the entire error-handling stack is exposed to software and drivers adequately report the syndromes when ECC errors happen. On other systems, drivers do not always properly report the syndromes (e.g., Intel-1) and on yet other systems, syndromes are lacking altogether (e.g., Intel-2 and Intel-3). We developed our own driver for reading syndromes for Intel-1. For Intel-2 and Intel-3, it is possible to use the available error counters (for which we also developed drivers) and rely on Proposition 1 to reverse engineer the ECC function. However, this approach is error-prone and requires more manual effort. Instead, we rely on a cold boot attack for reverse engineering the ECC functions on these systems.

### D. Cold boot attacks

Cold boot attacks, previously used to breach privacy and reverse engineer the data scrambling performed inside memory controllers [32], [33], consist of three main steps: 1) interesting data is written in memory, 2) the temperature of the memory is lowered such that data retention of the DDR module is high, and 3) the memory is read back after a reboot, for instance by removing the DIMM and immediately plugging it into another machine and booting.

To read the ECC bits, the attacker can perform a cold boot attack, where the first two steps are similar to other cold boot attacks. However, because the ECC bits are not exposed explicitly by the memory controller, we cannot directly access them in Step 3. We can use a custom FPGA-based memory controller to read the ECC control bits. While there are existing solutions to do so for normal DIMMs [34], we did not find a cost-effective solution for ECC memory. Instead, we opted for using an off-the-shelf motherboard and CPU combination for which we already recovered and verified the ECC function with methods presented in Section V-B. Knowing 1) the data that was written, 2) the data that we read after the cold boot, 3) the expected ECC value and 4) the observed syndrome, we can reconstruct the ECC value that was stored by the victim system for certain data patterns.

One challenge is that ECC memory is normally always initialized at boot time by the target system to avoid spurious ECC errors when accessing the memory. This initialization is usually done by the firmware (BIOS) and stops us from performing our cold boot attack. To achieve our goal, we bypassed the memory initialization by reverse engineering and modifying the parts of the binary BIOS code that performs DRAM initialization. We will open-source this patch along with all other necessary details to allow others to build a generic ECC memory dumper.

### E. Reverse engineering approach

Table II summarizes the pros and cons of our available reverse engineering mechanisms. We now briefly describe how we employed these mechanisms to reverse engineer ECC functions on the machines described in Table I.

**Machine AMD-1.** Here, the data sheet includes the syndrome table decoding technique for locating ECC errors. The system supports symbols of 4 or 8 bits wide and uses 128 bits (two 64-bits *beats* interleaved) to compute the ECC control bits. The data sheet further claims that the code can correct any number of errors in a single symbol and detect two symbols data corruption, hinting at a variant of the BCH code. We recover the complete ECC function using the syndrome table. To find out that the system indeed uses the same ECC functions to find the mapping of the data bits to DRAM pins, we employ our shunt probe. Our results conclude that AMD-1's memory controller accurately reports errors and we further find how data bits are mapped to DRAM pins. The mapping of data bits to DRAM pins is helpful when reverse engineering with cold boot attacks.

The data sheet of a newer version of the AMD-1 CPU model mentions the support for error injection. We therefore wrote a driver for injecting errors through the memory controller of this system and confirmed that it also supports this mechanism. We used the error injection functionality to also confirm that bit errors in different symbols are uncorrectable.

**Machine Intel-1.** The ECC function for this system is not documented. While it has support for error injection through the memory controller, unfortunately driver support for this functionality at the moment of writing is non-existent. Given that writing a device driver for error injection in this processor

TABLE II: Advantages ($\triangle$) and disadvantages ($\triangledown$) of the proposed ECC recovery methods in this paper ($\ominus$ indicates 'neutral or fixable').

| Method | Compatibility | Price | Setup Time | Precision |
|---|---|---|---|---|
| **Needle FI** | $\triangle\triangle$ works on any hardware | $\triangle\triangle$ a few dollars | $\triangledown$ fiddly | $\triangle\triangle$ recovers signal mapping |
| **Mem. cntr.** | $\triangledown\triangledown$ not always available | $\triangle\triangle$ free | $\triangledown$ software support is rare | $\ominus$ potentially imprecise<br>$\triangledown$ no signal mapping |
| **Rowhammer FI** | $\triangle$ targets' performance | $\triangle$ vulnerable DIMMs | $\triangle$ quick | $\triangledown$ no signal mapping |
| **Cold-boot** | $\triangle\triangle$ works on any hardware | $\triangledown$ initial investment<br>$\triangledown$ cooling spray | $\triangledown$ rather slow | $\triangle\triangle$ recovers signal mapping |

TABLE III: Properties of recovered ECC algorithms.

| ID | $d_{min}(cw)$ | $d_{min}(data)$ | symbol size |
|---|---|---|---|
| AMD-1 | 3 | 4 | 8 |
| Intel-1 | 4 | 4 | 4 |

TABLE IV: ECC error handling software with a default Debian 9.

| ID | OS log | Firmware log | Crash on UE |
|---|---|---|---|
| AMD-1 | yes | yes | yes |
| Intel-1 | no | yes | no |

TABLE V: Error patterns that can circumvent ECC.

| ID | Pattern | Config. | # flips | Flips location |
|---|---|---|---|---|
| AMD-1 | $[\mathcal{P}_1]$ | Ideal | 3-BF-16 | 3 symbols, 1 in control bits |
| AMD-1 | $[\mathcal{P}_2]$ | Ideal | 4-BF-16 | Min. 2 symbols |
| Intel-1 | $[\mathcal{P}_3]$ | Ideal | 4-BF-8 | Min. 2 symbols |
| Intel-1 | $[\mathcal{P}_4]$ | Default | 2-BF-8 | Min. 2 symbols |

is much more involved than just reading information (such as syndromes), we opted for using Rowhammer bit flips themselves for reverse engineering the ECC function. The data sheet of Intel-1 exposes the ECC error counters and syndromes of the ECC error. We had to write our own drivers to access this information. We previously already built a database of vulnerable bits and DIMMs and used a novel side-channel attack to leak whether the ECC unit is correcting a bit flip (which we explain in Section VI-A). Using our database of bit flips on these vulnerable DIMMs, we found the syndromes for each vulnerable bit position—only three DIMMs were required for a complete recovery. We validated our results using the shunt probe, which showed that the memory controller shuffles the data when sending them to various data pins on the DIMMs.

**Machines Intel-2 and Intel-3.** These two machines are the least friendly in terms of documentation, but their data sheets do mention that ECC is generated over 64 bits of information at a time. Using our shunt probe, we realized that the software stack in these machines does not report ECC errors. To reverse engineer the ECC functions on these machines, we employ our cold boot attack and rely on the already reverse engineered ECC function on AMD-1 to stage the last step of the cold boot attack. We re-flashed the BIOS of AMD-1 with changes that bypass the memory initialization. In this process, we used an old version of the memory initialization that was contributed by the manufacturer to the `coreboot` project [35]. Note that the two-beats ECC computation and residual errors due to cold boot complicate the complete recovery of the parity matrix on these machines. As a result, the recovered ECC functions for these machines still contain a few incorrect cases.

### *F. Results*

For brevity, since AMD-1 and Intel-1 are representative of the general trends we observed across all setups, and the recovery on Intel-2 and Intel-3 is not entirely complete due to residual errors in the cold boot attacks, we focus on AMD-1 and Intel-1 in the remainder of the paper. Even so, all the recovered parity matrices for the configurations in Table I can be found in Figures 8 and 9 in the Appendix.

**Ideal guarantees.** We first discuss the ideal guarantees provided by the ECC functions in the two systems. In an ideal setting, correctable errors should be detected and corrected, while uncorrectable errors that are detected should result in a process or system crash. In this configuration, the only way an attacker can compromise the system is by triggering *enough bit flips* at the *right* positions to ensure that the ECC function does not detect a corruption. Table III shows the minimum number of bit flips required in either data bits (i.e., $d_{min}(data)$) or data bits plus control bits (i.e., $d_{min}(cw)$). Triggering these many bit flips close to each other is difficult on most DIMMs that are vulnerable to Rowhammer. However, it is much easier to trigger corruptions on Intel-1 as discussed next.

**State of practice.** As shown in Table IV, we found that in Intel-1 detected uncorrectable errors do *not* crash the system and are not even reported by the OS. The main cause seems to be improper software support for the memory controller in the OS, i.e. the error reporting driver fails to recognize and initialize the resources of the error reporting mechanism. As a consequence, an attacker can exploit the system, in its default configuration, with a smaller number of bit flips than necessary with the ideal guarantee provided with the ECC function.

**Exploitable patterns.** We use Z3, a constraint solver, to mine exploitable patterns of the ECC functions for AMD-1 and Intel-1. Table V shows the results for the ideal and default configurations. For AMD-1, the attacker requires at least three bit flips in 16 bytes (i.e., an ECC word) when one of the bit flips is in the control bits ($[\mathcal{P}_1]$). The other two bit flips should target two distinct symbols (i.e., be at least 8 bits apart). When targeting data bits alone, four bit flips should land in at least two distinct symbols in an ECC word ($[\mathcal{P}_2]$).

For Intel-1, in an ideal configuration, an attacker needs to find four bit flips in at least two distinct symbols (i.e., at least 4 bits apart) in eight bytes ($[\mathcal{P}_3]$). However, given that Intel-1 does not crash on detected uncorrectable errors, with only two bit flips in distinct symbols in an ECC word, it is possible to

TABLE VI: Percentages of rows with corruptions in an ECC DIMM.

| $[\mathcal{P}_1]$ | $[\mathcal{P}_2]$ | $[\mathcal{P}_3]$ | $[\mathcal{P}_4]$ |
|---|---|---|---|
| 0.12% | 0.12% | 0.06% | 0.60% |

TABLE VII: Percentages of rows with corruptions in the flip database of Tatar et al. [37] with 14 DIMMs.

| ID | Bit flips | $[\mathcal{P}_1]$ | $[\mathcal{P}_2]$ | $[\mathcal{P}_3]$ | $[\mathcal{P}_4]$ |
|---|---|---|---|---|---|
| $A_1$ | 200468 | 18.38% | 04.41% | 00.79% | 29.51% |
| $A_2$ | 21542 | 00.23% | 00.03% | 00.03% | 02.81% |
| $A_3$ | 2926 | 00.00% | 00.00% | 00.00% | 00.30% |
| $A_4$ | 256359 | 26.80% | 08.52% | 02.10% | 37.52% |
| $B_1$ | 1504 | 00.00% | 00.00% | 00.00% | 00.00% |
| $C_1$ | 16489 | 00.09% | 00.00% | 00.00% | 01.32% |
| $D_1$ | 2131 | 00.00% | 00.00% | 00.00% | 00.66% |
| $E_1$ | 202630 | 06.30% | 00.76% | 00.14% | 17.16% |
| $E_2$ | 24587 | 00.06% | 00.00% | 00.00% | 01.51% |
| $F_1$ | 413796 | 51.09% | 26.02% | 06.00% | 53.03% |
| $G_1$ | 15990 | 00.06% | 00.00% | 00.00% | 00.93% |
| $H_1$ | 16087 | 00.03% | 00.00% | 00.00% | 00.77% |
| $I_1$ | 130187 | 00.82% | 00.03% | 00.00% | 06.24% |
| $J_1$ | 7185 | 00.00% | 00.00% | 00.00% | 00.70% |
| AVG | 93705 | 7.42% | 2.84% | 0.65% | 10.89% |

exploit the system ($[\mathcal{P}_4]$).

**Exploitable ECC DIMMs.** We ordered ECC DIMMs from four different DRAM chip manufacturers. We chose ECC DIMMs with DRAM chips based on previously published work [10], [36], [37]. Note that the exact same DRAM chips are used both in ECC and non-ECC DIMMs. We found that one out of the four manufacturers produces DIMMs that cause corruption on both AMD-1 and Intel-1. Table VI shows the results of hammering 109k pairs of aggressor-victim-rows and the percentage of rows that have enough bit flips to escape the patterns discussed in Table V. We later use this DIMM to evaluate our end-to-end exploits in Section VII.

**Other DIMMs.** Table VII shows the ECC protection for the public database of bit flips published by Tatar et al. [37] that contains 14 desktop DIMMs with the kind of chips that are used in ECC DIMMs also. We find that every DIMM but one exhibits bit flips that ECC cannot correct and 10 contain potentially uncorrectable corruptions that the ECC algorithm cannot detect. When the ECC detection is used correctly (i.e., $[\mathcal{P}_1]$, $[\mathcal{P}_2]$ and $[\mathcal{P}_3]$), 0.65%-7.42% of all bit flips still cause silent corruptions. On the default configuration ($[\mathcal{P}_4]$), on average up to 10.89% of the bit flips cannot be corrected.

## VI. Challenge $\mathcal{C}_2$: ECC-aware Rowhammer

This section addresses $[\mathcal{C}_2]$ and shows how an attacker armed with details on the ECC function can reliably trigger Rowhammer bit flips that bypass ECC memory with no crashes. To this end, we show an attacker can *observe* bit flips using a side channel and then *control* bit flips using carefully selected data patterns in memory.

### A. Observing bit flips

We now present a novel side channel that allows an attacker to observe bit flips that trigger correctable ECC errors. For this purpose, we use double-sided Rowhammer (i.e., accessing two aggressor rows targeting a victim row in between) to trigger bit flips and then measure the number of clock cycles it takes to access the victim row. On setup Intel-1, we select 3K aggressor-victim pairs and measure the DRAM access time on the victim row after Rowhammer. In case of a bit flip in the victim, this access triggers a correctable ECC error. We also randomly select 3K pairs that are potential targets for Rowhammer (i.e., map to adjacent rows), but that do not trigger any error after Rowhammer. To confirm ECC error correction is triggered, we read platform-specific hardware registers that record the presence of an ECC correctable error.

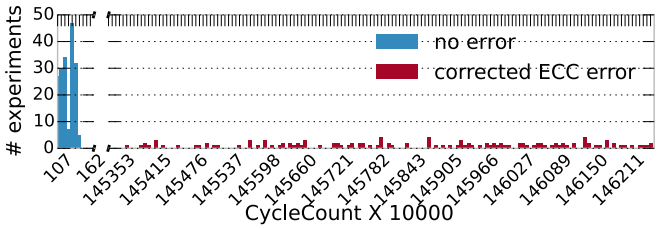



Fig. 3: ECC memory access time distribution across 3K aggressor-victim pairs for corrupted vs. uncorrupted data.

Figure 3 shows that accesses to data triggering correctable ECC errors are slower than those to data with no bit flips. The timing difference is three orders of magnitude, yielding a reliable timing side channel to distinguish between the two cases. Furthermore, we note that, in the error case, the access time has higher dispersion compared to the error-free case.

To show this side channel is present on different platforms, we target a single vulnerable aggressors-victim pair across our setups. In this experiment, each pair is *hammered* in two rounds each comprising 100 Rowhammer iterations. In the first round, we choose data such that errors are triggered. In the second round we change the data such that no errors are triggered. On setup Intel-1, we confirm the error case is slower by a factor of 563.1x compared to the error-free case. On setup AMD-1, however, we observe a difference of only a factor of 1.01x. To closely examine the latter scenario, we randomly pick 5 vulnerable victim rows, hammer them, and measure the DRAM access time for each 8-byte word in the victim row. We repeat this experiment 100 times per victim row and report the average access time in Figure 4. As evident by the peaks in the figure (marking synchronously corrected ECC errors), even a minimal difference in the number of cycles to access the victim row is sufficient to reliably distinguish error from error-free cases. Interestingly, we also observe that, in some cases, error accesses are faster than error-free ones. Such negative peaks (first and fourth subplot in Figure 4) seem to only occur in the case of 0-to-1 bit flips. We leave the study of this phenomenon as future work.

In summary, the presented side channel is reliable enough to observe bit flips triggering ECC error corrections. Moreover, the side channel can reveal the exact location and direction of the bit flip. In the following, we investigate the source of the side channel in hardware and software.

**ECC error handling architecture.** ECC error detection is synchronous with respect to a given memory access. In par-

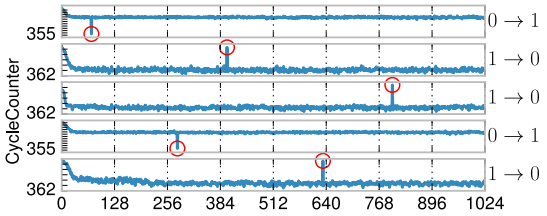



Fig. 4: ECC memory access times for all the 8-byte chunks in 5 victim rows. The peaks correspond to bit flip-induced ECC errors corrected by hardware.

ticular, in response to a memory access request from the CPU, the memory controller immediately retrieves the data and its associated ECC bits from memory. Before returning the data to the CPU, the controller checks the data for errors. Note that, when so-called *scrubbing* is enabled, the controller can also periodically check the memory for errors with no CPU synchronization. However, given the low scanning frequency (a few hours for a full memory scan), its impact can be safely ignored for our purposes (short-lived Rowhammer attacks).

Once an error is detected by the memory controller, error-correcting operations are immediately performed by the hardware. Since the hardware has to correct (and to write back) the data via a slow path, this may introduce a measurable latency on the corresponding memory access and give rise to a timing side channel. In addition, the hardware needs to inform the system of the event using one of the following options (depending on the boot-time configuration): raise an exception at the software level or invoke a system management interrupt (SMI) handler.

With the first option, a machine check exception (MCE) is triggered as soon as the error is detected—even if interrupts are disabled [38]. With a failing memory cell, correctable machine check interrupts (CMCIs) become frequent, resulting in non-trivial system overhead due to excessive time spent servicing interrupts. To reduce the overhead, an OS driver may dynamically switch to polling mode, where CMCIs are blocked and error accounting registers are polled explicitly. In both cases, errors are logged inside the OS and, depending on the OS configuration, the memory page containing the error is masked, the system is restarted, or the faulting process is killed [39]. However, the OS does not have accurate knowledge of the physical location of the error (e.g., the exact DIMM, DRAM address, etc.), which makes it hard to implement sophisticated error handling policies.

This problem is solved with the second option, where an SMI handler can use platform-specific information to recover the exact physical location of the error. This information can then be saved in Advanced Configuration and Power Interface (ACPI) tables or other error-reporting registers. To inform the OS of the event, the SMI handler ultimately raises an MCE. This option is widely used on recent Intel Xeon machines and it is known as Enhanced Machine Check Architecture [40].

In both cases, a software chain that involves expensive operations is synchronously executed as soon as an error is triggered in response to a given memory access. This may introduce significant access latency and give rise to another timing side channel to detect ECC correctable errors.

**ECC error handling in practice.** As evidenced earlier, ECC error handling side channels may originate from both hardware and software operations. We now revisit our earlier experiments across setups to exemplify their availability on real-world platforms in their default configurations.

On setup AMD-1, uncorrectable errors crash the system. Correctable errors are reported by the OS driver and appended to a dedicated MCE log file (other than being logged at the firmware level). These synchronous software operations are lengthy and give rise to the strong timing signal we observed in Figure 3. Had an SMI handler been enabled in our setup, the signal would have been even stronger, given that studies show that handling an SMI is up to 171x times slower than simply triggering an MCE [31]. In addition, we observe that, by default, on the Debian 9 distribution (Linux kernel 4.9.3) used in our setup, the MCE log file[1] is world-readable, yielding an even more convenient side channel to observe bit flips.

On setup Intel-1, uncorrectable errors do not crash the system. In addition, the available OS driver recognizes the memory controller but does not report correctable errors. In other words, no MCE event is logged by the OS. Correctable and uncorrectable errors are logged in a firmware log, but only after a certain threshold is reached. While no logging or other software/firmware operations take place in the common case, the error handling operations performed by the hardware at memory access time are still sufficiently lengthy to give rise to the crisp timing signal we observed in Figure 4.

In summary, while ECC-equipped platforms may be configured in several different ways, error correcting operations carried out in hardware or software are consistently observable across platforms through a variety of side channels. This allows attackers to reliably observe bit flips as a prelude to end-to-end Rowhammer attacks on ECC-equipped platforms.

### B. Controlling and composing bit flips

It has been long known that Rowhammer bit flips are data-dependent. For example, the original Rowhammer paper [5] showed that a *stripe* pattern in DRAM's array-of-rows organization (even/odd rows populated with 0s/1s or vice versa) induced the most errors. Since then, similar patterns have been used to maximize the number of bit flips and ease Rowhammer exploitation. We now aim to show that such data-dependent behavior can also be used to *control* and *compose* bit flips and enable ECC-aware Rowhammer exploitation. We start with showing how data patterns can be used to enable/disable individual bit flips and later show such behavior is independent of neighboring flips or data patterns enabling composability.

**Controlling individual bit flips.** We start by exhaustively testing our memory chips using double-sided Rowhammer with 4 possible data patterns: (i) *0/1-stripe* (aggressor rows populated with all 0s, victim rows populated with all 1s), (ii) *1/0-stripe* (aggressor rows populated with all 1s, victim

[1] `/var/log/mcelog`

rows populated with all 0s), (iii) *0-uniform* (aggressor and victim rows populated with all 0s), and (iv) *1-uniform* (aggressor and victim rows populated with all 1s). Across our setups, we observe numerous bit flips in the two *stripe* configurations and no bit flips in the *uniform* ones. To confirm the latter result, we progressively reduce the DRAM refresh rate until we observe bit flips for the *uniform* patterns. This only happens for unstable system configurations with very low refresh rates, where bit flips occur even without Rowhammer.

This experiment empirically shows an important property of Rowhammer: bit flips occur due to parasitic current [41], which induces capacitors storing opposite electric charges (i.e., data values) to interfere with one another and cause charge leakage in the victim cells. The direction of the bit flip ($1 \rightarrow 0$ vs. $0 \rightarrow 1$) triggered by a particular *stripe* pattern (*0/1-stripe* vs. *0/1-stripe*) is an artifact of *data scrambling* operated by the memory controller, which stores 0s (or 1s) as a charged (or non-charged) state. However, since scrambling on commodity systems operates by XORing data values with an address-dependent bitmask that repeats consistently across (adjacent) rows [33], the bitwise *stripe* pattern is preserved even in the presence of scrambling. In other words, for every bit $i$ in a given aggressor-victim-aggressor row tuple, data scrambling can (if at all) turn a $0-1-0$ bit column (assuming *0/1-stripe*) into a $1-0-1$ column (and vice versa), but always preserve the *stripe* (or in other cases *uniform*) pattern at the bit granularity. This property shows that, somewhat counterintuitively, we can ignore data scrambling to control Rowhammer bit flips with (*stripe*) data patterns. It also suggests we can enforce bit-granular *stripe* patterns to control individual bit flips.

To confirm this intuition, for each bit flip triggered in the previous experiment, we flip the corresponding (column-wide) bits in the aggressor rows to enforce a bit-granular *uniform* pattern and hammer again. Across our setups, we observe this is consistently sufficient to disable the original individual bit flips. Restoring the original bit-granular *stripe* pattern consistently re-enables every given bit flip. This experiment shows we can reliably control individual bit flips. In other words, for every bit $i$ in a given aggressor-victim-aggressor row tuple, setting aggressor bit values to enforce a column-wide *uniform* pattern ($0-0-0$ or $1-1-1$) prevents occurrence of any flips in the victim bit, while setting aggressor bit values to enforce a column-wide *stripe* pattern ($0-1-0$ or $1-0-1$) induces flips in the victim bit (assuming the underlying cell is vulnerable). We can then switch between the two patterns to selectively enable/disable individual bit flips.

**Impact of neighboring bit flips.** We now have the ability to control individual bit flips starting from a given data pattern configuration in an aggressor-victim-aggressor row tuple. We now want to verify whether controlling multiple bit flips in the same ECC word at the same time is viable. This property is necessary to ensure composability of bit flips and is only realistic with no cross-bit-flip interference. To confirm the absence of such interference, we select all the victim ECC words that revealed multiple *stripe*-induced bit flips in our previous experiment, and exhaustively test all the relevant combinations of aggressor bit values. For example, given a victim ECC word with only two bit flips at offset $i$ and $j$ with the *0/1-stripe* pattern, we test the 4 possible combinations of column-wide *0/1-stripe* (or *1/0-stripe*) at offset $i$ and column-wide *0/1-stripe* (or *1/0-stripe*) at offset $j$. We say that there is no cross-bit-flip interference in a given victim ECC word iff the bit flip $i$ ($j$) is solely dependent on the aggressor bit values at offset $i$ ($j$). Across our setups, we observe no interference in any vulnerable ECC word, empirically confirming we can control multiple bit flips at the same time in a given word.

**Impact of neighboring data.** Our last experiment showed we can control individual bit flips with no interference from neighboring bit flips nor neighboring aggressor bit values. This was the case even for adjacent bit flips, showing that value changes in the aggressor bits at offset $i+1$ (or $i-1$) have no impact on a bit flip at offset $i$. To achieve fully unconstrained bit flip composability, however, we also need to study the impact of neighboring data values in the victim row.

For this purpose, we set up a new experiment, in which we select all the aggressor-victim-aggressor row tuples that trigger a single bit flip and randomly assign them one of the following data patterns: $D$ (column-wide *1/0-stripe* pattern in the bit flip location, random values elsewhere in the aggressor rows, and 0s elsewhere in the victim row) and $N$ (same as $D$, but 1s are used elsewhere in the victim row). The patterns are designed to stress the extreme cases of data values following (or not following) the direction of the bit flip (respectively). For this reason, we present results with data scrambling disabled, but we observed a similar trend with data scrambling enabled.

Figure 5 presents our results, depicting the probability distribution of the difference between the number of bit flips induced by $D$ and $N$ patterns as a function of the probability of the occurrence of the $D$ pattern (which we vary in every experiment). As the difference is generally less than 2.5% across setups, this shows that even neighboring data values in the victim row have little or no influence on a given bit flip. This confirms an attacker can surgically manipulate aggressor data bits to obtain fully composable, data-controlled bit flips and target arbitrary victim data in a given ECC word.

Interestingly, in some setups (e.g., AMD-1), there seems to be less interference, showing that, while the properties we described well-approximate DRAM behavior across setups, they cannot perfectly model all the physical constraints in general. However, our approximations are sufficient to reliably mount practical attacks, as shown by our end-to-end exploit.

## VII. Challenge $\mathcal{C}_3$: A practical *ECCploit*

In this section, we present *ECCploit* and show how an attacker—armed with knowledge of the ECC function, a side channel to observe bit flips, and the ability to control/compose bit flips via data patterns in aggressor rows—can mount practical end-to-end Rowhammer exploits on ECC-equipped systems. *ECCploit* consists of three phases. First, we *template* memory to find correctable bit flips. Second, we try to *combine* multiple of these bit flips to create error patterns that the

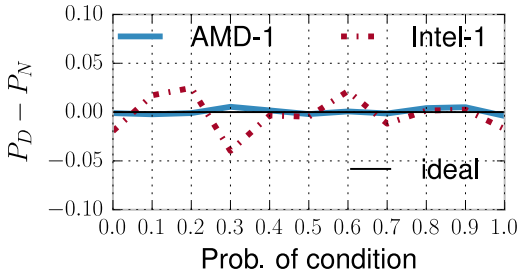



Fig. 5: Probability distribution of the difference between the number of $D$ and $N$ pattern-induced bit flips vs. probability of occurrence of the $D$ pattern.

ECC function is unable to detect. Finally, we use these patterns to launch *exploits* on three different victims: page table entries [6], RSA public keys [10], and binary code [13].

### A. Templating correctable errors

In the templating phase, we probe the memory to see if we can safely trigger bit flips using Rowhammer. In particular, we only want to cause errors that the ECC function can correct automatically. Although the error correction ensures that we cannot observe these bit flips directly, the side channel presented in Section VI still lets us detect them.

**Target address selection.** Templating starts with a list of potential aggressor locations ($a_1$ and $a_2$ in the case of double-sided Rowhammer) and victim ($v$) addresses which should both map to the same bank but different (neighboring) rows. Obtaining this list is trivial if we know the mapping between virtual and physical addresses. In our exploits, we rely on existing reverse engineering techniques to reconstruct such mapping [42]. However, even if this information is absent, the attack can start with an exhaustive list of addresses—slowing down, but not stopping, the attack.

**Pattern selection.** Our attack uses double-sided Rowhammer to detect usable tuples of aggressor-victim-aggressor $(a_1, v, a_2)$. To ensure a crash-free templating strategy (i.e., only triggering correctable ECC errors in vulnerable locations), we arrange values in aggressor and victim rows such that the Hamming distance is less than or equal to the number of errors $E$ that the ECC algorithm is capable of correcting. In other words, we make sure that for each ECC word in the victim row, the corresponding ECC words in the aggressor rows are only $E$ bit flips apart. Assuming $x$ is the value stored in an ECC word, and $x'$ is the value with $E$ bits flipped, we can either store $x$ in the victim ECC word and $x'$ in the aggressor ECC words or $x'$ in the victim ECC row and $x$ in the aggressor ECC words to check for correctable bit flips in either $1 \rightarrow 0$ or $0 \rightarrow 1$ directions due to the resulting striping patterns.

**Search strategy.** Rather than targeting a single ECC word and single word offset for each Rowhammer trial, we target all the words in the victim row at the same time during each hammering attempt. For each word, we consider a different set of $E$ bits in subsequent attempts. For instance, if the ECC corrects single bit errors, we hammer first with bit patterns in the aggressor and victim rows such that aggressors and

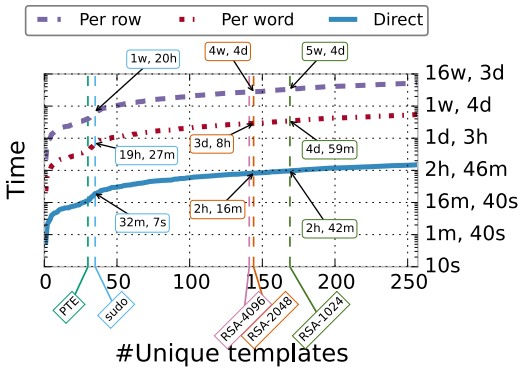



Fig. 6: Templating with ECC memory.

victim differ only in the most significant bit of each of the ECC words in the row, then with patterns that differ only in the next bit, and so on. At each trial, we read from the entire victim row all at once and use our side channel to detect bit flips anywhere in the row—we found this is reliable even at the row granularity. This strategy exploits composability of bit flips and allows us to batch many independent tests and increase the templating efficiency. For instance, if the ECC corrects single bit errors, this strategy requires only as many trials per tuple as the number of bits in a single ECC word. ECC algorithms that use multiple-bit symbols (e.g., ChipKill) require even fewer trials as a row contains fewer symbols.

If we detect bit flip(s) anywhere in the victim row, we need to hammer the tuple a few more times to identify the flipping ECC word(s). For this purpose, we perform a (pseudo-)binary search—omitting *stripe* patterns in words we are not testing—until we reproduce the bit flip(s) on one or more words. The entire process is repeated twice for each tuple using the two possible *stripe* patterns. This is to identify vulnerable bits in both directions ($1 \rightarrow 0$ or $0 \rightarrow 1$). After scanning all the tuples in memory, we note down all the vulnerable *1-bit templates* with the corresponding $(a_1, v, a_2)$ tuple, the ECC word, the word offset, and the direction of the bit flip in the victim row.

### B. Combining bit flips

Given our knowledge of the ECC algorithm and the 1-bit templates inducing correctable bit flips from the previous step, the goal of this phase is to combine multiple bit flips in a single ECC word and produce new words that escape ECC detection. As a first step, we group together all the 1-bit templates that have the same aggressor rows, victim row, direction, and ECC word in a *template group*.

Next, we generate possible flipped words that, when induced via Rowhammer, bypass the target ECC algorithm. Specifically, for every template group, we want to find a combination of $k$ 1-bit templates that would induce $k$ bit flips that result in a corruption that ECC does not correct ([$\mathcal{P}_4$]) or even detect ([$\mathcal{P}_1$], [$\mathcal{P}_2$] and [$\mathcal{P}_3$]). For simplicity, the current version of *ECCploit* only targets flips in the data bits and not in the control bits. While this is enough for our setup, one can optimize *ECCploit* further to take control bits into account.

Figure 6 shows the results of our templating step on the Intel-1 machine. On this machine, we can directly observe (detectable) corruptions without crashing the system. When we cannot directly observe uncorrectable errors (e.g., AMD-1), we can instead use the side channels discussed in Section VI-A. Overall, we only have 265 templates available. When directly observing bit flips, it takes 4 hours to find these templates. Using the word-level side channel, it takes 6 days, and using the row-level side channel, it takes us 8 weeks to find these templates. To compare, assuming no ECC support, it would take us at most 1 minute to find 265 templates. This shows that ECC does significantly reduce the attack surface of Rowhammer attacks, by forcing the attacker to go through a much lengthier templating step. However, this is typically unimportant in practical attack settings, where the attacker can run code on demand on the victim machine and complete a templating step of hours or even days in complete isolation without interfering with the rest of the system. After templating is over, ECC has essentially no impact on the exploitation step, which completes in seconds or minutes similar to existing non-ECC exploits. Next we discuss how we use our templates to build practical exploits on ECC memory.

### C. Exploitation

Armed with vulnerable ECC-aware templates, an attacker can now mount practical exploits by (i) massaging the target data onto the vulnerable location, (ii) setting the corresponding aggressor bit values as dictated by the templates, and (iii) and hammering to reliably reproduce the (composed) bit flips on the victim data. This exploitation strategy is similar, in spirit, to the one employed by existing reliable Rowhammer attacks [10]. The key difference—and challenge for ECC-aware exploitation—is that the number of useful templates is now much lower, given that we need a carefully-selected combination of bit flips to bypass ECC. Furthermore, unlike existing Rowhammer exploits, ECC templates corrupt multiple bits and this can complicate existing Rowhammer attacks.

To study the effectiveness of our *ECCploit* attack in real-world exploitation settings, we reproduce three existing Rowhammer attacks on Intel-1. (i) The original Rowhammer attack by Seaborn [6], which flips bits in page table entries (PTE) to map an unauthorized page (ideally a page table page) for privilege escalation, (ii) the attack introduced by Razavi et al. [10] which flips bits in a RSA key to compromise its cryptographic strength for authentication bypass, and (iii) the attack introduced by Gruss et al. [13] that flips bits in opcodes, leading to user authentication bypass in the `sudo` command.

**Page Table Entry (PTE) ECCploit.** Like the original attack by Seaborn et al. [6], we spray physical memory with page tables and then try to gain access to an inaccessible page by flipping a bit in a PTE. To implement this attack, we need to consider the format of the PTE. The format of the PTE can vary across different architectures. In modern Intel and AMD machines, PTEs are 64 bits wide and store the physical address of a page in bits 12 to $L$, where $L$ is the number of bits required to address the machine's physical memory. Importantly, Intel requires that bits $L$ to 51 are zero, lest any access triggers a general protection fault which would crash the machine. AMD even prescribes a zero value for all bits between $L$ and 63. Given this, *useful* templates contain at least one bit flip between bit 12 and $L$ in 64 bits chunks and do not trigger a $0 \rightarrow 1$ bit flip in the $L$:51 range on Intel machines and $L$:63 range on AMD machines. Note that bit flips on the first 12 bits are often harmless (e.g., cacheable flag).

**Results.** From our discovered 265 templates, 6.15% are exploitable. The rest are templates that would crash the system because bits would flip in the reserved field of the PTE. As shown in Figure 6, we find the first suitable template after 19 minutes if we can directly observe the bit flips, and 12 hours or 4 days using the side channels respectively. Without ECC, it would take less than 2 seconds to find a suitable template.

Summarizing, even with an imperfect page table spraying strategy of the Seaborn attack, we were able to map unauthorized memory pages with a success rate of 39.9% and a page table page with a 2.5% success rate. In the remaining cases, the attack fails to modify any PTE of the attack process, but no crashes occur. By tracking the correctable error counters, we confirmed that when there is no change in the PTE, as either no bit flip occurs or ECC corrects the error. This happens because the victim PTE does not always have the target bits set in the direction of the chosen template.

Brasser et al. [36] report a 5% success rate in a similar non-ECC setting for mapping page table pages, which shows that our ECC-based exploitation strategy has relatively little impact on the success of the attack compared to traditional Rowhammer exploits. On our testbed, a more sophisticated massaging strategy such as the one employed by Drammer [11] can obtain a significantly higher success rate in mapping a page table page in the address space (39.9% in the ideal case).

**RSA ECCploit.** RSA [43] is a public-key crypto system which relies on the infeasibility of factorizing the product ($n$) of two large prime numbers with a similar number of bits. The attack uses the fact that a single-bit-faulted $n$ ($n_1$) is easy to factorize as the chance of the factors of $n_1$ being of similar size is very low—the probability to efficiently factorize $n_1$ is 12-22% [10]. We claim that in the presence of $t$ bit faults ($t \geq 2$), $n_t$ is efficiently factorizable with at least the same probability as $n_1$. This is because flipping a single bit versus flipping $t$ bits in $n$ only changes the quantity that is added or subtracted to $n$. The result in both cases is a natural number with the same probability of being easily factorizable. Formally, using the Erdös-Kac [44] theorem, the number of distinct prime factors of $n_1$ and of $n_t$ follows the standard normal distribution with the mean and variance $\log\log n_*$. Because $n_t$ and $n_1$ are of similar sizes, the probability to efficiently factorize the faulty $n$ is the same in both cases—12-22%.

**Results.** To experimentally confirm this claim, we use 1337 randomly generated RSA keys from each size class of 1024 bit, 2048 bit and 4096 bit. We then replicate Flip Feng Shui [10] using our ECC templates. On average, our 265 templates could only mutate a given 1024 bit key 2.8 times, a given 2048 bit

key 5.5 times, and a given 4096 bit key 9.4 times. Given a 1 hour cutoff time to ECM [45], we can factorize 45.1% of the 1024 bit keys, 37% of the 2048 bit keys and 28.7% of the 4096 bit keys. Without considering the factorization and memory deduplication delay, if we can directly observe the errors it takes us on average 2 hours, and 3 days or 4 weeks if we use the side channels as shown in Figure 6. Without ECC, it takes us less than a minute to achieve similar success rates.

**Opcode modification ECCploit.** This attack corrupts instructions in memory to bypass certain security checks [13]. As already mentioned, on ECC protected memory, more than one bit flip within the same ECC word is necessary to bypass the ECC protection. On synthetic `x86_64` binaries that mimic authentications, we find that the probability of the code being successfully attacked slowly grows from 5% to 10% when the number of bit flips in 8 bytes increases from 1 to 4 respectively. On the other hand, the probability of the program to crash is 55% when 4 bits are changed as opposed to 20% when a single bit is flipped. To investigate whether corrupting opcodes is feasible with ECC templates in a real application, we target `sudoers.so` which is responsible for privilege elevation functionality provided by the `sudo` command.

**Results.** In the same version of the binary, Gruss et al. [13] find 29 candidate instructions in which a single bit flip yields unauthorized access. Template #36 flips bit 0 and 5 of a single byte, changing a conditional branch instruction (`jne $8fa0` at offset `0xbdc0`) to a `mov` instruction (`mov 0x1da(%rbp),%eax`), leading to an authentication bypass. When observing ECC errors directly, we find this template in 32 minutes, and it takes 12 hours or 4 days when using the side channels as shown in Figure 6. Without ECC, we can target any of the 29 candidate instructions without worrying about crashes. We can find such a flip in 6 minutes.

## VIII. Related Work

**Rowhammer.** After the initial disclosure of Rowhammer [5], security researchers showed advanced Rowhammer-based exploitation of browsers [6]–[8], [46], clouds [10], [29] and mobile phones [11], [12], and even managed to flip bits across the network [46]. Although it was always clear that it is possible that more bits flip than an ECC function can handle, properly implemented ECC memory is still perceived as a practical mitigation for Rowhammer exploits [6], [13]. However, some researchers already questioned whether ECC is enough, and consistent with our findings, discovered that some systems do not always report ECC events [47]. We are the first to show that reliable Rowhammer attacks are possible, even if the system reports these events correctly.

**Hardware reverse engineering.** There are many undocumented features modern hardware systems. The complex hashing function that decides how physical addresses map to CPU cache sets is an example which is important for a variety of cache attacks [48]–[51]. Maurice et al. [52] reverse engineers this mapping. DRAMA [42] reverse engineers the mapping function from physical addresses to DRAM addresses. Inside memory chips, each DRAM address is further decoded in banks, rows and columns. Jung et al. [53] reverse engineer this physical decoding scheme by applying a temperature gradient to memory chips. GPU architectures are sometimes undocumented, Frigo et al. [12] reverse engineer a common integrated GPU in mobile phones. In this paper, we reversed engineered the ECC functions in common processors and used this to mount successful and reliable Rowhammer attacks.

**ECC error handling and error injection.** While others have studied the overhead of SMI handling [31], [54], the overhead of handling ECC exceptions is only briefly noted in the context of memory reliability [9], [55], [56]. Recently, Gottscho et al. [31] injected faults in memory with the help of a custom proprietary device and focused on the overhead of these errors. Instead, we proposed several new and cheaper ways to induce memory errors (e.g., a simple syringe needle probe).

## IX. Mitigations

We have shown that ECC alone is not an adequate Rowhammer mitigation. One way to strengthen ECC is to combine it with Target Row Refresh (TRR) [19]—another hardware mechanism, designed specifically to protect against Rowhammer. While there are reports of bit flips on memory with TRR [11], [13], we expect that a combination of ECC with TRR will make Rowhammer exploitation much harder.

State-of-the-art ECC algorithms in use today all target error patterns of off-the-shelf DRAM under normal conditions [1], [9] rather than adversarial cases. Another avenue for mitigations is to devise new Rowhammer-aware ECC algorithms that can be deployed either in hardware or software [57]. Moreover, to improve the guarantees of new ECC algorithms [58]–[61], we may explicitly augment them with defenses against Rowhammer, either in software [36], [46], [57], [62]–[64] or in hardware—e.g., in the memory controllers or inside the memory chips themselves. As an example, in-DRAM ECC [65]–[68], where the ECC engine resides inside each chip can co-exist with rank-level ECC implemented in the memory controller [68]. The in-DRAM ECC helps to mitigate Rowhammer, while potentially masking the side channel presented in this paper (since the errors are corrected on die).

Another common solution against Rowhammer is to increase the DRAM refresh rate, but doing so wastes power. Also, the current trend in practice is exactly the opposite: manufacturers have started lowering the DRAM refresh rate to save power and relying on ECC for memory integrity [16], [66], [69], [70]. Since lowering the refresh rate dramatically increases the number of Rowhammer bit flips [5], [7], [8], doing so makes it easier to bypass ECC—we believe that it is time to reconsider such strategies in the Rowhammer era.

## X. Conclusion

Rowhammer has evolved into a serious threat to computer systems from the smallest mobile devices to very large clouds, but so far machinery with high-end memory with error correcting code (ECC) has been free from such attacks. This has

been due to the complex challenge of reverse engineering commodity ECC functions and, more importantly, to the narrow margins within which attackers must operate: multiple bits must flip in order to bypass the error correcting functionality, but flipping the wrong number of bits may crash the system. Thus, many believed that Rowhammer on ECC memory, even if plausible in theory, is simply impractical. This paper shows this to be false: while harder, Rowhammer attacks are still a realistic threat even to modern ECC-equipped systems. This is particularly worrying, because all other existing defenses have already been proven insecure. Given the proliferation of Rowhammer vulnerabilities across a broad range of systems, we urgently need better defenses against these attacks.

## ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct) and No. 825377 (UNICORE) as well as by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing", NWO 639.021.753 VENI "PantaRhei", NWO 016.Veni.192.262, and NWO 628.001.005 CYBSEC "OpenSesame". This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

## REFERENCES

[1] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design," in *SIGPLAN '12*.

[2] S. Satoh, Y. Tosaka, and S. Wender, "Geometric effect of multiple-bit soft errors induced by cosmic ray neutrons on DRAM's," IEEE. 2000.

[3] P. McLellan, "We Live on a Radioactive Planet Bombarded by Cosmic Rays," https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/single-event-effects (Accessed on 12/05/2018).

[4] H. Kobayashi, K. Shiraishi, H. Tsuchiya, H. Usuki, Y. Nagai, and K. Takahisa, "Evaluation of LSI soft errors induced by terrestrial cosmic rays and alpha particles," tech. rep., Sony Corporation and RCNP Osaka University, 2001.

[5] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," ISCA'14.

[6] M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," *Black Hat*, 2015.

[7] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup est machina: Memory deduplication as an advanced exploitation vector," in *S&P'16*.

[8] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in Javascript," in *DIMVA'16*.

[9] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *DSN'15*.

[10] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *SEC'16*.

[11] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *CCS'16*.

[12] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *S&P'18*.

[13] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," in *S&P'18*.

[14] "Amazon EC2 uses ECC memory," https://aws.amazon.com/ec2/faqs/#Does_AmazonEC2_use_ECC_memory (Accessed on 13/05/2018).

[15] "Mac Pro (Late 2013): Memory specifications." https://support.apple.com/en-us/HT202892 (Accessed on 13/05/2018).

[16] "ECC Brings Reliability and Power Efficiency to Mobile Devices." https://www.micron.com/resource-details/28c643e4-9f86-49cd-9471-f386a0b812ca (Accessed on 13/05/2018).

[17] N. Kwak, S.-H. Kim, K. H. Lee, C.-K. Baek, M. S. Jang, Y. Joo, S.-H. Lee, W. Y. Lee, E. Lee, D. Han, *et al.*, "23.3 A 4.8 Gb/s/pin 2Gb LPDDR4 SDRAM with sub-100$\mu$A self-refresh current for IoT applications," in *ISSCC'17*, 2017.

[18] "DDR3 SDRAM Unbuffered DIMM Design Specification," *JEDEC Standard*, vol. No. 21C, pp. 4.20.19–1, 2013. (Accessed on 04/24/2018).

[19] "DDR4 SDRAM Registered DIMM Design Specification," *JEDEC Standard*, vol. No. 21C, pp. 4.20.28–1, 2014. (Accessed on 04/24/2018).

[20] "2GB, 4GB, 8GB (x72, ECC, DR) 240-Pin DDR3 UDIMM," https://www.micron.com/~/media/documents/products/data-sheet/modules/unbuffered_dimm/jsf18c256_512_1gx72az.pdf (Accessed on 04/27/2018).

[21] T. J. Dell, "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," in *IBM Microelectron. Div. '97*.

[22] W. Ryan and S. Lin, *Channel Codes: Classical and Modern*. Cambridge University Press, 2009.

[23] R. W. Hamming, "Error detecting and error correcting codes," 1950.

[24] R. C. Bose and D. K. Ray-Chaudhuri, "On a class of error correcting binary group codes," 1960.

[25] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*. 1960.

[26] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, "Future scaling of processor-memory interfaces," in *SC'09*.

[27] "Google Cloud CPU Platforms — Compute Engine Documentation," https://cloud.google.com/compute/docs/cpu-platforms (Accessed on 02/08/2018).

[28] "Amazon EC2 Instance Types – Amazon Web Services (AWS)," https://aws.amazon.com/ec2/instance-types/, (Accessed on 02/08/2018).

[29] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation," in *SEC'16)*.

[30] "Memory Error Injection (MEI) Test Card and Utility ," https://designintools.intel.com/Memory_Error_Injection_MEI_Test_Card_and_Utility_p/stlgrn61.htm (Accessed on 04/18/2018).

[31] M. Gottscho, M. Shoaib, S. Govindan, B. Sharma, D. Wang, and P. Gupta, "Measuring the Impact of Memory Errors on Application Performance," CAL'17.

[32] J. Bauer, M. Gruhn, and F. C. Freiling, "Lest we forget: Cold-boot attacks on scrambled DDR3 memory," Digit. Investig. '16.

[33] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin, "Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors," in *HPCA'17*.

[34] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA'17*.

[35] "Coreboot." https://www.coreboot.org/ (Accessed on 02/05/2018).

[36] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "CAnt Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory," in *SEC'17*.

[37] A. Tatar, K. Razavi, H. Bos, and C. Giuffrida, "Defeating Software Mitigations against Rowhammer: a Surgical Precision Hammer," in *RAID'18*. https://github.com/vusec/hammertime.

[38] A. Kleen, "Machine check handling on Linux," https://www.halobates.de/mce.pdf (2004, SUSE Labs).

[39] "Mcelog – the linux hardware error daemon," https://mcelog.org/ (Accessed on 04/18/2018).

[40] "MCA Enhancements in Intel® Xeon® Processors.book," https://software.intel.com/sites/default/files/managed/d0/d2/329176-mca-enhancements-in-intel-xeon-processors.pdf (Accessed on 04/18/2018).

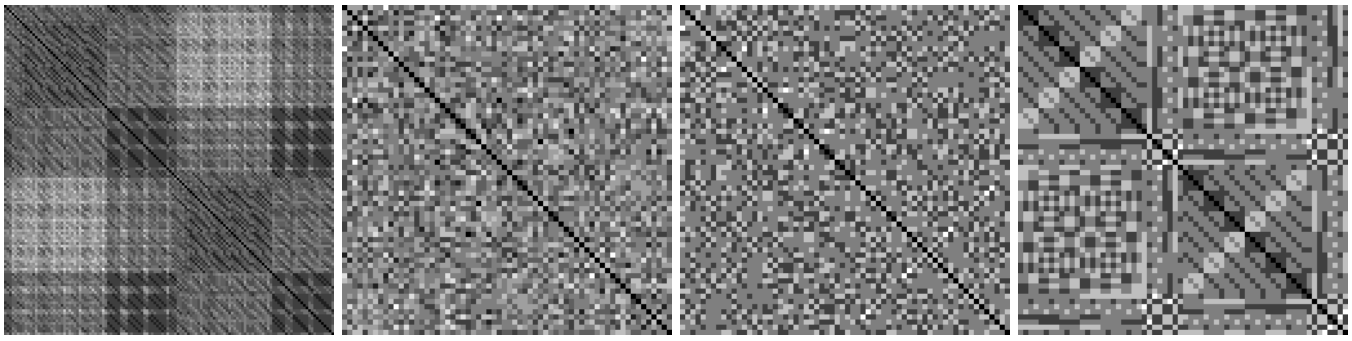
[41] K. S. Bains, J. B. Halbert, C. P. Mozak, T. Z. Schoenborn, and Z. Greenfield, "Row hammer refresh command," 2018. US Patent 9,865,326.

[42] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks.," in *SEC'16*.

[43] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," ACM'78.

[44] P. Erdös and M. Kac, "The Gaussian law of errors in the theory of additive number theoretic functions," Am. J. Math. 1940.
[45] H. W. Lenstra, "Factoring Integers with Elliptic Curves," *Annals of Mathematics*, 1987.
[46] A. Tatar, R. Krishnan, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer Attacks over the Network and Defenses," in *USENIX ATC'18*.
[47] M. Lanteigne, "How rowhammer could be used to exploit weaknesses in computer hardware." http://www.thirdio.com/rowhammer.pdf, 2016.
[48] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *SEC'14*.
[49] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *RSA'06*.
[50] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ Flush: a fast and stealthy cache attack," in *DIMVA'16*.
[51] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," *NDSS'17*.
[52] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering Intel last-level cache complex addressing using performance counters," in *RAID'15*.
[53] M. Jung, C. C. Rheinländer, C. Weis, and N. Wehn, "Reverse Engineering of DRAMs: Row Hammer with Crosshair," MEMSYS '16, ACM.
[54] B. Delgado and K. L. Karavanic, "Performance implications of system management mode," in *IISWC'13*.
[55] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," DSN '14.
[56] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly," SIGARCH'15.
[57] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks," in *OSDI'18*.
[58] A. N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N. P. Jouppi, "LOT-ECC: Localized and Tiered Reliability Mechanisms for Commodity Memory Systems," SIGARCH'12.
[59] D. H. Yoon and M. Erez, "Virtualized and flexible ECC for main memory," in *SIGARCH'10* .
[60] J. Kim, M. Sullivan, and M. Erez, "Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory," in *HPCA'15*.
[61] P. J. Nair, V. Sridharan, and M. K. Qureshi, "XED: Exposing on-die error detection information for strong memory reliability," in *ISCA'16*.
[62] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks," ASPLOS '16, pp. 743–755.
[63] M. Oliverio, K. Razavi, H. Bos, and C. Giuffrida, "Secure page fusion with vusion," in *SOSP'17*.
[64] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "GuardION: Practical mitigation of DMA-based Rowhammer attacks on ARM," in *DIMVA'18*.
[65] S.-H. Kim, W.-O. Lee, J.-H. Kim, S.-S. Lee, S.-Y. Hwang, C.-I. Kim, T.-W. Kwon, B.-S. Han, S.-K. Cho, D.-H. Kim, and others, "A low power and highly reliable 400Mbps mobile DDR SDRAM with on-chip distributed ECC," in *ASSCC'07*.
[66] U. Kang, H.-s. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. S. Choi, "Co-architecting controllers and DRAM to enhance DRAM process scaling," in *The Memory Forum'14*.
[67] T. Y. Oh, H. Chung, Y. C. Cho, J. W. Ryu, K. Lee, C. Lee, J. I. Lee, H. J. Kim, M. S. Jang, G. H. Han, K. Kim, D. Moon, S. Bae, J. Y. Park, K. S. Ha, J. Lee, S. Y. Doo, J. B. Shin, C. H. Shin, K. Oh, D. Hwang, T. Jang, C. Park, K. Park, J. B. Lee, and J. S. Choi, "25.1 A 3.2Gb/s/pin 8Gb 1.0V LPDDR4 SDRAM with integrated ECC engine for sub-1V DRAM core operation," in *ISSCC'14*.
[68] S. Cha, O. Seongil, H. Shin, S. Hwang, K. Park, S. J. Jang, J. S. Choi, G. Y. Jin, Y. H. Son, H. Cho, and others, "Defect analysis and cost-effective resilience architecture for future DRAM devices," in *HPCA'17*.
[69] C. Chou, P. Nair, and M. K. Qureshi, "Reducing refresh power in mobile devices with morphable ECC," in *DSN'15*.
[70] D. A. Klein and J. Schreck, "Memory system and method using ECC to achieve low power refresh," US Patent 7,184,352.
[71] "Encoder and decoder for an SEC-DED-S4ED rotational code." US Patent 5,856,987.

## APPENDIX



(a) AMD-1 (b) Intel-2 and Intel-3 (c) Intel-1 (d) SEC-DED-S4ED [71]

Fig. 7: Hamming Distance (HD) of ECC function results.

To quickly visualize the ECC properties, In Figure 7 we show the HD of the various ECC algorithms that we recovered. A pixel of coordinate $x$, $y$ has a brightness level of the HD between the ECC result of $data_x$ and $data_y$. Where $data_i$ means that bit on position $i$ is asserted and all the others are de-asserted. A black pixel (lowest brightness and HD) means that the ECC are the same. On AMD-1 (Figure 7a we observe a distinct pattern at 8 bits intervals. This is expected, as the ECC algorithm treats 8 bits as a single symbol. Repetitions are also observed in Figure 7d at 4 bits. This implementation corresponds to an Intel patent [71] which can detect up to 4 bits (SEC-DED-S4ED). These patterns are not always obvious, for example Intel-1 (Figure 7c) uses the same ECC algorithm (and values) but the bits are considered in a different order.

---

```
ECCIntelHaswell                 ECCAmdFam10h

000 11110001 032 00100011       000 1000000000010100  032 1000000000100111 064 0001010010000000 096 0010011110000000
001 01000011 033 01100010       001 0100000000001010  033 0100000101011011 065 0000101001000000 097 1010101101000000
002 01101000 034 10010001       002 0010000000000101  034 0010000011101101 066 0000010100100000 098 1110110100100000
003 10010100 035 00001011       003 0001000010111010  035 0001000011001110 067 1011101000010000 099 1100111000010000
004 00001110 036 11111000       004 0000100001011101  036 0000100001100111 068 0101110100001000 100 0110011100001000
005 01010001 037 01010100       005 0000010010010110  037 0000010010001011 069 1001011000000100 101 1000101100000100
006 10100010 038 10101000       006 0000001001001011  038 0000001011111101 070 0100101100000010 102 1111110100000010
007 00011100 039 01001100       007 0000000110011101  039 0000000111000110 071 1001110100000001 103 1100011000000001
008 00011111 040 00110010       008 1000000011101101  040 1000000000010000 072 1110110110000000 104 0001000010000000
009 00110100 041 00100110       009 0100000011001110  041 0100000000001000 073 1100111001000000 105 0000100001000000
010 10000110 042 00011001       010 0010000001100111  042 0010000000000100 074 0110011100100000 106 0000010000100000
011 01001001 043 10110000       011 0001000010001011  043 0001000000000010 075 1000101100010000 107 0000001000010000
012 11100000 044 10001111       012 0000100011111101  044 0000100000000001 076 1111110100001000 108 0000000100001000
013 00010101 045 01000101       013 0000010011000110  045 0000010010111000 077 1100011000000100 109 1011100000000100
014 00101010 046 10001010       014 0000001001100011  046 0000001001011100 078 0110001100000010 110 0101110000000010
015 11000001 047 11000100       015 0000000110001001  047 0000000100101110 079 1000100100000001 111 0010111000000001
016 00101111 048 00010011       016 1000000001001110  048 1000000000100000 080 0100111010000000 112 0010000010000000
017 10000011 049 01100100       017 0100000000100111  049 0100000000010000 081 0010011101000000 113 0001000001000000
018 01100001 050 10010010       018 0010000010101011  050 0010000000001000 082 1010101100100000 114 0000100000100000
019 10011000 051 01110000       019 0001000011101101  051 0001000000000100 083 1110110100010000 115 0000010000010000
020 11010000 052 01001111       020 0000100011001110  052 0000100000000010 084 1100111000001000 116 0000001000001000
021 01010010 053 01011000       021 0000010001100111  053 0000010000000001 085 0110011100000100 117 0000000100000100
022 10100100 054 10100001       022 0000001010001011  054 0000001010111000 086 1000101100000010 118 1011100000000010
023 00101100 055 10001100       023 0000000111111101  055 0000000101011100 087 1111110100000001 119 0101110000000001
024 11110010 056 00110001       024 1000000010101011  056 1000000001000000 088 1010101110000000 120 0100000010000000
025 00111000 057 01000110       025 0100000011101101  057 0100000000100000 089 1110110101000000 121 0010000001000000
026 00010110 058 00101001       026 0010000011001110  058 0010000000010000 090 1100111000100000 122 0001000000100000
027 10001001 059 00000111       027 0001000001100111  059 0001000000001000 091 0110011100010000 123 0000100000010000
028 00001101 060 11110100       028 0000100010001011  060 0000100000000100 092 1000101100001000 124 0000010000001000
029 00100101 061 10000101       029 0000010011111101  061 0000010000000010 093 1111110100000100 125 0000001000000100
030 01001010 062 00011010       030 0000001011001110  062 0000001000000001 094 1100011000000010 126 0000000100000010
031 11000010 063 11001000       031 0000000101100011  063 0000000110111000 095 0110001100000001 127 1011100000000001
```

---

Fig. 8: Recovered parity matrices (Intel-1 and AMD-1 respectively).

```
ECCIntelSandy64

000 10011101 032 10000000
001 01010000 033 11101101
002 10110010 034 01011101
003 01110001 035 00011100
004 00001101 036 01100100
005 01110000 037 11011000
006 00110101 038 00000101
007 11110111 039 10111011
008 01100010 040 11010000
009 10111001 041 10010100
010 01010000 042 10111001
011 11011010 043 11011111
012 01010001 044 10001010
013 01111000 045 11110001
014 10011000 046 01001011
015 11011110 047 11100000
016 10100011 048 10111101
017 10011011 049 01000000
018 01001101 050 00011011
019 01110000 051 11101001
020 10110101 052 01000010
021 01010101 053 00101010
022 01011111 054 00110001
023 01111010 055 00011010
024 10001010 056 11000111
025 10010010 057 11010110
026 00000101 058 10101111
027 01111110 059 01001011
028 00001010 060 11000101
029 01110011 061 01011010
030 01001101 062 10101110
031 11111000 063 11001100
```

Fig. 9: Recovered parity matrix with cold-boot attack on Intel-2 and Intel-3.

**Hardware details.** The *Intel-1* setup uses the Intel Xeon E3-1270 v3 CPU built on the Haswell microarchitecture and a Supermicro X10SLL-F motherboard (BIOS version: 3.0a). Setup *AMD-1* contains the AMD Opteron 6376 CPU that is part of the Bulldozer Family 15h microarchitecture. This CPU was mounted on the Supermicro H8SGL-F motherboard with the BIOS: 5.925, version: 3.5a). *Intel-2* is the HP Proliant DL360p Gen8 Server that uses the Intel Xeon E5-2650 v1 (Sandy Bridge) CPU with default configuration of BIOS (version P71). *Intel-3* is the SuperServer 1026GT that uses the Intel Xeon E5-2620 v1 CPU (Sandy Bridge) and a Supermicro X9DRG-HF motherboard with BIOS version 1.0c.

In our experiments we tested several memory modules from different manufacturers. We confirm a significant amount of Rowhammer bit flips in a DIMM similar to the one on which Brasser et al. [36] reported the highest successful exploitation rate. As Rowhammer is a fundamental architecture issue, many other combinations of CPU (memory controllers) and memory modules are susceptible to this class of attacks. We stress that the configurations that we mention here represents just some arbitrary setups that we came accross in our research and we do not blame one manufacturer or another.

**Disclosure.** We disclosed our findings to the affected parties. *CVE-2018-18904* tracks the timing side-channel of the error correction. Information about operating systems' drivers of several Linux distribution can be found in *CVE-2018-18905* and in *CVE-2018-18906*.