usenix

THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Half-Double: Hammering From the Next Row Over

Andreas Kogler, *Graz University of Technology;* Jonas Juffinger, *Graz University of Technology and Lamarr Security Research;* Salman Qazi and Yoongu Kim, *Google;* Moritz Lipp, *Amazon Web Services;* Nicolas Boichat, *Google;* Eric Shiu, *Rivos;* Mattias Nissler, *Google;* Daniel Gruss, *Graz University of Technology*

https://www.usenix.org/conference/usenixsecurity22/presentation/kogler-half-double

978-1-939133-31-1

Half-Double: Hammering From the Next Row Over

Andreas Kogler[1] Jonas Juffinger[1,2] Salman Qazi[3] Yoongu Kim[3] Moritz Lipp[4]*
Nicolas Boichat[3] Eric Shiu[5] Mattias Nissler[3] Daniel Gruss[1]

[1]*Graz University of Technology* [2]*Lamarr Security Research* [3]*Google*
[4]*Amazon Web Services* [5]*Rivos*

Abstract

Rowhammer is a vulnerability in modern DRAM where repeated accesses to one row (the aggressor) give off electrical disturbance whose cumulative effect flips the bits in an adjacent row (the victim). Consequently, Rowhammer defenses presuppose the adjacency of aggressor-victim pairs, including those in LPDDR4 and DDR4, most notably TRR.

In this paper, we present Half-Double[1], an escalation of Rowhammer to rows beyond immediate neighbors. Using Half-Double, we induce errors in a victim by combining many accesses to a distance-2 row with just a few to a distance-1 row. Our experiments show that the cumulative effect of these leads to a sufficient electrical disturbance in the victim row, inducing bit flips. We demonstrate the practical relevance of Half-Double in a proof-of-concept attack on a fully up-to-date system. We use side channels, a new technique called *Blind-Hammering*, a new spraying technique, and a Spectre attack in our end-to-end Half-Double Attack. On recent Chromebooks with ECC- and TRR-protected LPDDR4x memory, the attack takes less than 45 minutes on average.

1 Introduction

Rowhammer is a widespread DRAM issue caused by the unintended coupling between its constituent rows [31]. By repeatedly accessing one row (*i.e.*, *aggressor*), an attacker can corrupt data in adjacent rows (*i.e.*, *victims*) by accelerating their charge leakage. As a powerful means of bypassing hardware and software memory protection, Rowhammer has been used as the basis for many different attacks (Section 2.3).

Previously, Rowhammer was understood to operate at a distance of one row: an aggressor would only flip bits in its two immediate neighbors, one on each side. This makes intuitive sense: as a coupling phenomenon [54], the Rowhammer effect should be the strongest at closest proximity. Indeed, this assumption underpins many countermeasures (Section 2.3) that have been proposed against Rowhammer, especially the ones that rely on detecting aggressors and refreshing the charge in their intended victims (e.g., [31,35,40]). In fact, *Target Row Refresh (TRR)*, a productionized countermeasure widely deployed as part of LPDDR4/DDR4 chips, falls into this *detect-and-refresh* category [15].

In this paper, we present *Half-Double*, a new escalation of Rowhammer where we show its effect to extend beyond just the immediate neighbors. Using Half-Double, we are able to flip bits in the victim by combining many accesses to a *far aggressor* (at distance two) with just a few to a *near aggressor* (at distance one). Both aggressors are necessary: accessing just the former does not flip bits in a row that's two away, whereas accessing just the latter devolves into a classic attack that's easily mitigated. Based on our experiments, the near aggressor appears to act as a bridge, *transporting* the Rowhammer effect of the far aggressor onto the victim. Concerningly, TRR facilitates Half-Double through its mitigative refreshes, turning their recipient row into the near aggressor that co-conspires with the far one that necessitated the refresh in the first place. In effect, the cure becomes the disease.

While the discovery and evaluation of Half-Double is the main contribution of this work, we also demonstrate its practical relevance in a proof-of-concept exploit. However, current systems limit the attacker's control, introducing 4 challenges: First (**C1**), the adversary needs to allocate memory contiguous in a DRAM bank. However, without access to physical addresses [48] and huge pages [13, 19], we have to introduce a novel approach combining *buddy allocator* information with a DRAM timing side channel to reliably detect contiguous memory. Second (**C2**), ECC-protected memory can make bit flips unobservable depending on the victim data which the attacker does not control, the adversary cannot template the memory like in previous Rowhammer attacks as hammering requires knowledge of the cell data. As state-of-the-art [8, 13, 15, 19, 45, 51] does not solve this problem, we introduce a novel technique called *Blind-Hammering* to induce bit flips despite the ECC mechanism of LPDDR4x. Third (**C3**), reduced address space sizes on recent ARM-based sys-

[*] Work done while affiliated with Graz University of Technology.

[1] Named after a crochet stitch taller than a single but shorter than a double.

tems break the page table spraying mechanism from previous attacks [19, 44, 48, 51]. Therefore, we develop a new spraying technique that is still unmitigated. Finally (**C4**), without templating, we need an oracle telling whether Rowhammer induced an exploitable bit flip, without crashing the exploit. For this, we introduce a novel approach using a Spectre-based oracle for exploitable bit flips. We combine these techniques into an end-to-end proof-of-concept, the Half-Double Attack[2], which escalates an unprivileged attacker to arbitrary system memory read and write access, *i.e.*, kernel privileges. The Half-Double Attack runs within 45 minutes on a fully updated Chromebook with TRR-protected LPDDR4x memory.

To summarize, we make the following contributions:

1. We discover a new Rowhammer effect: Half-Double, and evaluate a set of devices and modules for susceptibility.
2. We perform a thorough root-cause analysis to empirically prove that TRR is responsible for the Half-Double effect.
3. We analyze the stop-gap mitigations present in today's systems and show that with a new exploit using Half-Double, we can bypass them and build an end-to-end attack.
4. Our end-to-end Half-Double Attack runs on up-to-date Chromebooks and combines the Half-Double effect with exploit techniques, side channels, and a Spectre attack.

Outline. We provide background in Section 2 and introduce a new Rowhammer pattern notation in Section 3. We overview the the Half-Double effect in Section 4 and empirically verify that it is a new effect in Section 5. We develop the end-to-end attack in Section 6. We discuss the related work and implications in Section 7 and conclude in Section 8.

Responsible Disclosure. Pre-existing contractual obligations between a subset of the authors and the memory vendors mean we cannot provide details as to the effectiveness or substance of efforts to remediate the flaws. We believe the work should nonetheless be published as the impact of disclosure is unlikely to impact consumer security substantially, as the presence of Rowhammer-type vulnerabilities is a known limitation in DRAM design prior to our publication [15, 31]. Therefore, we believe that publicly disclosing our new variant will help rather than hinder the safe deployment of systems.

We responsibly disclosed Half-Double by notifying the affected memory vendors, triggering a customary embargo. The vulnerability was made public via a blog post after the expiration of the embargo [43].

2 Background

In this section, we provide background on DRAM, the Rowhammer effect, and the broadly deployed TRR mitigation.

2.1 DRAM Organization

The main memory system consists of multiple *channels*, which are independent links between memory controller and DRAM chips. Since DRAM chips have a narrow data bus, several of them are grouped into a *rank* whose aggregated data-bus width matches that of the channel. Multiple ranks can time-share a channel. Chips in a rank run in lockstep, *i.e.*, organizationally, like a single larger chip. Hence, we use the terms "rank" and "chip" interchangeably. Each rank consists of *rows* of capacitor-based DRAM *cells*. To access a row, the voltage of its wordline must be raised, which connects its cells to their respective bitlines. Referred to as *activation*, this procedure then involves what's called the *row-buffer* – situated at the other end of the bitlines – to sense the voltage perturbations and to amplify them to either '0' or '1'. This brings us full circle as the cells are restored to their original state: fully discharged or fully charged. As long as the same row remains activated, subsequent accesses are served from the row-buffer. Such *row hits* are faster than *row conflicts* which must activate a different row. To increase the probability of a row hit, rows within a rank are partitioned into *banks* with dedicated row buffers. Capacitors and, thus, DRAM cells lose charge over time. Thus, all rows must be *refreshed* at a regular interval which is typically 32-64ms [27]. Refreshes are spread out evenly over time, *i.e.*, refreshing a small subset of rows with each *refresh* command. We emphasize that *refreshing a row is exactly the same as activating it* [38] (see Section 3).

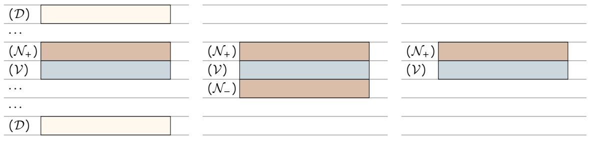
2.2 DRAM Address Reverse Engineering

Various attacks require specific placement of data in DRAM, motivating several works to reverse-engineer DRAM addressing functions. Pessl et al. [42] and later also Barenghi et al. [6] used the row buffer timing side channel. Jung et al. [29] reverse-engineered even the physical on-chip location via heat-based hardware-fault attacks. Helm et al. [21] used performance counters to measure row hits and misses. All of these methods group addresses into sets of addresses that see row conflicts or row hits with each other, *i.e.*, they are in the same bank. They then compute which combination of bits indicates the set, which is often a linear XOR-combination of bits. Helm et al. [21] showed that addressing functions can vary between different address ranges or channels. While older works found the row index to be just a subset of address bits [19, 42, 48, 56], more recent works also found XOR combinations to be used for index bits as well [49]. As a response to Rowhammer, physical addresses today are hidden from user programs [32], rendering approaches that rely on them unapplicable for attacks on up-to-date systems.

2.3 Rowhammer

At higher densities, chips are more likely to suffer from disturbance errors caused by intra-chip crosstalk [39]. In 2014,

[2] Our open-source proof-of-concept implementations can be found at: https://github.com/iaik/halfdouble



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

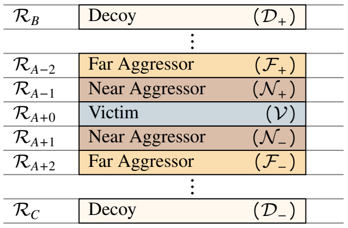
Figure 1: Rowhammer access patterns: red rectangles (□) represent hammered rows, *i.e.*, the near aggressors $\mathcal{N}$, while blue rectangles (□) represent the most likely row for bit flips, *i.e.*, the victim row $\mathcal{V}$. Single-sided hammering accesses a set of unrelated rows, which we call decoys $\mathcal{D}$ (□).

Kim et al. [31] demonstrated row-to-row disturbance errors in DRAM chips from memory accesses and called it *row hammer* [23]. Recently, Walker et al. [54] provided a comprehensive analysis of the underlying physics of Rowhammer.

Existing Rowhammer access patterns (Figure 1) vary depending on relative location of victim and aggressor(s). First, in the *single-sided* pattern [48], an attacker alternates accesses to two rows: an aggressor and what we call a *decoy*. Accesses to the decoy (an arbitrary row in the same bank) are needed to thrash the row-buffer to ensure that the aggressor is indeed activated. There is also an "amplified" variant [19] where two aggressors are placed next to each other. As the name suggests, they sometimes work to reinforce each other, yielding more bit flips in their respective victims than otherwise. Second, in the *double-sided* pattern [48], the victim is sandwiched between two aggressors. This is known to be the worst-case access pattern that induces the most bit flips. There is also a "many-sided" extension [15] that involves a larger number of aggressors and victims with varying degrees of sandwiching. Third, the *one-location* pattern [18] is similar to the single-sided one except that it eschews the decoy. Instead, it waits for the memory controller to clear the row-buffer before accessing the aggressor again to ensure that it is activated.

The Rowhammer vulnerability has been demonstrated in sandboxed environments [48], in native environments [7, 18, 48,49], in virtual machines [26,45,56], in JavaScript [8,13,19], on mobile devices [14,51], and over the network [36,50]. Rowhammer exploits often borrow traditional exploit techniques such as memory spraying [19, 48, 56], grooming [51], and page deduplication [8, 45] to place the target data structure at the correct memory location. There are many proposals to improve hammering, with special instructions [44], load hazards [25], page table accesses [59], an onboard FPGA [55], and memory pressure from quality-of-service techniques [1].

Many defenses have been proposed [18], focused on detecting [10, 20, 22, 24, 35, 40, 41, 53, 58], neutralizing [8, 9, 19, 28, 45, 51], or eliminating [4, 9, 16, 30, 31] Rowhammer in software or hardware. A defense that has already been integrated into some DDR4 modules and the LPDDR4 standard [28] is Target Row Refresh (TRR), which we discuss in Section 2.4.



$\mathcal{R}_B$	Decoy	($\mathcal{D}_+$)
	⋮	
$\mathcal{R}_{A-2}$	Far Aggressor	($\mathcal{F}_+$)
$\mathcal{R}_{A-1}$	Near Aggressor	($\mathcal{N}_+$)
$\mathcal{R}_{A+0}$	Victim	($\mathcal{V}$)
$\mathcal{R}_{A+1}$	Near Aggressor	($\mathcal{N}_-$)
$\mathcal{R}_{A+2}$	Far Aggressor	($\mathcal{F}_-$)
	⋮	
$\mathcal{R}_C$	Decoy	($\mathcal{D}_-$)

Figure 2: Row annotations for the rows inside a single bank that surround the *victim* row.

2.4 Mitigative Refreshes (a.k.a. "TRR")

Starting from the (LP)DDR4 generation, vendors have implemented opaque and proprietary defenses inside their chips. Frigo et al. [15] found that such measures appear to involve two main components: (i) a *sampler* identifying potential aggressors, and (ii) an *inhibitor* performing *mitigative refreshes* on their potential victims. Furthermore, the sampler is limited in its tracking capability and can be fooled when an attacker interleaves activations to multiple rows.

In contrast, our Half-Double Attack capitalizes on the shortcomings of the inhibitor, which is hardwired to perform mitigative refreshes on just the immediate neighbors without accounting for the longer-ranged effects of Rowhammer. In fact, we show how mitigative refreshes actually facilitate Half-Double Attack by turning their recipient row into a co-conspirator – more specifically, it becomes the near aggressor to the far one that necessitated the mitigative refresh.

In this paper, we use the terms "mitigative refresh" and "TRR" (Target Row Refresh) interchangeably. Despite its usage in previous works, the latter is a slight misnomer since it refers to a previously proposed (but never adopted) DRAM command that allows the CPU's memory controller to send a row-address alongside a refresh command [5].[3]

3 A Systematic Rowhammer Pattern Notation

In this section, we introduce a new systematic notation for Rowhammer patterns, allowing us to categorize existing attacks and describe the Half-Double Attack effect in Section 4.

Our notation describes Rowhammer patterns and their locality concerning the actual row location inside a bank.We assume the row index represents the physical row position inside a bank. For the notation, we assume that rows with contiguous row indices are physically adjacent. Figure 2 denotes the rows inside a bank as follows. The *victim* row ($\mathcal{V}$) is the target of the Rowhammer attack, and the bit flips inside this row are used to measure the effectiveness of the pattern in the experiments. The direct neighbors of the *victim* row are the so-called *near aggressor* rows ($\mathcal{N}_+$, $\mathcal{N}_-$), which are directly followed by the *far aggressor* rows ($\mathcal{F}_+$, $\mathcal{F}_-$). These three

[3] "Pseudo TRR" is emulating that behavior by sending a pair of activation and precharge commands to refresh the desired row manually.

types of rows are located in a contiguous range inside a bank. We denote rows further away from this range as *decoy* rows ($\mathcal{D}$). The absolute row position of the upper *far aggressor* row ($\mathcal{F}_+$) is denoted with $\mathcal{R}_{A-2}$, this allows us to address these rows with an index. To characterize the Rowhammer patterns, we use a special notation, e.g., $(\mathcal{A}_i \rightarrow (\mathcal{B} \rightarrow \mathcal{C})^\beta)^\infty$, where i is the current repetition of the selected pattern. Hence, the first memory access goes to $\mathcal{A}_0$. Then, the pattern accesses the rows $\mathcal{B}$ and $\mathcal{C}$ and repeats these two accesses β times. After β accesses to $\mathcal{B}$ and $\mathcal{C}$, we continue with the next iteration, *i.e.*, row $\mathcal{A}_1$ is accessed next, and so on.

With this notation, we compare known Rowhammer patterns based on their row locality. Double-sided Rowhammer [48] uses two *near aggressors* to hammer the *victim* row, *i.e.*, we can express the pattern as $(\mathcal{N}_+ \rightarrow \mathcal{N}_-)^\infty$. Single-sided Rowhammer [48] effectively uses one *near aggressor* to hammer the *victim* row and 7 *decoy* accesses, *i.e.*, we can express the pattern as $(\mathcal{N}_+ \rightarrow (\mathcal{D}_i)^7)^\infty$. However, the purpose of these decoy rows is often to trigger a row conflict on DIMMs that use an open row policy, as otherwise, the accesses are served from the row buffer (cf. Section 2.1). More recent Rowhammer type attacks like TRRespass [15] and Smash [13], also use *near aggressor* and *decoy* rows. However, in both cases, multiple *victim* rows are targeted interleaved to exploit the limited TRR *sampler* size and deplete the number of protected rows. This allows the attack to induce flips in one of the *victim* rows hammered less often as the TRR mitigation no longer protects them.

In summary, all existing Rowhammer patterns use *near aggressor* rows to hammer, *i.e.*, they are *distance-1* patterns, directly surrounding single or multiple *victim* rows. The TRR mitigation is designed to mitigate these *distance-1* type attacks. TRR detects these repeated accesses to the *near aggressors* with the *sampler*, and then the *inhibitor* refreshes the *victim* row before a bit flip can occur (cf. Section 2.4). The detailed implementation of TRR refreshes is vendor specific and not publicly documented. We assume similar to Liu et al. [38], that TRR refreshes are implemented by closing the currently open row and then opening the *victim* row to load the row into the row buffer and, therefore, refresh the content of the victim. However, this raises the question of whether there are practically exploitable *distance-2* patterns.

4 The Half-Double Effect and Exploit

This section provides an overview of the Half-Double Attack, its new hammering patterns, and challenges for the attack.

4.1 The Half-Double Effect

With Half-Double Attack, we present two new Rowhammer patterns, the *Quad pattern* and the *Weighted pattern* (or more verbosely, Weighted Single Plus Decoys (WS+D)).

The *Quad pattern* (Pattern 1) shifts the double-sided Rowhammer pattern outwards by one row:

$$(\mathcal{F}_+ \rightarrow \mathcal{F}_-)^\infty. \qquad (1)$$

However, as this only drains a small amount of charge from the actual *victim* row, the *Quad pattern* incorporates the effects of TRR refreshes to hammer the *victim* row. The pattern uses the *far aggressors* ($\mathcal{F}_+$, $\mathcal{F}_-$) to hammer. Hammering the *far aggressors* is detected by the TRR *sampler*, and after a sufficient number of row activations, the TRR *inhibitor* issues (in an attempt to mitigate bit flips) a refresh command to the *near aggressors* ($\mathcal{N}_+, \mathcal{N}_-$). The TRR refresh mechanism closes the open row and activates the *near aggressor* rows successively to refresh them. These additional activations from the TRR refresh mechanism *assist* our hammering of the *victim* row by draining further charge from the *victim* row ($\mathcal{V}$).

The *Weighted pattern* (Pattern 2) distributes half the hammers to the upper *far aggressor* ($\mathcal{F}_+$), and the others to the rows *below* the *victim* row. Thus, we represent it as

$$(\mathcal{R}_{A+4+3\cdot i} \rightarrow \mathcal{F}_+ \rightarrow \mathcal{R}_{A+6+3\cdot i} \rightarrow \mathcal{F}_+)^\infty. \qquad (2)$$

The intuition of this pattern is to shift a double-sided Rowhammer pattern over the rows below the *victim*, while distributing half of the hammers to the *far aggressor* ($\mathcal{F}_+$). As the maximum number of rows inside a bank is limited, i is wrapped around to zero if $\mathcal{R}_{A+6+3\cdot i}$ is outside the physical row range, restarting the pattern below the *victim*. The first two repetitions of the *Weighted pattern* produce the following sequence: $\mathcal{R}_{A+4}$, $\mathcal{F}_+$, $\mathcal{R}_{A+6}$, $\mathcal{F}_+$, $\mathcal{R}_{A+7}$, $\mathcal{F}_+$, $\mathcal{R}_{A+9}$, $\mathcal{F}_+$. Similar to the *Quad pattern*, the *Weighted pattern* hammers the *far aggressor* ($\mathcal{F}_+$), but accesses *decoys* below the *victim*. The pattern accesses the *far aggressor* triggering the mitigative refresh mechanism (TRR) on the *near aggressor* ($\mathcal{N}_+$) *assisting* the hammering by draining further charge from the *victim* ($\mathcal{V}$).

We describe the effects of the Half-Double patterns with the following hypothesis $\mathcal{H}$ under which the Half-Double patterns induce flips into the *victim row*.

Hypothesis $\mathcal{H}$: Hammering *far aggressors* ($\mathcal{F}_+$, $\mathcal{F}_-$) triggers mitigative refreshes (TRR) on *near aggressors* ($\mathcal{N}_+$, $\mathcal{N}_-$), implicitly assisting the hammering of the *victim* row ($\mathcal{V}$), by draining charge from it. However, the refreshes of the *near aggressors* ($\mathcal{N}_+$, $\mathcal{N}_-$) cannot draw sufficient charge from the *victim* row without the activations of the *far aggressors* ($\mathcal{F}_+$, $\mathcal{F}_-$).

Compared with multi-sided Rowhammer [13, 15], Half-Double patterns do not rely on depletion of TRR resources. Instead, the patterns incorporate the TRR refresh mechanism such that is assists the Rowhammer attack. Therefore, this pattern is also applicable in a scenario where the TRR mechanism works perfectly against *distance-1* Rowhammer attacks. We evaluate and discuss the differences between Half-Double and state-of-the-art Rowhammer attacks in Sections 5.1.2 and 7.

4.2 The Half-Double Exploit

Rowhammer exploits typically involve solving several challenges beyond the bit flip. For Half-Double on state-of-the-art systems, we identified 4 challenges: **C1** the allocation of contiguous memory (without physical address information or huge pages), **C2** finding bit flips without templating (to bypass defenses against templating), **C3** memory spraying with constrained spraying resources, and **C4** bit flip verification (due to the uncertainty created by hammering blind). For Challenges 1 and 3 we can extend existing techniques. However, for Challenge 2, ECC memory hinders bit flip templating as the ECC code depends on the data in the corresponding cells unknown to the attacker. A novel approach we call *Blind-Hammering* circumvents this problem by not templating for bit flips. However, this introduces uncertainty, creating Challenge 4, which we resolve by combining *Blind-Hammering* with a Spectre attack. Hence, we put more focus on Challenge 2 and Challenge 4 as they require novel methods.

Challenge 1: Allocation of Contiguous Memory. The first challenge is to obtain access to adjacent rows in a bank. Physical address information is unavailable today due to previous Rowhammer attacks [32]. Huge pages or page fusion mechanisms are not available on all systems, making both approaches unapplicable for our attack. Therefore, we first design a novel contiguous memory detection, incorporating knowledge of the general structure of xor-based DRAM addressing functions to obtain information on the underlying physical addresses, even if the DRAM addressing functions of the device are unknown. Combined with knowledge of the behavior of the buddy allocator, we obtain information on the underlying physical addresses. Second, due to the precise row location requirements of the Half-Double patterns, we need to reverse-engineer the row indexing function of the bank using a timing side channel. Finally, we map a virtual address via the contiguous memory to a bank and row.

Challenge 2: An Alternative to Memory Templating. After controlling contiguous rows inside a bank, the next challenge is to find flippable locations inside the memory. Due to variances in the DRAM cells, some cells are more likely to flip than others [54]. The current state of the art is templating the memory in advance for locations that are susceptible to Rowhammer flips. However, some ECC memory hinder this step as bit flips are only reproducible in an attack if the templating was performed on the exact data or data that behaves identically for the ECC code. An attacker usually does not have this information, hindering the memory templating approach. Thus, we propose a new approach without memory templating, namely *Blind-Hammering*.

Challenge 3: Memory Massaging. This challenge focuses on filling the memory with targets that are exploitable with *Blind-Hammering*. The targets of our exploit are page table entries. We target the physical page number inside these page table entries. We use an approach where we map shared memory between multiple children of the parent process to fill the memory with additional page tables without filling the main memory with other non-exploitable data pages.

Challenge 4: Bit-Flip Verification. This challenge focuses on determining the location of an induced bit flip. Due to *Blind-Hammering*, we cannot directly check whether the hammering was successful or not, as accessing a potential corrupted page-table entry (PTE) is detected by the OS, terminating the exploit. We solve this problem with a novel Spectre oracle [33] determining whether the address is safe to access. We also develop an architectural alternative oracle and evaluate the advantages and drawbacks of both approaches.

We solve the above challenges in Section 6 and gain complete control over the system's main memory, proving that the Half-Double effect can be exploited on real-world systems.

5 Empirical Evaluation of Half-Double

To show that the hypothesis $\mathcal{H}$ holds and explains the Half-Double effect, we make the following observations:

1. We show that the Half-Double effect exists, *i.e.*, inducing bit flips on current TRR-protected systems (Section 5.1).
2. We show that Half-Double does not occur without (TRR-induced) refreshes on *near aggressors* and that there is a relation between TRR refreshes and the number of bit flips observed, *i.e.*, (counter-intuitively) more TRR refreshes lead to more bit flips in the *victim* row (Section 5.2).

To obtain noise-free observations, we use an FPGA board with full control over all refreshes and memory accesses, where we have no requirements on data retainment for stability (Section 5.3). Since the focus of this section is to show the above points and, thus, that the hypothesis $\mathcal{H}$ holds, we do not restrict ourselves to a specific threat model in this section.

5.1 Half-Double on TRR-protected LPDDR4x

In this subsection, we demonstrate Half-Double using the *Quad pattern* on TRR-protected systems.[4] We show that this pattern can generate bit flips and record the number of observed bit flips to measure the performance.

5.1.1 Test System and DRAM Addressing Functions

We use 10 commodity systems (see Table 9 for a full list). We reverse-engineer the DRAM addressing functions using the method by Pessl et al. [42] (cf. Section 2.2). Since their approach only maps a physical address to a given bank but does not recover the precise row index we need for the *Quad pattern*, we use an additional timing side channel between row hits and row conflicts within a bank [49] (cf. Section 2.2) to obtain information on the row indices. We discover that our two identical ARM-based Lenovo Chromebooks have the

[4] We analyze the *Weighted pattern* in Section 5.2 and Section 5.3.

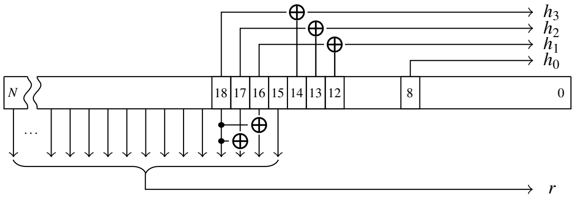


Figure 3: The reverse-engineered DRAM addressing functions from our Chromebooks.

same row scrambling functions described by Tatar et al. [49], where bit 3 is XORed onto bits 2 and 1 in the row index. We illustrate the full DRAM addressing and indexing functions for the Chromebooks in Figure 3. With the device-specific functions, we map physical addresses to banks and row indices and, thus, test the Half-Double patterns from Section 4.1.

5.1.2 Evaluation of the *Quad pattern*

We test the *Quad pattern* with two strategies to reach DRAM: uncacheable memory [52] and memory flushing [31]. For uncacheable memory, we mark the *far aggressors*, the *near aggressors*, and the *victim* as uncacheable, allowing more hammering attempts within one refresh interval. The memory flushing approach is much slower, relying on the architecture's flush instruction to flush the *far aggressors* from the cache.

For our evaluation, we allocate a large chunk of memory and use the Linux `pagemap` interface [32] to extract physical addresses. We analyze the physical addresses of the chunk and group the virtual addresses corresponding to their banks. Afterwards, we search for addresses from the same bank mapping to a consecutive range of rows representing the *Quad pattern*, *i.e.*, we find rows $\mathcal{R}_{A-2}$ to $\mathcal{R}_{A+2}$, cf. Figure 2.

Modern memory controllers scramble data by XORing a mask onto the row's data. Cojocar et al. [11] showed that the data mask is the same across all rows. We empirically observed the data scrambling and, correspondingly, set all bytes of the *far aggressor* and *near aggressor* rows to `0x55` and fill the bytes of the *victim* row with `0xaa`. We found that this maximizes the number of bit flips we see in the *Quad pattern* attack across the tested devices.

The hammering runs in a tight loop accessing the *far aggressors*. We run the *Quad pattern* for 20 000 000 iterations and check the *victim* row for bit flips. Table 1 shows the results of both approaches. The Chromebook$_2$ shows 36 times more flips than the identical Chromebook$_1$. The OnePlus 5T shows similar flip tendencies as the Chromebook$_1$. With uncacheable memory, we can induce 10 to 20 times more bit flips on the Chromebooks. However, the OnePlus 5T does not show a huge increase when using uncachable memory. We also observe more flips from 1 to 0, similar to Kim et al. [31]. However, we conclude that an attack is possible in either case, albeit faster if uncacheable memory is available.

Table 1: Performance of the *Quad pattern* with uncacheable memory and flush instruction on affected LPDDR4x systems.

System	**N**$_{Hammers}$	**UC**$_{0\to1}$	**UC**$_{1\to0}$	**Flush**$_{0\to1}$	**Flush**$_{1\to0}$
Chromebook$_1$	23 274	27	40	2	5
Chromebook$_2$	23 586	235	2379	12	101
OnePlus 5T	25 687	2	30	1	24
Pixel 3	32 921	11	5	0	0
HTC U11	21 840	-	-	3	17

Key Insight: Half-Double is capable of producing bit flips on TRR-protected memory.

To compare the Half-Double effect with current state-of-the-art multi-sided Rowhammer patterns, we performed three experiments using TRRespass [15] with up to 20 aggressors on our most susceptible commercial system, *i.e.*, the Chromebook$_2$. First, we ported the publicly available TRRespass tool to ARM, including row scrambling and uncacheable memory support to search for bit flips. Second, we implemented the patterns in our hammering tool for cross-validation. We evaluated hammering uncachable memory with the *Quad pattern* with one of a 12 aggressor multi-sided pattern under the same conditions. We did not observe any bit flips with multi-sided patterns, whereas the *Quad pattern* induced 956 flips over the same time frame. This experiment concludes that there are commodity devices that are affected by Half-Double but not (or less) by other state-of-the-art patterns. Section 7 provides further discussion of Half-Double and multi-sided Rowhammer.

5.2 Determining the Role of TRR

With the experiments on the commodity systems, we cannot rule out that the observed flips are *distance-1* flips induced solely by TRR (or other row refreshes), or that they are actually *distance-2* bit flips induced by *far aggressor* hammering. In line with prior work [54], we observe a small number of *distance-2* bit flips, too infrequent to explain Half-Double (cf. Section 5.3). Furthermore, Helm et al. [21] found complex addressing functions that change depending on the actual physical location. To exclude this possible source of error, we use a commercial SoC platform with LPDDR4x memory to measure the influence of refreshes, e.g., those from TRR.

We obtained precise but confidential information from the vendor on the relationship between the actual physical row location inside a bank and the physical address on this SoC platform. For this system, we can switch memory refreshes off and on. However, this switch disables not only the TRR refreshes but also refreshes issued by the memory controller to conform to the refresh interval (e.g., 64 ms) or by pTRR. Completely disabling refreshes renders the system unusable, as DRAM cells lose charge and corrupt data after a short period. To still be able to run actual software, we build a duty cycle mechanism alternating between enabled and disabled

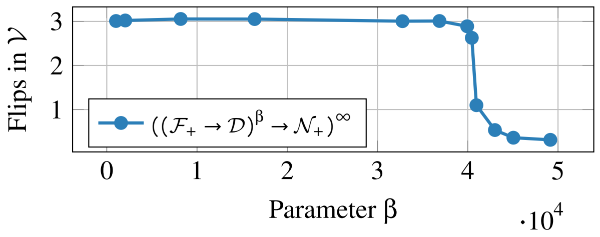


Figure 4: Number of observed bit flips in the *victim* over the dilution parameter for the single-sided case.

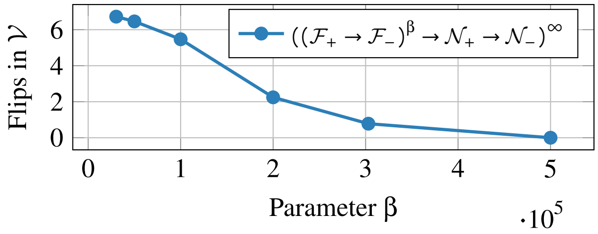


Figure 5: Number of observed bit flips in the *victim* over the dilution parameter for the double-sided case.

refreshes, we denote as *dance* (*i.e.*, dancing between refresh on and off). In these *dance* experiments, we enable refreshes for 25 % of the time, *i.e.*, 64 ms enabled and 192 ms disabled.

The *dance* experiments allow limiting the number of refreshes temporarily and, therefore, observe the correlation between refreshes and the number of bit flips. While refreshes are disabled, they cannot unintentionally assist our Half-Double patterns, *i.e.*, no interfering TRR refreshes. While the window where refreshes are disabled is longer than the standard refresh period (e.g., 64 ms), it is short enough to avoid instabilities. We expect a reduction of the number of bit flips with a decreasing number of refreshes if our hypothesis $\mathcal{H}$ holds, *i.e.*, the TRR refreshes assist the Half-Double effect.

To show that TRR refreshes *assist* the observed bit flips from Section 5.1, we run the following experiment: We design **three** pattern categories to demonstrate the Half-Double effect: the first to show that the effect is not explained by *distance-2* hammering, the second to show that simulated TRR refreshes trigger the effect as well, and the third to show that only the simulated TRR refreshes alone do not trigger the effect. The patterns used to verify the hypothesis are constructed around the *victim row*. These observations show that only the combination of the TRR refreshes (or other accesses) to the *near aggressors* combined with our accesses to the *far aggressors* trigger the Half-Double effect, confirming our hypothesis $\mathcal{H}$. For our experiment we use a *victim* row that was reliably susceptible to Rowhammer attacks, *i.e.*, we usually were able to induce three bit flips with the *Weighted pattern*.

The **first** category verifies that the observed bit flips are not caused solely by *distance-2* hammering. The single-sided Pattern S1 accesses *far aggressor* ($\mathcal{F}_+$) and a *decoy* ($\mathcal{D}$).

$$(\mathcal{F}_+ \to \mathcal{D})^\infty \tag{S1}$$

The double-sided Pattern D1 replaces this *decoy* with an access to the lower *far aggressor* ($\mathcal{F}_-$).

$$(\mathcal{F}_+ \to \mathcal{F}_-)^\infty \tag{D1}$$

While with TRR, we observed a significant number of bit flips, in our experiments, both patterns do not show any considerable number of bit flips with TRR refreshes disabled. The number of bit flips observed is far too low to visualize them or to explain the Half-Double effect by *distance-2* Rowhammer bit flips (as we detail further in Section 5.3). Hence, this indicates that TRR refreshes contribute to the Half-Double effect, in support of our hypothesis $\mathcal{H}$.

The **second** category of patterns extend the Half-Double patterns with simulated TRR refreshes to the *near aggressors* while TRR is disabled. Patterns S1 and D1 are repeated β times before simulated TRR accesses to the *near aggressors* are performed. The single-sided Pattern S2 simulates TRR by accessing the *near aggressor* ($\mathcal{N}_+$).

$$((\mathcal{F}_+ \to \mathcal{D})^\beta \to \mathcal{N}_+)^\infty \tag{S2}$$

The double-sided Pattern D2 simulates TRR by accessing both *near aggressors* ($\mathcal{N}_+, \mathcal{N}_-$).

$$((\mathcal{F}_+ \to \mathcal{F}_-)^\beta \to \mathcal{N}_+ \to \mathcal{N}_-)^\infty \tag{D2}$$

The parameter β is a *dilution* parameter allowing us to vary how many simulated TRR refreshes are performed, *i.e.*, we perform accesses to the *near aggressors* after a certain amount of accesses to the *far aggressors*.

For a low dilution parameter (*i.e.*, very high number of accesses to the *near aggressors*), these patterns behave like a traditional single-sided or double-sided Rowhammer attack without TRR protection, *i.e.*, inducing many bit flips. We confirmed this empirically, shown in Figure 4 for the singled-sided Pattern S2 and in Figure 5 for the double-sided Pattern D2. We see a correlation between the dilution parameter β and the number of bit flips, *i.e.*, fewer simulated refreshes lead to fewer bit flips. An alternative representation is also provided in Appendix B. From these observations, we conclude that accesses to the *near aggressor* contribute to the observed bit flips, supporting our hypothesis $\mathcal{H}$. However, we also need to test the null hypothesis, as we do in the following.

For this purpose, the **third** pattern category implements placebo patterns to verify that the bit flips in the second pattern category are not caused solely by the accesses to the *near aggressors* (*i.e.*, the null hypothesis). The patterns access *decoys* to keep the overall accesses rate to *near aggressors* the same as Patterns S2 and D2 for a given dilution. Thus, the single-sided Pattern S3 accesses one *near aggressor* ($\mathcal{N}_+$).

$$((\mathcal{D}_1 \to \mathcal{D}_2)^\beta \to \mathcal{N}_+)^\infty \tag{S3}$$

The double-sided Pattern D3 accesses both ($\mathcal{N}_+, \mathcal{N}_-$).

$$((\mathcal{D}_1 \to \mathcal{D}_2)^\beta \to \mathcal{N}_+ \to \mathcal{N}_-)^\infty \tag{D3}$$

Table 2: Number of bit flips observed in our FPGA setup (rows with bit flips in parentheses). The hammer duration determines the number of accesses (hammer count). The hammer duration has a stronger influence on the number of bit flips and affected rows than the dilution factor. Even a dilution factor of 3712, within one 64 ms refresh interval fits 950 272 accesses, 256 of which to the *near aggressors* simulating TRR (cf. Section 3), still induces bit flips in *all* 32 *rows*. Thus, only 256 accesses to the *near aggressors* combined with 950 016 accesses to the *far aggressors* are sufficient to attack any row.

	Accesses	296 960	356 352	415 744	475 136	534 528	593 920	653 312	712 704	772 096	831 488	890 880	950 272
	Duration	20 ms	24 ms	28 ms	32 ms	36 ms	40 ms	44 ms	48 ms	52 ms	56 ms	60 ms	64 ms
Dilution Factor	58	1 (1)	3 (3)	5 (5)	6 (6)	15 (12)	26 (19)	35 (20)	44 (23)	57 (28)	83 (30)	115 (32)	173 (32)
	116	1 (1)	3 (3)	4 (4)	6 (6)	14 (11)	24 (19)	32 (20)	40 (22)	51 (27)	73 (30)	117 (32)	152 (32)
	232	1 (1)	3 (3)	4 (4)	5 (5)	12 (10)	24 (19)	31 (20)	39 (21)	51 (27)	68 (30)	112 (32)	149 (32)
	464	1 (1)	2 (2)	3 (3)	5 (5)	11 (8)	24 (18)	32 (20)	39 (21)	49 (26)	70 (30)	109 (32)	148 (32)
	928	1 (1)	2 (2)	3 (3)	5 (5)	11 (8)	25 (18)	32 (20)	39 (21)	49 (25)	70 (29)	108 (32)	146 (32)
	1856	0 (0)	2 (2)	3 (3)	5 (5)	11 (8)	22 (17)	32 (20)	37 (21)	49 (25)	66 (29)	110 (32)	140 (32)
	3712	0 (0)	2 (2)	3 (3)	5 (5)	10 (7)	22 (16)	30 (20)	37 (21)	49 (25)	64 (27)	99 (31)	139 (32)
	7424	0 (0)	2 (2)	3 (3)	5 (5)	8 (6)	18 (15)	29 (19)	36 (20)	48 (25)	66 (27)	92 (31)	128 (31)
	14 848	0 (0)	0 (0)	2 (2)	4 (4)	7 (6)	15 (12)	22 (15)	32 (19)	40 (22)	58 (27)	80 (30)	109 (30)
	29 696	0 (0)	0 (0)	2 (2)	2 (2)	3 (3)	8 (7)	11 (9)	19 (14)	28 (18)	41 (25)	57 (27)	82 (29)

Table 3: The modules used in the FPGA analysis. M_1 is not affected by Half-Double, M_3 is affected within default refresh windows (64 ms), M_2 is affected with longer windows.

Module	**Freq.**	**Size**	**Ranks**	**Banks**	**Pins**	**Half-Double**
M_1	2666	4 GB	1	8	x16	✗
M_2	3200	4 GB	1	8	x16	✓(>64 ms)
M_3	3200	8 GB	1	8	x16	✓

When varying the dilution parameter, we observe a point at which the second category of patterns still produce bit flips in the *victim*, whereas the third category no longer does. More concretely, Pattern S3 shows a decrease in the bit flips before Pattern S2. We observe the same effect also for the double-sided case where the number of bit flips with Pattern D3 drops at a lower dilution parameter than the number of bit flips with Pattern D2. Since we only access *decoys* $\mathcal{D}_i$, unrelated to the *victim* row $\mathcal{V}$, this drop is explained by the missing accesses to the *far aggressors* $\mathcal{F}$, supporting our hypothesis $\mathcal{H}$. We conclude that the additional accesses of the TRR *inhibitor* assist the hammering of the *victim* for the Half-Double effect.

Key Insight: TRR refreshes assist Half-Double but are not the root cause. They alone induce no bit flips in the *victim*.

5.3 Noise-free FPGA Experiments

To confirm our results without noise, we use the *ZCU104* FPGA platform[5] where we have full control over all refreshes and memory accesses and no requirements on data retainment for stability. In contrast to Section 5.2, we can disable all refreshes on the FPGA-based platform because the platform itself does not store any data in the DIMM, *i.e.*, the FPGA does not use the DIMM as system memory.

We analyzed three off-the-shelf DDR4 DIMMs listed in Table 3 and found that M_1 is not susceptible to the Half-Double effect. While the other two are affected, we could only demonstrate bit flips with the default refresh interval of 64 ms on M_3, whereas M_2 required a doubled refresh interval (128 ms). Therefore, for our analysis, we focused on M_3, the DIMM susceptible to Half-Double by default. For our experiments, we use the same patterns as in Section 5.2 and confirm our results in this highly controlled and noise-free setup.

$$((\mathcal{F}_+ \rightarrow \mathcal{F}_-)^{\beta} \rightarrow \mathcal{N}_+ \rightarrow \mathcal{N}_-)^{\infty} \tag{D2}$$

With Pattern D2, we hammer 32 rows individually and vary both dilution parameter β and total hammer duration (*i.e.*, hammer count, number of accesses) in this experiment. Furthermore, we vary the hammer duration from 20 ms to 64 ms with a step size of 4 ms. In contrast to the dilution parameter, we introduce the dilution factor d_f. This factor slightly changes the representation of β. The relation between the dilution factor and the dilution parameter for Pattern D2 is $d_f = \beta + 1$. A dilution factor of 58 refers to 1 *distance-1* hammer in every 58 hammers. Therefore, we can compute the accesses to the *near aggressors* directly by dividing the total hammers by the dilution factor. The dilution factor is varied from 58 to 29 696 by doubling it in each step.

Table 2 shows the results of this experiment and we can observe two effects, as expected: First, the number of bit flips increases with the hammer duration. We can induce bit flips in all 32 rows within one default refresh interval (64 ms) regardless of the tested dilution factor. Second, the number of bit flips decreases with a higher dilution. However, the decrease in bit flips is much flatter than for the hammer duration. Even with the highest tested dilution, we induce flips into 29 out of 32 rows within one default refresh interval.

To underline that the Half-Double effect is a different phenomenon than *distance-1* and *distance-2* Rowhammer effects, we test two more patterns on the FPGA system and compare

[5] https://github.com/antmicro/litex-rowhammer-tester

Table 4: *Distance-1* double-sided hammering $(\mathcal{N}_+ \rightarrow \mathcal{N}_-)^\infty$ and the observed bit flips per cell and row.

Hammers	Time (ms)	Cells	Rows
18 000	1.212	2	1
24 000	1.616	23	18
30 000	2.020	136	31
36 000	2.425	495	32
42 000	2.829	1395	32
48 000	3.233	2870	32
54 000	3.637	5099	32
60 000	4.041	7749	32

Table 5: *Distance-2* double-sided hammering $(\mathcal{F}_+ \rightarrow \mathcal{F}_-)^\infty$ and the observed bit flips per cell and row.

Hammers	Time (ms)	Cells	Rows
4 000 000	270	1	1
5 000 000	336	1	1
6 000 000	404	2	2
7 000 000	472	2	2
8 000 000	538	3	3
9 000 000	606	2	2
10 000 000	674	3	3

them with the results obtained in the previous experiment. We use *distance-1* double-sided Rowhammer $(\mathcal{N}_+ \rightarrow \mathcal{N}_-)^\infty$ and the *distance-2* variant $(\mathcal{F}_+ \rightarrow \mathcal{F}_-)^\infty$. We again hammer 32 rows and measure the observed flips on a cell and row basis.

Table 4 shows the results of *distance-1* double-sided hammering, where the number of hammers required to induce flips into all 32 rows is only 36 000. This is 25 times smaller than with Half-Double, indicating that Half-Double is not just *distance-1* Rowhammer. However, 36 000 accesses are also much higher than what a TRR implementation could perform within the standard 64 ms refresh interval. Even at a low dilution factor like 58, this would require about 2 088 000 accesses within one refresh interval, *i.e.*, about twice as many accesses than fit in the standard refresh interval.

Table 5 shows *distance-2* double-sided hammering and we observe that we need 4 000 000 hammer accesses to obtain a *single distance-2* bit flip, *i.e.*, four times more accesses than fit within a 64 ms refresh interval. Hence, Half-Double can also not be explained with *distance-2* bit flips.

In line with Section 5.2, this again shows that $\mathcal{H}$ holds. With our results from Table 2, we can model when the Half-Double effect occurs. With a dilution factor of 3712 and 950 272 hammers, the total number of accesses to the *near aggressors* is 256. Therefore, only 256 accesses to the *near aggressors*, combined with 950 016 accesses to the *far aggressors* are sufficient to induce flips in each of the 32 rows. However, if we compare these numbers with the equivalent accesses in the *distance-1* and *distance-2* experiments, we are far below the required accesses to see even a single bit flip in both cases.

Key Insight: Both *distance-1* Rowhammer and *distance-2* Rowhammer effects would require more accesses than fit inside the standard 64 ms refresh interval if they would induce the Half-Double effect. Hence, we conclude that $\mathcal{H}$ is the most plausible explanation of Half-Double.

6 Half-Double Attack Exploit

In this section, we demonstrate the real-world attack capabilities of the Half-Double Attack. The attack is split into multiple phases, each tackling one of the challenges (see Section 4.2) to finally gain complete control over the system from within an untrusted executable. Our attack aims to induce a bit flip in the physical page-frame number of a PTE. If the corrupted page-frame number points to attacker-controlled data instead of the original page table, the adversary can forge additional page table entries. This grants the adversary arbitrary read and write access to the entire system memory.

Threat Model. We assume that the victim runs an untrusted executable or Android APP on either an ARM or x86 based system for our attack. Our attack does not exploit any software vulnerabilities in the OS or other running programs but only uses side-channel information and the provided interfaces by the OS. Furthermore, we assume that the LPDDR4x DRAM used on the system is both ECC- and TRR-protected. However, our attack does not rely on the exhaustion of TRR resources like previous Rowhammer attacks targeting TRR [15]. We evaluate the challenges on the Chromebooks, the OnePlus 5T and a Lenovo T490s to show the applicability across multiple architectures and operating systems (see Appendix A).

From Virtual Memory Accesses to Half-Double Patterns. In the first step of our exploit, we map virtual addresses to actual physical row locations inside the DRAM banks, a building block to hammer with the Half-Double patterns. While we can use the *Quad pattern* and the *Weighted pattern*, we focus on the *Quad pattern* in our exploit as it induces bit flips faster. For the *Quad pattern*, we need to control at least five adjacent rows where the middle row is unmapped and used by the victim process. With DRAM addressing functions (cf. Section 5.1.1 and Figure 3), we can determine the physical location inside a bank and row. However, the required physical address information is not available to the unprivileged executable. We solve this challenge (**C1**) in Section 6.1.

Inducing Bit Flips. In the second step, we need to place potential bit flip targets at the right memory locations. Templated bit flips are very likely not reproducible during the actual attack, as the data in the victim rows differs between templating and attack phase, and the integrated ECC mechanisms of the DRAM depend on the actual data stored in the victim row. Consequently, bit flips during templating may not occur when attempting to fault the targeted data during the attack. Therefore, in Section 6.2, we present two new ways to solve **C2**. The first technique uses an alternative templating process that is ECC-aware. The second technique,

Table 6: *Page distance* patterns on the Chromebooks. Each pattern has one unique *page distance* highlighted in yellow.

P	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}	d_{13}	d_{14}	d_{15}	d_{16}	…
$\mathbf{P}_0$	8	9	8	9	8	9	8	9	8	9	8	9	8	9	8	**1**	…
$\mathbf{P}_1$	8	7	8	11	8	7	8	11	8	7	8	11	8	7	8	**3**	…
$\mathbf{P}_2$	8	9	8	5	8	9	8	**13**	8	9	8	5	8	9	8	5	…
$\mathbf{P}_3$	8	7	8	7	8	7	8	**15**	8	7	8	7	8	7	8	7	…

called *Blind-Hammering*, is a versatile alternative to templating. However, verifying whether exploitable bit flips occurred becomes its own challenge then as we outline below.

Placing Exploitable Data. In the third step, we fill the system's memory with PTEs. However, due to address space limitations modern ARM-based devices enable for performance reasons, we cannot use the same spraying techniques as prior Rowhammer attacks. In Section 6.3, we solve this challenge **C3** by spawning child processes to increase the number of page tables in memory via multiple address spaces.

Bit-Flip Verification. We cannot directly access the hammered victim rows. Attempting to access a corrupted mapping is also fatal, as the Linux kernel detects corrupted PTEs upon faults and terminates the corresponding user-space process. In Section 6.4, we solve **C4** and use a Spectre attack to prevent irrecoverable crashes of our attacking app.

Combining all steps, we obtain a full **end-to-end exploit** with read and write access to the entire system's memory.

6.1 C1: Memory Allocation

To use the Half-Double patterns, the adversary needs access to at least 5 adjacent rows within the same bank. We present three distinct approaches to solve this challenge. Either via huge pages, using unique bank access patterns if the DRAM addressing functions are known, or by using the structure of unknown xor-based DRAM addressing functions

Via Huge Pages. The Chrome OS running on the Chromebooks as well as the Ubuntu running on the T490s has transparent huge pages activated. For 2 MB huge pages, the lowest 21 bits of virtual and physical address are the same. This covers all bits we need to find adjacent rows across the test systems, effectively solving this challenge (cf. Figure 3).

Via DRAM Addressing Functions. Disabling huge pages mitigates the aforementioned approach. Unfortunately, Half-Double requires specific row index information going beyond contiguency information from prior work [14, 34, 47]. However, we can combine information on DRAM addressing functions (cf. Figure 3) and the *buddy allocator* [17] used in Linux and Chrome OS to detect contiguous memory blocks and reconstruct the additional physical address bits we require.

When dealing with a contiguous range of physical memory, the pages of the memory range are distributed over multiple banks due to the DRAM addressing functions. We now select only pages of a given bank and iterate over all the allocated pages to analyze the *page distances*. The *page distance* is the distance between two pages in the same bank. If we fix the bank and analyze the allocated memory, we observe that the *page distances* on the Chromebooks follow one of four patterns. Table 6 shows these four *page distance* patterns. The patterns have a period of 16, and each pattern has one unique *page distance*, which is highlighted in yellow. We only find four patterns since we can skip bit 8 of the DRAM addressing functions, as we always control it with the virtual address, leaving only eight remaining banks. However, we only observe four patterns as two banks share the same pattern.

Table 7: Reconstruction of physical address bits via unique *page distances*.

Page	From							To						
Distance	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}
1	B	1	1	1	1	1	1	$\bar{B}$	0	0	0	0	0	0
3	B	1	1	1	1	1	0	$\bar{B}$	0	0	0	0	0	1
13	B	1	1	1	0	0	1	$\bar{B}$	0	0	0	1	1	0
15	B	1	1	1	0	0	0	$\bar{B}$	0	0	0	1	1	1

All four *page distance* patterns have one unique distance per period (1,3,13, or 15). This unique value allows us to reconstruct the physical address bits 12 to 17, since the DRAM addressing functions start at bit 12 (cf. Figure 3). If we advance the unique *page distance* in pages (4 kB) in the physical memory, the underlying address bits 12 to 17 of the physical address change. However, due to the *page distance* analysis with the timing side channel [34, 47], we know that the page advanced by the unique *page distance* falls into the same bank. Therefore, changing the address bits did not influence the outcome of the bank from the DRAM addressing functions. This can only be the case once for each bank. Due to the unique *page distance*, we know that we fall into one of two banks, and therefore, this analysis only leaves bit 18 unknown. Table 7 shows the reconstructed physical address bits for each unique *page distance* value, depending on bit 18. This 50 % probability acts as a corresponding slow down for the attack.

Via DRAM Addressing Structure. We generalize the previous approach by formulating the general structure of xor-based DRAM addressing functions in the Z3 theorem prover [12]. Schwarz et al. [47] use a similar solver to recover physical address parts from *known* DRAM addressing functions and Kwong et al. [34] use a solver to recover bank information from *contiguous* memory. We in contrast retrieve *contiguous* memory information via bank access patterns without *knowing* the DRAM addressing functions.

In the exploit, we record the bank access pattern when iterating over each page of a virtual address range and determine the corresponding bank affinity via the row-conflict timing side channel [42]. The solver uses the pattern and the underlying structure of the xor-based DRAM addressing functions to implement the following question: *Can this bank access pattern be generated via a xor-based addressing function when walking over contiguous physical memory?* If the constraints are unsatisfiable, the underlying memory range is not contigu-

Initial	0**1**1010**0**0	0**0**1010**0**0	0**1**1010**1**0
	↓	↓	↓
Flipped	0**0**1010**1**0	0**0**1010**1**0	0**0**1010**1**0
	↓	↓	↓
Corrected	0**0**1010**1**0	0**0**1010**0**0	0**1**1010**1**0
	(a) c_1 and c_6 flip	(b) c_1 flips but is corrected	(c) c_6 flips but is corrected

Figure 6: Error correction of 8 memory cells with ECC (from c_7 to c_0). We effectively never see single bit flips.

ous or, the addressing functions are not xor-based. However, if the constraints are satisfiable, the solver found addressing functions and a physical start offset that generate this access pattern. The constraints are detailed in Appendix C. We restrict the bits of the xor masks to only cover physical address bits 12 to 20, as the page offset controls bits below 12, and bits above 20 are not needed to find adjacent rows.

Evaluation. The memory scan takes less than 10 seconds with 2 MB pages and less than 3 minutes using the *page distance*, *i.e.*, 19.05 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.002) on Chromebook$_1$, 13.03 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.003) on Chromebook$_2$, 18.39 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.006) on the OnePlus 5T and 46.55 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.455) on the T490s.

Finally, we evaluated the correctness of the solver by generating physical address ranges of 512 pages consisting of uniformly generated contiguous memory blocks of up to 128 pages. This memory range is transformed via a DRAM addressing functions into a bank access pattern, where we additionally scrambled the bank index. We vary the pattern length that the solver receives as input and *slide* the solver over the whole memory range and compute the F-score metric. The solver achieves an average F-score of 0.97 with a bank access pattern length of 64 samples and an average scanning speed of 1.079 MB s^{-1}. Further details on performance and correctness of the functions are provided in Appendix C.

6.2 C2: Alternative to Memory Templating

Due to semiconductor production variances, some cells are more susceptible to Rowhammer than others [54]. Because of that, most Rowhammer bit flips are reproducible, and their direction (0 → 1 or 1 → 0) is fixed [31] as well. The affected systems (cf. Table 1) use LPDDR4x DRAM with ECC with a typical single-error-correction code[6]. We empirically verified this with our observation that we see no single, but only double bit flips. The reason is that the ECC memory requires at least two bit flips to show an effect within a code word. Otherwise, the bit flip is corrected and not exploitable. Figure 6 visualizes this effect with 8 data bits (parity bits are not shown). Therefore, we propose two techniques for the Half-Double Attack to work around this data dependency, an improved templating technique for ECC memory, and *Blind-Hammering*, which does not require any bit flip templating.

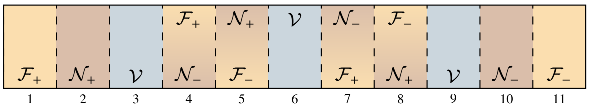


Figure 7: The *zebra* pattern used for *Blind-Hammering*. Notice how the *near aggressors* and *far aggressors* change depending on the *victim* row.

ECC-aware Templating. Classic memory templating fills the aggressor rows with a specific byte value and fills the victim row with the inverse of this value. During our experiments, we find many weak cells when filling the aggressor rows with `0x55` and the victim row with `0xaa`. However, when moving to the next stage of the exploit where the victim row is filled with PTEs, we initially no longer observed any flips. The reason for this is the one outlined before and illustrated in Figure 6: Bit flips on ECC memory depend on the data that is stored in the cells [34]. Therefore, we need to adapt the templating phase to incorporate a presumed structure of the data we target, *i.e.*, fake PTEs when targeting page-frame numbers. Hence, during templating, we fill the victim rows with fake PTEs where the page-frame number is filled as in the regular templating approach, e.g., we fill the aggressor rows with `0x5555555555555` and victim rows with `0x68000AAAAAAFD3`. We keep track of all addresses that produce flips at the offset of the page-frame number field during the evaluation. Afterwards, we use these addresses to induce bit flips in a target page placed in the victim row.

***Blind-Hammering*.** The disadvantage of ECC-templating is that it requires precise knowledge of the target data. *Blind-Hammering* generalizes our attack further and makes no assumptions about the target data in the victim row. Instead, it circumvents the ECC data dependency problem by not depending on the repeatability of the bit flips over changing victim data. *Blind-Hammering* skips the templating phase and hammers as many rows as possible with real page tables in potential victim rows. *Blind-Hammering* creates a *zebra* pattern (cf. Figure 7) by mapping contiguous memory (cf. Section 6.1) and then unmapping parts of the allocated memory to make room for the victim rows shown in blue. In essence, it performs the templating directly on the inaccessible victim rows and using the victim's own data for the attack. Consequently, the trade-off for *Blind-Hammering* is similar as for the templating phase. While *Blind-Hammering* enables targeting ECC memory, it has a clear drawback: The attacker cannot simply read the data anymore to check whether a bit has flipped. We elaborate this problem further in Section 6.3, motivating challenge **C4** that we then solve in Section 6.4, *i.e.*, the need to verify that a bit has flipped in an exploitable way without crashing the attacker process.

[6] We have not seen any freezes due to error detection, indicating that it is only single-error correction with no support for double-error detection.

Evaluation. We evaluate *Blind-Hammering* on Chromebook$_2$ and observed 30 exploitable bit flips (13 flips 0 → 1, 17 flips 1 → 0) within 11.6 h. This gives us an average of 2.59 exploitable flips per hour, or 23.2 minutes on average to produce an exploitable flip. We used the ratio of overall bit flips to exploitable bit flips of the Chromebook$_2$, to estimate the exploitation time on our other devices. On the OnePlus 5T it takes approximately 6.4 h, on the HTC U11 4.0 h and on the Chromebook$_1$ 4.2 h, to flip an exploitable bit in a PTE.

6.3 C3: Memory Preparation (Spraying)

In this section, we fill the memory of the target systems with our attack targets, the PTEs. Modern ARM-based platforms, *i.e.*, mobile platforms, can reduce the levels of page tables from the default of 4 page-table levels to only 3 page-table levels to optimize the performance of page walks (on TLB misses). However, this also decimates the available virtual address space for every process by a factor of 512. Since the affected devices (cf. Table 1) use this approach, we only have a virtual address space of 512 GB available. While this is still much more than the amount of physical memory the device has, it severely limits the practicality of page-table spraying using file mappings. Previous work has used file mappings and other memory mappings to fill memory with page tables [19, 44, 48, 51]. However, with only 512 GB of virtual address space available, we can only create mappings requiring less than 262 144 page tables, *i.e.*, taking up 1 GB of memory. Thus, we can only occupy, e.g., 25 % of the available 4 GB on the Chromebooks. This increases the attack duration by a factor of more than 8 (since not all pages can be mapped).

To bypass this aggravating effect, we propose a new technique called *Child Spray*. Instead of spraying only our own virtual memory with mappings to allocate page tables, *Child Spray* spawns child processes that share memory with the parent process. The shared memory is only once in the physical memory, but each process has its own page-table hierarchy, effectively spraying the physical memory with page tables. The only disadvantage of *Child Spray* is that a hammered PTE can point to a page table of a child process, leading to extra engineering steps for successful exploitation.

With this spraying approach and *Blind-Hammering*, bit flips can now occur in any page table any time, as we don't know which cells are vulnerable and where page tables are. We can check page tables periodically for changes by checking whether the shared memory mapping still has its expected content. However, as *Blind-Hammering* bit flips in page tables are, in contrast to templated bit flips, not predictable, PTEs often become invalid. Consequently, we need a method to test whether a bit flip in a page table occurred without crashing the attacker process. We solve this challenge in the following.

Evaluation. The *Child Spray* runs with two child processes at 79.39 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.24) on Chromebook$_1$, 54.09 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=1.437) on Chromebook$_2$, 25.42 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.346) on the OnePlus 5T, and 99.88 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.456) on the Lenovo T490s. Thus, the memory is filled with page tables within less than 1 minute on average. The *Child Spray* on the T490s is not required as the system supports 4 page table levels.

```
if (misprediction)
    access(probe + (pointer[0] & 1) + ... + (pointer[4] & 1));
if (flush_reload(probe) == CACHE_HIT)
    // Report valid address
```

Listing 1: Example code for our *Speculative Oracle*. The attacker learns whether `pointer[0]` till `pointer[4]` are accessible memory locations or would raise a CPU fault. If a fault is detected, the attacker probes each address individually.

6.4 C4: Robust Bit-Flip Verification

With *Blind-Hammering* we cannot verify the success of a bit flip. Reading the corrupted data (as templating does) is not possible as it is not in our process. Accessing potentially remapped memory often crashes the attacker's process due to corrupted PTEs (e.g., mapped to an invalid physical memory region, or setting of a reserved bit), as the OS detects this corruption upon a faulting access. Consequently, we develop a new technique, combining the Half-Double Rowhammer attack with a Spectre [33] side-channel mechanism allowing us to safely determine whether an address can be accessed or whether accessing it would crash the attacker's process. With the *Speculative Oracle*, we can check whether a mapping is corrupted or not without triggering the OS's own detection.

6.4.1 Speculative Oracle

If the physical page-frame number points to an illegal memory location, a read or write to it raises a CPU fault, e.g., a Data Abort [2]. As CPU faults cannot succeed during speculative execution on systems that are not susceptible to Meltdown [37], we can use speculative execution to determine whether the bit flip corrupted the entry in a defective way or, otherwise, in an exploitable way. This allows us to avoid accesses that would make the OS terminate our attack process.

Our *Speculative Oracle* uses Spectre similar to Lipp et al. [37] in the Meltdown attack for *exception suppression* on an ARM-based mobile phone. Our Chromebooks use a Mediatek MT8183 SoC with ARM cores that are not vulnerable to Meltdown [3]. Thus, loads depending on the faulting load are not user-visible executed on this ARM microarchitecture.

Our *Speculative Oracle* uses exception suppression by mistraining branch predictors to execute the probing code transiently [37], cf. Listing 1. We transiently load *probe* with an offset based on *pointer*, which is the address to test. There are two possible cases for the *Speculative Oracle*: *pointer* is **valid** and, hence, its value forwarded to the *probe* load. Thus,

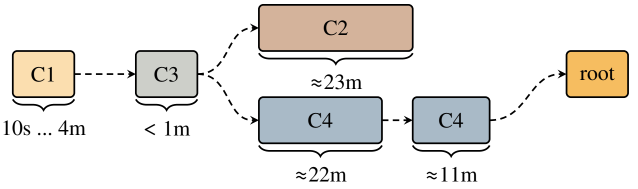


Figure 8: The timing durations for the end-to-end exploit executed on the Chromebook$_2$. For Challenges **C1** and **C4** faster alternatives might be available (cf. Sections 6.1 and 6.4). The overall runtime is bound by Challenge **C2**, *i.e.*, the time it takes to induce an *exploitable* bit flip. Afterwards it takes on average halve **C4** to find the bit flip.

Table 8: Overview of the challenges, their alternatives and availability across multiple platforms.

	Alternative	Requirement	Available on	Prior Work
C1	Physical Address Access	OS-enabled	Linux-based Systems	[48]
	Huge Pages	OS-enabled	Linux-based Systems	[13, 19]
	Bank Differences	Known Functions	DDR-based Memory	[34, 47]
	Solver	XOR-based Functions	DDR-based Memory	[34, 47]
C2	Templating	no ECC	Systems without ECC	[8, 13, 15, 19, 44, 48, 51]
	Blind-Hammering	Half-Double affected	see Tables 1 and 3	-
	ECC-Aware Templating	Half-Double affected	see Tables 1 and 3	[11, 34]
C3	Spray Children	`fork`	ARM64, x86	-
	Spray Page Tables	4 Page Table Levels	ARM64, x86	[19, 44, 48, 51]
C4	`vfork`	OS-enabled	Linux-based Systems	
	Speculative Oracle	Hardware	ARM64, x86	-

probe is loaded into the cache. *pointer* is **invalid** and, hence, the *probe* load is not executed and, thus, not loaded into the cache. Using Flush+Reload [57], we determine whether *probe* is cached or not and, thus, whether *pointer* is valid or not.

As our probing gadget runs in the transient domain, the misspeculated branch may be corrected before the load is issued or the branch may not be mispredicted at all. Thus, *probe* may not be cached even when *pointer* is valid. Hence, we repeat the Spectre attack several times to more likely see a cache hit if *pointer* is valid. However, if *pointer* is invalid, *probe* can never be loaded into the cache and, hence, we infer *pointer* to be invalid with a high probability if we can not observe a cache hit after a certain number of repetitions. Additionally, we can probe multiple addresses at once by chaining them as additional dependencies to the *probe* address as shown in Listing 1. This significantly improves the runtime by allowing to coarsely scan multiple address candidates at once before probing the candidates separatetely.

Evaluation. We evaluate the success rate and runtime of our *Speculative Oracle*. Since a cache hit on *probe* can only be observed if the address under test is valid, our method has a false negative rate (classifying an invalid address as valid) of zero. Therefore, we can only misclassify a valid address as invalid if we do not observe a cache hit within our repeated trials of triggering the probe gadget.

We evaluated the success rate of the target address classification and the runtime for different numbers of Spectre attacks. We probe 5 addresses at once where either all of them are valid or one random one of them is invalid. We repeat the experiment 10 000 times, 5000 times for valid and 5000 times for invalid addresses. With a single probe try, we already achieve a success rate of 99.01 % with an average runtime of 0.008 ms (n=10 000, $\sigma_{\bar{x}}$=0.002) on the Chromebook$_1$. Chromebook$_2$ achieves 99.68 % success rate with two tries and an average runtime of 0.025 ms (n=10 000, $\sigma_{\bar{x}}$=0.006). On the OnePlus 5T we achieve a success rate of 99.24 % with three tries and a runtime of 0.034 ms (n=10 000, $\sigma_{\bar{x}}$=0.011). We also evaluated this technique on x86-based systems where we used the RSB misprediction for performance reasons. The Lenovo T490s achieves a success rate of 99.94 % with 20 tries and a runtime of 0.018 ms (n=10 000, $\sigma_{\bar{x}}$=0.004).

Hence, since almost all addresses remain valid and without bit flips, the verification with our *Speculative Oracle* consumes 19.0 minutes of CPU time to scan 2 GB of PTEs for bit flips. However, this scan can run on a second core in the background during *Blind-Hammering* (see Figure 8).

Architectural Alternative. As an alternative to the *Speculative Oracle*, we propose an architectural approach, namely using the `vfork` system call. `vfork` creates, similarly to `fork`, an exact copy of the calling process with the only difference that the page tables are not copied. Its primary purpose is to provide a faster version of `fork` for child processes that immediately execute another process via `exec`. Our `vfork` oracle creates a child process that scans 2 GB of PTEs for bit flips. The child process gets killed by the kernel if the page translation is corrupted and returns cleanly otherwise. By checking how the child process died, we know whether the address range is safe to access. If the child process was killed we use shared memory to communicate the last address accessed to the parent process. This minimizes the number of `vfork` invocations down to one per corrupted PTE.

Evaluation. The `vfork` alternative scans 19.45 GB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.039) on the Chromebook$_2$, 256.47 GB s^{-1} (n=10, $\sigma_{\bar{x}}$=3.971) on the T490s. The bandwidth numbers approach nearly the maximum memory bandwidth of the systems. Hence, with this approach the verification consumes merely 56 s of CPU time on the Chromebook$_2$ and only 4.3 s on the T490s to scan 2 GB of PTEs for bit flips. The disadvantage of this approach is that it is trivial to mitigate by disabling our specific use of `vfork` in the kernel, e.g., as on the OnePlus 5T where the `vfork` instruction is aliased to `fork`. Nevertheless, the speculative oracle is still available.

6.5 End-to-End Attack Evaluation

Figure 8 shows the combined runtimes of all attack steps, with less than 5 minutes for contiguency (**C1**) and spraying (**C3**). *Blind-Hammering* (**C2**) takes on average 23.2 minutes on our Chromebook$_2$ to find an exploitable bit flip. Fourth, the *Speculative Oracle* (**C4**) runs in parallel to the *Blind-Hammering* and consumes 22 minutes. After the bit flip, the exploit cleans up, scans physical memory, and sets up page tables for convenient arbitrary read and write, in less than 3 minutes. Thus, the total runtime usually stays below 45 minutes but varies with the time until an exploitable bit flip occurs. On our other devices the total exploitation time is primarily determined by the time it takes to flip an exploitable bit, as the other exploit steps are negligible fast in comparison (cf. Section 6.2). Table 8 summarizes the challenges and the requirements for the solutions to be applicable.

7 Discussion

Recently, *TRRespass* [15] has fuzzed hammering patterns and showed that various TRR-protected DDR4 DIMMs are still susceptible to Rowhammer. The generated *many-sided* (3- to 19-sided) patterns worked on 13 out of 42 modules tested. The underlying effect they exploit is an optimization in TRR implementations, where DRAM modules count accesses only to a limited number of rows, which the attacker can exhaust. TRR then loses track of the number of accesses to *near aggressor* (*distance-1*) rows. Compared with the multi-sided Rowhammer patterns from TRRespass [15] and Smash [13], our Half-Double patterns do not rely on the depletion of TRR resources. Instead, the patterns directly incorporate the TRR refresh mechanism into the attack. Therefore, our patterns even work where the TRR mechanism works perfectly for detecting and mitigating *distance-1* Rowhammer.

Applicability to other Systems (including x86). We evaluated all building blocks of the end-to-end attack on other systems as well, in particular x86. Half-Double also exists on TRR-protected DDR4 memory on x86 systems, Some x86 processors, e.g., Xeon, use pTRR to introduce similar accesses to *near aggressors* and, hence, could be used in the Half-Double Attack attack. The contiguous memory detection is also applicable both on other Arm- and x86-based system. *Blind-Hammering* also works on both x86 and Arm. Spraying on x86 systems is easier as no child processes are required and techniques from prior work still apply. We show that the speculative oracle can either be adapted (e.g., using Spectre-RSB instead of Spectre-PHT, or adapting thresholds) to the specific system or be replaced by `vfork` which does not depend on microarchitectural behavior.

Mitigations. To mitigate Half-Double we discuss a short-term defense to protect the next generation of DRAM chips and a more generic Rowhammer defense. First, we propose (p)TRR±2, extending the existing victim-oriented refresh-based mitigations also to include *distance-2* aggressors. This mitigation would minimize the hardware changes as only the *inhibitor* of a TRR-based design needs to be adapted to refresh additional rows. Furthermore, as the results from Table 2 suggest, a more sophisticated design would refresh *distance-2* rows with a lower frequency than *distance-1* rows. Second, as DRAM cell density will further increase, we assume that the influence of the Half-Double effect will rise, increasing the need for a more generic Rowhammer protection. Saileshwar et al. [46] proposed to replace victim-oriented defenses like TRR with an attacker-oriented defense. They propose to swap attacker rows after a certain activation count is reached with another row within the same bank using a permutation layer. This mechanism statistically breaks the locality between aggressor and victim rows and makes it therefore highly unlikely to continuously hammer the same victim.

To harden affected systems against our end-to-end exploit, we propose to tackle the contiguous memory allocation, and the bit flip verification (cf. Sections 6.1 and 6.4). If the underlying system allocator ensures that the allocation never returns continuous pages, the attacker has to resort to a brute force approach to find the correct *far aggressors* to induce Half-Double bit flips. Furthermore, the `vfork` system call can be aliased to `fork` removing this gadget from the system.

8 Conclusion

We presented a new and unmitigated Rowhammer effect, Half-Double. Half-Double induces errors in a victim by combining a large number of accesses to a “far” aggressor (at distance two) with just a handful (dozens) to a “near” aggressor (at distance one). This is problematic on DRAM with mitigative refreshing as a Rowhammer protection (e.g., “TRR”), as protections implicitly access the near aggressors and, thus, instead of preventing Rowhammer assist Half-Double in inducing bit flips. We evaluate Half-Double thoroughly and demonstrate its practical relevance in an end-to-end Rowhammer attack. To overcome the challenges for an end-to-end attack on recent off-the-shelf devices, we used side-channel attacks, a novel technique called *Blind-Hammering*, a novel page table spraying technique, and a Spectre-based crash-resistant bit-flip verification. Our end-to-end proof-of-concept attack, the Half-Double Attack, gives an attacker arbitrary read and write access to the entire memory on fully up-to-date systems, as we showcase on Chromebooks with ECC- and TRR-protected LPDDR4x memory, in only 45 minutes average runtime.

Acknowledgments

We thank the anonymous reviewers, especially our shepherd, Shaanan Cohney, for their guidance, comments and suggestions. Part of the funding was provided by a generous gift from Amazon. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. When good protections go bad: Exploiting

anti-DoS measures to accelerate Rowhammer attacks. In *HOST*, 2017.

[2] ARM. *ARM Architecture Reference Manual ARMv8*. ARM, 2013.

[3] ARM. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism, 2018. URL: https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability.

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

[5] K.S. Bains, J.B. Halbert, C.P. Mozak, T.Z. Schoenborn, and Z. Greenfield. Row hammer refresh command, January 2014. US Patent App. 13/539,415. URL: https://google.com/patents/US20140006703.

[6] Alessandro Barenghi, Luca Breveglieri, Niccolò Izzo, and Gerardo Pelosi. Software-only reverse engineering of physical dram mappings for rowhammer attacks. In *International Verification and Security Workshop (IVSW)*, 2018.

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

[8] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P*, 2016.

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

[10] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. ePrint 2015/1034, 2015.

[11] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In *S&P*, 2019.

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

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

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

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

[16] Mohsen Ghasempour, Mikel Lujan, and Jim Garside. ARMOR: A Run-time Memory Hot-Row Detector, 2015. URL: http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer.

[17] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall Upper Saddle River, 2004.

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

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

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

[21] Christian Helm, Soramichi Akiyama, and Kenjiro Taura. Reliable reverse engineering of intel dram addressing using performance counters. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2020.

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

[23] Rei-Fu Huang, Hao-Yu Yang, Mango C.-T. Chao, and Shih-Chin Lin. Alternate hammering test for application-specific DRAMs and an industrial case study. In *Annual Design Automation Conference (DAC)*, 2012.

[24] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Stopping microarchitectural attacks before execution. ePrint 2016/1196, 2017.

[25] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and

Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security Symposium*, 2019.

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

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

[28] JEDEC Solid State Technology Association. Low Power Double Data Rate 4, 2017. URL: http://www.jedec.org/standards-documents/docs/jesd209-4b.

[29] Matthias Jung, Carl C Rheinländer, Christian Weis, and Norbert Wehn. Reverse engineering of drams: Row hammer with crosshair. In *International Symposium on Memory Systems*, 2016.

[30] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. Architectural support for mitigating row hammering in DRAM memories. *IEEE Computer Architecture Letters*, 14, 2015.

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

[32] Kirill A. Shutemov. Pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace, 2015. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce.

[33] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

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

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

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

[37] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[38] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. Raidr: Retention-aware intelligent dram refresh. *ACM SIGARCH Computer Architecture News*, 40(3):1–12, 2012.

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

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

[41] Matthias Payer. HexPADS: a platform to detect "stealth" attacks. In *ESSoS*, 2016.

[42] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*, 2016.

[43] Salman Qazi, Yoongu Kim, Nicolas Boichat, Eric Shiu, and Mattias Nissler. Introducing Half-Double: New hammering technique for DRAM Rowhammer bug, 2021. URL: https://security.googleblog.com/2021/05/introducing-half-double-new-hammering.html.

[44] Rui Qiao and Mark Seaborn. A New Approach for Rowhammer Attacks. In *International Symposium on Hardware Oriented Security and Trust*, 2016.

[45] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security Symposium*, 2016.

[46] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J Nair. Randomized row-swap: mitigating row hammer by breaking spatial correlation between aggressor and victim rows. In *ASPLOS*, pages 1056–1069, 2022.

[47] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

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

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

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

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

[52] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM. In *DIMVA*, 2018.

[53] Saru Vig, Siew-Kei Lam, Sarani Bhattacharya, and Debdeep Mukhopadhyay. Rapid detection of rowhammer attacks using dynamic skewed hash tree. In *Workshop on Hardware and Architectural Support for Security and Privacy*, 2018.

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

[55] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms. *arXiv:1912.11523*, 2019.

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

[57] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[58] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *RAID*, 2016.

[59] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, and Zhi Wang. TeleHammer: Cross-Privilege-Boundary Rowhammer through Implicit Accesses. *arXiv:1912.03076*, 2019.

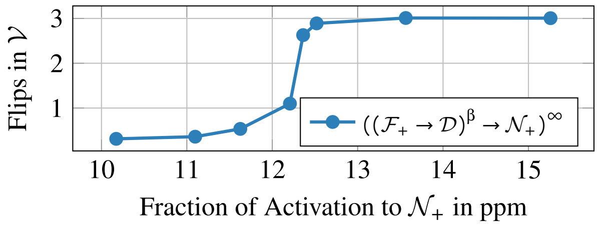


Figure 9: Number of observed bit flips in the *victim* over the fraction of accesses to the *near aggressor* ($\mathcal{N}_+$) in 1 million accesses for the single-sided case.

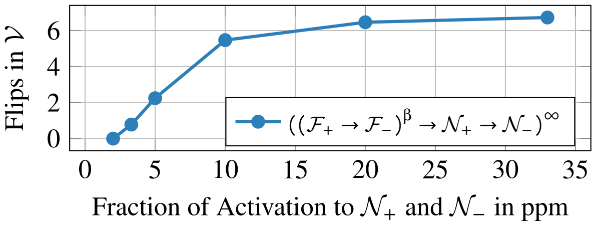


Figure 10: Number of observed bit flips in the *victim* over the fraction of accesses to the *near aggressors* ($\mathcal{N}_+, \mathcal{N}_-$) in 1 million accesses for the double-sided case.

A Summary of the Evaluated Memory Parts

Table 9 provides a comprehensive list of all DDR4, LPDDR4 and LPDDR4x parts we obtained and evaluated in this paper. First, we divide the parts into the DIMMs analyzed in Section 5.3 via the ZCU104 FPGA platform. For these DIMMs we have complete control over DRAM addressing and refresh intervals to evaluate the performance of *distance-1*, *distance-2*, and Half-Double-based hammering (cf. Tables 2, 4 and 5). Second, we evaluate the mobile devices and the PC parts in Section 5.1, where we first reverse engineer the DRAM addressing functions [42] and use the *Quad pattern* for hammering. Table 1 shows the resulting bit flips of the affected devices. We observed overall 7 parts that are affected by Half-Double. For the unaffected mobile and PC parts, we can only speculate whether the underlying memory is Half-Double resistant or whether unknown row scrambling prevented mounting the *Quad pattern*. Hence, we cannot conclude that the underlying memory is indeed unaffected from Half-Double.

B Alternative Representation for Bit Flips under Simulated TRR

In this section, we present a different representation for Figure 4 and Figure 5. Instead of using the β parameter, we plot the number of bit flips observed in the *victim* over the fraction of *near aggressor* in 1 million accesses. Figure 9 shows this data for the single-sided case (*i.e.*, the same data as Figure 4). Figure 10 shows this data for the double-sided case (*i.e.*, the same data as Figure 5),

Table 9: All evaluated memory parts, including their production date, the underlying memory structure, and information of the test system or operating system we evaluated them on. We indicate parts evidently affected by Half-Double.

	Name	Year-Week	CPU / SoC	RAM	Size	Manufacturer	Test System / Operating System	Half-Double
DIMMs	M_1	2019-48	-	DDR4	4 GB	Confidential	ZCU104 FPGA Platform	✗
	M_2	2020-32	-	DDR4	4 GB	Confidential	ZCU104 FPGA Platform	✓
	M_3	2020-42	-	DDR4	8 GB	Confidential	ZCU104 FPGA Platform	✓
Mobile Devices	Chromebook_1	2020-01	MT8183	LPDDR4x	4 GB	Unknown	Baseboard *Kukui* with Chrome OS Version 90.0.4430.218	✓
	Chromebook_2	2020-01	MT8183	LPDDR4x	4 GB	Unknown	Baseboard *Kukui* with Chrome OS Version 90.0.4430.218	✓
	Pixel 3	2018-40	SDM845	LPDDR4x	4 GB	Unknown	Android 11 LineageOS 18.1 with Kernel Version 4.9	✓
	HTC U11	2017-18	MSM8998	LPDDR4x	4 GB	Unknown	Android 9 with Kernel Version 4.4	✓
	OnePlus 5T	2017-47	SDM835	LPDDR4x	6 GB	Unknown	Android 11 LineageOS 18.1 with Kernel Version 4.4	✓
	Samsung S9 (SM-G960F/DS)	2018-10	Exynos 9810	LPDDR4x	4 GB	Unknown	Android 10 with Kernel Version 4.9	✗
	Samsung S7 (SM-G935F)	2016-10	Exynos 8890	LPDDR4	4 GB	Unknown	Android 8 with Kernel Version 3.18	✗
PC	Lenovo T490s	2019-13	Intel i5-8265U	DDR4	16 GB	Samsung	Ubuntu 20.04.3 LTS with Kernel Version 5.11	✗
	Minisforum TL50 MiniPC	2021-43	Intel i5-1135G7	LPDDR4	16 GB	SK Hynix	Ubuntu 20.04.3 LTS with Kernel Version 5.13	✗
	Minisforum X35G MiniPC	2020-43	Intel i3-1005G1	LPDDR4	16 GB	Micron	Ubuntu 20.04.1 LTS with Kernel Version 5.4	✗

C Contiguous Memory Solver

This section details the implementation of the solver and additional performance and correctness analysis based on the reconstructed DRAM addressing functions of the real devices.

Solver Implementation. The solver is implemented with the Z3 theorem prover [12]. To detect continuous memory regions, we first implement the structure of xor-based DRAM addressing functions as constraints. The solver solves for N xor masks we denote as M_i f or $0 \leq i < N$ and the base address of the contiguous physical range B. The general idea is to increment the base B for each of the given input samples, *i.e.*, the pages, as if the range would be contiguous, resulting in unsatisfiable constraints if not. Each of the input samples x_i comes from precisely one set $\mathbb{X}_i$, where x_i is the current sample index. First, we define the function $F_i(x)$ that computes the i-th set bit for the x-th page in the physical memory range:

$$F_i(x) = \bigoplus (M_i \wedge (B + x \cdot 0x1000)).$$

We denote $\bigoplus(x)$ as operation xor-ing all bits of x and $\wedge$ as the bitwise *and* operation. Second, we define a set index as a binary concatenation of each of the set's bits:

$$S(x) = F_0(x) \mid\mid \cdots \mid\mid F_{N-1}(x).$$

For each of the pages contained in one set we enforce that the set index is the same as of the first member of the set, *i.e.*, the first page of the set x_i^0:

$$\text{assert}(S(x_i) = S(x_i^0)) \forall x_i \in \mathbb{X}_i.$$

Finally, we restrict that all other page offset not contained in one set must have a different set index:

$$\text{assert}(S(y) \neq S(x_i^0)) \forall y \notin \mathbb{X}_i.$$

The underlying range can be generated via a xor-based DRAM addressing function if these constraints are satisfied. If unsatisfied, the memory region is not contiguous or the addressing functions are not xor-based.

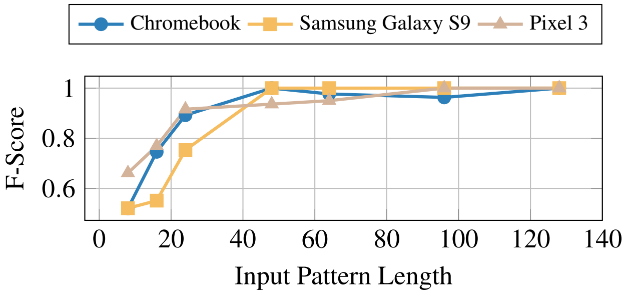


Figure 11: F-Score of our solver-based contiguency detection for different pattern lengths.

Evaluation. We verify the correctness of the solver by randomly concatenating contiguous ranges with up to 128 pages and converting these physical pages into bank access patterns with real reverse-engineered DRAM addressing functions. In the evaluation, we *slide* the solver over this generated pattern and vary the number of input samples the solver receives. Figure 11 shows the resulting F-score metric for different DRAM addressing functions and various pattern lengths. We observe that the F-score increases with increasing pattern length. This is as expected since the solver internally has more constraints to rely on. We see that with a pattern length of 128 pages, the solver achieves an F-score of > 0.99 for each pattern.