Assignment IV – Triggering your first Rowhammer bitflips

Due by Week 10 (November 18, 2024) at 23:59

Learning Objectives

The goal of this assignment is to understand a key weakness of modern DRAM devices known as Rowhammer [4]. Moreover, you will learn how to trigger Rowhammer *bitflips* (i.e., inverting bit values) in memory without accessing it. However, to achieve this, you need to know how the memory controller translates physical addresses into memory cells in DRAM. To learn this, you will use a reverse engineering method that exploits the timing differences caused by bank conflicts inside DRAM, known as DRAMA [7]. Further, you will acquire an understanding of how Rowhammer mitigations inside modern DDR4 chips operate and how to bypass them. To bypass them, you will implement a Rowhammer fuzzer, a simplified variant of *Blacksmith* [3] that crafts access patterns that Rowhammer mitigations fail to detect. Once these mitigations are bypassed, you can easily trigger bitflips on the DIMMs in the compute nodes of our cluster.

In assignment V, you will use the primitive you built in this assignment to craft an end-to-end exploit that results in root privileges based on prior work [5]. At the end of the DRAM part of this course, you have gone the whole way from reverse-engineering the DRAM subsystem, over creating an effective fuzzer, all the way up to building a practical, real-world exploit!

Overview

This assignment starts by exploring bank conflicts as a side channel. Equipped with this side channel, you reverse-engineer the *secret* DRAM address functions to precisely address DRAM rows of the same bank. Lastly, you build a simplified variant of the Blacksmith fuzzer and assess its performance across different DRAM modules in our cluster.

Week 1 – DRAMA

This week, you will reproduce parts of the DRAMA paper [7], which demonstrates how to reverse-engineer the secret DRAM addressing functions of Intel CPUs. Specifically, you will reverse-engineer the DRAM bank function bits (which select a bank) and row index bits (which select a row within that bank) in preparation for performing Rowhammer exercises in the following week.

Reverse engineering these functions can be divided into the following steps:

1. **Determining the bits involved in bank selection**: Identify which physical address bits are used to select the bank.
2. **Understanding bank functions**: Explore how these bits are combined to form bank selection functions.
3. **Identifying row selection bits**: Determine which bits are used for row selection within the bank.

We refer to the physical address bits involved in selecting the channel, DIMM, rank, bank, row, and column as *DRAM addressing bits*. The DRAM addressing bits used for rank and bank selection are XOR-ed together to form an index that selects a bank within a given rank. Each bit of this index is composed of an XOR function that takes **two physical address bits** as input [7, 8]. Hence, we refer to these XOR functions used to uniquely select a bank in the DRAM subsystem as bank functions.

Note: Our systems do not use multiple channels or DIMMs, so we disregard their respective addressing schemes, which otherwise also must be considered when selecting a given bank.

Task 1: Detecting Bank Conflicts

In this task, you show that you can reliably detect bank conflicts using timing measurements. A bank conflict occurs when a DRAM access forces a previously opened row (i.e., loaded into the row buffer) to be closed, thus resulting in a slowdown. By picking two random addresses and measuring their access times, you can infer if such a slowdown happened and the addresses are in the same bank. The slowdown is caused by a precharge command (`PRE`) that is sent by the memory controller in between the two accesses to activate a different row of a bank.

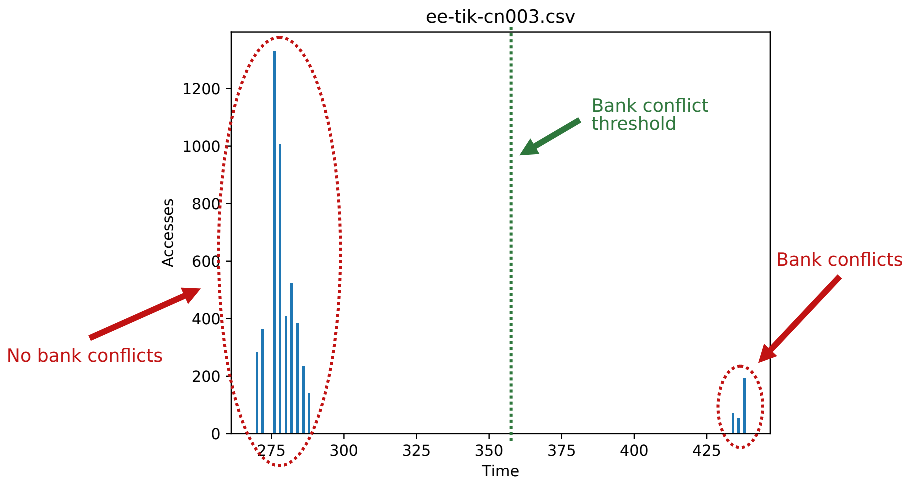

Figure 1: Access time histogram showing the distribution of times (in cycles) for a fixed base address. A possible bank-conflict threshold is indicated in green.

Contiguous physical memory. You start by allocating a *superpage*[1], a block of 1 GiB physically contiguous memory using `mmap(..)`, as shown in Listing 1. As the physically contiguous memory block will also be contiguous in virtual memory, this exposes and gives us control over the lower 30 bits of the physical address.

```
char* buffer = mmap(
  NULL,                  // address hint of the address of mapping (NULL=anywhere)
  (1<<30),               // allocation size (= 1GiB)
  PROT_READ|PROT_WRITE, // read/write permission
  // not shareable (private) | not file-backed (anonymous) | contig. phys. mem (hugetlb)
  MAP_PRIVATE|MAP_ANON|MAP_HUGETLB,
  -1, 0);

```

Listing 1: Allocation of a superpage. See `man 2 mmap` for more options. Huge pages are 1GB on our machines.

Observing conflicts. From the memory range, pick a random but fixed "base" address a^* and any other address a. Then, measure the total access time of the two addresses together. You will see that two accesses in different banks cause a lower access time than accesses to different rows of the same bank. This allows you to derive a "bank-conflict threshold" by choosing a value in the middle of the two peaks in your histogram as we show in Figure 1.

Timing measurements. There are some important aspects you must consider when measuring the access time:

[1] We refer to 1 GiB hugepages as *superpage* and to 2 MiB pages as *hugepages*.

- Accesses must be *uncached* so that the memory controller is actually sending an activate (`ACT`) followed by a read (`RD`) command to the DRAM device.
- Reading the timestamp counter, flushing caches, and accessing memory are subject to out-of-order execution. We can counteract by using `lfence`, `mfence`, or `sfence` instructions.
- Prevent the compiler from optimizing away your unused memory accesses.
- Repeat measurements many times to get stable results.

To better understand low-level instructions such as fence instructions and `clflushopt`, we recommend consulting the Intel SDM [2]. For your convenience, we provide you with an implementation of a timing measurement routine in the function `time_addresses` in the file `drama.c`.

Implementation. Given that we provide you with a primitive to measure the DRAM access time, your tasks are as follows:

1. Let the binary `./test__drama` output a CSV of the access times for different randomly picked addresses a_i while keeping a^* **fixed**.
2. From the CSV, create a histogram like in Figure 1.
3. Save the output CSV under `data/<hostname>__drama.csv` (**not** the shared folder `/data`).
4. Save the histogram as `figures/<hostname>__drama.pdf`.

Task 2: Detecting the Number of Banks

In this task, you repeat the same procedure as before, but for N different base addresses a_i^* with $i \in \{0, 1, 2, \dots, N-1\}$. For each a_i^*, create a set $\mathcal{B}_i$ of distinct addresses that all conflict with a_i^* but not with other base addresses.

More specifically, start by picking a random base address a_0^* and define its conflict set $\mathcal{B}_0 = \{a^*\}$. Then, time the picked addresses $a \neq a^*$ from your superpage against a^*.

- If an address a causes a bank conflict with a_i^*, let $\mathcal{B}_i := \mathcal{B}_i \cup \{a\}$.
- If an address a does *not* conflict with any existing $\mathcal{B}_x$, define a new conflict set $\mathcal{B}_{i+1} = \{a\}$.

Repeat this procedure until you cannot find new non-colliding addresses with any already-defined $\mathcal{B}$. The number of conflict sets found corresponds to the number of banks on your DRAM device. Note, there exists a case (think about it!) where two addresses map to the same bank but do not cause a conflict.

Implementation. Implement the function `build_conflict_sets` in `drama.c` that generates these conflict sets. Then, test your function by running `./test__drama -t <threshold>`, providing a threshold according to your histogram.

Task 3: Recovering bank selection mask

In this task, you will derive a bit mask of the address bits used in bank selection. Flipping a physical address bit used in bank selection will select a different bank. Consequently, the memory requests can be served from different row buffers. Hence, the addresses in their respective collision set will no longer cause bank conflicts with the new address. Based on this insight, you will derive a bit mask of the bits used in bank selection.

In the algorithm shown in Listing 2, we iterate over all addresses of a conflict set and systematically flip one bit after another using `flip_bit(..)` while measuring the total access time of the original (`addr`) and the flipped address (`new_addr`). If the access time is below the bank-conflict threshold, our flipped bit was likely part of the bank selection bits. Unfortunately, Listing 2 does not consider that access times below the threshold could also mean that the flipped bit is part of the column bits (i.e., `addr` and `new_addr` share the same bank and row). How can you distinguish between column bits and bank selection bits?

```
char** conflict_set;  // all addresses conflict pairwise with each other
uint64_t bit_mask;    // the result mask covering all DRAM addr. bits
for addr in conflict_set:
  for bit_pos in 0..30:
    new_addr = flip_bit(addr, bit_pos);  // flip bit at position bit_pos in addr
    if time(addr,new_addr) < threshold:  // measure access latency between addr and new_addr
      bit_mask |= (1<<bit_pos)  // => bit is significant for bank addressing
```

Listing 2: DRAM bank selection. Pseudocode to determine the DRAM bank selection mask.

Implementation. Running `./test__dram-fns -b` should output the bank selection mask as a hexadecimal with a preceding `0x`.

Task 4: Recovering bank selection functions

In this task, you will determine the XOR schemes (i.e., functions) of the bank selection bits used to select a bank. Two addresses map to the same bank if all functions return the same value (0 or 1) for both addresses. Therefore, we can think of the concatenation of these values as the *bank index*.

Example approach. Figure 2 demonstrates a method to derive the bank function bits. Exhaustively generate all possible 2-bit functions covering the 30-bit range of the superpage. Then, test each function candidate against your previously obtained conflict sets. Identify a bank function through its following characteristics:

- it produces the **same bit** (0 or 1) for all addresses for **any given** conflict set, and
- it produces **different values** for **different sets**. If it always produces the same value for all sets, it cannot be a DRAM address component.

You can speed up this brute-forcing process by considering only bit masks that cover the bits of the bank selection mask that you determined in the previous task.

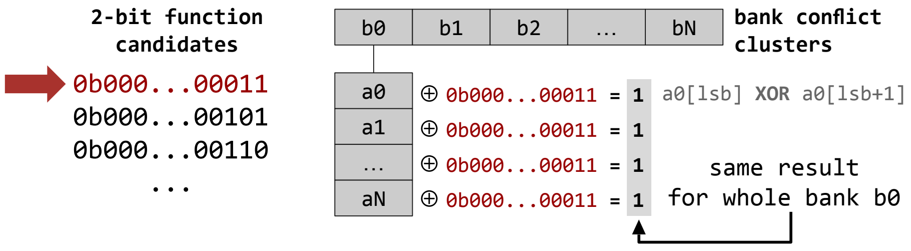

Figure 2: Bank function brute-forcing. Overview of the approach to recover the bank functions.

Implementation. Running `./test__dram-fns -f` should output the bank selector functions in hexadecimal representation.

Task 5: Recovering the DRAM row selection mask

The last task of week 2 is about recovering the row mask, meaning the bits that select the row. Unlike rank and bank bits, these are not subject to any XOR schemes. Unfortunate for us, the upper bank function bits may overlap with row bits. Luckily for us, however, the lower bank function bits never overlap with the row bits.

The general idea of the approach is given in Listing 3. We choose a random address `a0` and compute its DRAM bank addressing index. Next, we mutate a single bit of `a0`, resulting in `a0_x`. In case the single mutated bit resulted in a different bank addressing index, `ensure_bank(..)` restores its index by mutating a second bit in the address. Measuring the access time of `a0` and a0_x will tell whether we changed a row bit and both addresses are now in different rows of the same bank (slow), or we changed a column bit (fast).

Because we can assume that only the higher bit of the bank functions may ever overlap the row mask and that the row bits are laid out consecutively in the physical address, we can infer which bit out of the two belongs to it. Keep in mind that the column bits do not overlap, i.e., do not affect the bank or row selection.

```
char* a0;  // a randomly chosen address
int bank_idx = get_bank_idx(a0);  // the bank selection index of applying rank functions to a0
for bit_pos in 0..30:
    a0_x = flip_bit(a0, bit_pos);        // flip a single bit at position bit_pos in address a0
    a0_x = ensure_bank(a0_x, bank_idx);  // make sure a0_x still maps to bank_idx.
    time = time(a0, a0_x);
    if time > threshold:
      // => highest set bit of (a0 XOR a0_x) is part of the row mask
```

Listing 3: DRAM row mask reverse engineering. Pseudocode of the approach.

Implementation. Running `./test__dram-fns -m` should output the row mask. You may save your results in a struct defined as `dram_info_t`, declared in `dram_info.h`.

Week 2 – Fuzzing Rowhammer Patterns

In this week of the assignment, you will finally build a fuzzer to trigger your first bitflips! Later, you build upon your fuzzer by exploiting the discovered bitflips to gain access to a root shell.

Assuming that you managed to get the bank functions and row mask, you will next compose the hammering patterns. For this, you will implement a simplified variant of the frequency-based Blacksmith patterns. Blacksmith patterns take advantage of the assumptions the Target Row Refresh (TRR) mitigations put on hammering patterns. Then, you will implement the code that hammers a given Blacksmith pattern. Once you have hammered it, you will also need to detect bitflips in the hammered memory you evaluate your fuzzer's effectiveness.

Implementation. Implement the various tasks so that they can be easily tested by running `test__blacksmith` with different command line arguments.

Task 6: Generating simplified Blacksmith patterns

The original Blacksmith paper [3] defines a set of parameters from the frequency domain, namely *amplitude*, *phase*, and *frequency*. The parameters are randomized, under a given number of constraints, to compose non-uniform hammering patterns. Turn to the file `bs_pattern.c` to get an overview of the parameters that your Blacksmith pattern depends on. You may implement your pattern generation function here and export it from the respective `.h` file. You will generate the patterns as described in the original paper [3, see §IV and Appx. E], but use fewer fuzzing parameters than the original Blacksmith fuzzer. For example, we only work with aggressor *pairs* while the original Blacksmith considers aggressor *tuples* with two or more aggressors, and we always hammer aggressors that have a distance of 1 row from the victim row (like in a double-sided Rowhammer pattern).

A pattern consists of an even number of *aggressors* that all target the same, randomly chosen victim bank. The number is even because, for every tested row (i.e., victim row), we have two aggressors sandwiching it (i.e., double-sided Rowhammer). As shown in the paper, an aggressor pair appears in one or more *periods*, given its *frequency*. The number of single aggressors that one period fits is referred to as the *base period*. An aggressor pair is attributed with an *amplitude* that determines how many consecutive times it occurs within its assigned periods. An aggressor pair targets a victim row of a victim bank by hammering the immediate neighbor rows of the victim (i.e., “sandwiching” it). An aggressor pair reoccurs with the same phase and amplitude in all its periods. You may turn to the procedure described in the *Matching Frequencies* paragraph of Appendix F [3] to create aggressor pairs that do not overlap in the pattern.

Hint: Make sure to remain in the correct bank when picking conflicting addresses. This can be validated with your bank-conflict timing side-channel. You should not rely on the virtual addresses as the virtual and physical addresses are likely not aligned.

Table 1: Suggested parameter ranges for your simplified Blacksmith fuzzer. We denote the number of ACT commands (memory accesses) that you can do during the refresh interval (*tREFI*) (i.e., between two consecutive REF commands) as ACT_{tREFI}.

Parameter Name	Min	Max
Pattern length	ACT_{tREFI}	$8 \times ACT_{\text{tREFI}}$
Frequency	$1/\#periods$	1
Amplitude	1	16

As for the fuzzing parameter ranges, you can use the narrowed-down parameter ranges given in Table 1 as a starting point, which in our experiments reliably triggers bitflips on Samsung DIMMs. We recommend fully randomizing the parameters that are not mentioned in this table (e.g., phase). Of course, you are free to experiment with the parameter ranges and optimize them during your experiments.

Implementation. Implement the function `new_bs_pattern` in `bs_pattern.c` with your pattern generation logic. You can assume for this task a fixed ACT_{tREFI} of 180. You will later, in Task 9, implement the logic to determine the exact ACT_{tREFI} value, which is system-specific. To demonstrate that you have created patterns properly, calling `./test__blacksmith -d` should show 100 generated patterns.

Validation. To make sure that your pattern is correct, we recommend you to print the hammered row indices and add a line break between each period. In this way, it is easy to verify the pattern's correctness by looking at the aggressor pairs and their frequencies. In the example in Listing 4, for example, the double-sided aggressor pair `r01/r03` is the first access in the pattern and repeated every second period. You should also print another pattern repetition (`rep`) to ensure that frequencies are respected across pattern repetitions. For example, the pair `r62/r64` should be accessed again after accessing `r01/r03` in the first period of the second repetition and not earlier.

```
rep 1, period 1:   r01/r03  r01/r03  r01/r03  r07/r09  r11/r13  r55/r57  r14/r16  r14/r16
rep 1, period 2:   r62/r64  r62/r64  r62/r64  r22/r24  r11/r13  r19/r21  r42/r44  r42/r44
rep 1, period 3:   r01/r03  r01/r03  r01/r03  r07/r09  r11/r13  r32/r34  r14/r16  r14/r16
rep 1, period 4:   r62/r64  r62/r64  r62/r64  r22/r24  r11/r13  r36/r38  r47/r49  r47/r49

rep 2, period 1:   r01/r03  r01/r03  r01/r03  r07/r09  r11/r13  r55/r57  r14/r16  r14/r16
rep 2, period 2:   r62/r64  r62/r64  r62/r64  r22/r24  r11/r13  r19/r21  r42/r44  r42/r44
...
```

Listing 4: Example pattern. A simple Blacksmith pattern with `rX` referring to row `X`.

Task 7: Hammering patterns

In this task, you will implement the hammering function and integrate it into the functions provided by us in `blacksmith.c` to initialize your memory with a random data pattern and check it for bitflips. Note that the bitflip distribution and direction heavily depend on the data pattern you wrote into your aggressor and victims rows [6]. For now, we will use a simple random data pattern to initialize the memory.

Your hammering function should take a list of aggressors and, one after another, access them pair-wise (you may try more as an optimization). As aggressor pairs are sometimes accessed more than once (*amplitude* larger than one), you must flush them from caches before accessing them again — every aggressor access should hit DRAM.

After running a pattern for a sufficient number of repetitions (e.g., 3 M `ACTs` in total), iterate over the hammered memory area to check for any bitflips. Note that due to the on-die row remapping of some DRAM DIMMs, bit flips might appear far away from hammered aggressors. You do not need to care as long as you know which bit flips can be caused by a pattern.

Implementation. To demonstrate that you have implemented memory initialization, hammering, and bitflip checking correctly, running `./test__blacksmith` should show the results of 100 generated patterns. Further, `./test__blacksmith -e` should run your fuzzer for 15 minutes and report the results as described in the deliverables section.

Task 8: Detecting ACT_{tREFI} & evaluating your fuzzer's performance

An essential aspect that lets the Blacksmith fuzzer bypass TRR is its synchronization with the refresh (`REF`) command. Due to constraints in the DRAM specification, the device can only mitigate an attack during `REF` commands. To make your fuzzer more effective, it needs to become *`REF`-aware*.

Determine the ACT_{tREFI}, which is the number of `ACT` commands you can send between two subsequent `REF` commands. You can detect a `REF` command by measuring the access times of a fixed address pair and detecting a peak in access latency. Once you have derived ACT_{tREFI}, change your pattern generation procedure from Task 7 to select a pattern length using a random power of 2 that is a multiple of ACT_{tREFI}.

Although your pattern is now better synchronized with the periodic `REF` commands, a certain drift due to system noise is unavoidable. To remain in sync, you may want to resynchronize with a `REF` command each time you iterate over the pattern by awaiting a new `ACT`-latency peak in between pattern repetitions. Once (re-)synchronized, execute (i.e., *hammer*) your pattern. In theory, you may stop hammering once you are certain that a *refresh window* is over, meaning that all rows of the device have received at least one `REF`. In an entire refresh window, you can issue around 1.6 M `ACTs`. However, as not all rows are refreshed at the same time, we do not know when our victim row was refreshed the last time. Therefore, we recommend to hammer for at least 3 M `ACTs`.

In the evaluation of Rowhammer fuzzers [1, 3], we are generally interested in two metrics: (i) the number *effective patterns* (i.e., patterns causing bitflips) discovered over a fixed period of time, and (ii) the total number of bitflips when sweeping the most effective pattern (i.e., *best pattern*) over a physical memory range while incrementing each aggressors' row index. Your task is to evaluate your fuzzer on these two aspects. By sweeping the pattern over a physical memory range (e.g., 32 MiB), you transform each aggressor of the pattern by displacing it by one row while remaining in the same bank. Test each transformed pattern while accumulating the total number of bitflips.

Perform a mini-evaluation of your fuzzer by running it on three different nodes for 15 minutes each, where at least one node should use a different number of ranks than the others. Include your results in `report.txt`

Implementation. Running `./test__blacksmith -s` should run a 15 minutes evaluation followed by a sweep with the best pattern over the memory range.

Deliverables for Code Submission

- Code
  - A `tar.gz` bundle of your code that builds all artifacts with `make`
  - `./test__drama` works as `./test__drama -h` advertises
  - `./test__dram-fns` works as `./test__dram-fns -h` advertises
  - `./test__blacksmith` works as `./test__blacksmith -h` advertises
- Include results of your experiments from **three different nodes**, `<hostname>`, where at least one node has a **different rank count** than the other two. We expect this for all deliverables that include `hostname` in their output.
- `data/patterns.csv`: 100 generated patterns with the following CSV columns: `bank_idx, base_period, pattern`.
  - `bank_idx`: a logical bank index composed of the bits produced by the bank functions.
  - `pattern`: a space-separated list of memory offsets within your hammered memory range (i.e., subtract the base pointer from the aggressors).
- `figures/<hostname>__drama.pdf`: a histogram plot showing the clustering of access times formed by bank conflicts, like in Figure 1.
- `data/<hostname>__drama.csv`: the raw data used to generate your histogram.
- `report.txt`
  - Include `hostname, threshold, #banks, bank_functions, row_mask` (Tasks 1–5).
    - → `threshold`: the determined bank-conflict threshold.
    - → `bank_functions`: a space-separated list of hexadecimal masks for the different XOR schemes that together select a unique bank.
  - Fuzzer evaluation over 15 minutes, showing: `hostname, #tested patterns, #effective patterns, #bitflips`.
  - Best pattern evaluation, sweeping it over a 32 MiB physical memory range, showing: `hostname, #displacements, #bitflips, bank_idx, pattern`.
    - → `pattern`: the best pattern in the same format as in `data/<hostname>__patterns.csv`.
    - → `#displacements`: total number of variations of the best pattern tested.

Grading Scheme

- 4.0 → Correctly detecting the number of banks in the system.
- 5.0 → Brute-forcing of the DRAM bank selector functions and row mask on single-rank and dual-rank DIMMs.
- 6.0 → Generating effective hammering patterns with a simplified version of Blacksmith with `REF` synchronization and showing its effectiveness.

Presentation Submission

You must present this assignment in a 5-minute presentation on 10 December 2024. For this, you should prepare slides that detail the assignment and your steps taken to solve it. Upload it to Moodle at the "Assignment X | Presentation Submission" form. All team members should be able to present the slides and answer questions about the assignment, the lecture, and the papers we presented in the lecture.

References

[1] P. Frigo et al. “TRRespass: Exploiting the Many Sides of Target Row Refresh”. In: *IEEE S&P ’20*. 2020, pp. 747–762. DOI: 10.1109/SP40000.2020.00090. URL: https://download.vusec.net/papers/trrespass_sp20.pdf.

[2] *Intel® 64 and IA-32 Architectures Software Developer Manuals*. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (visited on 05/21/2023).

[3] Patrick Jattke et al. “BLACKSMITH: Scalable Rowhammering in the Frequency Domain”. In: *IEEE S&P, May ’22*. Nov. 2021. URL: https://comsec.ethz.ch/wp-content/files/blacksmith_sp22.pdf.

[4] Yoongu Kim et al. “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors”. In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 361–372.

[5] Seaborn Mark and Thomas Dullien. *Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges*. Mar. 2015. URL: https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[6] Lois Orosa et al. “A deeper look into rowhammer’s sensitivities: Experimental analysis of real dram chips and implications on future attacks and defenses”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 2021, pp. 1182–1197.

[7] Peter Pessl et al. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *USENIX Security ’16*. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl.

[8] Minghua Wang et al. “DRAMDig: A Knowledge-assisted Tool to Uncover DRAM Address Mapping”. In: *arXiv:2004.02354 [cs]* (July 2020). arXiv: 2004.02354 [cs]. URL: http://arxiv.org/abs/2004.02354 (visited on 12/18/2020).