Reverse-Engineering Bank Addressing Functions on AMD CPUs

Martin Heckel*†, Florian Adamsky*
*Institute of Information Systems (iisys)
Hof University of Applied Sciences
Hof, Germany
firstname.lastname@hof-university.de
†Institute of Applied Information Processing and Communications (IAIK)
Graz University of Technology
Graz, Austria

***Abstract*—The memory controller of the CPU uses bank addressing functions to determine physical locations within DRAM DIMMs. There are many fields of application for these addressing functions, particularly in security. For example, many Rowhammer proof-of-concepts use bank addressing functions to select addresses located on the same bank but in different rows to produce row conflicts. AMD provides these addressing functions for older CPU models. Hence, research on reverse-engineering addressing functions mainly targeted Intel CPUs since Intel did not publish these functions. However, AMD stopped to publish the DRAM addressing functions several years ago. AMD manufactures roughly a third of the sold CPUs in today's CPU market. We analyze the reverse-engineering tool for addressing functions published by Pessl et al. and find that it does not work with AMD CPUs, hindering reverse-engineering attempts and Rowhammer attacks on systems with AMD CPUs. In this paper, we introduce an approach to reverse-engineer the addressing functions of AMD CPUs, which facilitates future Rowhammer experiments on AMD CPUs.**

1. Introduction

Memory is a substantial component in every information processing device, and due to the drastic increase in the requirements for performance and capacity, memory chips have become very dense. The operating system (OS), with the help of the memory management unit (MMU), translates virtual to physical memory addresses. The memory controller then translates the physical memory addresses into a DRAM location, i. e., channels, DIMMs, ranks, and banks, using DRAM addressing functions.

Addressing functions have many fields of application. For example, researchers in IT security are interested in these functions to understand DRAM attacks better. One such attack is *Rowhammer*, in which an attacker can flip bits by rapidly accessing the content of nearby memory rows. An attacker can conduct better targeted Rowhammer attacks if the addressing functions are known. Pessl et al. [8] presented an approach to measure DRAM addressing functions entirely in software removing the requirement to perform physical probing. However, these addressing functions can also be used for performance optimization, such as application-aware memory channel partitioning [7] or variable page sizes [9] for more efficient row-buffer usage.

In contrast to AMD, Intel has not published the addressing functions of their CPUs. Therefore, the scientific community focused on reverse-engineering the DRAM addressing functions of Intel CPUs [8, 10, 5, 3]. AMD had a market share of 35.2 % (Intel had 62.8 %) in the market of x86 CPUs in Q3 2022 [4]. So, roughly one-third of x86 CPUs are manufactured by AMD. AMD published the DRAM banks' addressing functions in the BIOS and Kernel Developer's Guide (BKDG) up to microarchitecture `16h` [1]. Beginning with microarchitecture `17h` (released in 2017 [2]), no BKDG was released, so the addressing functions of the DRAM banks are not publicly available anymore.

Our paper makes the following contributions:

- We present an adapted approach based on the one introduced by Pessl et al. [8] to reverse-engineer the addressing functions on AMD CPUs.
- We evaluate our approach on four CPUs with two different DRAM settings.
- We provide the addressing functions that we have found with our approach.
- We publish our AMD DRAM addressing function reverse-engineering tool[1].

However, to the best of our knowledge, this paper is the first to focus on reverse engineering the addressing functions of AMD CPUs.

2. Backgrounds

This section briefly overviews how DRAM memory is organized and how the memory controller uses the addressing functions.

1. https://anonymous.4open.science/r/amdre-poc-A085/

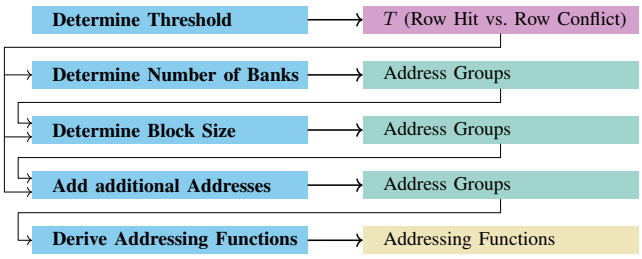

Figure 1. Overview of the reverse-engineering process. Actions (shown on the left side) lead to results (shown on the right side) and require some of the results of previous actions. Different results are depicted in different colors. Actions that require and return address groups add more addresses to the existing groups.

DRAM memory, in general, is organized in channels, DIMMs, ranks, and banks. The channel is a bus that connects the DIMMs with the CPU. In the case of a multi-channel memory architecture, DIMMs can be on different channels or share the same channel. A DIMM contains the actual DRAM chips, containing *banks*, which can be organized in groups called *ranks*. DRAM banks store data in cells consisting of capacitors and transistors organized in arrays of rows and columns. Additionally, a bank contains a *row buffer* that stores the entire row accessed last.

Since reading destroys the data saved in a row, the data has to be written back before the next row can be read. Writing the content of the row buffer back to the DRAM array before the next row is fetched (called *row conflict*) is significantly slower than if the correct row is already in the row buffer (called *row hit*).

The memory controller selects the bank by applying an XOR operation to the bits of the physical address masked with each of the addressing functions. The result of this operation is used as one addressing bit of the DRAM bank [8] for each DRAM addressing function. These functions split the available physical addresses equally to the physically available DRAM banks. If there are n_{banks} DRAM banks and n_{banks} is a power of two, there should be $\log_2(n_{\text{banks}})$ addressing functions. If n_{banks} is not a power of two, non-linear functions are used.

Since the memory controller is part of the CPU, different addressing functions exist on different CPUs. However, Pessl et al. [8] showed that the addressing functions also depend on the system's DIMM configuration. That means that the addressing functions can differ on systems with the same CPU.

3. Reverse-Engineering of DRAM Addressing Functions on AMD CPUs

This paper introduces an approach to reverse-engineer AMD CPUs' addressing functions that only depends on access to physical addresses and measurements of access timings. Figure 1 gives an overview of the procedure.

3.1. Determine the Threshold between Row Hit and Conflict

The first step is determining the threshold T between a row hit and a row conflict, which is required to measure whether addresses belong to the same bank. For that, we need to measure the access time of memory addresses. If the access is slow, we assume a row conflict was triggered. Thus, the memory addresses are in the same bank but in different rows. If the access is fast, we assume a row hit was triggered, and the memory addresses are located in different banks or the same row in the same bank. We need to determine a threshold T to distinguish between the two cases.

Our proof-of-concept allocates a 2 MiB transparent hugepage and measures the access time of the first and the second page (e. g., the first and the 4096$^{\text{th}}$ byte) n (by default, 200) times to rule out noise. We use the `rdtscp` instruction to measure the access time. To avoid that the CPU caches the results, our proof-of-concept uses the `clflush` instruction between measurements to remove the data from the cache. Then, the proof-of-concept computes the average by dividing the measured access time by n. Next, we repeat the measurement for all other pages within the transparent hugepage, group all measurements by access time and search for a gap in the access times, which should be between row hits and conflicts. If the gap is found, the middle of the gap is used as the threshold value.

This measurement is repeated for multiple transparent hugepages to increase the accuracy. The number of transparent hugepages can be configured and is set to 21 by default. The median of all measurements is used as the threshold.

3.2. Measuring the Number of Banks

In this step, we measure the number of banks by grouping addresses with row conflicts. Addresses that end up in one group belong to the same bank. Initially, there are no groups. We take every 4096$^{\text{th}}$ address, i. e. one per 4 KiB page, and compare it with x randomly selected addresses of each group. If the address does not fit in a group or there are no groups, this address goes into a new group. If a group has fewer than x addresses, the address is compared to all addresses in this group. The comparison works as follows: When the access time (measured the same way as described in Section 3.1) is higher than the threshold determined before, there is a row conflict, and the address is at the same bank as the other addresses in the group; otherwise, there is a row hit. When multiple groups have access times longer than the threshold, which might occur due to measurement errors, the group with the longest access time is used. We set $x = 9$ in our experiments which worked best for us, but this parameter can be changed via a command line option.

At least two 4 KiB pages fit into one row on x86_64 systems, assuming a size of 8 KiB for a row. As a 4 KiB page can span multiple banks and rows, it is possible to fit parts of more than two 4 KiB pages into one row on one

bank [8]. Thus, comparing two addresses that are located in the same row will produce a row hit even if they belong to the same bank. To avoid this problem, we regroup the initial groups. We iterate through the groups, removing each one after another, and try to add the addresses that belong to the removed group to the remaining groups. If one of the remaining groups matches (e. g., a row conflict occurs), our proof-of-concept adds the address to that group. Otherwise, our proof-of-concept adds the address to a new group. This regrouping step is repeated until the number of groups is plausible, meaning it is a power of two. Afterwards, the number of groups is fixed, e. g., no new groups are created by the following steps.

3.3. Detecting the Block Size

The next step is to detect how many consecutive memory addresses are located in the same bank. This *block size* helps us determining which bits we can ignore when inferring the addressing functions. Additionally, it decreases the number of addresses we need to group, speeding up the process.

First, we guess the block size, starting with 4096 B. Then, we count the number of consecutive memory addresses for the guessed block size. When we start with 4096 B, we need to find memory addresses at offsets of `0x1000` ($\widehat{=}$ 4096). If we do not find consecutive memory addresses, we divide the block size by two and try again. The next block size would be 2048 B. We already found the number of banks, but since we halved the block size, we have more addresses, which we need to sort again into the groups. That is the case since only every 4096$^{\text{th}}$ address was grouped before. Now, every 2048$^{\text{th}}$ address has to be grouped. In contrast to the grouping described before, no new groups are created at this stage. When an address does not fit into any existing group, it is discarded.

If the number of consecutive memory addresses is one, the block size is smaller than or equal to the currently guessed block size. If it is two, the system's block size is twice the currently guessed block size, so the block size is known in this case. Therefore, we must halve the block size until the number of consecutive memory addresses is two. We will repeat this process until we find the block size or reach 64 B, which is the size of one cache line.

3.4. Add Additional Addresses to the Groups

When we found the block size, we can allocate more transparent hugepages and add more memory addresses to the groups. This procedure is similar to the one described in Section 3.2, except that we will not create new groups and use the measured block size. If an address cannot be added to any group, e. g., due to measurement errors or noise in the system, we discard the address. This optional step was not necessary for the tested systems in Section 4. However, there might be systems where the addressing function uses a bit before the 21$^{\text{st}}$ bit. In that case, one transparent hugepage is insufficient to derive the full addressing functions. In that case, additional transparent hugepages are required.

3.5. Deriving the Addressing Functions

The final step is to derive the linear addressing functions. We implemented this step with multi-threading to reduce the time to derive the addressing functions. Each thread computes a *mask candidate* similar to Drama [8]. Initially, we try one bit (e. g., `0x01`) followed by the next larger combinations with the same number of bits. If this is no longer possible, two bits (e. g., `0x11`) followed by the next larger combinations with the same number of bits are tried. This is repeated until max_mask_bits bits are set. Thus, all possible masks with $1 \leq n \leq$ max_mask_bits are computed in total. We set the max_mask_bits = 7 by default to limit the number of candidates and, thereby, the time required for mask detection. Each mask is then checked to see whether it is valid based on the following criteria:

- The function yields the same results for all addresses within the same group for all groups.
- The function splits the groups equally, e. g.,, it provides the results 0 and 1 for 50 % of the the groups.
- The mask is not equivalent to a simpler mask. This is analyzed by toggling the ones in the mask to zeroes in all possible combinations. If one of the modified masks yields the same results, the current mask is equivalent to a mask with fewer bits and, therefore, does not have to be added.

After the threads are done, the detected masks are unified since the algorithm described above does also detect linear combinations of other masks. Finally, it is verified that the number of detected masks equals $\log_2 n_{banks}$. If that is the case, the detected masks are returned.

3.6. Runtime Estimation

The runtime of the approach described before depends on multiple parameters. We divide the reverse engineering process into three stages: In the first initialization stage (S1), our proof-of-concept measures the threshold between row hit and conflict. In the grouping stage (S2), our proof-of-concept groups memory blocks based on their timings when accessed alternatingly. In the last stage (S3), our proof-of-concept derives the addressing functions with multiple threads.

First, we look at the number of banks n_b depending on the number of addresses n_a. In stage S2, when $n_a \leq n_b \cdot x$, the runtime does not depend on n_b. When $n_a \geq n_b \cdot x$, the runtime has a linear dependency to n_b. Each address is compared to at most x (9 by default) randomly selected other addresses from the same bank. Therefore, if there are more than x addresses in the bank, the number of comparisons for each new address is limited to x per bank.

Next, we look at the number of transparent hugepages n_t. For both stages, S1 and S2, the runtime depends on n_t linearly. The block size n_s influence stages S1 and S2 inverse proportionally. If our proof-of-concept is run with multiple threads n_c, the last stage S3 depends inverse proportional on n_c.

Regarding the number of bits set at most in the mask candidates n_m: For S3, we need to sum up the binomial coefficient since all combinations with $1 \leq x \leq n_m$ bits are selected.

See Section 4 for runtime measurements during the performed experiments.

4. Experimental Evaluation

We reverse-engineered the addressing functions on several systems with different CPUs and multiple DIMM configurations. In addition, we performed these measurements for different proof-of-concepts. In order to ensure that our approach works on Intel CPUs as well, we run our proof-of-concept on several Intel CPUs in addition to the AMD CPUs. First, we analyze the existing Drama proof-of-concept [8]. Next, we evaluate the proof-of-concept introduced in this paper. Table 3 provides an overview of the systems used for experimental evaluation.

As the approach shown by Pessl et al. [8] is similar to our approach, it is used for evaluation. Their proof-of-concept returns all reverse-engineered addressing functions (might be more than addressing functions on the system) and their probability. Afterwards, the user has to guess which of the returned addressing functions are the correct ones. Therefore, the $\log_2(n_B)$ most probable addressing functions are listed for systems with n_B banks in the evaluation. Additionally, we provide the total number of identified addressing functions. The proof-of-concept was executed on several systems with AMD and Intel CPUs. Table 2 shows the results of our measurements.

The number of DIMMs in Table 2 is depicted as n_D and the number of DRAM banks is depicted as n_B. Not all identified addressing functions are listed due to space limits, but only the most probable ones. We provide the total number of addressing functions if more addressing functions were identified.

The proof-of-concept introduced in this paper was executed on the same systems with AMD and Intel CPUs. The configuration of the systems significantly affects the runtime (see Section 3.6 for detailed estimation of runtime impacts). The results of the measurements are shown in Table 1.

It should be noted that both the proof-of-concept by Pessl et al. [8] and our approach identified addressing functions on systems with AMD CPUs. The addressing functions reverse-engineered on the Intel systems are the same for both tools with the exception of the i7-4800MQ where Drama did not find all required sets. However, the ones on AMD systems are different. The addressing functions derived by Drama were not stable on the AMD systems, e. g., different when the tool was executed multiple times. Our tool took longer to execute but was more stable than Drama. Therefore, we conclude that Drama does not run stable on systems with AMD CPUs. However, we depict the addressing functions identified by Drama for completeness.

5. Limitations

Like prior approaches [8], our reverse-engineering approach introduced in this paper only works on systems with linear DRAM addressing functions, i. e., the total number of DRAM banks is a power of two. Moreover, since it is exclusively based on timings, it is impossible to get further semantic information related to the detected addressing functions, e. g., determining whether a detected function addresses a channel, rank, DIMM, bank group, or bank. Our proof-of-concept needs root privileges to get the physical addresses mapped to virtual addresses using `/proc/self/pagemap`. Since the addressing functions can only be applied to physical addresses (or parts of virtual addresses that are directly mapped to physical addresses), the restriction with the root privileges should not be a problem in typical application scenarios.

6. Related Work

In 2014, Kim et al. [6] analyzed the effect that bits in DRAM flipped (e. g., a 1 changed to a 0 or vice versa) when spatially nearby memory locations were repeatedly accessed. In addition, the authors reverse-engineered addressing functions on the Intel CPUs they used in a way. They found that the two selected addresses should have a distance of 8 MiB to be on the same bank.

Later, Pessl et al. [8] introduced a more generic approach to reverse-engineer addressing functions without requiring physical access, by using timing instead. The primary approach is to measure the time a specific number of alternating accesses to two addresses takes when the accesses are not cached. If that access time is considered *high*, there is a row conflict, and both addresses are on the same bank in different rows. Otherwise, there is a row hit, and the addresses are either on different banks or on the same bank in the same row. Based on the conflicts, addresses are grouped, mapping to banks. Afterwards, the addressing functions can be derived using the physical addresses in the groups. The authors evaluated their approach on several systems with Intel CPUs by physically probing the accessed components and comparing the results of the reverse-engineered addressing functions to the probed values. They showed that different addressing functions are used depending on the configuration of DIMMs in the systems. In contrast to their tool, our tool measures the number of DRAM banks, so it does not have to be specified manually and can be used as additional sanity check. Additionally, we do not select addresses randomly out of an address pool but group entire 2 MiB transparent hugepages. We modified the measurements of the timings to get more precise results and reduce the impact of noise. However, this approach slows down the measurements, so our tool brings more stable results at the cost of a higher runtime.

In 2018, Barenghi et al. [3] showed that the addressing functions used by the memory controller depend on the configuration of DIMMs in the system. Their approach is similar to the one showed by Pessl et al. [8].

TABLE 1. RESULTS OF THE REVERSE-ENGINEERING TOOL INTRODUCED IN THIS PAPER.

CPU	n_D	n_B	Blocksize	Addressing Functions (Bit Mask)	Time
AMD Ryzen 9 5950X	2	64	64 B	0x003fc0, 0x004100, 0x008000, 0x070000, 0x090000, 0x120000	2673 s
	1	32	64 B	0x004000, 0x048000, 0x090000, 0x103fc0, 0x138000	8048 s
AMD Ryzen 9 3900X	2	64	64 B	0x003fc0, 0x004100, 0x008000, 0x070000, 0x090000, 0x120000	2675 s
	1	32	64 B	0x004000, 0x048000, 0x090000, 0x103fc0, 0x138000	16 201 s
Intel Core i9-10900K	2	32	64 B	0x004080, 0x01b300, 0x048000, 0x090000, 0x120000	742 s
	1	16	8192 B	0x002000, 0x024000, 0x048000, 0x090000	18 s
Intel Core i7-4800MQ	2	32	64 B	0x00f380, 0x020000, 0x044000, 0x087380, 0x130000	3644 s
	1	16	8192 B	0x022000, 0x044000, 0x088000, 0x110000	58 s

TABLE 2. RESULTS OF THE REVERSE-ENGINEERING TOOL INTRODUCED BY PESSL ET AL. [8].

CPU	n_D	n_B	Blocksize	Addressing Functions (Bit Mask)	Time
AMD Ryzen 9 5950X	2	64	64 B	(none)	131 s
	1	32	64 B	0x10290000, 0x80, 0x20010200, 0xc100200, 0x22204000, ... (14)	555 s
AMD Ryzen 9 3900X	2	64	64 B	0x40, 0x100, 0x400, 0x800, 0x800000, 0x21000, ... (12)	562 s
	1	32	64 B	0x8000, 0x4000000, 0x41000, 0x401000, 0x8001000, ... (20)	42 s
Intel Core i9-10900K	2	32	64 B	0x004080, 0x01b300, 0x048000, 0x090000, 0x120000, ... (6)	371 s
	1	16	8192 B	0x002000, 0x024000, 0x048000, 0x090000, ... (5)	135 s
Intel Core i7-4800MQ	2	32	64 B	0x00f380, 0x00040, 0x044000, 0x088000, 0x110000	111 s
	1	16	8192 B	0x022000, 0x044000, 0x088000, 0x110000, ... (5)	96 s

TABLE 3. SYSTEMS USED FOR EXPERIMENTAL EVALUATION

CPU	DRAM type	Number of DIMMs
AMD Ryzen 9 5950X	DDR4	1, 2
AMD Ryzen 9 3900X	DDR4	1, 2
Intel Core i9-10900K	DDR4	1, 2
Intel Core i7-4800MQ	DDR3	1, 2

Two years later, Wang et al. [10] combined the approach introduced by Pessl et al. [8] with additional knowledge about the DIMM configuration and addressing functions in general. Thereby, they were able to reverse-engineer DRAM bank addresses significantly faster. Additionally, their approach gave information about the parts of addresses used to address rows and columns within the same bank. The authors claimed to open-source their tool, which seems not have happened until now. Upon a request we send to the authors we did not get any response.

In the same year, Helm et al. [5] introduced an approach that uses CPU performance counters instead of timings to group addresses by DRAM bank. Due to the usage of the counters, the results are more precise than the ones received via timing. Since the timers the authors are using are not supported on the AMD systems we used for our experiments, we can not evaluate their approach on AMD CPUs.

However, all of these publications focus on Intel CPUs, and none of them verified their approach on AMD. Our work closes this gap and extends over prior techniques.

7. Conclusion and Future Work

In this paper, we introduced a new approach extending the one by Pessl et al. [8]. We evaluated the proof-of-concept from prior work [8] for reverse-engineering the addressing functions on systems with AMD and Intel CPUs. Additionally, we evaluated our new approach on these systems and showed that prior approaches did not yield good results on AMD-based systems.

In the future, the scientific community should evaluate techniques to reverse-engineer DRAM addressing functions further on AMD CPUs to better understand both, what DRAM addressing functions on AMD in general look like and which patterns they follow, and which corner cases on different microarchitectures limit the applicability of reverse-engineering techniques.

Acknowledgment

We thank the Deutsche Forschungsgemeinschaft (DFG) for their support and Daniel Gruss for his review and fruitful discussions, which influenced the work of this paper. This work was funded by the DFG under grant number 503876675.

References

[1] *AMD Developer Guides, Manuals and ISA Documents*. 2020. URL: https://developer.amd.com/resources/developer-guides-manuals/.

[2] *AMD's Zen CPU is now called Ryzen, and it might actually challenge Intel*. arstechnica, 2016. URL: https://arstechnica.com/gadgets/2016/12/amd-zen-performance-details-release-date/.

[3] Alessandro Barenghi et al. "Software-only Reverse Engineering of Physical DRAM Mappings for Rowhammer Attacks". In: *IVSW*. 2018, pp. 19–24. DOI: 10.1109/IVSW.2018.8494868.

[4] *Distribution of Intel and AMD x86 computer central processing units (CPUs) worldwide from 2012 to 2022, by quarter*. statista, 2023. URL: https://www.statista.com/statistics/735904/worldwide-x86-intel-amd-market-share/.

[5] Christian Helm, Soramichi Akiyama, and Kenjiro Taura. "Reliable Reverse Engineering of Intel DRAM Addressing Using Performance Counters". In: *MASCOTS*. 2020. URL: https://ieeexplore.ieee.org/document/9285962/.

[6] Yoongu Kim et al. "Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors". In: *SIGARCH* (2014). URL: https://doi.org/10.1145/2678373.2665726.

[7] Sai Prashanth Muralidhara et al. "Reducing memory interference in multicore systems via application-aware memory channel partitioning". In: *MICRO*. 2011. URL: https://dl.acm.org/doi/10.1145/2155620.2155664/.

[8] Peter Pessl et al. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: *USENIX Security Symposium*. 2016.

[9] Kshitij Sudan et al. "Micro-pages: increasing DRAM efficiency with locality-aware data placement". In: *SIGARCH*. 2010. URL: https://dl.acm.org/doi/10.1145/1735970.1736045/.

[10] Minghua Wang et al. "DRAMDig: A Knowledge-assisted Tool to Uncover DRAM Address Mapping". In: *DAC*. 2020. URL: https://ieeexplore.ieee.org/document/9218599/.