RAMBleed: Reading Bits in Memory Without Accessing Them

Andrew Kwong
University of Michigan
ankwong@umich.edu

Daniel Genkin
University of Michigan
genkin@umich.edu

Daniel Gruss
Graz University of Technology
daniel.gruss@iaik.tugraz.at

Yuval Yarom
University of Adelaide and Data61
yval@cs.adelaide.edu.au

***Abstract*—The Rowhammer bug is a reliability issue in DRAM cells that can enable an unprivileged adversary to flip the values of bits in neighboring rows on the memory module. Previous work has exploited this for various types of fault attacks across security boundaries, where the attacker flips inaccessible bits, often resulting in privilege escalation. It is widely assumed however, that bit flips within the adversary's own private memory have no security implications, as the attacker can already modify its private memory via regular write operations.**

We demonstrate that this assumption is incorrect, by employing Rowhammer as a *read* side channel. More specifically, we show how an unprivileged attacker can exploit the data dependence between Rowhammer induced bit flips and the bits in nearby rows to deduce these bits, including values belonging to other processes and the kernel. Thus, the primary contribution of this work is to show that Rowhammer is a threat to not only integrity, but to confidentiality as well.

Furthermore, in contrast to Rowhammer write side channels, which require persistent bit flips, our read channel succeeds even when ECC memory detects and corrects every bit flip. Thus, we demonstrate the first security implication of successfully-corrected bit flips, which were previously considered benign.

To demonstrate the implications of this read side channel, we present an end-to-end attack on OpenSSH 7.9 that extracts an RSA-2048 key from the root level SSH daemon. To accomplish this, we develop novel techniques for massaging memory from user space into an exploitable state, and use the DRAM row-buffer timing side channel to locate physically contiguous memory necessary for double-sided Rowhammering. Unlike previous Rowhammer attacks, our attack does not require the use of huge pages, and it works on Ubuntu Linux under its default configuration settings.

***Index Terms*—Side channels, Rowhammer, OpenSSH**

I. Introduction

In recent years, the discrepancy between the abstract model used to reason about computers and their actual hardware implementation has lead to a myriad of security issues. These range from microarchitectural attacks [15] that exploit contention on internal components to leak information such as cryptographic keys or keystroke timing [18, 45, 65], through transient execution attacks [10, 35, 39, 60, 63] that break down fundamental OS isolation guarantees, to memory integrity attacks [9, 32, 34, 36] that exploit hardware limitations to change the contents of data stored in the device.

Rowhammer [19, 34, 55] is a fault attack, in which the attacker uses a specific sequence of memory accesses that results in bit flips, i.e., changes in bit values, in locations *other* than those accessed. Because the attacker does not directly access the changed memory location, the change is not visible to the processor or the operating system, and is not subject to any permission checks. Thus far, this ability to reliably flip bits across security boundaries has been exploited for sandbox escapes [19, 55], privilege escalation attacks on operating systems and hypervisors [19, 21, 51, 55, 61, 64], denial-of-service attacks [21, 28], and even for fault injection in cryptographic protocols [6].

A common theme for all past Rowhammer attacks is that they break memory *integrity*. Namely, the attacker uses Rowhammer to obtain a (limited) write primitive into otherwise inaccessible memory, and subsequently modifies the contents of that memory in a way that aligns with the attacker's goals. This observation has led to various mitigation proposals designed to secure the target's memory by using integrity checks [62], or by employing ECC (error-correcting code) memory to ensure memory integrity. The latter, in particular, has long been touted as a defense against Rowhammer-based attacks. Even when an attacker flips a bit in memory, the ECC mechanism corrects the error, halting the attack. While recent work has demonstrated that an attacker can defeat the ECC mechanism, resulting in observable bit-flips after error correction [13], successfully corrected flips are still considered benign, without any security implications. Thus, in this paper we pose the following questions:

- *Is the threat posed by Rowhammer limited only to memory integrity and, in particular, can the Rowhammer effect be exploited for breaching confidentiality?*
- *What are the security implications of* corrected *bit flips? Can an attacker use Rowhammer to breach confidentiality even when ECC memory corrects all flipped bits?*

A. Our Contributions

In this paper, we answer these questions in the affirmative. More specifically, we present *RAMBleed*, a new Rowhammer-based attack that breaks memory confidentiality guarantees by acquiring secret information from other processes running on the same hardware. Remarkably, RAMBleed can break memory confidentiality of ECC memory, even if all bit flips are successfully corrected by the ECC mechanism. After profiling the target's memory, we show how RAMBleed can leak secrets stored within the target's physical memory, achieving a read speed of about 3–4 bits per second. Finally, we demonstrate

the threat posed by RAMBleed by recovering an RSA 2048-bit signing key from an OpenSSH server using only user level permissions.

Data-Dependent Bit Flips. The main observation behind RAMBleed is that bit flips depend not only on the bit's *orientation*, i.e., whether it flips from 1 to 0 or from 0 to 1, but also on the values of neighboring bits [34]. Specifically, true bits tend to flip from 1 to 0 when the bits above and below them are 0, but not when the bits above and below them are 1. Similarly, anti bits tend to flip from 0 to 1 when the bits above and below them are 1, but not when the bits above and below them are 0. While this observation dates back to the very first Rowhammer paper [34], we show how attackers can use it to obtain a read primitive, thereby learning the values of nearby bits which they might not be allowed to access.

RAMBleed Overview. Suppose an attacker wants to determine the value of a bit in a victim's secret. The attacker first *templates* the computer memory to find a flippable bit at the same offset in a memory page as the secret bit. (For the rest of the discussion we assume a true bit, i.e., one that flips from 1 to 0.) The attacker then manipulates the memory layout to achieve the arrangement depicted below:

Row Activation Page	Secret
Unused	Sampling Page
Row Activation Page	Secret

Here, each memory row spans two memory pages of size 4 KiB. The attacker uses the *Row Activation* pages for hammering, the *Sampling* page contains the flippable bit, which is initialized to 1, and *Secret* pages contain the secret victim data that the attacker aims to learn. If the value of the secret bit is 0, the layout results in a flippable 0-1-0 configuration, i.e., the flippable bit is set to 1, and the bits directly above and below it are 0. Otherwise, the secret bit is 1, resulting in a 1-1-1 configuration, which is not flippable.

Next, the attacker repeatedly accesses the two activation pages she controls (left top and bottom rows), thereby hammering the middle row. Because the Rowhammer effects are data dependent, this hammering induces a bit flip in the sampling page in the case that the secret bit is 0. The attacker then accesses the sampling page directly, checking for a bit flip. If the bit has flipped, the attacker deduces that the value of the secret bit is 0. Otherwise, the attacker deduces that the value is 1. Repeating the procedure with flippable bits at different offsets in the page allows the attacker to recover all of the bits of the victim's secret.

We note here that neither the victim nor the attacker access the secrets in any way. Instead, by accessing the attacker-controlled row activation pages, the attacker uses the victim's data to influence Rowhammer-induced bit flips in her own private pages. Finally, the attacker directly checks the sampling page for bit flips, thereby deducing the victim's bits. As such, RAMBleed is a cross address space attack.

ECC Memory. ECC memory has traditionally been considered an effective defense against Rowhammer-based attacks. Even when an attacker flips a bit in memory, the hardware's ECC mechanisms simply revert back any Rowhammer-induced bit flips. However, recent work has demonstrated that an attacker can defeat the ECC mechanism by inducing enough carefully-placed flips in a single codeword, resulting in observable bit-flips after error correction [13].

In this paper, however, we show that even ECC-corrected bit flips may have security implications. This is because RAMBleed does not necessarily require the attacker to read the bit to determine if it has flipped. Instead, all the attacker requires for mounting RAMBleed is an indication that a bit in the sampling page has flipped (and subsequently corrected). Unfortunately, as Cojocar et al. [13] show, the synchronous nature of the ECC correction algorithm typically exposes such information through a timing channel, where memory accesses that require error correction are measurably slower than normal accesses.

Thus, we can exploit Rowhammer-induced timing variation to read data even from ECC memory. In particular, our work is the first to highlight the security implications of successfully corrected flips, hitherto considered to be benign.

Memory Massaging. One of the main challenges for mounting RAMBleed, and Rowhammer-based attacks in general, is achieving the required data layout in memory. Past approaches rely on one or more mechanisms which we now describe. The first practical Rowhammer attack relied on operating system interfaces (e.g., `/proc/pid/pagemap` in Linux) to perform virtual-to-physical address translation for user processes [55]. Later attacks leveraged huge pages, which give access to large chunks of consecutive physical memory [19], thereby providing sufficient information about the physical addresses to mount an attack. Other attacks utilized memory grooming or massaging techniques [61], which prepare memory allocators such that the target page is placed at the attacker-chosen physical memory location with a high probability. An alternative approach is exploiting memory deduplication [7, 51], which merges physical pages with the same contents. The attacker then hammers its shared read-only page, which is mapped to the same physical memory location as the target page.

However, many of these mechanisms are no longer available for security reasons [42, 52, 57, 61]. Thus, as a secondary contribution of this paper, we present a new approach for *massaging* memory to achieve the desired placement. Our approach builds on past works that exploit the Linux buddy allocator to allocate blocks of consecutive physical memory [11, 61]. We extend these works by demonstrating how an attacker can acquire some physical address bits from the allocated memory. We further show how to place secret-containing pages at desired locations in the physical memory.

Finally, we note that this method may have independent value for mounting Prime+Probe last-level cache attacks [40]. This is since it allows the attacker to deduce physical addresses of memory regions, thereby aiding eviction set construction.

Extracting Cryptographic Keys. To demonstrate the effectiveness of RAMBleed, we use it to leak secrets across process boundaries. Specifically, we use RAMBleed against an OpenSSH 7.9 server (newest version at time of writ-

ing), and successfully read the bits of an RSA-2048 key at a rate of 0.3 bits per second, with 82% accuracy. We combine the attack with a variant of the Heninger-Shacham algorithm [23, 24, 46] designed to recover RSA keys from partial information, achieving complete key recovery.

Summary of Contributions. In this paper we make the following contributions:

- We demonstrate the first Rowhammer attack that breaches confidentiality, rather than integrity (Section IV).
- We abuse the Linux buddy allocator to allocate a large block of consecutive *physical* addresses, and show how to recover some of the physical address bits (Section V-A).
- We design a new mechanism, which we call *Frame Feng Shui*, for placing victim program pages at a desired location in the physical memory (Section V-C).
- We demonstrate a Rowhammer-based attack that leaks keys from OpenSSH while only flipping bits in memory locations the attacker is allowed to modify (Section VII).
- Finally, we demonstrate RAMBleed against ECC memory, highlighting security implications of successfully-corrected Rowhammer-induced bit flips (Section VIII).

B. Responsible Disclosure

Following the practice of responsible disclosure, we have notified Intel, AMD, OpenSSH, Microsoft, Apple, and Red Hat about our findings. The results contained in this paper (and in particular our memory massaging technique) were assigned CVE-2019-0174 by Intel.

C. Related Works

Security Implications of Rowhammer. The potential for sporadic bit flips was well known in the DRAM manufacturing industry, but was considered a reliability issue rather than a security threat. Kim et al. [34] were the first to demonstrate a reliable method for inducing bit flips by repeatedly accessing pairs of rows in the same bank. Subsequently, Seaborn and Dullien [55] showed that Rowhammer is a security concern by using Rowhammer-induced flips to break out of Chrome's Native Client sandbox [67] and obtain root privileges.

Since the initial Rowhammer-based exploit of [55], researchers have demonstrated numerous other avenues for Rowhammer exploitation. Gruss et al. [19] demonstrated that page-table bits can be flipped via Rowhammer from JavaScript, while Bosman et al. [7] flipped the types of JavaScript objects through the browser. Aweke et al. [2] also demonstrated Rowhammer flips without the use of CLFLUSH, and with a halved DRAM refresh interval. Van Der Veen et al. [61] used Rowhammer to gain root on mobile phones, while Lipp et al. [38] and Tatar et al. [59] used network requests to induce Rowhammer flips via a completely remote attack. Frigo et al. [14] managed to induce bit flips from the browser's interface to the GPU. ECC memory was shown to be vulnerable to Rowhammer by Cojocar et al. [13].

Lou et al. [41] systematically categorize Rowhammer attacks in a framework to better understand the problem and uncover new types of Rowhammer attacks. Their methodology, however, is limited and completely ignores the possibility of using Rowhammer as a read side channel.

Defenses. Various defenses have been proposed for Rowhammer attacks, aiming to detect ongoing attacks [2, 12, 20, 27, 47, 69], neutralize the effect of bits being flipped [19, 61], or eliminate the possibility of Rowhammer bit flips in the first place [8, 31, 33, 34].

II. Background

This section provides the necessary background on DRAM architecture, the row-buffer timing side channel described by Pessl et al. [49], and the Rowhammer bug. We begin by briefly overviewing DRAM organization and hierarchy.

A. DRAM Organization

DRAM Hierarchy. DRAM (dynamic random access memory) is organized in a hierarchy of cells, banks, ranks, and DIMMs, which are connected to one or more channels.

More specifically, at the lowest level DRAM stores bits in units called *cells*, each consisting of a capacitor paired with a transistor. The charge on the capacitor determines the value of the bit stored in the cell, while the transistor is used to access the stored value. For *true cells*, a fully charged capacitor represents a '1' and a discharged capacitor represents a '0' while the opposite holds true for *anti cells*.

Memory cells are arranged in a grid of rows and columns called a *bank*. Cells in each row are connected via a *word line*, while cells in each column are connected across *bit lines*. Banks are then grouped together to form a *rank*, which often corresponds to one side of a DIMM. Each DIMM is inserted, possibly with other DIMMs, into a single *channel*, which is a physical connection to the CPU's memory controller.

DRAM Operation. Access to a DRAM bank operates at a resolution of a row, typically consisting of 65536 cells, or 8 KiB. To *activate* a row, the memory controller raises the word line for the row. This produces minute currents on the bit lines, which depend on the charge in the cells of the active row. *Sense amplifiers* capture these currents at each column and amplify the signal to both copy the logical value of the cell into a latch and refresh the charge in the active row. Data can then be transferred between the CPU and the *row buffer*, which consists of the latches that store the values of the cells in the active row.

Over time, the charge in the cell capacitors in DRAM leaks. To prevent data loss through leakage, the charges need to be refreshed periodically. Refreshing is handled by the memory controller, that ensures that each row is opened at least once every *refresh interval*, which is generally 64 ms [30] for DDR3 and DDR4. LPDDR4 defines temperature-dependent adaptations for the refresh interval [29].

DRAM Addressing. Modern memory controllers use a complex function to map a physical address to the correct physical location in memory (i.e., to a specific channel, DIMM, rank, bank, row, and column). While these functions are proprietary and undocumented for Intel processors, they can be reverse

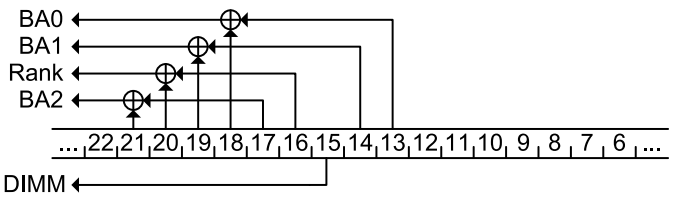
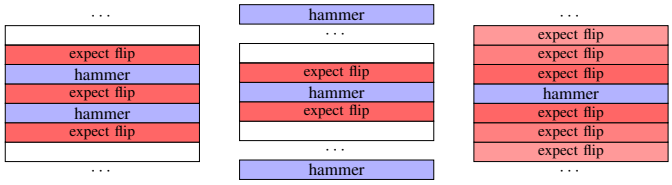


Fig. 1: Reverse engineered DDR3 single channel mapping (2 DIMM per channel) for Ivy Bridge / Haswell (from [49]).



(a) Double-sided (b) Single-sided (c) One-location

Fig. 2: Different hammering techniques as presented by [21].

engineered through both software- and hardware-based techniques [49]. For example, Section II-A shows the DRAM mapping for a typical configuration found in Ivy Bridge and Haswell systems. As the figure shows, the bank and the rank are determined based on bits 13–21 of the physical address. We have verified that the mapping matches the Haswell processor we use in our experiments.

Row Addressing. As discussed above, DRAM rows have a fixed size of typically 8 KiB. However, from the implementation side, it is usually more important to know what amount of memory has the same row index. This is sometimes referred to as same-row [19, 55]. If the address goes to the same row and the same bank, it is called *same-row same-bank*; if it goes to different banks but has the same row index, it is called *same-row different-bank* [55].

In our experimental setup, we have a total of 32 DRAM banks, and thus an aligned block of 256 KiB $= 2^{18}$ B of memory has the same row index. In other words, the row index on our system is directly determined by bits 18 and above of the physical address. Pessl et al. [49] provide a more extensive discussion.

B. Row-Buffer Timing Side Channel

Opening a row and loading its contents into the row buffer results in a measurable latency. Even more so, repeatedly alternating accesses to two uncached memory locations will be significantly slower if these two memory locations happen to be mapped to different rows of the same bank [49]. In Section V, we use this timing difference to identify virtual addresses whose contents lie within the same bank, and also uncover the lower 22 physical addressing bits, thereby enabling double-sided Rowhammer attacks.

C. Rowhammer

The trend towards increasing DRAM cell density and decreasing capacitor size over the past decades has given rise to a reliability issue known as *Rowhammer*. Specifically, repeated accesses to rows in DRAM can lead to bit flips in neighboring rows (not only the direct neighbors), even if these neighboring rows are not accessed [34].

The Root Cause of Rowhammer. Due to the proximity of word lines in DRAMs, when a word line is activated, crosstalk effects on neighboring rows result in partial activation, which leads to increased charge leakage from cells in neighboring rows. Consequently, when a row is repeatedly opened, some cells lose enough charge before being refreshed to drop to an uncharged state, resulting in bit flips in memory.

Performing Uncached Memory Accesses. A central requirement for triggering Rowhammer bit flips is the capability to make the memory controller open and close DRAM rows rapidly. For this, the adversary needs to generate a sequence of memory accesses to alternating DRAM rows that bypass the CPU cache. Several approaches have been suggested for bypassing the cache.

- **Manually Flush Cache Lines.** The x86 instruction set provides the CLFLUSH instruction, which flushes the cache line containing its destination address from all of the levels of the cache hierarchy. Crucially, CLFLUSH only requires read access to the flushed address, facilitating Rowhammer attacks from unprivileged user-level code. On ARM platforms, prior to ARMv8, the equivalent cache line flush instruction could only be executed in kernel mode; ARMv8 does, however, offers operating systems the option to enable an unprivileged cache line flush operation.
- **Cache Eviction.** In cases where the CLFLUSH instruction is not available (e.g. in the browser), an attacker can force contention on cache sets to cause cache eviction [2, 19].
- **Uncached DMA Memory.** Van Der Veen et al. [61] report that the cache eviction method above is not fast enough to cause bit flips on contemporary ARM-based smartphones. Instead, they used the Android ION feature to allocate uncacheable memory to unprivileged userspace applications.
- **Non-temporal instructions.** Non-temporal load and store instructions direct the CPU not to cache their results. Avoiding caching means that subsequent accesses to the same address bypass the cache and are served from memory [50].

Another important distinction between Rowhammer attacks is the strategy in which DRAM rows are activated, i.e., how aggressor rows are selected. See Figure 2.

Double-sided Rowhammer. The highest amount of Rowhammer-induced bit flips occur when the attacker *hammers*, that is repeatedly opens and closes, the two rows adjacent to a target row. This approach maximizes the number of neighboring row activations, and consequently the charge leakage from the target row (Figure 2a). However, for double-sided hammering, the attacker needs to locate addresses in the two adjacent rows, which may be difficult without knowledge of the physical addresses and their mapping to rows. Previous attacks exploited the Linux *pagemap* interface, which maps virtual to physical addresses. However, to mitigate the Seaborn

and Dullien [55] attack, recent versions of Linux only allow root access to the pagemap interface.

Another avenue used by previous works for finding adjacent rows is to use huge pages, e.g., transparent huge pages (THP), to obtain large blocks of physically contiguous memory [19].

Single-sided Rowhammer. To avoid the need for finding the two rows adjacent to the target row, an adversary can take a more opportunistic approach, which aims to cause bit flips in any row in memory (Figure 2b). This can be achieved by guessing several addresses at random, e.g., 8 addresses, in the hope that some fall within two rows in the same bank. With B banks, the probability of having at least one such a pair is $1 - \prod_{i=1}^{n} \frac{B-i}{B}$, i.e., 61.4% for 8 addresses and 32 banks.

Alternatively, the adversary can take a more disciplined approach and use the row-buffer timing channel (Section II-B) to identify rows in the same bank [6, 61].

Because only one of the rows being hammered is located near the target row, single-sided Rowhammer results in fewer bit flips than double-sided Rowhammer [2].

One-location Rowhammer. Finally, one-location hammering [21], is the least restrictive strategy, but also generates the fewest number of bit flips (Figure 2c). Here, the attacker repeatedly flushes and then reads from a single row. The presumed cause of flips, in this case, is that newer memory controller policies automatically close DRAM rows after a small amount of time. This obviates the need to open different rows in the same bank.

D. RSA Background

As the end-to-end attack described in this paper recovers RSA private keys from an OpenSSH server, we now briefly overview the RSA [53] cryptosystem and signature scheme.

A user creates an RSA key pair by first generating two random primes, p and q, a public exponent e, and a private exponent d such that $e \cdot d \equiv 1 \pmod{(p-1)(q-1)}$. The public key is then set to be (e, N) where $N = pq$, and the private key is set to be (d, N). To sign a message m, the signer uses its private key to compute $\sigma \leftarrow z^d \bmod N$, where z is a collision resistant hashing of m. To verify a signature σ, the verifier first hashes the message by herself and obtains a digest z'. She then computes $z'' \leftarrow \sigma^e \bmod N$ using the public key and verifies that $z' = z''$, and rejects the signature otherwise.

The Chinese Remainder Theorem. A common optimization used by most applications to compute $\sigma \leftarrow z^d \bmod N$ is the Chinese Remainder Theorem (CRT). Here, the private key is first augmented with $d_p \leftarrow d \bmod (p-1)$ and $d_q \leftarrow d \bmod (q-1)$. Next, instead of computing $z^d \bmod N$ directly, the signer computes $\sigma_p \leftarrow z^{d_p} \bmod p$ and $\sigma_q \leftarrow z^{d_q} \bmod q$. Finally, the signer computes σ from σ_p and σ_q using the CRT.

Partial Key Recovery. Cryptographic keys recovered through a side channel are typically subject to some measure of noise. Often only a fraction of the key bits are recovered, and their values are not known with certainty. Various researchers have exploited the redundancy present in private key material to correct the errors [5, 25, 44, 46, 48, 66].

III. Threat Model

We assume an attacker that runs unprivileged software within the same operating system (OS) as the victim software. The OS maintains isolation between the victim program and the attacker. In particular, we assume that the OS works correctly. We further assume that the attacker cannot exploit microarchitectural side channel leakage from the victim, either because the victim does not leak over microarchitectural channels or because the OS enforces time isolation [16]. We do assume that the machine is vulnerable to the Rowhammer attack. However, we assume that the attacker only changes its own private memory to bypass any countermeasures and detection mechanisms. Finally, we assume that the attacker is able to somehow trigger the victim to perform allocations of secret data (for example using an incoming SSH connections for the OpenSSH attack in Section VII).

IV. RAMBleed

Previous research mostly considers Rowhammer as a threat to data integrity, allowing an unprivileged attacker to modify data without accessing it. With RAMBleed, however, we show that Rowhammer effects also have implications on data confidentiality, allowing an unprivileged attacker to leverage Rowhammer-induced bit flips in order to *read* the value of neighboring bits. Furthermore, as not every bit in DRAM can be flipped via Rowhammer, we also present novel memory massaging techniques that aim to locate and subsequently exploit Rowhammer flippable bits. This enables the attacker to read otherwise inaccessible information such as secret key bits. Finally, as our techniques only require the attacker to allocate and deallocate memory and to measure instruction timings, RAMBleed allows an unprivileged attacker to read secret data using the default configuration of many systems (e.g., Ubuntu Linux), without requiring any special configurations (e.g., access to pagemap, huge pages, or memory deduplication).

A. The Root Cause of RAMBleed.

RAMBleed exploits a physical phenomenon in DRAM DIMMs wherein the likelihood of a Rowhammer induced bit flip depends on the values of the bits immediately above and below it. Bits only flip when the bits both immediately above and below them are in their discharged state [13]. This is in agreement with observations by Kim et al. [34] that hammering with a striped pattern, where rows alternate between all zeros and all ones, generates many more flips than with a uniform pattern.

Data-Dependent Bit Flips. Put simply, bits tend to flip to the same value of the bits in the adjacent rows. That is, a charged cell is most likely to flip when it is surrounded by uncharged cells. This is likely due to capacitors of opposite charges inducing parasitic currents in one another, which cause the capacitors to leak charge more quickly [3]. For our attack to work, is it also crucial that bit flips are influenced only by bits in the same column, and not by the neighboring bits within the same row. This isolation is what allows us to deduce one

bit at a time. Cojocar et al. [13] experimentally demonstrate this to be the case.

A Toy Example. To illustrate with a concrete example, we introduce the notation of an x-y-z configuration to describe the situation in which three adjacent bits in the same column have the values x, y, and z, respectively, where $x, y, z \in \{0, 1\}$. The key reasoning behind our attack is as follows.

- **True Cells.** For cells where a one-valued bit is represented as the cell being *charged*, the 0-1-0 configuration is the most likely to flip, changing to an all zero configuration (0-0-0) when rows of the first and the last zero-valued cells are hammered. In this case, the surrounding zero-bits in the aggressor rows *enable* the bit flip in the victim row.
- **Anti Cells.** For cells where a one-valued bit is represented by an *uncharged* cell, a 1-0-1 configuration is more likely to flip and change to an all one configuration (1-1-1) when rows of the first and the last one-valued cells are hammered.

Notation. We adopt Cocojar et al.'s [13] terminology of calling 0-1-0 and 1-0-1 configurations "stripe" patterns, and naming 1-1-1 and 0-0-0 configurations "uniform" patterns. Given this data dependency, we now proceed to build a read side channel in which we read the bits in surrounding rows by observing flips, or lack thereof, in the attacker's row.

B. Memory Scrambling

One potential obstacle to building our read channel is that modern memory controllers employ *memory scrambling*, which is designed to avoid circuit damage due to resonant frequency [68] as well as to serve as a mitigation to cold-boot attacks [22]. Memory scrambling applies a weak stream cipher to the data prior to sending it to the DRAM. That is, the memory scrambler XORs the data with the output of a pseudo-random number generator (PRNG). The seed for the PRNG depends on the physical address of the data and on a random number generated at boot time [26, 43]. The PRNG is cryptographically weak, and given access to the physical data in the DRAMs, an adversary can reverse engineer it and recover the contents of the memory [4, 68].

Bypassing Memory Scrambling. Under our threat model we cannot use the techniques of Yitbarek et al. [68], as we do not assume physical access. However, we can take advantage of the weaknesses of the PRNG. In particular, The boot-time random seed is identical for all rows, and the physical address bits included in the seed are such that several adjacent rows can have the same bits in their addresses. Thus, adjacent rows typically use the same seed, and have the same mask applied. Applying the same mask across multiple rows means that adjacent bits either remain unchanged or are all inverted. Either way, as observed by [13], striped configurations remain striped after scrambling. Hence, writing a striped configuration to memory results in a striped configuration appearing in the DIMM, maintaining the crucial property that a bit will only flip if the bits immediately above and below have the opposite value.

C. Exploiting Data-Dependent Bit Flips

We now show how to exploit the data-dependent bit flips presented above to read data without accessing it.

A Leaky Memory Layout. We begin by considering the memory layout presented in Figure 3a, where every DRAM row contains two 4 KiB pages. In this layout, we assume that A0, A1, and A2 are the attacker-controlled pages containing known data, S is a page with the victim's secret, and R0 is an arbitrary page. All three rows reside in the same bank. Next, note that attacker pages A0 and A2 reside in the same rows as the copies of S. Since DRAM row-buffers operate at an 8 KiB granularity, accessing a value in A0 activates the entire first row, including the page containing the secret S. Similarly, accessing a value in A2 activates the entire third row, again including the page that contains S. Thus, by repeatedly accessing A0 and A2, the attacker can indirectly use the victim pages containing S for hammering, despite not having any permissions to access them.

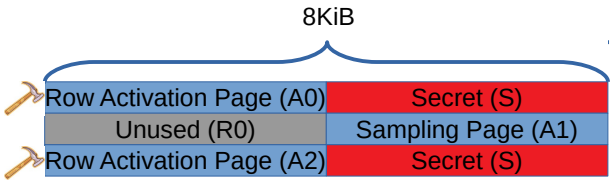
Hammering. By hammering the attacker-controlled pages A0 and A2, the attacker induces analog disturbance and interaction between S and A1. Page A1 also belongs to the attacker, who can therefore detect bit flips in it. From these bit flips, the attacker can infer the values of bits in S.

Reading Secret Bit Values. Given a page P, we denote by $P[i]$ the i-th bit in P, where $i \in \{0, 1, \ldots, 32766, 32767\}$. At a high level, given a known flippable bit A1[i] in the page A1, we can read the corresponding bit S[i] (i.e., the bit at the same offset within the frame) in S as follows:

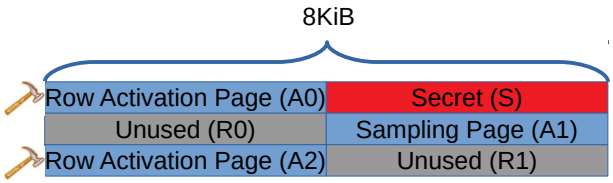
1) **Initialize.** Assuming that the bits are true cells, the attacker first populates all of A1 with ones before hammering.
2) **Hammer.** The attacker repeatedly reads her own pages A0 and A2, thereby using the victim's secret-containing pages to perform double-sided hammering on A1.
3) **Observe.** After hammering, the attacker reads the value of the bit A1[i], which is accessible to her because the page A1 is located inside the *attacker's* own private memory space. We argue that after hammering, the value of A1[i] is equal to the value of S[i]. Indeed, if S[i] equals 0, then before hammering A1[i] would have been in the center of a 0-1-0 stripe configuration. Since A1[i] sits in the center of a flippable stripe configuration, A1[i] will flip from one to zero after hammering. Conversely, if S[i] equal to 1, then A1[i] will be in the center of a 1-1-1 uniform configuration, and will retain its value of 1 after hammering. Thus, in both cases, the attacker reads A1[i] from her own private memory after hammering, which directly reveals S[i].

Double-sided RAMBleed. In the case of anti-cells, the only change we make is that in step 1, we populate A1 with zeros instead of ones. Thus, by observing bit flips in her own pages, the attacker can deduce the values of surrounding cells. Since the secret S surrounds A1 from both sides, we call this "double-sided RAMBleed".

Single Sided RAMBleed. Figure 3b presents the memory layout for what we call "single-sided RAMBleed", which differs from the double-side case only in the bottom right



(a) Double-sided Rambleed. Here, the sampling page (A1) is sandwiched between two copies of S.



(b) Single-sided Rambleed. Here, the sampling page (A1) is neighbored by the secret-containing page (S) on a single side.

Fig. 3: Page layout for reading out the victim's secret. Each cell represents a 4 KiB page, meaning that each row represents an 8 KiB row in a DRAM bank. The attacker repeatedly accesses her row activation pages A0 and A2, activating the top and bottom rows. She then reads out corresponding bits in page S by observing bit flips in the sampling page A1.

frame; instead of another copy of S, an arbitrary page R1 resides below A1. With this configuration, we can still read out bits of S by following the same steps as in the double-sided scenario, albeit with reduced accuracy. The reduction in accuracy is because the value of R1[i] may differ from that of S[i]. Assuming a uniform distribution of bits in R1, in half of the cases, the starting configuration is one of 1-1-0 and 0-1-1, which are neither striped nor uniform. With such configurations, bits tend to flip less than with striped configurations introducing uncertainty to the read values. Yet, in half of the cases R1[i]=S[i], resulting in the same outcome as for the double-sided RAMBleed scenario.

While double-sided RAMBleed maximizes the disturbance interactions between the secret bits and A1, it is also more challenging to execute in practice because it requires two copies of the same data in memory. Nevertheless, in Section VII we show how an attacker can reliably obtain two copies of S, demonstrating an end-to-end attack on OpenSSH.

V. Memory Massaging

The descriptions from Section IV assume that the attacker can place the victim's secrets in the layout shown in Figure 3, where A0–A2 are allocated to the attacker, and that the attacker knows which bits can flip and in which direction. We now present novel memory massaging primitives that achieve both goals without requiring elevated permissions or special operating system configuration settings (i.e., avoiding huge pages, page map access, memory deduplication).

A. Obtaining Physically Consecutive Pages

As we can see in Figure 3, the attack requires pages located in three consecutive 8 KiB rows in the same bank. While this task was previously achieved using the Android ION allocator [61], no such interface is available in non-Android Linux. Instead, we exploit the Linux buddy allocator [17] to allocate a 2 MiB block of physically consecutive memory. As the same-row-index size (See Section II-A) on our system is 256 KiB, we are guaranteed to be able to build the layout of Figure 3 using some of the pages in the block provided by the allocator. We now proceed to provide a short overview of Linux's buddy allocator. See Gorman [17] for further details.

Linux Buddy Allocator. Linux uses the buddy allocator to allocate physical memory upon requests from userspace. The kernel stores memory in physically consecutive blocks that are arranged by *order*, where the nth order block consists of $4096 \cdot 2^n$ physically consecutive bytes. The kernel maintains free lists for blocks of orders between 0 and 10. To reduce fragmentation, the buddy allocator always attempts to serve requests using the smallest available blocks. If no small block is available, the allocator splits the next smallest block into two "buddy" halves. These halves are coalesced into one block when they are both free again.

The user space interface to the buddy allocator, however, can only make requests for blocks of order 0. If, for example, a user program requests 16 KiB, the buddy allocator treats this as four requests for one 4 KiB block each. This means that irrespective of their size, user space requests are first handled from the free list of 0 order blocks. Only once the allocator runs out of free 0 order blocks, it will start serving memory requests by splitting larger blocks to generate new 0 order blocks. Thus, while obtaining a virtually consecutive 2 MiB block is trivial and only requires a single memory allocation, obtaining a *physically* consecutive block requires a more careful strategy, which we now describe.

Obtaining a Physically Consecutive 2 MiB Block. We now exploit the deterministic behavior of the buddy allocator to coerce the kernel into providing us with physically consecutive memory, using the following steps:

- **Phase 1: Exhausting Small Blocks.** First, we allocate memory using the mmap system call with the MAP_POPULATE flag, which ensures that the kernel eagerly allocates the pages in physical memory, instead of the default lazy strategy that waits for them to be accessed first. Next, we use the `/proc/pagetypeinfo` interface to monitor available block sizes in the kernel free lists, and continue to allocate memory until less than 2 MiB of free space remains in blocks of order less than 10.
- **Phase 2: Obtaining a Consecutive 2 MiB Block.** Once free space in blocks of order below 10 is less then 2 MiB, we make two requests of size 2 MiB each. Thus, to serve the first request after exhausting the smaller blocks, the kernel needs to split one of the 10th order blocks (whose size is 4 MiB each). This leaves more than 2 MiB in the free list, where all such space comes from the newly-split 4 MiB block, and is served in-order. Thus, the memory allocated for

the second request consists of consecutive physical memory blocks, which is exactly what we require.

While the region we obtain in the second allocation is physically consecutive, this approach does not guarantee that the obtained area will be 2 MiB-aligned in the physical memory. Thus, to use the obtained region for Rowhammer, we require an additional step to recover more information about the physical address of the obtained 2 MiB region.[1]

Recovering Physical Addressing Bits. Next, for double-sided hammering, we need to locate addresses in three consecutive rows within the same bank. As some of the physical address bits of the 2 MiB block are used for determining the banks of individual 4 KiB pages, we must somehow obtain these addressing bits for every 4 KiB page in our block.

Since 2 MiB$= 2^{21}$ bytes, and our 2 MiB block is physically sequential, obtaining the low 21 bits of the physical addresses amounts to finding the block's offset from being 2 MiB aligned (where the low 21 bits are 0). In older Linux kernels, an attacker could use the pagemap interface to translate virtual addresses to physical addresses. However, in the current Linux kernel, the interface requires root privileges due to security concerns [55]. Instead of using the pagemap interface, we exploit the row-buffer timing channel of Pessl et al. [49] to recover the block offset.

Computing Offsets. To find a block's offset from a 2 MiB aligned address, we take advantage of the fact that our 2 MiB block is physically contiguous and that the set of *distances* between co-banked addresses uniquely defines the block's offset. Figure 4 illustrates this concept. The blue block is a 2 MiB aligned block originally found in the fragmented order 10 block, while the red, 2 MiB unaligned block is the region we have obtained from our attack on the allocator. The colored vertical stripes are 4 KiB pages, where two pages of the same color indicate that they reside in the same bank.

The distances d_i, $i \in \{0, 1, 2, \ldots, n\}$ are the differences between the addresses of the i-th page in our block and the very next address located in the same bank. Together, the set $\{d_0, d_1, d_2, \ldots, d_n\}$ forms a *distance pattern* for our block. There are 512 possible offsets for a 4 KiB page within a 2 MiB block; simulations of DRAM addressing confirm that these patterns uniquely identify the block's offset.

Recovering Distance Patterns. We can now use Pessl et al.'s [49] row-buffer timing side channel to find the distances $\{d_0, \cdots, d_n\}$ between pages located in the same bank. Once we have uncovered enough of the distance pattern to uniquely identify a single offset, we have succeeded in computing the offset of our 2 MiB block. This typically occurs after finding fewer than ten distances.

We compute a distance d_i by alternating read accesses between p_i and p_j for $j \in \{i{+}1, i{+}2, \ldots, i{+}2n{-}2, i{+}2n{-}1\}$, where p_i is the page at the i-th offset within the block, and

[1]The more naive strategy of first exhausting all smaller blocks and then using one larger request in the hope that it is served from a single large block tends not to work in practice. Any block of order 0 released during the exhaustion phase will be recycled before splitting the large block and will result in a non-consecutive allocation.

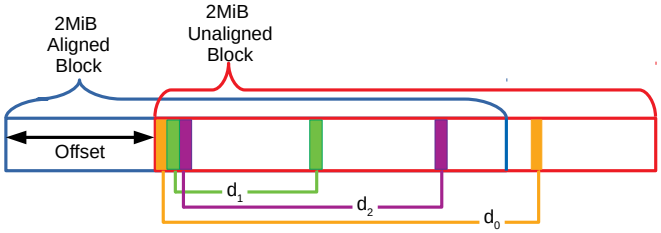


Fig. 4: The blue block is the 2 MiB aligned block that was originally found in the fragmented order 10 block, while the red, 2 MiB unaligned block is the block we have obtained from our attack on the allocator. We compute the offset by finding the distances between co-banked pages d_i, $i \in \{0, 1, 2, .., n\}$, which uniquely identify the offset.

n is the number of pages with the same row index. We then time how long it takes to access both addresses, and average the results over 8,000 trials; the page that corresponds to the greatest read time is identified as residing in the same bank as p_i. The distance d_i is then equal to the difference in the page offset between the two.

The reason we search over the next *two* rows of any bank (i.e., 512 KiB), and not just the next, is that the nature of the DRAM addressing scheme means that the two co-banked pages in consecutive rows can potentially lie anywhere within the memory range with the same row index. When we compute the distances, we make use of Schwarz's [54] optimizations for confusing the memory controller to obtain accurate timing measurements. We empirically find over many trials that this method works with a 100% success rate.

Recovering Bit 21. So far, we have uncovered bits 0–20 of the physical address. As Pessl et al. [49] show, however, DRAM addressing on our system depends on bits 0–21. The naive solution is to simply adjust our attack on the memory allocator to obtain a physically contiguous 4 MiB block. This solution, however, is infeasible as the buddy allocator does not track 8 MiB blocks, and thus cannot split an 8 MiB block into two contiguous 4 MiB blocks. Another solution is to simply guess the value of bit 21, doubling the attack's running time.

We can, however, overcome this through an insight into the DRAM addressing scheme. On our system (a Haswell machine with two DIMMs on a single channel) there are three *bank addressing bits* used to select between the eight banks within a single rank. As specified by [49], bit 21 is only used for computing the third bank addressing bit by XORing bits 17 and 21 of the physical address. Thus, to find two physical addresses a^0, a^1 located in the same bank in consecutive 8 KiB rows, we need to ensure that

$$a^0_{17} \oplus a^0_{21} = a^1_{17} \oplus a^1_{21}$$

where a^i_j is the j-th least significant bit in the i-th physical addresses (a^0, a^1). Then, given a physical address a^0 in the 2 MiB block, when we want to find another physical address a^1 in the same bank, but located in the row above. First we set a^1 to be a^0 plus the size until the next row index. Then, we

adjust a_{17}^1 to preserve the above equation. Even though we do not know a_{21}^0 nor a_{21}^1, we can examine bits 0 till 20 in a^0 to see if the addition of the size of row index done for computing a^1 had resulted in a carry for bit 21. If so, we compensate by flipping a_{17}^1 in order to preserve the above equation.

B. Memory Templating

After obtaining blocks of contiguous memory, we proceed to search them for bits that can be flipped via Rowhammer. We refer to this as the *templating* phase, which is performed as follows. We first use our technique to obtain 2 MiB blocks of physically contiguous memory. Then, we locate addresses that belong to the same bank using the method described above. Next, we perform double-sided hammering with both 1-0-1 and 0-1-0 striped configurations. Finally, we record the locations of these flips for later use with RAMBleed.

C. Placing Secrets Near Flippable Bits

After templating memory, we exploit the determinism of the Linux physical memory allocator to place the victim's page in the desired physical locations as outlined in Figure 3. While a similar task was achieved in [61] on Android's ION allocator by exhausting most of the available memory to control the placement of the victim, we achieve the same result on Linux's buddy allocator without memory exhaustion. Following the convention of [61][51][58], we call this technique "Frame Feng Shui", as we are coercing the allocator into placing select pages into a frame of our choosing.

Exploiting Linux's Buddy Allocator. The buddy allocator stores blocks of equal order in a first-in-last-out (FILO) stack-like data structure, and upon receipt of a request of order n, the allocator returns the most recently freed block from the n-th order's bucket. Thus, if we assume that the victim, after being triggered, allocates a predictable number of pages before allocating the secret-containing page, we can force Linux's memory allocator to place the victim's secret containing page in a page frame of our choice by the following:

- **Step 1: Dummy Allocations.** The attacker allocates n 4 KiB pages by calling mmap with the MAP_POPULATE flag, where n is the number of pages that the victim will allocate before allocating its secret containing page.
- **Step 2: Deallocation.** The attacker inspects her own address space and chooses the target page frame for the victim's secret to land on (one that neighbors the flippable bits). Next, the attacker calls munmap and deallocates the selected frame. The attacker then immediately unmaps all the pages mapped during Step 1. After doing so, the allocator's stack-like data structure for the 0th order blocks will have the n pages on top, followed by the target page.
- **Step 3: Triggering the Victim.** After Steps 1 and 2, the attacker immediately triggers the victim process, letting it perform its memory allocations. In Section VII, we accomplish this by initiating an SSH connection, which is served by the SSH daemon. After being triggered, the victim allocates n pages, which then land in the frames vacated by the pages mapped in Step 1. Finally, the victim allocates its secret-containing page, which then lands in the desired frame, as it will be located on top of the allocator's stack-like data structure for 0th order blocks at this point.

D. Putting It All Together

With the above techniques in place, we can now describe our end-to-end attack. which consists of two phases.

Offline. The attack starts by allocating 2 MiB blocks and dividing them into physically consecutive pages as described in Section V-A. The attacker then templates her blocks and locates Rowhammer induced bit flips using the methodology described in Section V-B. Notice that this phase is done offline, entirely within the attacker's address space, and without any interaction with the victim. Finally, after the attacker obtains enough Rowhammer induced bit flips to read the victim's secret, the attacker begins the online phase described below.

Online. In this step, the attacker uses Frame Feng Shui to get the victim to place his secret in the physical memory locations desired by the attacker (e.g., using the layout in Figure 3). The attacker then performs the RAMBleed attack described in Section IV-C to exploit the data-dependency with the victim's bits, and subsequently deduces some of their values. Finally, the attacker repeats the online phase step until a sufficient number of secret bits where leaked from the victim (e.g., around 66% percent of the victim's RSA secret key, which is sufficient to mathematically recover of the remaining bits).

VI. Experimental Evaluation

To measure RAMBleed's capacity as a read side channel, we measure the rate and accuracy of RAMBleed's ability to extract bits across process boundaries and address spaces under ideal conditions and predictable victim behavior.

Next, after evaluating both double-sided and single-sided RAMBleed, in Section VII we evaluate RAMBleed against an OpenSSH 7.9 server (which is a popular SSH server), extracting the server's secret RSA signing keys.

The Victim Process. In the proof-of-concept victim code, the victim waits for an incoming TCP connection, and then copies the secret key into a freshly allocated page (using an anonymous mmap) upon each TCP connection request. This behavior is akin to a server that runs a decryption routine every time the attacker makes a request, thereby using its secret key.

The Attacker Process. The attacking process uses the techniques described in Section V-A to obtain 2 MiB physically consecutive blocks, and subsequently templates memory for flippable cells using the methods outlined in Section V-B. Finally, the attacker uses Frame Feng Shui to place the secret-containing page above and below a flippable bit (for single-sided, we only place it above). Concretely, we accomplish this by unmapping the target location and then initiating a TCP connection with the victim. Since $n = 0$ in this case, meaning that the secret is the first allocation upon context switching, the secret-containing page should land in the recently vacated frame. The attacker then hammers the surrounding rows and leaks the secret bits by reading out the flips from its own page. We run both processes as taskset with the same CPU affinity.

Type	Read Accuracy Percents		
	Overall	*False Positive*	*False Negative*
Double-sided	90%	5%	15%
Single-sided	74%	19%	29%

TABLE I: "false positive" events, where a uniform configuration still flips are more rare than "false negative" events, in which a striped configuration refuses to flip.

Hardware. We use an HP Prodesk 600 desktop running Ubuntu 18.04, featuring an i5-4570 CPU and two Axiom DDR3 4 GiB 1333 MHz non-ECC DIMMs, model number 51264Y3D3N13811, in a single-channel configuration.

Experimental Results. While [13] report that bit flips are deterministic with regards to the surrounding bits (i.e. a bit flips if and only if it is in a striped configuration), on our systems we observe the more general case where the bit flips are probabilistic. Next, the probability of a bit flip highly depends on the type of configuration (striped or uniform). This uncertainty adds noise to our read-channel, which we handle with a variant of the Heninger-Shacham technique [24].

Memory Templating. The time required to template memory and find the needed flips is entirely dependent upon how easily the underlying DIMMs yield bit flips. While [37] and [21] report finding thousands of flips within minutes, we found flips at a more modest rate of 41 flips per minute.

Reading Secret Bits. After templating the memory with a striped 0-1-0 pattern, our experimental code can read out the victim's secret at a rate of 3–4 bits/second. As we can see from the results in Table I, this works with 90% accuracy overall, and 95% accuracy when it comes to identifying 1-bits. This is because "false positive" events, that is, when a 1-1-1 uniform configuration still results in the center bit flipping from one to zero, are much rarer than "false negative" events, in which a 0-1-0 stripe refuses to flip. We can then template with the opposite stripe pattern (1-0-1) and achieve a 95% accuracy rate on the zero-valued bits.

VII. Attacking OpenSSH

To demonstrate the practical risk that RAMBleed poses to memory confidentiality, in this section we present an end-to-end attack against OpenSSH 7.9 that allows an unprivileged attacker to extract the server's 2048-bit RSA private signing key. This key is what allows an SSH server to authenticate itself to incoming connections. As such, a break of this key enables the attacker to masquerade as the server, thereby allowing her to conduct man-in-the-middle (MITM) attacks and decrypt all traffic from the compromised sessions.

At a high level, our attack operates by coercing the server's SSH daemon to repeatedly allocate and place its private key material at vulnerable physical locations. We then use double-sided RAMBleed to recover a portion of the bits that make up the server's RSA key. Finally, we utilize the mathematical redundancy in RSA keys to correct for errors in extracted bits, as well as recover missing bits that we were unable to read directly. Before describing our attacks, we now describe how OpenSSH manages and uses its keys in response to incoming SSH requests, and how we adapted the techniques from Section V to specifically target OpenSSH.

A. Overview of OpenSSH

The OpenSSH daemon is a root-level process that binds to port 22 and has access to a root-accessible file, which stores the server's RSA private key. As shown in Figure 5, when a TCP connection arrives on port 22, the daemon spawns a child process that handles the authentication phase of incoming SSH connections. The child is responsible for both authenticating the server to the client as well as authenticating the client to the server. While the latter can be done either via public-private key pair, or by supplying a password, the former is done by having the server use its RSA private key to sign a challenge issued by the client. Finally, once authentication is complete, the child process spawns an unprivileged grandchild for handling the user's connection. See Figure 5.

Key Memory Management. The child process that is spawned by the SSH demon for mutually authenticating an incoming SSH request must first read in the server's private key from the key file into a temporary buffer. At this point, the key will actually be located in memory in two places: namely, the temporary buffer and the OS' page cache. Unfortunately, we cannot read either of these memory locations via RAMBleed. For the former, this buffer gets overwritten immediately, before we have any chance to read even a single bit using RAMBleed. The latter copy is also inaccessible as it is stored inside the OS' page cache, which is located in a static region of physical memory that is not moved around. Luckily, OpenSSH's authentication process then proceeds to copy the keys into a new buffer maintained by a global structure, aptly named "sensitive_data". This buffer remains in physical memory for the duration of the connection. Thus, our attack aims to read the private key material from this structure.

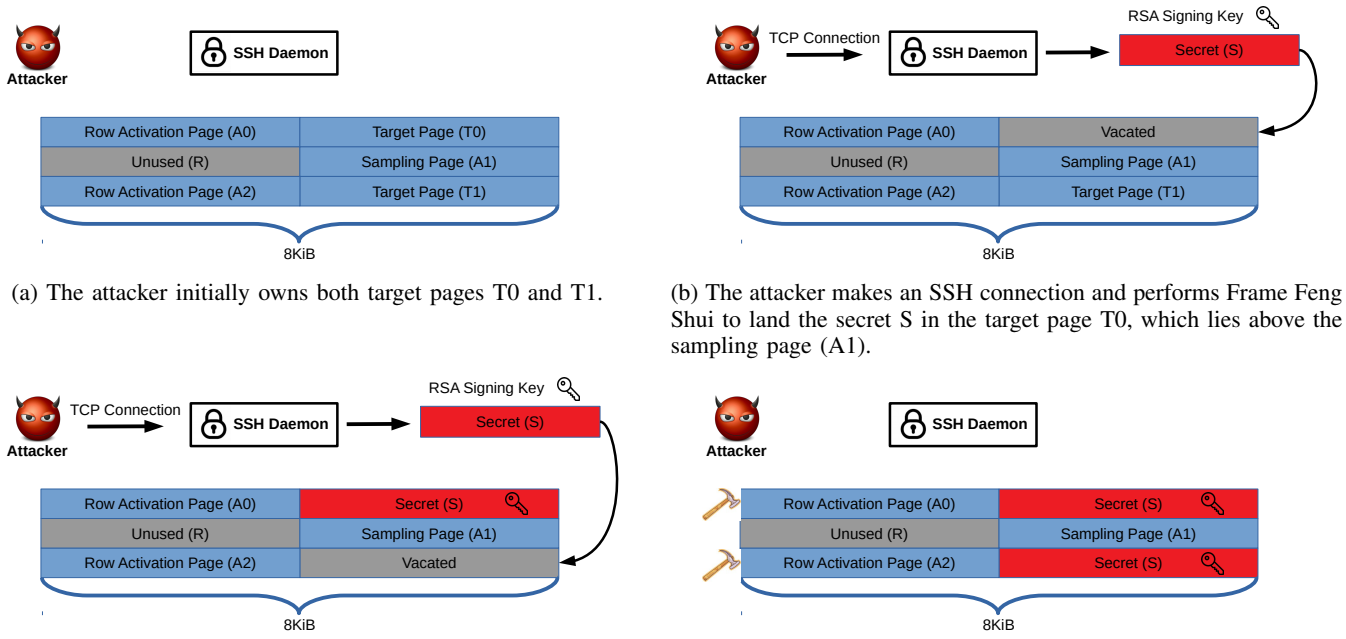
We now proceed to describe our attack on OpenSSH.

B. Attack Overview

Our first step is to profile memory, looking for flippable bits. We do this in the same manner described in Section V-B. After finding a sufficient number of flips, we begin the *reading* phase, in which we perform RAMBleed to leak a single bit at a time. At a high level, for each templated bit, we use Frame Feng Shui to place private key material in the configuration shown in Figure 3, where A1 is the page containing the flippable bit. We then perform double-sided RAMBleed to leak the bit's value and proceed to the next bit.

C. Overcoming OpenSSH's Memory Allocation Pattern

To use Frame Feng Shui against OpenSSH, we must determine the value n, which is the number of pages we must unmap after vacating the target frame in order to cause OpenSSH to place the secret in the targeted frame location. Examining the behavior of OpenSSH 7.9 on our system, we found that its allocations pattern is predictable, which allows us to use Frame Feng Shui with a high success rate. More



(a) The attacker initially owns both target pages T0 and T1.

(b) The attacker makes an SSH connection and performs Frame Feng Shui to land the secret S in the target page T0, which lies above the sampling page (A1).

(c) The attacker repeats the Frame Feng Shui process to land S in the target page T1, below the sampling page (A1).

(d) After achieving the double-sided RAMBleed position, the attacker now hammers the activation pages (A0 and A2) to induce flips in the sampling page (A1).

Fig. 5: Overview of our attack on OpenSSH

specifically, we found that OpenSSH uses the default RSA key size of 2048 bits, with the following allocation pattern.

- First, the page containing d, the RSA private exponent, is allocated 101 pages after the daemon accepts a new TCP connection. See Section II-D for RSA notation.
- Next, a single page containing both p and q is allocated 102 pages after the daemon accepts a new connection.
- Finally, a single page that contains both d_p and d_q is allocated 104 pages after accepting a new connection.

Furthermore, all the private key values mentioned above are located at the same offset within their page upon every incoming connection. Thus, we fix $n = 100, 101$, and 104 respectively for d, p and q, and d_p, d_q. Next, to obtain the configuration in Figure 3, we call munmap on the page above A1 and follow it with n munmaps on random pages. We then immediately make a TCP connection, causing the SSH daemon to make n allocations, followed by allocating the secret-containing page, which will then be placed in the target frame. By holding the TCP connection open, we can repeat the process to place the page in the frame below A1, thereby creating two copies of the secret in memory to facilitate double-sided RAMBleed.

Accounting for Allocation Noise. The memory placement technique described above is much more susceptible to noise, as many CPU cycles pass between the point of the original unmapping by the attacker and when the victim maps the key-containing page. Thus, if any pages are allocated or deallocated in that time frame by another process, the key-containing pages will not be placed in the desired locations. To minimize this noise, the attacker yields the scheduler before performing the page deallocaitons, allowing other scheduled system activity to execute. Next, we also use a busy loop after unmapping the pages and before reading the bits, waiting a fixed amount of time for OpenSSH to perform the required allocations. We note here that if we replace the busy loop with a sleep operation, this will likely cause the system to schedule another process and destroy the memory layout. After using RAMBleed to read the bit(s), we close the connections, triggering the daemon to kill the two children.

After mitigating noise in this manner, the memory placement process succeeds against OpenSSH with 83% probability. This means that we will be in the double-sided-RAMBleed situation $0.83^2 = 68.89\%$ of the time, in single-sided RAMBleed $2 \cdot 0.83 \cdot 0.17 = 28.22\%$ of the time, and $0.17^2 = 2.39\%$ of the time we will be unable to place the target page near the flipping row, resulting in random guessing. This, along with potential for RAMBleed to misread bits, gives us an overall accuracy of 82% when reading the OpenSSH host key.

Key Recovery. To recover the key from the noisy bits, we use a variant of Paterson et al. [46]'s algorithm, an adaptation of the Heninger-Shacham algorithm [24] for the case that key bits are only known with some probability. Specifically, the algorithm aims to reconstruct the key, bit by bit, starting from the least significant bit. By relating the public (N, e) and private (d, p, q, d_p, and d_q) key components, the algorithm prunes potential keys and dramatically reduces the search space. The algorithm explores a search tree of potential keys while pruning branches that contradict known bits or have a large number of mismatches with probabilistically recovered

Type	Probability
Double-sided RAMBleed	68.89%
Single-sided RAMBleed	28.22%
Unable to place victim	2.39%

TABLE II: Probability of OpenSSH placing pages containing private key material into double-sided, single-sided, or unable-to-place situations.

bits. Our approach is similar to Paterson et al. [46], but instead uses a depth-first search in place of a bread-first search.

Through a series of simulations on random RSA 2048 bit keys, we empirically found that our amended Heninger-Shacham algorithm requires 68% recovery of the private key material (d, p, q, d_p, d_q) with an 82% accuracy. This implies that 4200 distinct bits of private key material is sufficient to extract the complete key.

D. Overall Attack Performance

Memory Templating. We begin our attack by locating the flippable bits in the memory of the target machine. Using the techniques presented in Sections IV and V, we profiled the machine's memory to locate Rowhammer induced bit flips. We note here that the time required to template memory and find the required flips is entirely dependent upon the susceptibility of underlying DIMMs to Rowhammer attacks. While [21, 37] report finding thousands of flips within minutes, we found flips at a more modest rate of 41 flips per minute, giving us a running time of 34 hours to locate the 84K bit flips required for the next phase of the attack.[2]

We note here that this phase can be performed ahead of time and with user level permissions, without the need to interact with the victim application or its secrets.

Removing Useless Bits. Next, we note that not all of these bitflips are useful for key extraction. First, given OpenSSH memory layout and the location of the key elements in their respective pages, only a $\frac{6144}{32768} = \frac{3}{16}$ fraction of the bits (corresponding to offsets of d, p, q, d_p and d_q) are useful for key recovery. Out of the 84K bit flips recovered in the previous phase, this leaves approximately 15750 bits flips which have the potential to reveal bits of the secret key. Next, we note that these bit flips also contain repetitions in their locations in the page, meaning that two or more bit flips might actually correspond to the same bit of the secret key. After removing such duplicates, we are left with 4.2K bit flips in distinct locations that are useful for key extraction.

Reading Private Key Material. After placing the key containing pages in the desired locations to achieve one of the RAMBleed configurations, we then proceed to hammer A0 and A2 (See Figure 3). We have no way of determining if we are in the double-sided, single-sided, or unable-to-place RAMBleed situation, but given the probabilities in Section VII-C, it is likely that the bit flip in A1 will depend upon the secret bit values. Overall, this process resulted in recovering 68% of the private key, or 4200 key bits, at a rate of 0.31 bits/second at an accuracy rate of 82% against OpenSSH. We conjecture that the deceased accuracy is due to the combined noise from both the inaccuracy of RAMBleed and Frame Feng Shui.

Key Recovery. As mentioned above, we recover 68% of the key bits with 82% accuracy. Using our amended Heninger-Shacham algorithm, we recover the entire RSA private key in about 3 minutes on a consumer laptop (Dell XPS 15 featuring an Intel i7-6700 3.4 GHz CPU and 32 GiB of RAM).

[2]We empirically found that 84K bit flips was approximately the threshold for locating 4200 usable, unique, flippable bits.

VIII. RAMBleed on ECC Memory

In this section we show how to use RAMBleed to read secret information stored on DIMMs that use ECC memory. Unlike Section IV, which shows how RAMBleed can exploit visible bit flips to read secret information, here we show how an attacker can exploit bit flips that were successfully corrected by ECC to read information from the victim's address space.

We begin by providing background on ECC memory.

A. ECC Memory Background

Memory manufacturers originally designed ECC memory for correcting rare, spontaneous bit flips, such as those caused by cosmic rays. As such, ECC memory uses error correcting codes that can only correct a small number of bits in a single code word, typically only one or two. This is commonly known as SECDED (Single error correction double error detection).

Correction Mechanism. When an ECC enabled system writes data to DRAM, the memory controller writes both the data bits and an additional string of bits, called the *check bits*. These bits offer the redundancy that enables detection and correction of errors. Together, the data and check bits make up a codeword, where the typical sizes for data and check bits are 64 and 8 bits, respectively. Upon serving of a read request from DRAM, the memory controller reads both the data and check bits, and checks for errors. If an uncorrectable error is detected, the controller typically crashes the machine, rather than letting the software operate on corrupted data. Alternately, if the error can be corrected, the memory controller first corrects the error, and only then passes the corrected value to the software. We note that ECC correction and detection occurs only during read requests, and that a bit flip will go undetected until a codeword is read from the DIMM.

Detecting Bit Flips. As Cojocar et al. [13] describe, this synchronous error correction results in a timing side channel that allows an attacker to determine if a single-bit error has occurred. They found that the overhead incurred by correctable bit flips is on the order of hundreds of thousands of cycles, which the attacker can easily measure.

Concretely, we can detect the presence of a bit flip in any given word by measuring the read latency from the word. When we read from a word with a single-bit error, the hardware must first complete the ECC algorithm, and often log the error in the firmware log, before the value from the read is returned. If we observe a much longer read latency, it indicates that a bit flip occurred sometime after the last time that the same 64 bit word was read from. This effect is illustrated in

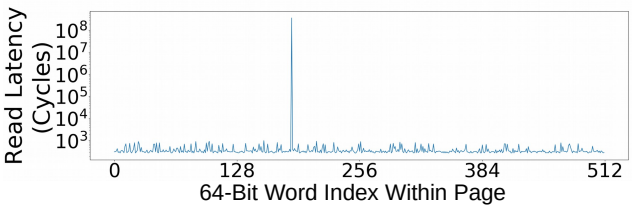


Fig. 6: Read latencies for the 64-bit words in a single page. When ECC corrects an error, the latency is 5 orders of magnitude greater than the common case. This can be seen by the peak for the 186th word, which indicates a bit flip.

Figure 6; after performing double-sided hammering on the two aggressor rows, we read from the victim row and observe a crisp peak for the 186th word, clearly indicating a bit flip.

B. RAMBleed on ECC Memory

We now show how we can leverage the ability to detect the presence of corrected bit flips to read information from the victim's address space. To the best of our knowledge, this is the first demonstration of security implications of corrected bit flips.

Experimental Setup. Following the Intel-1 setup of Cojocar et al. [13], we demonstrate the RAMBleed attack on ECC memory on a Supermicro X10SLL-F motherboard (BIOS version 3.0a) equipped with an Intel Xeon E3-1270 v3 CPU and a using a pair of Kingston 8GB 1333 MHz ECC DIMMs, model number KVR1333D3E9SK2.

Templating. As with the non-ECC attack, we begin by first templating memory to locate bit flips. We do so in much the same manner of [13], only with an algorithmic improvement for determining which bit in a row is the flippable bit.

Cojocar et al. [13] locate bit flips by performing double sided Rowhammer, and then using the timing side channel to locate a word containing a bit flip. They determine which of the 64 bits flipped by setting exactly one of the bits to its charged state, while all the rest are discharged. This results in the targeted bit being in the middle of a striped configuration, while all the other bits in the word are part of a uniform configuration. Next, a long read latency indicates that the single charged bit flipped. Finally, they repeat the process for each bit to determine which bits can be flipped.

To speed up the process of templating memory for bit flips, we replace the single-bit iteration phase with a binary search over the possible locations for the bit flip. That is, after locating a word with a bit flip, we set half of the bits to their charged state, with the other half discharged. We then hammer the aggressor rows again, and record the read latency. If it is long, then the bit flip lies in the half with the charged bits; otherwise, it lies in the other half. We repeatedly reduce the search space by half in this manner, until we have pinpointed the location of the bit flip. Overall this speeds up the templating phase of Cojocar et al. [13] by a factor of 10.

Reading Bits. After profiling memory and recording the precise locations of flippable bits, we use the memory massaging and Frame Feng Shui techniques described in Section V to achieve the double-sided RAMBleed configuration. In the non-ECC RAMBleed case, we hammered the aggressor rows and subsequently directly read the victim row for a Rowhammer-induced bit flip, thereby leaking values of secret bits. With ECC, we cannot observe the flips directly. Instead we use the timing side channel and look for long read latencies. As such latencies occur only due to Rowhammer-induced flips, they can be used to reveal the value of the secret bit as described in Section IV.

Experimental Results. We can successfully read bits via RAMBleed against ECC memory with a 73% accuracy at a reading rate of 0.64 bits/second in our setup. Since ECC DIMMs are typically built using the same chips as used on non-ECC DIMMs, but with an additional chip for storing the check bits, we attribute the drop in accuracy to the fact that they are simply different sets of DIMMS.

IX. Mitigations

Unlike previous Rowhammer attacks which compromise integrity, RAMBleed is an attack which compromises confidentiality. Moreover, to leak information cross process and cross address space, RAMBleed only requires that the attacker can read and hammer her own private memory, and does not involve any access or modification to the target's data, code, or address space. As such, RAMBleed can bypass software-based integrity checks that might be applied to the target, such as using message authentication codes (MAC) to protect the target's data. Moreover, techniques designed to protect cryptographic systems against fault attacks (such as Shamir's countermeasure [56]) are also ineffective as they again protect the integrity of the cryptographic computation and not its confidentiality. Other software defenses, such as Brasser et al.'s [8] memory partitioning scheme do not mitigate our attack, as we are not trying to read from kernel memory.

A. Hardware Mitigations

There are, however, a few commonly proposed hardware-based mitigations that have the potential to mitigate RAMBleed. Kim et al. [34] propose PARA (probabilistc adjacent row activation), wherein activating a row causes nearby rows to activate with some probability. Repeated hammering of an address then increases the likelihood that nearby victim rows will be refreshed, thereby restoring their cells' charges and preventing Rowhammer. PARA has not been widely adopted, as it can only proide a probabilistic security guarantee.

Targeted Row Refresh (TRR). The more recent LPDDR4 standard supports the ability to refresh a targeted row with TRR, where after a row is accessed a set number of times, the nearby rows are automatically refreshed [31]. Despite this mitigation, [21, 61] already report the ability to induce Rowhammer bit flips in the presence of TRR.

Increasing Refresh Intervals. Doubling DRAM refresh rate by halving the refresh interval from 64ms to 32ms is an attempt at reducing the number of bit flips by refreshing victim rows. However, this is impractical on mobile systems due to

the increased power demands. Worse yet, Aweke et al. [2] and Gruss et al. [21] demonstrate bit flips even under this setting.

Using Error Correcting Codes (ECC). An oft-touted panacea for Rowhammer is the usage of ECC memory, as any bit flip will simply be corrected by the hardware without affecting the software layer. However, as we show in Section VIII, the hardware error correction implementation actually produces sufficient side channel information for mounting RAMBleed. Thus, while ECC significantly slows RAMBleed, it does not offer complete protection.

B. Memory Encryption

One defense that does in fact protect against RAMBleed is memory encryption. This is because RAMBleed reads bits directly from memory, which are ciphertext bits in the case that memory is encrypted. Trusted execution environments, such as Intel's Software Guard Extensions (SGX), ARM's Trust Zone, and AMD's Secure Encrypted Virtualization (SEV), in fact fully encrypt the enclave's memory, thereby protecting them from RAMBleed. It should be noted, however, that some enclaves, such as SGX, perform integrity checking on encypted memory; Jang et al. [28] and Gruss et al. [21] show that Rowhammer induced flips in enclave memory halt the entire machine, necessitating a power cycle.

C. Flushing Keys from Memory

For systems that use sensitive data for a short amount of time (e.g., cryptographic keys), zeroing out the data immediately after use [22] would significantly reduce the risk from RAMBleed. This is because RAMBleed cannot accurately read bits of keys that do not remain in memory for at least one refresh interval (64ms by default). While this countermeasure is effective for protecting short lived data, it cannot by used for data that needs to stay in memory for long durations.

D. Probabilistic Memory Allocator

Our Frame Feng Shui technique exploits the deterministic behavior of the Linux buddy allocator to place the victim's pages in specific locations. Consequently, introducing a sufficient amount of non-determinism into the allocation algorithm will prevent the attacker from placing secrets into vulnerable locations. Such a defense would not, however, necessarily defeat a RAMBleed attacks that use probabilistic memory spraying techniques similar to [55]. The attacker could potentially keep many SSH connections open at once, and then hammer and read from the locations with the correct RAMBleed configurations. The attacker could use the row-buffer timing side-channel to detect the correct configurations.

X. Limitations and Future Work

RAMBleed's primary limitation is that it requires the victim process to allocate memory for its secret in a predictable manner in order to reliably read bits of interest. Otherwise, the Frame Feng Shui process described in Section V-C will not place the secret page in the intended frame. It may be possible, however, to bypass this limitation by using Yarom and Falkner's [65] Flush and Reload technique to determine when the secret page is about to be allocated.

Another limitation is that our attack against OpenSSH 7.9 required the the daemon to allocate the key multiple times. We conjecture, however, that it may be possible to read secrets even when they are never reallocated by the victim. If the secret lies in the page cache, it is likely possible to use Gruss et al.'s [21] memory waylaying technique to repeatedly evict the secret and then bring it back into memory, thereby changing its physical address. Even if it does not lie in the page cache, the attacking process can still evict it by exhausting enough memory to start paging memory to disk. Both of these strategies would, however, be defeated by using Linux's `mlock` system call to lock secret pages into memory, thereby preventing them from ever being evicted to disk.

Next, while we demonstrated our attack on a system using DDR3 DRAM, we do not suspect DDR4 to be a fundamental limitation, assuming that DDR4 memory retains the property that Rowhammer-induced bit flips are data-dependent. Our techniques for recovering physically sequential blocks depend only on the operating system's memory allocation algorithm, and are thus hardware agnostic. With regard to finding pairs of addresses in different rows of the same bank, [49] have already demonstrated how to reverse engineer the DRAM addressing scheme in DDR4 systems. Furthermore, Rowhammer-induced bit flips in DDR4 have been demonstrated by [1, 21, 37]. We leave the composition of these results to achieve RAMBleed on DDR4 memory to future work.

Finally, RAMBleed's rate of reading memory is modest, toping at around 3–4 bits per second. This allows sufficient time for memory scrubbing countermeasures to remove short-lived secret data from the target's memory. We thus leave the task of improving RAMBleed's read rate to future work.

XI. Conclusion

In this paper, we have shifted Rowhammer from being a threat only to integrity to also being a threat to confidentiality. We demonstrated the practical severity of RAMBleed by conducting and end-to-end exploit against OpenSSH 7.9, in which we extracted the complete 2048 bit RSA private signing key. To do so, we also developed memory massaging methods and a technique called *Frame Feng Shui* that allows an attacker to place the victim's secret-containing pages in chosen physical frames. By uncovering another channel for Rowhammer based exploitation, we have highlighted the need to further explore and understand the complete capabilities of Rowhammer.

Acknowledgments

This research was partially supported by a gift from Intel.

References

[1] M. T. Aga, Z. B. Aweke, and T. Austin, "When good protections go bad: Exploiting anti-dos measures to accelerate rowhammer attacks," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2017, pp. 8–13.

[2] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-based protection against next-generation Rowhammer attacks," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 743–755, 2016.

[3] K. Bains, J. Halbert, C. Mozak, T. Schoenborn, and Z. Greenfield, "Row hammer refresh command," US Patent Application 2014/0006703A1, 2014.

[4] J. Bauer, M. Gruhn, and F. C. Freiling, "Lest we forget: Cold-boot attacks on scrambled DDR3 memory," *Digital Investigation*, vol. 16, pp. S65–S74, 2016.

[5] D. J. Bernstein, J. Breitner, D. Genkin, L. G. Bruinderink, N. Heninger, T. Lange, C. van Vredendaal, and Y. Yarom, "Sliding right into disaster: Left-to-right sliding windows leak," in *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017, pp. 555–576.

[6] S. Bhattacharya and D. Mukhopadhyay, "Curious case of Rowhammer: Flipping secret exponent bits using timing analysis," in *CHES*, 2016.

[7] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory deduplication as an advanced exploitation vector," in *IEEE SP*, 2016.

[8] 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 *USENIX Security*, 2017, pp. 117–130.

[9] Y. Cai, S. Ghose, Y. Luo, K. Mai, O. Mutlu, and E. F. Haratsch, "Vulnerabilities in MLC NAND flash memory programming: Experimental analysis, exploits, and mitigation techniques," in *HPCA*, 2017, pp. 49–60.

[10] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," *arXiv*, vol. 1811.05441, 2018.

[11] Y. Cheng, Z. Zhang, and S. Nepal, "Still hammerable and exploitable: on the effectiveness of software-only physical kernel isolation," *arXiv*, vol. 1802.07060, 2018.

[12] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.

[13] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ECC memory against Rowhammer attacks," in *IEEE SP*, 2019.

[14] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand pwning unit: Accelerating microarchitectural attacks with the GPU," in *IEEE SP*, 2018, pp. 195–210.

[15] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *J. Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.

[16] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: the missing OS abstraction," in *EuroSys*, 2019.

[17] M. Gorman, *Understanding the Linux virtual memory manager*. Prentice Hall, 2004.

[18] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security*, 2015, pp. 897–912.

[19] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in JavaScript," in *DIMVA*, 2016, pp. 300–321.

[20] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: a fast and stealthy cache attack," in *DIMVA*, 2016, pp. 279–299.

[21] 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 *IEEE SP*, 2018, pp. 245–261.

[22] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *CACM*, vol. 52, no. 5, pp. 91–98, 2009.

[23] W. Henecka, A. May, and A. Meurer, "Correcting errors in RSA private keys," in *CRYPTO*, 2010, pp. 351–369.

[24] N. Heninger and H. Shacham, "Reconstructing RSA private keys from random key bits," in *CRYPTO*, 2009, pp. 1–17.

[25] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache attacks enable bulk key recovery on the cloud," in *CHES*, 2016, pp. 368–388.

[26] Intel Corporation, "6th generation Intel processor datasheet for S-Platforms," 2015.

[27] G. Irazoqui, T. Eisenbarth, and B. Sunar, "MASCAT: preventing microarchitectural attacks before distribution," in *CODASPY*, 2018, pp. 377–388.

[28] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-Bomb: Locking down the processor via Rowhammer attack," in *SysTEX*, 2017, p. 5.

[29] JEDEC Solid State Technology Association, "Low power double data rate 4," http://www.jedec.org/standards-documents/docs/jesd209-4b, 2017.

[30] ——, "JEDEC. Standard No. 79-3F. DDR3 SDRAM Specification," 2012.

[31] ——, "Low power double data rate 4," 2017.

[32] N. Karimi, A. K. Kanuparthi, X. Wang, O. Sinanoglu, and R. Karri, "MAGIC: Malicious aging in circuits/cores," *ACM (TACO)*, vol. 12, no. 1, p. 5, 2015.

[33] D.-H. Kim, P. J. Nair, and M. K. Qureshi, "Architectural support for mitigating row hammering in DRAM memories," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 9–12, 2015.

[34] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits

in memory without accessing them: An experimental study of DRAM disturbance errors," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, 2014, pp. 361–372.

[35] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE SP*, 2019.

[36] A. Kurmus, N. Ioannou, N. Papandreou, and T. P. Parnell, "From random block corruption to privilege escalation: A filesystem attack vector for Rowhammer-like attacks," in *WOOT*, 2017.

[37] M. Lanteigne, "How Rowhammer could be used to exploit weaknesses in computer hardware," http://www.thirdio.com/rowhammer.pdf, 2016.

[38] M. Lipp, M. T. Aga, M. Schwarz, D. Gruss, C. Maurice, L. Raab, and L. Lamster, "Nethammer: Inducing Rowhammer faults through network requests," *arXiv*, vol. 1805.04956, 2018.

[39] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018, pp. 973–990.

[40] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 605–622.

[41] X. Lou, F. Zhang, Z. L. Chua, Z. Liang, Y. Cheng, and Y. Zhou, "Understanding Rowhammer attacks through the lens of a unified reference framework," *arXiv*, vol. 1901.03538, 2019.

[42] Microsoft, "Cache and memory manager improvements," https://docs.microsoft.com/en-us/windows-server/administration/performance-tuning/subsystem/cache-memory-management/improvements-in-windows-server, Apr. 2017.

[43] P. Mosalikanti, C. Mozak, and N. A. Kurd, "High performance DDR architecture in Intel Core processors using 32nm CMOS high-K metal-gate process," in *VLSI-DAT*, 2011, pp. 154–157.

[44] K. Oonishi and N. Kunihiro, "Attacking noisy secret CRT-RSA exponents in binary method," in *ICISC*, 2018, pp. 37–54.

[45] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *CT-RSA*, 2006, pp. 1–20.

[46] K. G. Paterson, A. Polychroniadou, and D. L. Sibborn, "A coding-theoretic approach to recovering noisy RSA keys," in *ASIACRYPT*, 2012, pp. 386–403.

[47] M. Payer, "HexPADS: a platform to detect "stealth" attacks," in *ESSoS*, 2016, pp. 138–154.

[48] C. Percival, "Cache missing for fun and profit," in *BSDCan 2005*, 2005.

[49] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: exploiting DRAM addressing for cross-CPU attacks," in *USENIX Security*, 2016, pp. 565–581.

[50] R. Qiao and M. Seaborn, "A new approach for Rowhammer attacks," in *HOST*, 2016, pp. 161–166.

[51] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *USENIX Security*, 2016, pp. 1–18.

[52] Red Hat, *Red Hat Enterprise Linux 7 - Virtualization Tuning and Optimization Guide*, 2017.

[53] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *CACM*, vol. 21, no. 2, pp. 120–126, 1978.

[54] M. Schwarz, "DRAMA: Exploiting DRAM buffers for fun and profit," Ph.D. dissertation, Graz University of Technology, 2016.

[55] M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer bug to gain kernel privileges," https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.

[56] A. Shamir, "Method and apparatus for protecting public key schemes from timing and fault attacks," US Patent 5,991,415A, 1999.

[57] K. A. Shutemov, "Pagemap: Do not leak physical addresses to non-privileged userspace," https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce, Mar. 2015, retrieved on November 10, 2015.

[58] A. Sotirov, "Heap feng shui in JavaScript," in *BlackHat Europe*, 2007.

[59] A. Tatar, R. Krishnan, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer attacks over the network and defenses," in *USENIX ATC*, 2018.

[60] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security*, 2018, pp. 991–1008.

[61] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer attacks on mobile platforms," in *CCS*, 2016, pp. 1675–1689.

[62] S. Vig, S. K. Lam, S. Bhattacharya, and D. Mukhopadhyay, "Rapid detection of Rowhammer attacks using dynamic skewed hash tree," in *HASP@ISCA*, 2018, pp. 7:1–7:8.

[63] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," https://foreshadowattack.eu/foreshadow-NG.pdf, 2018.

[64] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation," in *USENIX Security*, 2016.

[65] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high

resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014, pp. 719–732.

[66] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: a timing attack on OpenSSL constant-time RSA," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[67] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *IEEE SP*, 2009, pp. 79–93.

[68] 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*, 2017, pp. 313–324.

[69] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *RAID*, 2016, pp. 118–140.