UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Attack on DDR4 Memory Scrambling

Angriff auf DDR4 Memory Scrambling

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

Vorgelegt von
Gilian Henke

Ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

Mit Unterstützung von
Jan Wichelmann

Lübeck, den 30. October 2020

Abstract

As part of this thesis we try to induce errors in the data transmission to the memory, which could lead to possible security vulnerabilities. We induce errors by overshooting the bus by transmitting specific data patterns. However, this approach is complicated by the usage of Memory Scrambling inside the MCU, which tries to randomize the sent data by using LFSRs. To remove the effects of Memory Scrambling, we analyse how the Scramble Code is generated. Because we need to construct reliable test sets efficiently, we use hugepages to obtain pairs of physical addresses and corresponding scramble code. From these sets we create potential equations for the LFSRs, which we check for satisfiability. As a result we are able to improve existing tests to identify scramble code.
Additionally, we send specific patterns in a normal user interaction via the bus to the memory to induce errors. This process is optimized in regard to the runtime. For that purpose we use non-temporal operations. We verify the transmission of the data by measuring the transferred bits with an oscilloscope. As a result, we show that it is not feasible to induce errors with these operations. Finally, we give a perspective on possible further studies on this topic.

Kurzfassung

Innerhalb dieser Arbeit versuchen wir, in der Datenübertragung zum Arbeitsspeicher Fehler zu erzeugen, welche potentiell Sicherheitslücken sein können. Diese Fehler wollen wir dadurch erzeugen, indem wir den Bus durch Übertragung spezifischer Daten zum Überschwingen bringen. Dieser Ansatz wird durch den Einsatz von Memory Scrambling innerhalb der MCU erschwert, wodurch die versendeten Daten durch den Einsatz von LFSRs randomisiert werden. Um den Effekt des Memory Scramblings zu verhindern, analysieren wir auf welche Art und Weise der Scramble Code erzeugt wird. Dafür generieren wir unter Verwendung von Hugepages effizient Paare von physischen Adressen und zugehörigem Scramble Code. Aus diesen erzeugen wir potentielle Gleichungen für die zugrundelegenden LFSRs, welche wir dann auf Erfüllbarkeit testen. Hierbei haben wir bestehende Tests verbessert.
Zusätzlich senden wir spezifische Muster als normale Nutzerinteraktionen über den Bus zum Speicher, um dadurch Fehler zu erzeugen. Diesen Prozess optimieren wir bezüglich dessen Laufzeit. Dafür verwenden wir non-temporale Operationen. Wir verifizieren die

tatsächlich überragenden Bit-Muster indem wir jene mit einem Oszilloskop messen. Hierbei zeigen wir, dass es nicht praktikabel ist, mit non-temporalen Operationen Fehler zu erzeugen. Zum Abschluss der Arbeit geben wir einen abschließenden Ausblick über mögliche fortführende Themen.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

_____

Lübeck, 30. Oktober 2020

Contents

1 Introduction

It is important to keep private information and especially passwords secure. This naturally extends to data digitally stored on a computer system. Therefore, the data has to be protected in some way or another. On persistent memory it was always obvious that the data has to be encrypted. But in the beginning, the temporary data didn't get the same attention. Before DDR3 the internal memory was not protected in any way and the same holds true for the cache. Some of the first attacks were Cache Attacks relying on shared memory like Flush + Reload [YF14] or Flush + Flush [GMWM16], and attacks which don't need that like Prime + Probe [Per05] . These Cache Attacks also can be executed across virtual machines. [IIES14]
But not only the cache is vulnerable to attacks, but also the DRAM. One of the most prominent attacks is the Rowhammer Attack [KDK+14], where an attacker can attain bit-flips in chosen memory locations, which enables root access. Another prime example are Cold Boot Attacks [HSH+09] where the attacker can read the complete DRAM.
With further advances of the memory modules the aforementioned attacks are no longer easily possible. The introduction of Memory Scrambling was only intended to improve the data integrity, but randomizing data inside the memory also makes Cold Boot Attacks more difficult. [YADA17] If Memory Scrambling is used in combination with target row refreshes bit-flips become extremely unlikely.
We have a good understanding of how the Memory Scrambling is conducted on DDR3 memory[MMK11, BGF16]. But, the same cannot be said about DDR4. Therefore, as a part of the thesis we want to especially investigate the workings of the Memory Scrambler and its underlying mechanisms on DDR4 memory.
On DDR3-DRAM the functionality of the Memory Scrambler is mostly analysed [BGF16]. But the exact same techniques do not seem to work on DDR4-DRAM. Therefore, we want to built upon their results to understand how Memory Scrambling works on DDR4 and what differentiates it from the one used in DDR3. Furthermore it has been shown possible to induce errors inside the RAM by simply transferring data in very specific patterns [MMK11]. Normally that wouldn't be a problem because the Memory Scrambler obfuscates the sent data enough by XORing it with pseudorandom patterns. But with sufficient knowledge about the Memory Scrambling mechanism for DDR4, it should be possible to sent specific patterns which if scrambled will lead to errors.

We analyse the communication between the MCU and the DDR4-DRAM in greater detail. Thereby, we want to to find a possibility to send arbitrary data as a normal user, while still being able to control the largest possible part of the data transmission, i.e. as many of the sent bits as possible. Furthermore, we want to find error inducing patterns and include them into the data transmissions. The process is rendered more complicated by the Memory Scrambling. Therefore we need to find a way around the obfuscation mechanism. To achieve our goal, we will perform three major steps: First, we generate usable data from our DDR4-RAM, where we can infer the workings of the underlying system. Given the data, we then can perform the second step, where we analyse how the Memory Scrambling mechanism works for our given RAM and processor. Then, we verify and generalize this to other systems.
The last step is finding out which data transmissions have the possibility to induce errors in our RAM. We analyse how likely errors can occur and if they are reproducible.
The combination of the last two steps gives us the possibility to create data transmissions, which when scrambled can induce errors in the DDR4-RAM because our modification of the data neutralizes the effect of Memory Scrambling. Thereby, we can induce errors, which otherwise would have been corrected by the Memory Scrambling.
Some motherboards posses the feature to disable the Memory Scrambling in the BIOS. We use this feature to make the steps easier and independent from each other.

In chapter 2 we will first discuss the previous relevant work for this thesis and on whose results and methods we build upon. Afterwards, we will examine the necessary background knowledge for the DRAM and the communication protocol between the DRAM and the MCU in greater detail. In the following chapter 3 we will analyse the workings of the Memory Scrambler and how its usage will impact our attack. At last, in chapter 4 we will try to inject errors via the bus onto the DDR4 memory. We will show the steps we took to get the closest possible utilization of the memory bus and how it enables us to induce errors.

2 Preliminaries

2.1 Related Work

Before presenting the results of our work, we briefly describe the most important works and attacks, which are connected to our goal and on whose results we depend to achieve it.

2.1.1 Rowhammer Bug

The further refinement of DRAM led to smaller and denser packed cells, which can be operated with a lower voltage. Thus even smaller changes of the voltage are sufficient to change the value of a single cell. With random access patterns that poses no further problems, but certain patterns can cause problems. This is called the Rowhammer Bug [KDK+14], which can be exploited as shown by Seaborn [Sea20a] to fault single bits. The flip of a single bit can then be used for kernel privilege escalation attacks.
The main idea is as follows: Because voltage in a single cell is hold low, there is only a small difference between the possible values. Enough rapid changes in a neighbouring cell can influence the cell just enough to flip its value. The process of repeatedly changing the value in the neighbouring cell is called *hammering*.
There are generally three methods of hammering: One-sided, Double-sided and One-location. An attacker can use One-sided and Double-sided Rowhammer to hammer one or two sides of the target addresses to try to provoke a bit-flip. One-location Rowhammer however needs to attack only one single address [GLS+18]. This method however is slower to produce bit-flips, but is used in stealthy applications to bypass Rowhammer defences.
With later DRAM (DDR3) the Rowhammer bug has not been fundamentally solved, but some methods were introduced to prevent it from occurring. One of these is to refresh all rows after some time, which has the disadvantage of negatively affecting the performance. A more sophisticated approach is the active counting of frequently accessed rows and then refreshing their neighbours. This process is called TRR (Target Row Refresh). There are however disadvantages to this method. On the one hand the power consumption increases and on the other hand there is a need for a more complex framework to manage counting the accessed rows. Solutions like a simple ECC (Error Correcting Code) may prevent single bit-flips, but they can't prevent multiple flips, which can often appear

in one single row [KDK+14].
With the introduction of DDR4 the Rowhammer bug seemed to be resolved at first because the previous attacks no longer produced bit-flips. Nonetheless, there are still reports of bit-flips happening. All successful attacks we know of have been executed on chips produced by Micron [PGM+16, Ma16]. Therefore, it was unsure if DDR4 is still susceptible to Rowhammer attacks or if the tested chips are faulty.

In March 2020 a new kind of Rowhammer attack against DRAM called TRRespass has been found [FVH+20]. TRRespass attacks Target Row Refresh by overloading it with information of a Many-sided Rowhammer attack. TRR needs to keep track of how often a row has been accessed. If a threshold is reached, it refreshes the neighbouring rows. All previous described Rowhammer attacks only access two different rows at most, which can be easily handled by the TRR. With TRRespass many more rows, in practice about 10, are accessed. With this amount of attacked rows the TRR can't keep track of all accessed rows any more and bit-flips can happen once again because only so many rows can be monitored at the same time. They were able to reproduce their attack on about 30% of the tested DIMMs.

2.1.2 Off-Chip Side-Channel Attack

In 2016 Gruss and Pessl [PGM+16] showed that DDR3 and even DDR4 memory are still vulnerable via a row buffer attack. They observed the difference of latency between two different read or write requests to DRAM memory. Depending on whether consecutive memory accesses belong to the same bank or to a different one, they have measurable differences in their access times.
Furthermore, they reverse engineered the mapping of the physical addresses to their corresponding ranks, banks and channels. For their method they used two different methods. The first one is a physical probing of the relevant pins and then calculating the underlying functions for the mapping. The other approach depends on the row buffer. If two successive accesses to the memory happen to be in the same bank, rank and channel, but in a different row, then the time to retrieve the information is significantly longer because the current row has to be precharged to prevent loss of information before another row can be activated. This leads to a non-negligible difference in the access times. Therefore, we can group the physical addresses together, if the access time between them is over some threshold. Then, the grouped addresses must be in the same bank, rank and channel. Thereby they were able to calculate the underlying functions. With the second method physical access is no longer necessary.
Overall they achieved three different results. First, they used the timing differences to

build a covert channel. Second, they used the row buffer as a side channel to monitor access to specific addresses. Third, they proposed the possibility to speed up Rowhammer attacks because it is not necessary to hammer if the two addresses can't be adjacent. And the addresses can only be adjacent, if they are in the same rank, bank and channel. Another Off-Chip Attack based on this was used by Lee [LJF+19] to leverage the memory bus to gain access to sensitive data.

2.1.3 Cold Boot Attacks

Originally it was assumed that data in DRAM is lost on reboot or power loss. Therefore, often sensitive data was stored unencrypted. In 2009 it was shown for DDR and DDR2 that transferring the memory from the machine of the victim to the attacker leaves the data intact [HSH+09]. To decelerate the loss of data they cooled the DRAM, hence the naming Cold Boot Attacks.
On DDR3 and DDR4 this has become increasingly difficult because of the introduction of Memory Scrambling. The data is scrambled with pseudo-random bits before being written to memory, which makes conventional Cold Boot attacks difficult.
But the Memory Scrambling is not a sufficient protection against Cold Boot Attacks. In 2016 Bauer et al. [BGF16] presented a template attack on scrambled DDR3 memory. They have shown that a LFSR with a periodicity of 64 bytes was used. With this they were able to descramble the whole memory when given only 64 bytes.
In 2017 Yitbarek et al. [YADA17] have shown a different attack, which works on DDR3 and DDR4 memory. They extracted keys from the memory and analysed them further. Their attempts have shown that 4096 different keys per channel were generated. Additionally, all the extracted keys contain certain patterns, which are sufficient to reliably identify scramble codes. Furthermore, they were able to construct an AES key litmus test to identify AES keys in a scrambled memory.

2.2 Background

Before we begin to analyse the Memory Scrambling on DDR4 we describe briefly all relevant components: The memory management unit (MMU), the Memory Controller Unit (MCU), the bus which transfers the data, and the DDR4-SDRAM where we want to induce the error. The terminology for DDR-SDRAM is not consistent over all publications. Therefore, in the following we will use the terminology by the JEDEC [JED05] as explained by [JNW07].
DDR-SDRAM stands for Double Data Rate Synchronous Dynamic Random Access Memory. Double data rate means that the data can be transmitted on the rising and the falling

edge of the clock signal. This only applies to the transfer of the data itself not for other accesses, e.g. commands. The execution of operations is synchronized by an external clock. All the data is stored dynamically and therefore has to be periodically refreshed. Each random 64 byte block of memory can be arbitrarily accessed in any given order.

2.2.1 Memory

The memory space of a given system can be divided into physical and virtual addresses. Physical addresses are the actual addresses of the data in the DRAM, whereas virtual addresses are the ones by which they can be accessed by a program.
The physical addresses can be further separated into bank address, bank group address, row address and column address. The addressing function is publicly documented by AMD [Dev13], but Intel has not made their mapping publicly known. There exists reverse engineered functions for some configurations [Sea20b], but these functions change depending on the used setup (processor, DRAM). Therefore, it is necessary to reverse engineer the functions for each system individually [PGM+16]. We will examine this for our system in section 3.5.
The virtual address space is partitioned into even-sized fragments called pages. In the physical address space these are called page frames. The virtual address space can be larger than the physical one. With that programs transcending the physical address space can be written. Additionally, the virtual addresses hinder programs to access memory, which they should not access.

2.2.2 MMU

The Memory Management Unit maps the virtual addresses onto the physical addresses. When the CPU tries to access a physical address via a virtual address, the MMU translates the address via a page table in the translation look aside buffer (TLB). The TLB can also contain information whether the page was modified or when it was last accessed. This information is used for eviction strategies. The introduction of a mapping also enables handling of non-continuous memory. If we have multiple continuous pages, it is possible to map each one to a non-continuous page frame, which is generally done to increase performance. Larger blocks of memory are split across multiple banks because it takes a longer time to access memory in the same bank.

2.2.3 MCU

The Memory Controller Unit is responsible for the reading and writing to DRAM. To access data inside the RAM we need to further split the physical address into row address, column address, bank address, bank group and channel. The exact separation is different depending on CPU and the DRAM setup. Multiple bits of the physical address are XORed for the corresponding result of a single bit. Which bits are used is found out via a row buffer conflict as stated by Gruss [PGM+16]. But this method does not allow to match the found bits to their function. To get this information probing on the pins is necessary.
The mapping is generally used to increase the performance of accesses to contiguous memory regions [WJ05]. On most systems we will switch between different banks before we try to access a new row in the same bank. That is done because access to a different row in the same bank takes a lot of time. Then, if these accesses are in different banks the process can be done independently in each bank. Additionally, on multi-channel systems the channel is changed after each 64 byte block to further increase parallelism.

2.2.4 DDR-SDRAM

The memory is connected to the processor via buses, which transmit the data and the commands. For each channel one dedicated bus is used. A DRAM module consists of multiple DRAM chips. Different modules can be controlled independently at the same time. Each chip can be roughly partitioned into the DRAM array, where the data is stored and the overlying peripheral circuits, which control the data.
The DRAM can be further divided into channels, DIMMs, ranks, bank groups, banks and rows. In a typical system we only have one channel with a width of 64 Bits to transfer data. Other channel setups are possible, but we will not discuss them further [JNW07].
A DIMM (Dual Inline Memory Module) is one physical module. But there are different segmentations of this, therefore we will give a physical and a logical one. On the physical side a memory module consists of two sides, a front side and a back side. DRAM chips can be positioned on both sides or only on one. These two sides can only be operated in lockstep, which means that only one side can be accessed at the same time. Additionally, a small time frame is needed to give I/O access from one rank to another one. We denote these sides as ranks. Therefore, each DIMM has always one or two ranks.
In each rank we have multiple DRAM chips, typical 16 in DDR4, which are independent of each other and can therefore be controlled in parallel. Each DRAM chip consists of multiple 2-dimensional arrays of DRAM cells, where all the cells are vertically and horizontally connected. The number of arrays depends on the type. Typically, there are either 4, 8 or 16 arrays inside a chip. This is stated as x4, x8, x16 in their description. An example

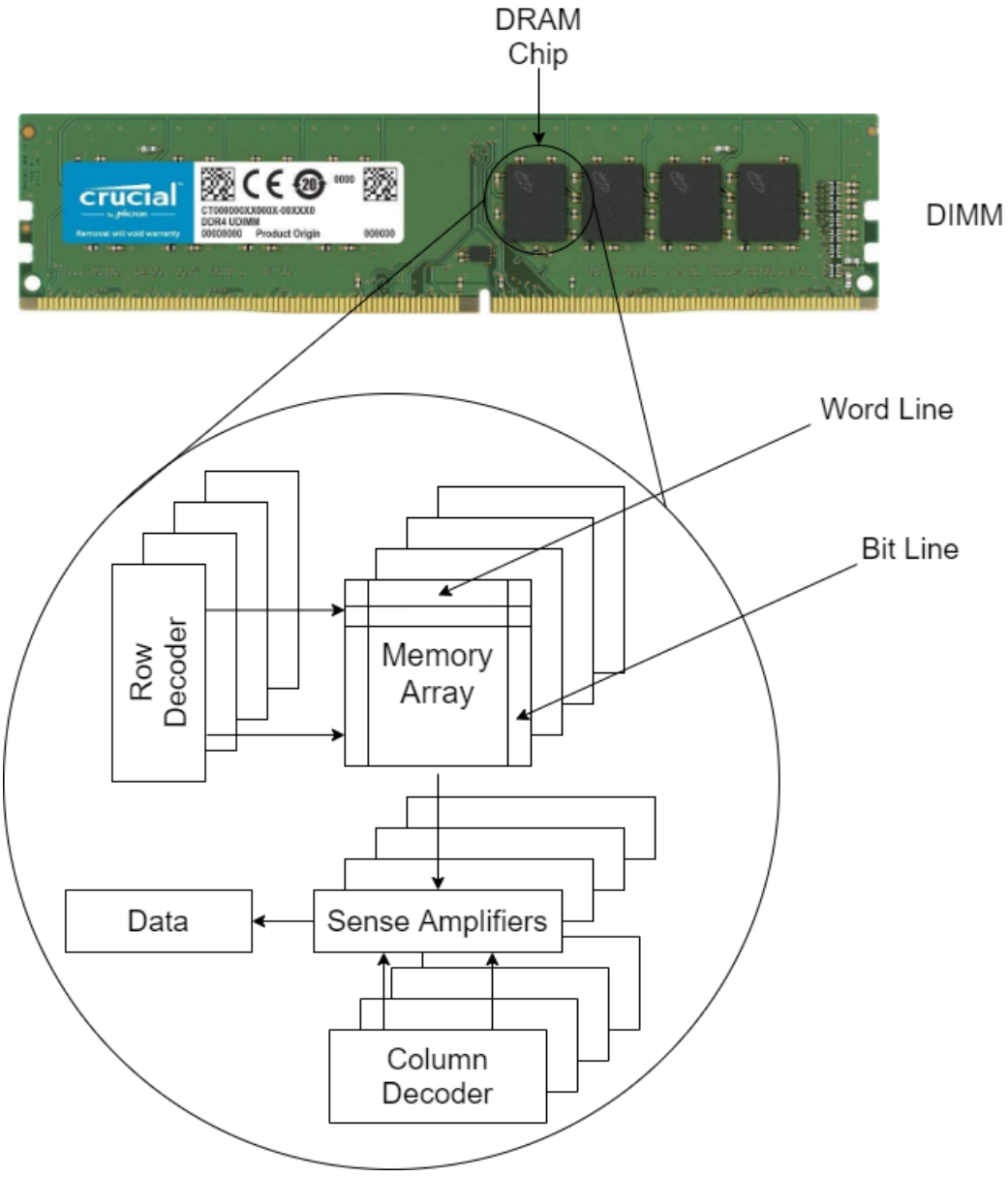


Figure 2.1: Schema of a x4 DRAM chip

for x4 can be seen in figure 2.1.
Each single cell in an array can store one bit of information via a capacitor. The horizontal lines of cells are called word lines, whereas the width of a word line, given by the number of arrays, is called the bit line. When a word line is activated, data is transferred from the word line to the sense amplifiers. The line of sense amplifiers is called the row buffer, where the data is temporally stored, so that multiple accesses to different locations in the same word line are more easily manageable.
On the logical side we call the memory arrays, which can be independently accessed, banks. Beginning with DDR4, multiple banks together form a bank group, as seen in figure 2.2. The advantage of this is to increase I/O parallelism [SJC+17]. On x4/x8 systems we have 4 bank groups and on x16 systems we have only 2. Further we divide a bank into multiple rows, where one row can be loaded into the row buffer. A single row consists of multiple columns, where each one contains the data transferred in a single burst. The banks in a bank group share the local I/O gating. But there still exists a global I/O gating for all bank groups.
The peripheral circuits have three major components: I/O-Component, Delay-Lock Loop (DLL) and Control Logic. The I/O-Component receives commands and transfers data between the DRAM and the MCU. The Control logic decodes the received commands, selects the chosen rows/columns and reads/writes the data. The DLL is used to synchronize data-transfers via an external clock.
In the following we give an overview for the most important I/O bits of the 288 pins of the DDR4 DRAM, a general list can be found in appendix C:

- $RESET\_n$: If high DRAM is active
- CS_n: Chip select, has to be low to read other inputs
- CKE: Enables internal clock
- CK_t/CK_c: Input for differential clocks
- DQ/DQS: Data bus and data strobe
- $RAS\_n/A16, CAS\_n/A15, WE\_n/A14$: Dual function inputs, if ACT_n is low they are address bits otherwise they are command bits
- ACT_n: Activates commands
- $BG0/1, BA0/1$: Bank Group, Bank Address
- $A0 - A13$: Input for addresses (Row and Column, number used depends on RAM size)

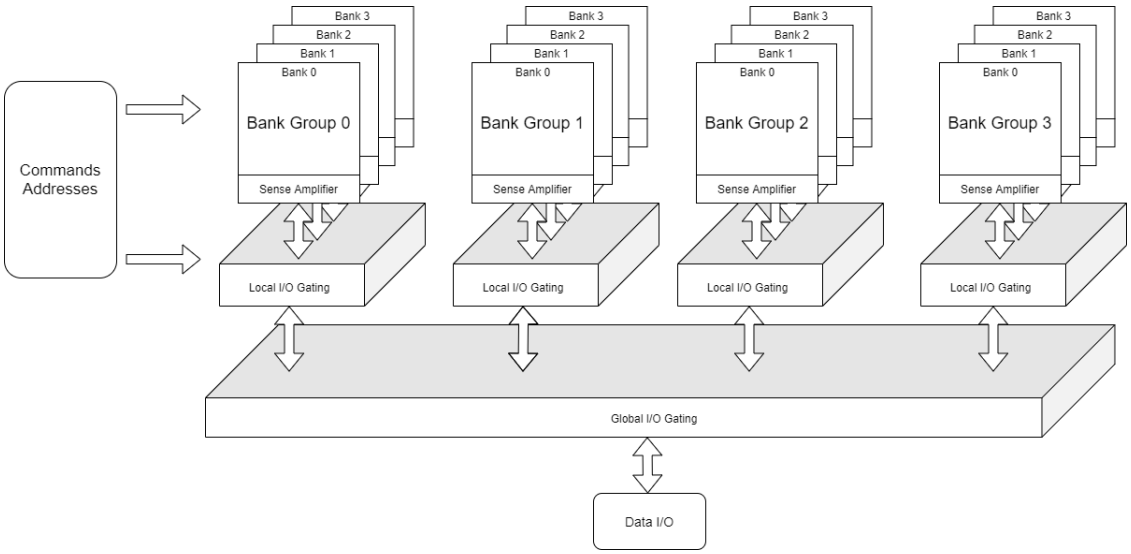


Figure 2.2: Schema of banks and bank groups

2.2.5 Communication Protocol

To read or write data to the memory, a communication protocol between the DRAM and the MCU is used. Depending on the ACT_n, CS_n, RAS_n, CAS_n and WE_n bits multiple different commands can be sent. The most important ones are Activate, Read, Write and Precharge. A simple read request consists of Activate, Read and Precharge.
Hereafter, we describe a READ request in greater detail. At first an Activate command is sent, which denotes the target row, bank and bank group via the corresponding pins. As a result, the electrical charge propagates from the chosen row to the row buffer. At the same time this drains energy from the chosen row. Therefore, its charge has to be restored later to prevent data loss. We now have one row in our row buffer, which typically has a size of 1 kiB. The activation of rows is done in each bank independently.
But the data in a row is too large to be sent in a single transfer. Therefore, the Read command denotes which columns shall actually be sent back to the processor. The entire transfer of the data is done in a so-called Data Burst, where over 8 cycles the corresponding data bits are sent. With one full Data Burst 64 bytes of data are sent.
The Precharge command is then used to prepare for the next access. Herby, the target active row will be closed and the value of it will be restored. If the following accesses are to the same row, it is not necessary to use a Precharge. Multiple Reads or Writes can be used sequentially. The reading of sequential data will be done without another Activate because the target row is already open. We just need to get the target column via a Read command.
There also exists Auto-Precharge commands, which will automatically precharge after a

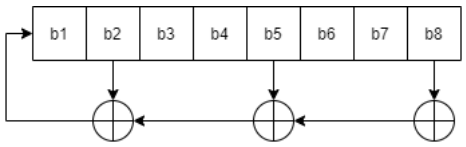


Figure 2.3: LFSR with polynomial $x^8 + x^5 + x^2 + 1$

Read or Write command without the need for another Precharge.
Additionally, each row has to be periodically refreshed even when not accessed, so that it does not lose its values. For that exists another command, Row Refresh, which will be periodically sent to the DRAM. When this command is sent each and every row in the memory will be refreshed.

2.2.6 Memory Scrambling

In older systems, which used DDR or DDR2-DRAM, the data was still stored in plain text. With newer and faster buses the scrambling was introduced as a means to improve signal integrity and to reduce power supply noise. Under normal workloads the observable traffic is not random. When the bus either fluctuates at some special frequency or multiple lines switch in parallel we can get high *di/dt* noise. This negatively effects the signal integrity. Therefore, Memory Scrambling is used to randomize the input s.t. each bit-flips from one time point to another one nearly 50% of the time. This eliminates the dangerous fluctuations and reduces the probability that an error occurs in the transmission. An example for DDR3 is shown by Mosalikanti [MMK11]. With no scrambling we get vastly different peak to peak noises, which can impact the integrity of the data. With scrambling activated the transmitted data becomes effectively white noise, i.e. like a random signal. The peak to peak noise becomes constant over all tested data patterns, so that the integrity of the data is guaranteed. For DDR4 much less information is given, therefore the detailed examination of Memory Scrambling will be the main point of chapter 3.

2.2.7 LFSR

The Memory Scrambling on DDR3 is commonly achieved by using a Linear-Feedback-Shift-Register (LFSR) as shown by Bauer et al. [BGF16]. A LFSR consists of a shift register and a feedback function. The function takes as input the entire register then modifies it by XORing some of its values and outputs a single bit. In the next step this output is then taken as the leftmost input for the shift register, while the remaining ones shift by one position to the right. Hereby, we can get multiple pseudo-random periodic sequences

depending on the initial state of the LFSR. In this context we use as an input for the LFSR a part of the address bits combined with a random seed. This is shown in figure 2.3. Alternatively we can also consider our input as a polynomial over $\mathbb{F}_2^n$, where the n state bits b_i represent the coefficients of the polynomial. The output sequence can here be calculated by multiplying the input with x and then doing a polynomial reduction with the characteristic polynomial.

2.2.8 Test System

For our test we use a system with the following components:

- ASUS ROG MAXIMUS X HERO Mainboard Socket 1151
- Intel Core i5-8600K
- Crucial DIMM 8 GB DDR4-2666

We verified the accuracy of our results by using the following system:

- ASUS PRIME H310-MK
- Intel Core i7-8700
- Micron DIMM 8ATF1G64AZ-2G6E1

2.2.9 Attacker Model

We perform the analysis and a possible attack under the following assumptions. As a preparation for the attack the attacker needs to have a system with the same parameters as the to be attacked system, to gain knowledge of how the underlying system operates. The preparations can be performed independently of the actual attack. On the attacked system the attacker needs only user privileges to read and write data (iff the Memory Scrambler is predictable). Additionally, the attacker needs a means to get the mapping from virtual to physical addresses or use the hugepages (or another means to get knowledge about the used physical addresses).

3 Memory Scrambling

The first advancements regarding Memory Scrambling were made by Bauer et al. [BGF16], who tried to gain further information about Memory Scrambling and its unknown mechanisms. We compare their results to Intel's Description of the Memory Scrambling on DDR4. To construct the attack, we then need to understand how the data is actually scrambled and send to the memory. Also, we need to learn how the scrambling depends on the input (address, data) and on random numbers. Furthermore, we not only need to know the stored data, but also the transmitted data, and the connection between these two. Then, we discuss our own results on Memory Scrambling on DDR4 and how it stands in relation to these two. Finally, we analyse how we can use our new knowledge to improve our attack and which problems remain to be solved.

3.1 Scrambling on DDR3

The first point of reference regarding DDR3 Memory Scrambling was given by Mosalikanti et al. [MMK11]. They stated that LFSRs are used to generate the scramble code. But most of their stated information is kept vague. Nonetheless, some general structures were described as following. For every Write command we generate 8 random strings of 16 bits, one for each burst of data. Additionally, the LFSR is shifted 16 times in every cycle. Each of the 16 bit wide codes is repeated 4 times to fully cover the 64 bit wide bus.
Next we examine the work from Bauer et al. [BGF16], who looked closer into the memory scrambling mechanisms on DDR3. In their work they stated four reasons why the descrambling proves to be difficult on DDR3. First, the information stated by Intel is unclear, therefore it is unknown which mechanisms are used in practice. Second, they acquired data via Cold Boot Attacks, which introduced errors into the scramble code and made the reconstruction even more difficult. Third, all data they acquired via a Cold Boot Attack contained the unknown and highly hardware specific ground state of the DRAM. And Fourth, the interleaved memory made it necessary to match data and channel because different channels use different Memory Scramblers.
To eliminate the ground state they used the differential of two scramble codes. On the created differential scramble codes they determined the periodicity and grouped the code into 64 byte groups. Inside these groups they removed errors via a majority function. It was shown that only 16 keys for each channel are generated, which leads to a possible

correlation between different blocks. Additionally, rereading scrambled memory after a reboot nullifies the effect of scrambling on a decently high part of the memory, about $\frac{1}{16}$. With this they were able to find a 64 byte key for differential key streams.

3.2 Changes from DDR3 to DDR4

The previous mentioned methods from Bauer no longer achieve the desired results on DDR4. That may be caused by a change of the Memory Scrambler or by more general changes from DDR3 to DDR4. On a high level the following improvements are made for DDR4: Higher bandwidth, reduced power consumption, increased density, and more reliability.

The first improvement is the higher bandwidth. DDR3 was only specified by the JEDEC up to 2133 Mhz. With overclocking this could be increased and most processors could scale up to 2400 Mhz with still good performances. But that has its limits and becomes exponentially more expensive. DDR4 at launch supported 2400 Mhz, but its specifications allowed for even higher speeds.

The second improvement is the reduced power consumption. DDR3 requires 1.5 V while DDR4 only needs 1.2 V. Additionally, DDR4 has a second voltage specification, so that the default one no longer has to be scaled down internally. In this context new methods to refresh the memory were also introduced. The intention of them is to reduce power consumption depending on the current operations. These changes to refresh timings may interfere with our later attack.

The the improvement is the increased density of a single DIMM. While normal DDR3 is 8 GB, DDR4 doubled the size to 16 GB. Additionally, the internal structure got reorganized to cluster 4 banks together by introducing bank groups. This changed the mapping from physical addresses to row addresses, column addresses, bank addresses and bank group addresses.

Finally, DDR4 improved the reliability of the memory. For the transmission of the commands and addresses a parity error detection and recovery has been introduced. For writing to the memory a cyclic redundancy check (CRC) has been added. The usage of CRC has possible implications on our attack and will be further discussed in section 4.6.

All these changes do not mean that we can't use any of the previous methods at all, but we need to evaluate every method again.

As shown by Yitbarek et al. [YADA17] the previous methods no longer work on DDR4 scramble code. In their work they showed that now 4096 different keys are generated for each boot, so that the differential analysis no longer works. Nonetheless, they were still able to identify AES keys inside the scramble code by using the equations which hold true.

But they didn't try to find out how the scramble codes are generated.

3.2.1 Litmus-Test

For DDR3 Bauer et al. [BGF16] described a Litmus-Test to identify the scramble code. Yitbarek et al. [YADA17] showed that the Litmus-Test also works for DDR4. This Litmus-Test takes the form of some equations which always hold true for each 64 Byte block, if it is a scramble code. In a more detailed analysis we learned that their equations can be made even stricter, which may be useful if only partial data of a scramble code is known. The equations for DDR4 inside a single aligned 64-byte scramble code are the following:

$$\begin{aligned}
K[i:i+1] \oplus K[i+2:i+3] &= K[i+8:i+9] \oplus K[i+10:i+11] \\
K[i:i+1] \oplus K[i+4:i+5] &= K[i+8:i+9] \oplus K[i+12:i+13] \\
K[i:i+1] \oplus K[i+6:i+7] &= K[i+8:i+9] \oplus K[i+14:i+15] \\
K[i+2:i+3] \oplus K[i+4:i+5] &= K[i+10:i+11] \oplus K[i+12:i+13] \\
\text{for } i &\in \{0, 16, 32, 48\}
\end{aligned} \tag{3.1}$$

3.3 Intel's Description of Memory Scrambling

Regarding Memory Scrambling Intel has always been scarce with the publicly available information. In [MMK11] it became known that for DDR3 LFSRs are actually used. They stated there that the LFSRs are 16 bits long and seeded with 16 bits of the address, which parts exactly is unknown. The address may also be XORed with a random string.
In another Intel patent [Ee20] some more complex procedures are briefly stated. But all procedures are still described imprecise to give a plethora of possibilities. Therefore, it is unclear if any of them are actually used in a practical application. In the following the most complicated process of Memory Scrambling, as described by the patent, is structured into four steps: Generating parallel LFSR outputs, reordering and remapping logic, Data Bus Inversion (DBI), and selective switching. For none of these is an actual implementation given, nonetheless in the following part we will try to explain the procedures. Additionally, it is important to remember that the actual use of these mechanisms is held pretty vague. Steps 2 to 4 are perpetually described as optional. Therefore, we can't be sure if any of the procedures are actually used. With these steps a scramble code of some fixed but unknown length is generated.
In the first and only non-optional step the base for the scramble code is generated. Here they use a seed and an unknown number of parallel LFSRs. The content of the seed is not specified, but pseudorandom numbers or the last scrambling code were mentioned as

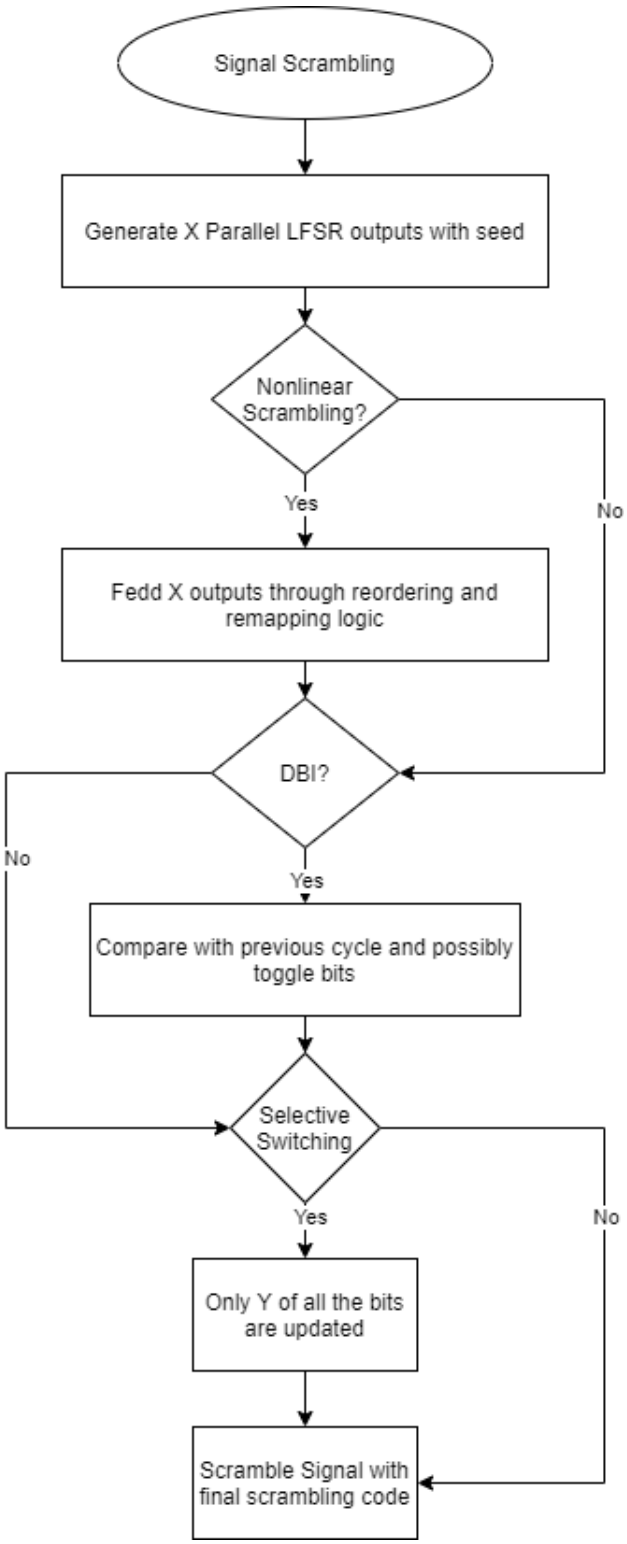


Figure 3.1: Schematic description of the memory scrambling as described by patent [Ee20]

examples.
In the next step some kind of reordering and remapping logic can be used to obfuscate the outcome of the LFSR. To implement this, substitution boxes could be used.
In the next step DBI (Data Bus Inversion) may be applied. We will discuss this in section 4.3 in greater detail.
At last they talk about selective switching. Hereby not the entire scramble code will be used as an output. The scramble code from the previous step will be used as a base, but some of its bits are updated with the newly generated ones.
The final scramble code is then the output of a LFSR manipulated by some scrambling techniques. After it has been generated via these means, it is XORed with the data and sent to the DRAM.
If we compare the description with our knowledge gained from Bauer on DDR3 we can see, that the additional scrambling techniques proposed by the patent were not used on DDR3. The first step to generate the base may be the same, but it is also possible that this step is more complicated, e.g. through the introduction of more parallel LFSRs or the use of different LFSRs for different banks. Which techniques may be used will be further examined in section 3.7

3.4 Data Acquisition

To reverse engineer the Memory Scrambling we need to get access to reliable scrambling data. We try to construct three sets of data: The raw data we write to memory, the scrambled data, which is actually transferred, and the corresponding physical addresses. Because we choose the raw data and sent it to the memory, we have complete knowledge of it. In the following we will describe methods to get the actual data written to the memory.
One method to gain this data is via a Cold Boot Attack as proposed by Bauer [BGF16]. For their attack we reboot the system from a cold state and then read the memory. The problem of this method is the unknown ground state of the memory. If we read the memory with this method we only get an XOR of the ground state G and our unknown scramble code K_1. Therefore, we only know $G \oplus K_1$.
To remove the ground state we can write a known text T to the memory so that it contains $T \oplus K_1$ and then turn the computer off and on. Hereby we get a different scramble code K_2 for the memory access. If we then read the data again we get $T \oplus K_1 \oplus K_2$. To get a new seed the computer has to be properly turned off. A simple reboot will not change the scramble code. But turning the system off leads to the problem that many bits are lost. To prevent this from occurring, it is possible to use the methods used by Cold Boot Attacks

to read the data with a different computer by freezing and swapping the memory. This method my retain more bits, but we still only get a differential of two scramble codes. It is possible to use the differential, but we will try a different approach to get a scramble code by itself.
We utilize a feature, which a few current motherboards posses: The ability to disable the Memory Scrambling via the BIOS. Now, we are able to formulate a different approach to read the memory. The general procedure is as follows: First we disable the Memory Scrambler, then we write our known text T to the memory. After that we reboot the computer and enable the Memory Scrambler with scramble code K_1. Now we can read from our memory $T \oplus K_1$. In all our experiments we are able to get a potential scramble code with this method. In practice, we write only zeros to the memory, therefore eliminating the need to XOR the read and written data. Additionally, in our tests the Memory Scrambler has to be turned off and then turned on to read scramble codes. If we reverse the order, we were never able to generate any scramble codes. By turning the Memory Scrambler of, writing data, and then turning the Memory Scrambler back on, we only read non-scrambled data in every single memory region.
For our method of acquisition we have no guarantee that the 64 byte blocks are not changed in between write and read. We accomplish this by simply testing the scramble code candidates with the aforementioned Litmus-Test. Therefore, we can determine if our data contains a scramble code. But the process is still inefficient because we can't access every block easily and many of them may be overwritten.

3.4.1 Refinement via Hugepages

While we are able to get scramble codes with the aforementioned method we want to have more context for them, to easily compare and evaluate our data regarding their appertaining memory addresses. By using this method, we want to achieve two things. First, we want to have a constant set of physical addresses for all experiments and especially we want to assure that we read and write to the same addresses. And second, we want to have consecutive addresses to more easily evaluate the experiment. Nonetheless, the following improvement is not required to perform the experiment, but it makes it easier to generate larger sample sizes of continuous memory. We can still get pairs of addresses and scramble codes by repeatedly conducting the experiment.
To achieve this we will use hugepages. Hugepages of a size of 1 GB compared to smaller hugepages have the advantage that they don't change their physical addresses between reboots. If we write something to a specific address, we can easily access the same address after a reboot. This makes it easy to send data to a specific memory location and after a

reboot read out the result of the scrambling of the sent data on the specific location. Additionally, the hugepages are created in low physical addresses and will therefore seldom be used. In our tests the hugepages were only overwritten, if we issued instructions, which take up nearly the entire memory. With the use of hugepages it is easier to generate lots of address and scramble code pairs than with the other mentioned methods.
Information to set up hugepages can be found under [Cok17].

3.5 Finding the Mapping

To gain further information for the input of the LFSRs we need to find the mapping from physical addresses to row addresses, column addresses, bank groups and bank addresses. Depending on the number of channels, size of the DRAM, and the processor the mapping is different. We use the mapping to further analyse, which parts of the addresses will be used as a seed for the LFSR. Generally the lowest 6 bits are used as the byte index inside a 64 byte block. They are followed by the upper part of the column address and then the row address. But some bits in the physical address do not belong to the previous mentioned addresses. These bits are used to indicate the bank the addresses belong to. For that they are often XORed with some bits of the row address. To find out the mapping we use the attack on the row buffer timing by Gruss et al., which was discussed in section 2.1.2. On our system we get the result that the combination of bits 6+13, 14+17, 15+18 and 16+19 of the physical address is used to indicate the bank. To learn, which combination of bits correspond to BG0, BG1, BA0 or BA1, we would need to measure the physical bits itself, but for our purpose the identification of the bits is enough.

3.6 Explanation of Verification Methods

We have now seen, how we can get verified data pairs of physical address and the corresponding scramble code, but we still have no knowledge how the scrambling itself is actually conducted. In the following sections we will describe the methods used to find possible scrambling functions and then the tests we conducted with them. The basic functionality of a LFSR has been explained in section 2.2.7. It is to note that the described methods only reflect our methods to calculate the results in a mathematical approach. The actual implementation of these methods may be different to optimize its calculation.

We formulate the mathematical groundwork for the methods and tests as follows. Let $a = (a_{k-1} \dots a_0)$ be an address of length k and L be an LFSR of also length k, where each bit in L means that this bit is used for the next step. As a result we can get the next bit by computing the vector product of a and L. All our following examples use the value

of $k = 5$. In practice a value of $k \geq 16$ has to be used to achieve outputs similar to the scramble code.

$$\begin{pmatrix} a_4 & a_3 & a_2 & a_1 & a_0 \end{pmatrix} \cdot \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} = \begin{pmatrix} a_5 \end{pmatrix} \tag{3.2}$$

Now, we can define a Matrix $\hat{L}$ which computes the content of the address register in the next step, i.e. one shift of the LFSR, by using the value of the current one as shown in the following.

$$\begin{pmatrix} a_4 & a_3 & a_2 & a_1 & a_0 \end{pmatrix} \cdot \begin{pmatrix} c_0 & 1 & 0 & 0 & 0 \\ c_1 & 0 & 1 & 0 & 0 \\ c_2 & 0 & 0 & 1 & 0 \\ c_3 & 0 & 0 & 0 & 1 \\ c_4 & 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} a_5 & a_4 & a_3 & a_2 & a_1 \end{pmatrix} \tag{3.3}$$

Hereby, we get $a_5 = \sum_{i=0}^{4} c_i \cdot a_{4-i}$ the same way we calculated the next bit before. But with this matrix we still have the bits from before to make the calculation of the next step possible. Our matrix $\hat{L}$ consists of the values c_i of the LFSR and a permutation matrix to keep the old bits inside the register.
The matrix can be further modified to get the next steps of the register, i.e. what happens after k shifts. To achieve this we have to apply $\hat{L}$ to a exactly k times. Given a concrete size of the register we can use this to more easily calculate the concrete values of the LFSR.

$$\begin{pmatrix} a_4 & a_3 & a_2 & a_1 & a_0 \end{pmatrix} \cdot \begin{pmatrix} c_0 & 1 & 0 & 0 & 0 \\ c_1 & 0 & 1 & 0 & 0 \\ c_2 & 0 & 0 & 1 & 0 \\ c_3 & 0 & 0 & 0 & 1 \\ c_4 & 0 & 0 & 0 & 0 \end{pmatrix}^5 = \begin{pmatrix} a_9 & a_8 & a_7 & a_6 & a_5 \end{pmatrix} \tag{3.4}$$

The general matrix for an LFSR can then be multiplied out. And then, if we have knowledge about the in- and output, this formulates some solvable equations, which are only dependent on the c_i, the unknown values of our LFSR. If we then solve these equations

we can get the actual values for the LFSR.

$$\begin{pmatrix} c_0 & 1 & 0 & 0 & 0 \\ c_1 & 0 & 1 & 0 & 0 \\ c_2 & 0 & 0 & 1 & 0 \\ c_3 & 0 & 0 & 0 & 1 \\ c_4 & 0 & 0 & 0 & 0 \end{pmatrix}^2 = \begin{pmatrix} c_0^2 + c_1 & c_0 & 1 & 0 & 0 \\ c_0c_1 + c_2 & c_1 & 0 & 1 & 0 \\ c_0c_2 + c_3 & c_2 & 0 & 0 & 1 \\ c_0c_3 + c_4 & c_3 & 0 & 0 & 0 \\ c_0c_4 & c_4 & 0 & 0 & 0 \end{pmatrix} \tag{3.5}$$

The equations of the full expansion of $\hat{L}^5$ can be seen inside the appendix A. We can also use non-quadratic matrices for our LFSR, if we are interested in a longer output sequence. For example to generate $2k$ output bits we can use $[\hat{L}^{2k}|\hat{L}^k]$.

We have seen that as long as a LFSR is used, we have some kind of linear dependency between our outputs. We can emulate the linear dependencies as a matrix multiplication. This leads to the formulation of two different tests. In the first test we try to find the linear dependencies. Let us say we want to find out if $x_1 = (x_{1,1}, x_{1,2}, \ldots, x_{1,k})$ and $x_2 = (x_{2,1}, x_{2,2}, \ldots, x_{2,k})$ of length k depend on the same LFSR for their outputs $y_1 = (y_{1,1}, y_{1,2}, \ldots, y_{1,r})$ and $y_2 = (y_{2,1}, y_{2,2}, \ldots, y_{2,r})$ of length r. We can calculate if there exists an arbitrary Matrix M, which contains linear equations, so that $x_0 \cdot M = y_0$ and $x_1 \cdot M = y_1$. If a solution doesn't exist then there can be no LFSR. But even if a solution for M is found, it does not mean that a LFSR actually does exist.

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,k} \\ \vdots & & \ldots & \vdots \\ x_{s,1} & x_{s,2} & \cdots & x_{s,k} \end{pmatrix} \cdot M = \begin{pmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,r} \\ \vdots & & \ldots & \vdots \\ y_{s,1} & y_{s,2} & \cdots & y_{s,r} \end{pmatrix} \tag{3.6}$$

The first test is only an indicator that there may be a LFSR. Then, we can verify the assumption with the second test, if such a LFSR exists. For this we use the general matrix $\hat{L^k}$ of size k. Then we solve the equation for $x_0 \cdot \hat{L^k} = y_0$. To use the test we need k bits of information for the input x_0 and k bits of information for the output y_0. If we solve the equations we get solutions which may be candidates for the LFSR. Such solutions are not unique if k is small.

To solve the equations efficiently we have used *SageMath* inside a python script. *SageMath* contains the important *m4ri* methods [Bar09, Alb12], which we use to efficiently solve the equations. Additionally, it is possible to solve the much more complicated second test as shown in figure 3.2. We construct an ideal from our multivariate equations over a Boolean Polynomial Ring R. For this we use one possible input for the LFSR x the generated general LFSR L and the expected output y. On the ideal we can then find all possible solutions

```
equations=construct(x,L,y)
J=R.ideal(equations)
Solution=J.variety()
```

Figure 3.2: Solving LFSR equation with $SageMath$

```
BB 13 7D 0B 1E 87 2F 41
BA 13 7C 0B 1F 87 2E 41
B2 13 74 0B 17 87 26 41
B2 12 74 0A 17 86 26 40
B2 12 74 0A 17 86 26 40
B6 32 70 2A 13 A6 22 60
A6 32 60 2A 03 A6 32 60
A6 32 60 2A 03 A6 32 60
```

Figure 3.3: Exemplary aligned 64 byte block of scramble code for address 0x000040

for our variables as varieties of the ideal. The second test does take a longer time to be completed, about 2 seconds per test. The biggest problem of the second test is that we need to have exact information about the input and output. Therefore, we will use the first test as an indicator for a possible LFSR and then use the second one to find its values.

3.7 Different Tests and their Results

3.7.1 Testing Basic Structure

As a first experiment we aim to verify, that our read data is actually generated by a simple XOR-operation of the sent data and the scramble code. For this we write not only zeros to the memory, but more complicated patterns. Then we XOR sent and read data to calculate the possible scramble code. Scramble code which has been generated without turning the power of is completely the same. If power had been turned off, we would get different scramble codes, but they still have the same internal structure as described in section 3.2.1. Now we can be sure that the Memory Scrambler uses an XOR-operation to unify sent data and scramble code.
In this context we can further examine the internal structure of the scramble code. An exemplary 64 byte block is shown in figure 3.3. Every aligned block has the same kind of structure, but with different values. Structure in our context means that the equations which hold true for DDR3 also hold true for each aligned 64 bytes of DDR4 scramble code. In the next step, we try to find out what is used as the input for the LFSR. For this we examine a single scramble code of 1 GB length. Inside the scramble code all the 64 bytes

blocks are repeating in a certain manner. If the lower 20 bits are the same, the 64 byte block is also the same. Inside each cycle every block appears 8 times, when either the bits 14 and 17, 15 and 18 or 16 and 19 are the same. This corresponds with our previous analysis of the physical addresses, where the XOR of these bits is used to calculate bank address and bank group. The lower 6 bits of the addresses are only used for the byte position inside each block. Therefore, we have found 11 bits, which uniquely identify each single of the 2048 different 64 byte blocks. The entire structure is constant between reboots, but is different on another system when they have different mappings.
The scramble codes between reboots on the same system contains many differences. Therefore, some kind of random string has to be involved. This random seed changes if the power is turned off, but on our system a reboot or even turning the Memory Scrambler on and off does not change it. The structure in each 64 byte block is always the same, but the actual parameters of it are always changing with no consistency between two different scramble codes. With simple observations we are unable to find further patterns.

3.7.2 Tests between different 64 byte blocks

Our previous described methods model the default LFSR to generate the base scramble code. We can not model arbitrary non-linear substitutions, but we can easily model reordering of the bits via a permutation matrix. Nonetheless, some basic substitutions can be included by modelling them as linear equations in a Matrix. If we expand our input by 1 column and our Matrix by 1 row we can also model constant offsets. Additionally, if we only use the first test to indicate linear relationship between the blocks, these matrices are integrated into our result matrix by a simple matrix multiplication. The DBI itself does not need to be considered because DBI only effects the transmitted bits and not the bits which are stored or retrieved. Therefore, our data does not contain any effects of DBI. For this test specifically we also do not need to consider selective switching, if we only use the first part of each scramble code which can not be updated because a previous scramble code does not exist.
As a result we can construct a generic test. We use the previous mentioned 11 bits from the addresses a_i, plus one bit for some basic substitutions, and the corresponding scramble codes c_i. For the first experiment we assume that the same LFSR L and the same random seed r is used. This leads to the following equation:

$$\begin{pmatrix} a_i \oplus r \\ \vdots \\ a_n \oplus r \end{pmatrix} \cdot L = \begin{pmatrix} c_i \\ \vdots \\ c_n \end{pmatrix} \tag{3.7}$$

To test this we iterate over all 2^{12} possibilities for r. If a_i is shorter than r, it would be filled with leading zeros. It is not necessary to test for r longer than 12 because the additional length would only create a constant offset for each bit in our solution. But the equation does not yield any results for a large enough sample size.
In the next step we change the random string, s.t. it is no longer a constant. Here we assume that the random values change consistently, s.t. we can find pairs were the XOR of the random values is the same. That would be the case, if for example 2 or 4 random values are used, or they increase linearly.

$$\begin{aligned}
(a_0 \oplus r_0) \cdot L &= c_0 \\
(a_1 \oplus r_1) \cdot L &= c_1 \\
\implies (a_0 \oplus a_1 \oplus r) \cdot L &= c_0 \oplus c_1 \\
(a_2 \oplus r_2) \cdot L &= c_2 \\
(a_3 \oplus r_3) \cdot L &= c_3 \\
\implies (a_2 \oplus a_3 \oplus r) \cdot L &= c_2 \oplus c_3 \\
\implies \begin{pmatrix} a_0 \oplus a_1 \oplus r \\ a_2 \oplus a_3 \oplus r \end{pmatrix} \cdot L &= \begin{pmatrix} c_0 \oplus c_1 \\ c_2 \oplus c_3 \end{pmatrix}
\end{aligned} \tag{3.8}$$

With these assumptions we can solve this equation the same way as the one before. We tried this for multiple possible combinations, but we also get no solution.
The general procedure can be expanded to also test scramble code with different random seeds, so that we eliminate the addresses.

$$\begin{aligned}
(a_0 \oplus r_0) \cdot L &= c_0 \\
(a_0 \oplus r_1) \cdot L &= \hat{c_0} \\
(a_1 \oplus r_0) \cdot L &= c_1 \\
(a_1 \oplus r_1) \cdot L &= \hat{c_1} \\
\implies \begin{pmatrix} a_0 \oplus a_0 \oplus r_0 \oplus r_1 \\ a_1 \oplus a_1 \oplus r_0 \oplus r_1 \end{pmatrix} \cdot L &= \begin{pmatrix} c_0 \oplus \hat{c_0} \\ c_1 \oplus \hat{c_1} \end{pmatrix}
\end{aligned} \tag{3.9}$$

If the scramble code is generated by some kind of linear matrix, we would assume that the differential of two scramble codes with different random seeds, but the same addresses, is the same. But that is not the case. Additionally, the differential always fulfils the aforementioned equations.
In another step we tried the idea of changing LFSRs. Hereby we assumed that multiple

LFSRs are used, but they are reused multiple times. We did not assume when they would be reused. For this we used many pairs of 16 byte and the following 16 byte, all aligned inside our 64 byte blocks. Then we used our second method on all the pairs and counted all possible solutions. Inside all the possible solutions we tried to find any solution for the LFSRs which appeared more often than other ones. But even the most common solutions appeared to be reused only by chance.

3.7.3 Tests inside a single 64 byte block

In the following we want to examine closer the equations inside a single 64 byte block. In section 3.2.1 we have seen the equations which hold true for DDR3. On our sample data all the equations also hold true for DDR4. In a revised version we can describe the equations in the form $(v_{j*4+i} >> 1) \oplus p_i = v_{j*4+i+1}$ for $i \in \{0, 1, 2\}$ and $j \in \{0, \ldots, 7\}$, where v_i is a 2 byte block inside our 64 byte block. This indicates a possible LFSR relationship between neighbouring 16 bits. In the following we will examine the relationship between these in greater detail.
As a preliminary step we try to verify that this relationship is in fact a linear one. In all the following tests we will neither need the input addresses nor some random seed. We assume that the first part of the scramble code, the first 2 bytes, are used to generate the next 2 bytes. We want to verify if the following equation has a solution:

$$\left(v_{j*4+i}\right)_{i\in\{0,1,2\},j\in\{0,\ldots,7\}} \cdot L = \left(v_{j*4+i+1}\right)_{i\in\{0,1,2\},j\in\{0,\ldots,7\}} \tag{3.10}$$

On our samples we are able to find a solution for the equation, but it is not directly useable as a LFSR. Therefore, we can now use the second test. But the result indicates that there is no possibility to have a single LFSR, which fulfils this equation. In the next step we try to emulate the possibility that multiple different LFSRs are used. We expand our scope to other 64 byte blocks and look at all possible LFSRs in our result set, to see if there are multiple results, which appear with a higher frequency. Nonetheless, we can not find a single LFSR appearing significantly more often than other ones and even amongst the ones with the highest occurrence there is no identifiable set, which seems to be used as LFSRs.
In the later parts of our research we found out that we can improve the aforementioned equations if we invert the byte order of each 2 byte block. Now we can reformulate our equations as $(v_{j*4+i} >> 1) \oplus p = v_{j*4+i+1}$ for $i \in \{0, 1, 2\}$ and $j \in \{0, \ldots, 7\}$, where v_i is a inverted 2 byte block. We only need one single polynomial for each 64 byte block. Over

all possible scramble codes only 16 different polynomials were found, which appeared evenly distributed, but without any kind of identifiable pattern. We tried to reconstruct the LFSR from these polynomials by using the second test, but even for a single block no common LFSR could be found.

3.8 Interpretation

With these experiments we have tried to properly reverse engineer the Memory Scrambler. We have seen that a simple expansion of the scope from DDR3 to DDR4 is not possible and more complex methods need to be used. We were only able to find linear dependencies inside a 64 byte block, but none between the scramble code and the part of the physical address used to generate the scramble code. Even inside a single 64 byte block we couldn't find a LFSR relationship between 16 bit strings. Therefore, we assume that some obfuscation measures are used. It seems unlikely that a proper non-linear substitution is used because we are still able to verify linear dependencies. But it is likely that either a substitution or reordering is used. To verify that with our methods is not feasible because if we consider all $2^{2^{20}}$ possible substitutions and $16!$ permutations neither the amount of samples we get for a single seed nor the efficiency of our methods seem sufficient. Nonetheless, we could slightly improve the existing tests.
Our overall goal is to gain knowledge in such a way that it is possible to remove the influence of the Memory Scrambler. There are systems where the scramble code is constant between reboots. On these systems it is easily possible to remove the influence of the Memory Scrambler by simply gaining its values with one of the previous described methods and then using this scramble code at any time. If the scramble code changes between reboots, the same is possible as long as the power is not turned off completely. If this is not possible, it is necessary to gain information about a lot of the bits inside the scramble code, but with our improvements to the tests, slightly fewer bits need to be found.
The scramble code itself still contains a lot of structure, which we were not able to properly implement into our test. Therefore, we are optimistic that with enough time it is possible to improve the equations further and maybe even find a solution for the LFSR.

4 Attack on the bus

We want to transfer arbitrary patterns over the memory bus to possibly induce errors. For this we need to examine further how the communication protocol works exactly. For that reason we identify which lanes can be used to send a continuous pattern. Additionally, we examine possibilities to send bits continuously with as few interruptions as achievable. The optimal result would be, if we are able to send bits on each edge of the clock. To verify if the patterns are actually transferred in an expected manner, we measure the transferred bits with an oscilloscope.

After we have established a basis to sent our data, we will try to use the optimized methods to generate errors with the transfer of data. That this is theoretically possible on DDR3, has been shown by Mosalikanti [MMK11]. In their experiment the data was sent in a continuous pattern via external means over all possible lanes at the same time. While their results only show that errors are possible, it does not imply, that they can be induced by standard user interactions. Additionally, they have not explicitly stated, that the found patterns lead to errors, only that they have high peak to peak differences. We attempt to induce the errors by transferring specific data patterns to the DRAM via the normal Communication Protocol. In general, there are two ways to find error inducing sequences, if they exist. The first possibility is to calculate the *di/dt* fluctuations and construct sequences out of them. Second, we can try to find patterns experimentally. In the following we use the experimental approach. For that purpose we have obtained a motherboard, where we can disable the Memory Scrambling. With the help of this we can send arbitrary patterns to the DRAM, independently of our knowledge about the Memory Scrambling.

4.1 Communication Protocol

We show exemplarily which bits are transferred with a WRITE request. In doing so we will not look into a READ request, because for the following analysis they are similar. The differences between these two will be further explained in section 4.2.
The bits transferred in a single WRITE request are shown in figure 4.2. At the beginning the row address and the column address are sent over four cycles via the pins A0-A15, BG0/1, BA0/1, which we will in the following only call the address pins. Additionally, the pins to signify that this is a WRITE request, which are CS_n, RAS_n, CAS_n, WE_n,

Command	CS_n	ACT_n	$RAS\_n/A16$	$CAS\_n/A15$	$WE\_n/A14$	$A10/AP$
Refresh	Low	High	Low	Low	High	Low or High
Single Bank Precharge	Low	High	Low	High	Low	Low
Bank Activate	Low	Low	Address	Address	Address	Address
Write	Low	High	High	Low	Low	Low
Write with Precharge	Low	High	High	Low	Low	High
Read	Low	High	High	Low	High	Low
Read with Precharge	Low	High	High	Low	High	High

Figure 4.1: Commands for the DRAM

are set to the required value. In more details we sent a Bank Activate over the command pins while specifying the opened row. Then we sent the Write request while we specify which column should be opened. More information about the relevant pins was already stated in section 2.2.4. All the different commands are stated in table 4.1. Then, in the next step over 4 cycles the corresponding data is transferred. The data is sent over 64 lanes with two bits per cycle, one on each edge of the clock. Hereby, 64 bytes of data are transferred. Transferring 1 GB of data with single instructions would generate too many delays, therefore we will discuss how data can be transferred as efficient as possible.
If we look upon this trivial example, the only lanes where bits are continuously sent, are the data lanes DQ0-DQ63. We can control and manipulate the values sent through the address pins, but these lanes are only used for a small amount of time. Additionally, it is not possible to freely access any address without some delays occurring. Therefore, in the following we manipulate the data sent via DQ0-DQ63 to transfer our patterns. But if we simply transfer data in such a manner there would still be a lot of unused time between the transfers of data to different addresses. As a result, the most efficient approach would be to use the data lanes and find out how they can be best pipe lined as shown in figure 4.3. This approach is generally possible and normally supported by the MCU except if it violates some timing constraints between different operations configured inside the BIOS. More details on the necessary timings between different operations can be found in the appendix D. If we read the memory out of order, e.g. two accesses to different rows in the same bank we get longer delays between the commands. The effect of this has been discussed in section 2.1.2. If we access the memory in a random order, it is likely that some of these delays occur. Therefore, it is advantageous to access the memory continuously.

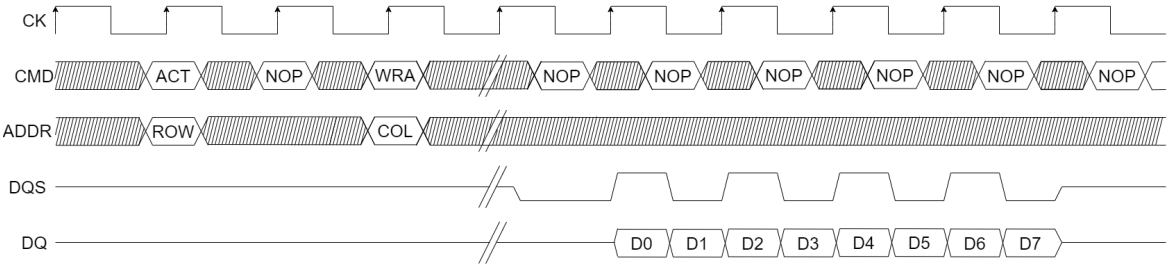


Figure 4.2: Schema of Writing 64 byte to memory

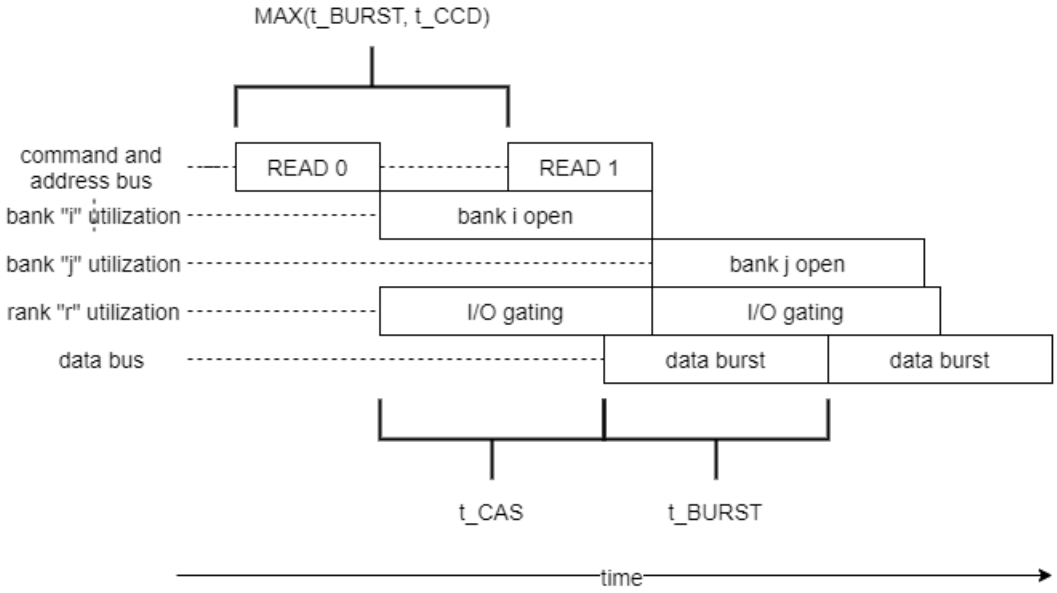


Figure 4.3: Pipelining of reads to different banks in the same rank

Method	Trivial	v4si	AVX2	AVX2 NT	AVX2 NT opt.	Theoretical opt.
Time	0.22	0.139	0.140	0.076	0.058	0.0503

Figure 4.4: Timing measurements of sending 1 GB of data to the memory via different implementations

4.2 Implementation

We have seen that different access patterns lead to different access timings, because some patterns require delays in between to precharge and/or switch the banks. Therefore, we want to reduce these delays as much as possible. The best way to achieve this is to access continuous memory regions. As described in section 3.5 the memory is structured in such a way that accesses to the next memory region are always fast. Therefore, we use this to reduce the delays. In practice, we will once again use hugepages, which are discussed in section 3.4.1, to access 1 GB of continuously allocated memory. Depending on the actual timings, there could be some delays, because the timings are configurable. But in our tests with the default parameters the approach leads to no identifiable delays caused by the access patterns.
We now want to improve our code in such a way that we use the best instructions to transfer the data without any unnecessary delays. For these improvements we define a timing goal, which describes the theoretical optimal time to transfer 1 GB of data, if the data is transferred in each cycle and all row and column addresses are completely pipelined. This is bottlenecked by the speed of our DRAM. The initial row and column addresses will not be considered in the calculation, because they are negligible small. We measure the time, which our system needs to write 1 GB of data to the hugepage and compare this with the timing goal.
Our system contains an 2667 Mhz DRAM. This means we can do 2667 MT/s. Each single transfer consists of 64 bits. Combined we want to transfer 1 GB = 8589934592 bits of data. This leads to:

$$\frac{8589934592 \text{ bits}}{\frac{2667 \cdot 10^6}{\text{s}} \cdot 64 \text{ bits}} = 0.050325 \text{ s} \tag{4.1}$$

In the following we describe which optimizations have been tried, to get as close to this time as possible. All following time measurements are made with a transfer of 1 GB with a 2667 Mhz DDR4 DRAM averaged over 1000 different accesses on the same memory region. The results of this are shown in figure 4.4

Trivial

The first attempt to get a baseline was to have no optimizations and no special instructions. Hereby we write all data as 64-bit integers to our memory. This leads to an average time of 0.22 seconds, which is significantly slower than our goal. A method with this time is not able to transfer enough continuous data to induce an error.

Vector v4si

In the next step we used vector instructions to improve the data transfer. The first attempt was via a compiler based method called v4si [Con20] . This is a feature of *GCC*, which enables vector extensions. With this multiple variables can be combined more efficiently together. In its basic usage it can be used to group 4 integers together to a single vector on which then more efficient operations can be performed. Generally we can define with v4si a vector of a given byte size, which then consists of multiple integer sized parts.
For our problem we will group 16 integers together to a single vector and then write the 64 byte vector to the memory. Hereby, we are able to improve the time to 0.139 seconds, which, while significantly faster, is not fast enough for our goals. On an assembly level this is implemented by using multiple MOVQ instructions, which are efficiently chained together. Each MOVQ operation represents a write for a 64 bit word. They are then further packed by the MCU. Noteworthy hereby is that the vectors we defined are only used by the compiler to optimize the code. Inside the assembly code our defined vectors are nowhere to be found.

AVX

An alternative approach is via the single instruction multiple data (SIMD) instruction set extensions for the x86 architecture. The first extensions were the Streaming SIMD Extensions (SSE), which can operate on up to 128 bits. For our purposes it would be optimal to have instructions for 512 bits. By using these extensions we hope to work more efficiently than using 64 bit sized integer. We use a later version of SSE called AVX (Advanced Vector Extensions) [Int20b]. AVX2 supports 256 bits and AVX-512 512 bits. But AVX requires support from the CPU. Current CPUs support AVX2, but only some support AVX-512.
This approach leads to the same timings of 0.14 seconds when compared to v4si. On the assembly level however different instructions are used. Here VMOVDQA is used, which depending on the system, can move 128, 256 or 512 bits. All our systems only supported AVX2, i.e. every instruction uses 256 bits. Therefore, there may be some remaining speed-up if AVX-512 would be used. But as we will see in the next part, it is not a necessary requirement to get the optimal runtime.

AVX non-temporal

Regardless of the actual instructions used, it is impossible to transfer the data significantly faster. A single write request by itself does not only contain a WRITE request, but also a READ request. This is normally done to preserve the integrity of the data, so that nothing is lost if only partial information is written. If less than 64 bytes are written to the memory, the content of the remainder would be overwritten, because a single request always transfers 64 bytes of data. Therefore, to preserve the integrity the whole page is read and then the updated parts are changed and then written back. Additionally, the value can, depending on the cache replacement policies used, be kept in the cache. Generally, that is a good feature, but if a large amount of data is written to different addresses, we get many unnecessary reads and possible useful data is replaced inside the cache. For our means the additional reads are not helpful, because we can't control each transferred bit individually easily. It is still possible to influence the transferred data by splitting our pattern in 64 byte blocks. Every second block has to be written to memory beforehand. This by itself would not make the data transfer impossible only more complicated. But what really makes it unviable are the delays which exist between READ and WRITE. The delays are caused by the necessity to change the direction of the bus. Thereby we will get some cycles where no data can be transferred, which makes a continuous data transfer impossible.
But there is another possibility to just simply not read the data with every write request. AVX supports so called non-temporal instructions. A non-temporal instruction ignores any cache-consistency rules and writes the result directly to the memory. The name itself means, that data is produced but will not be used in the near future, therefore there is no need to temporarily cache it.

To use these instructions directly in C as an AVX instruction it is necessary to use four `_mm_stream_si128` instructions as described by [Dre07]. All addresses have to be 8 or 16 byte aligned and need to happen shortly after another, so that the write-combining buffer can be utilized to construct one 64 byte block. The change is also reflected inside the assembly code. Instead of VMOVDQA now the non-temporal version VMOVNTDQA is used. This also leads to a better run time of 0.076 seconds which nearly halves the required time. The reduction is expected, because we removed all the read requests.

Further Optimizations

The usage of non-temporal instructions, was necessary to get a controllable bus, but we can make it even better. The assembly code produced by the aforementioned statements is not as efficient as possible. A solution would be to write the critical sections directly in inline assembly, but our tests have shown that it is sufficient to let the compiler optimize it by itself. In *C* with *GCC* this would be done by using the parameter `-O1`. Hereby we can further improve the run time to 0.058 seconds. This will be the last optimization step which can be done inside the code itself.

If we analyse the transferred data directly via an oscilloscope we see that the data is transferred in a continuous pattern. We will prove that later in section 4.4. For the last optimization we need to remove or delay the row refreshes. The more we delay the row refreshes the closer we get to the optimal time. If we quarter the occurrence the run time drops to 0.052 seconds, and if we could fully disable it, we would be able to reach our goal of 0.05 seconds.

READ vs. WRITE

Our previous optimizations assumed, that we use WRITE requests, but it is possible to change the used instructions to enable READ request. Generally there is only one key difference which has to be noted, read instructions do not have any kind of non-temporal operations associated with them. It is sufficient to use normal ones. This seems to make the preparations for an attack more easy, but it makes the whole attack longer, because we still need to write the correct data to the memory before. Furthermore, it makes checking for errors much more difficult. While we can take a long time to check the errors, which have been written to memory, errors induced by reading are temporal and need to be either checked immediately or need to be stored. Both of these need to be done very efficiently to not interfere with the read of the next word. Because the requirements for READ requests seem difficult, we proceed to use WRITE requests for our tests. There may be other advantages to using READ in the future, which will come up in section 4.6. But for our purposes WRITE seems to be the best.

4.3 Data Bus Inversion

Another feature supported by DRAM and used by the MCU is Data Bus Inversion (DBI) [LLJ18]. On most buses it is more expensive to switch between 0 and 1 than to keep the power constant. Therefore, DBI is used to minimize the number of switches necessary. In

	without DBI	with DBI
v_1	0101 1100	0101 1100
v_2	1010 0001	1010 0001
$hamming(v_1, v_2)$	7	7
$\hat{v_2}$	1010 0001	0101 1110
$hamming(v_1, \hat{v_2})$	7	1

Figure 4.5: Example of Data Bus Inversion on 8 bits

its most simple form the reduction of switching can be done via the hamming-distance. A visualization of this is shown in figure 4.5. Let us have two data vectors v_1 and v_2 of length n. The first vector v_1 will be transferred normally, but for the second one, the hamming-distance of v_1 and v_2 is calculated. If $hamming(v_1, v_2)$ is greater than $n/2$ it means that more than half the bits would be flipped. But if we invert v_2, less than $n/2$ bits are flipped. If this method is used, at most $n/2$ of the transferred bits are flipped and the average power consumption goes down. But an additional bit is needed to be sent to sign if the data is inverted or not. If the corresponding bit is activated while the data is sent, it is inverted in the memory again to write the normal data again. Therefore, the data can only be inverted while it is transmitted.

Whereas this was a trivial example, some variation of it is used on most DRAM. A typical version would be to calculate the hamming-distances for each byte of data and then invert each byte part of the 8 transferred byte individually.
For our attack using DBI means that we can't send our error inducing codes on all lanes simultaneously. We would send the same streams of zeroes and ones in each lane. If DBI is used on such patterns, the zeroes or the ones would be inverted, so that only zeroes or ones are always sent. Only the inversion bit would flip and mimic our pattern. While it may be possible to induce errors on the inversion bit, there also may be other measures in place so that the failure of a single lane does not invalidate the whole transmission. Therefore, the easiest solution would be to send our pattern on only one or very few lanes. Hereby we keep the amount of change bits inside on burst so small, that the DBI is not used.

4.4 Verification of Transferred Data

With the described methods we are able to efficiently send data, but we have no guarantee that the desired data is actually transmitted. We only know that all the data is transferred in a nearly optimal time window. There may still be some mechanisms left which we have not considered. To remove any remaining doubt, we measured the transferred bits via an oscilloscope to see if we actually transfer the expected bits with the expected frequency

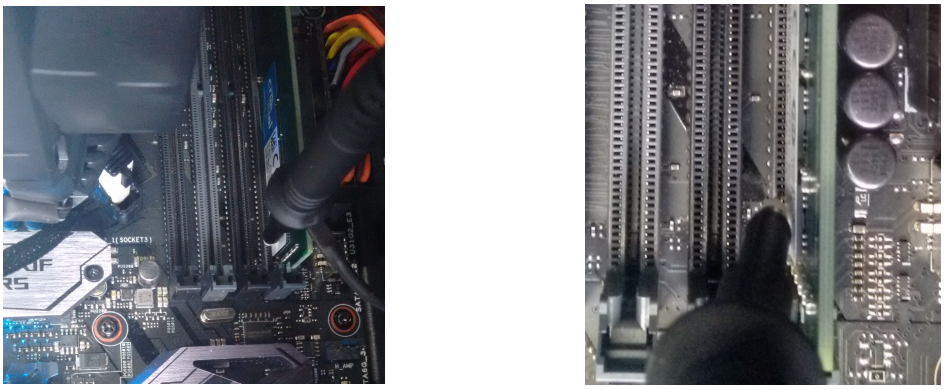


Figure 4.6: Measuring bits with an EM probe

in the expected order. To have better and more detailed results we under-clocked the DRAM to 800 Mhz via the BIOS. We also disabled the Memory Scrambler. All following measurements have been taken with this frequency. More precisely we used an EM-probe which we connected to a single bit of the DRAM and grounded it on the motherboard as seen in figure 4.6. We measure the change in voltage on a single pin to see if we can reconstruct the bits itself. For that reason we send the data pattern over a single DQ pin, while keeping all other ones constant.

In figure 4.7 we can see the result of one simple data transfer. Here we sent a recurring string of 01 to the memory, and we can observe the transmitted bits individually. Also, we can see that it takes about 1.16 ns to transfer a single bit. This is slightly faster than the stated frequency of 800 Mhz, but still within reason. Additionally, we can see that the bits we want to transfer are actually transferred in the specified pattern and that on this low level no further delays exist: Therefore, there is no delay remaining between different WRITE requests. All instructions are properly pipelined. If we remember from section 4.1 a single WRITE request would consist of 8 bursts over 4 cycles, which is shorter than the observed time frame.

In figure 4.8 we see the same data in a lower resolution. We see that the data we want to transfer is generally transmitted, but sometimes there will be short transfers which we can not control, because they are issued by the system. Therefore, it is necessary to conduct all following tests under low load on the system. The most prominent feature are the recurring periodic pauses, where nothing from our pattern is transmitted. After the modulation of different timings inside the BIOS, these interrupts turned out to be the row refreshes, the command from the MCU to refresh every single row inside the memory. During the row refreshes no other data can be transmitted. Inside the BIOS we can control the frequency of it occurring, but we can not and should not disable the row refreshes.

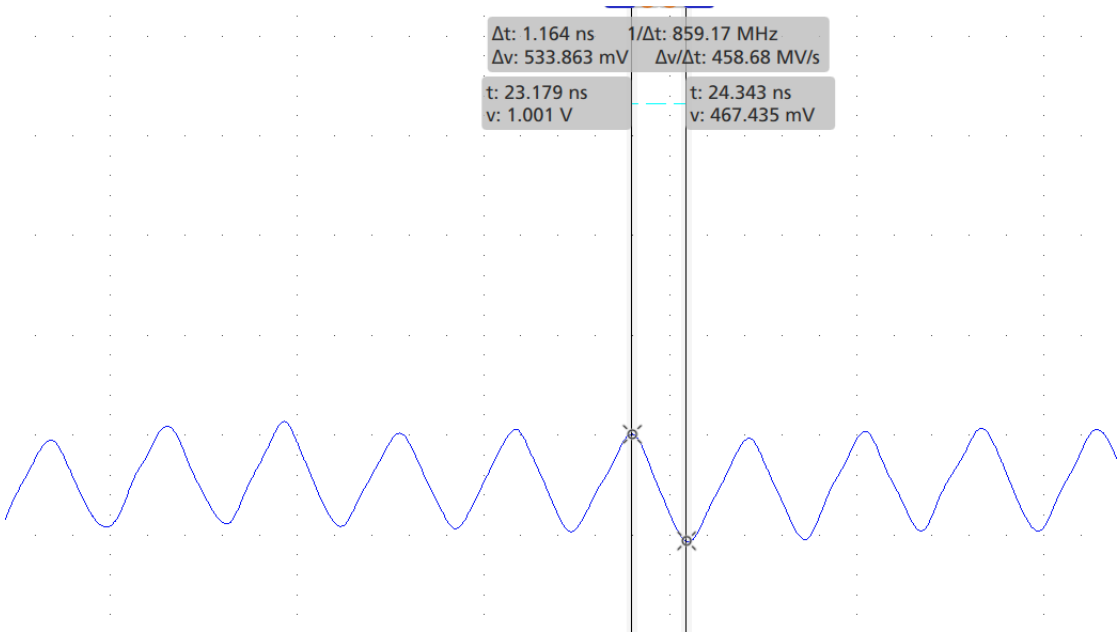


Figure 4.7: Sending patterns of 01 in high resolution

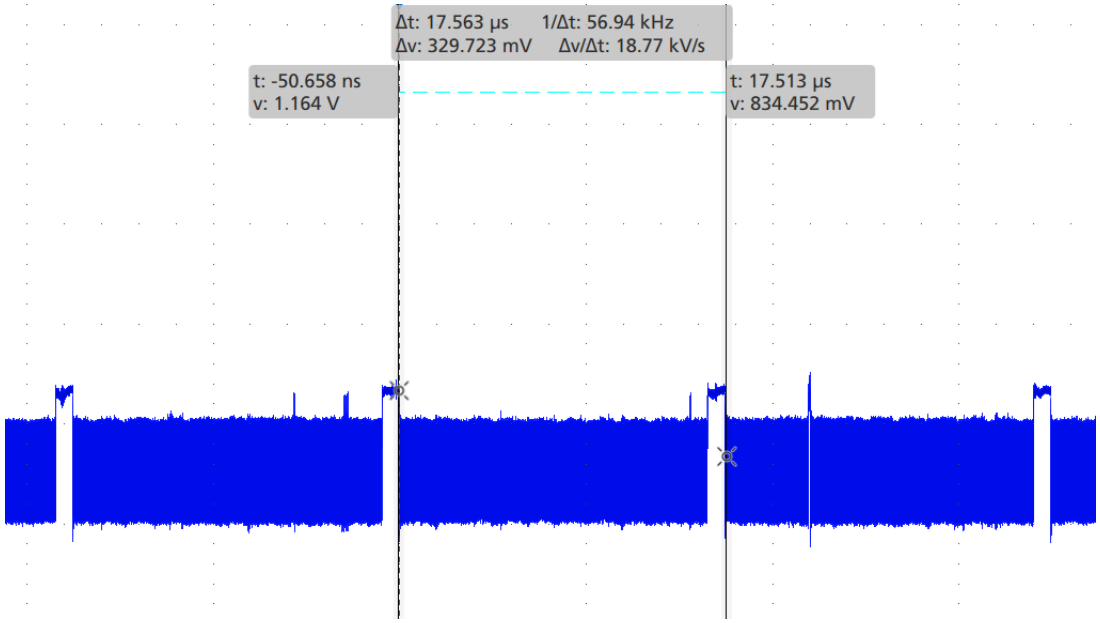
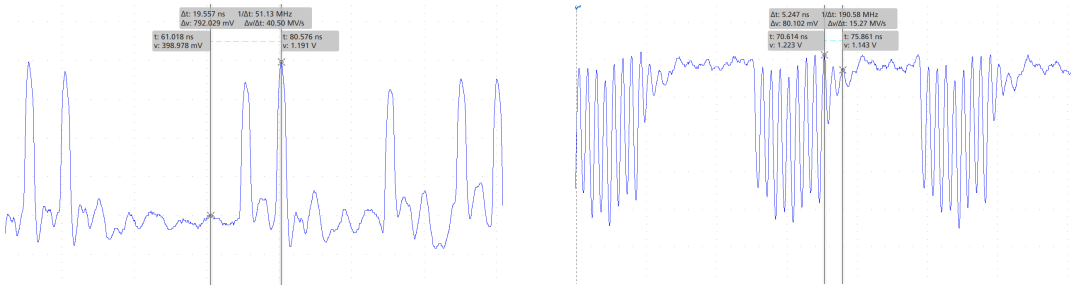


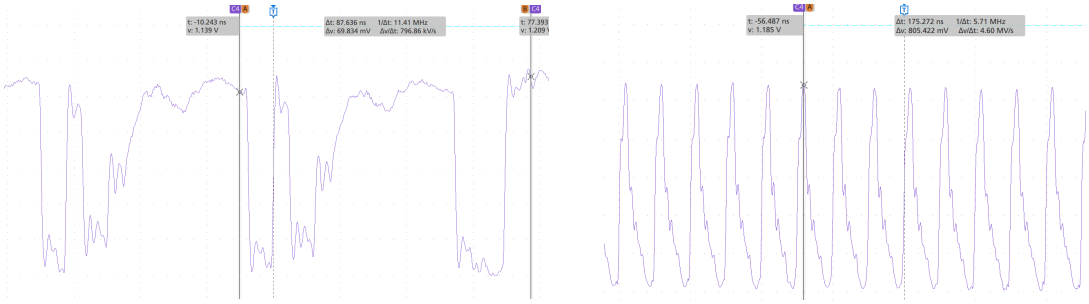
Figure 4.8: Sending patterns of 01 in low resolution



(a) Transfer without memory barriers of pattern 10000000 00000000

(b) Transfer with memory barriers of pattern 01010101

Figure 4.9: Images of data transfer with and without memory barrier



(a) Writing with memory barriers after 2 bursts

(b) Reading without memory barriers

Figure 4.10: Images of data transfer of pattern 00000011 11000000

Even so we can turn its frequency down and try to transfer as many data patterns as possible inside the time frame. On our system the default time between 2 row refreshes was about 15 ns, which leads to 13000 transfers between them. The number of transfers should be sufficient to induce an error, but it can also easily be increased.

But if we want to transfer different data, more precisely data, which is non-periodic over 8 cycles, we get vastly different results, as seen in figure 4.9a. On a first look the now transferred data may seem random and without any structure, but under further analysis it can be seen that our WRITE requests are executed out of order. Each stream of 8 bits belongs to one 8 bit stream which we actually want to transfer, in our example 10000000 or 00000000, but the order between them is mixed up. If we only wanted to execute the WRITE request in the given order, that could be easily achieved with memory barriers, in this case `sfence` would be the best one. This barrier means that all store instructions before it have to actually be completed before the store instructions after the barrier. There exists an equivalent one for READ request `lfence`, or for both `mfence`. But we choose the most specific one, because the more general ones take more time to ensure. Therefore,

we tried to use a memory barrier after every single WRITE request. Hereby, we are able to remove the out-of-order executions, but as seen in figure 4.9b their introduction leads to heavy delay after every WRITE. With so many interruptions the induction of errors seems impossible to achieve.
To alleviate the effect we use memory barriers less frequently, but even if only two WRITE requests are used consecutively, the order of them may be swapped, as seen in figure 4.10a. This makes it much more difficult to find any kinds of errors, because the desired pattern is only sent every so often, and we would need it to be repeated as many times as possible to get the desired effect. Therefore, the following tests only have reliable results for writing 8 bit patterns. Finally, we sent the same pattern via READ requests. Here the different requests are not reordered as shown in figure 4.10b.
As long as only 8 bit patterns are sent via WRITE requests we can now be sure that the corresponding data is actually transferred in the desired order without any delays. But if we sent any other pattern we can't guarantee the order. It is still possible to do a probabilistic test with only a low chance of success.

4.5 Results

With the preliminary preparations done, we can now try to induce errors by transmitting specific patterns. For that we will try two different methods. The first one is an exhaustive search over all 8 bit patterns and the second one will only test specific repetitions of zeroes and ones.
The first test can be reliably done with our methods. We rotate through all possible 256 different patterns and write them to our 1 GB memory region. This process will be repeated multiple times for each pattern. After every single run we check our memory against the transferred pattern for an occurring error. After our previous observations we can guarantee, that the pattern is actually sent via the specified lane. We varied the lanes over which the pattern is sent and also tried to send the pattern over multiple lanes at the same time, but none of our attempts lead to an error inside the memory.
For the second test we send patterns of the form $(0^a1^b)^*$. A single run iterates over all possible a and b up to an upper range and sends the specified pattern to the memory. Hereby we also could not induce an error, but after our previous observations, we can't be sure that the patterns are transmitted as we hope they are. On our system the order of the WRITE requests can not be guaranteed. Therefore, we try to alleviate this by repeating the patterns more often to increase the likelihood that we transmit our pattern, but none of our attempts did ever induce an error.
We are able to send arbitrary patterns with READ requests. Due to time constraints we

were not able to verify, if we can induce errors with READ requests. Therefore, it remains open if temporal errors can be induced by some pattern.

4.6 Into the future: ECC and CRC

There are some optional features in DDR4, which could make this kind of attack more difficult in the future: Error Correcting Code (ECC) and Cyclic Redundancy Check (CRC). ECC is currently only used on some DIMMS. If they are used in each burst of 64 bits an additional 8 bits are sent which can be used to correct one and identify two bit errors. In our attack we have only sent data via a single lane. If that leads to errors, every one of them could be corrected. Therefore, it is necessary to send the data via multiple lanes, s.t. they can no longer be identified by the ECC. But it is not possible to use all lanes freely because some patterns may be changed by the DBI. In any case it would reduce the probability for a successful bit-flip.
DDR4 memory itself supports CRC, but it is not used on most current systems. Using it would lead to 25% longer transfer times, which is only necessary for critical systems. But what is interesting for us is how CRC can impact our attack. After every 8 bursts we send, one burst which contains the CRC checksum. Then another 0 bit burst is sent to realign with the clock. The first burst can be manipulated by changing bits in different lanes, and since the CRC-function is known, we can easily change its value in our lane to the one we want. But the second 0 bit burst is constant and can't be changed. In this context it may be interesting to take a look at the READ instead of the WRITE because the CRC in DDR4 is only supported for WRITE, but never for any READ. All that is still theoretically since none of our systems supported CRC or ECC on DDR4. Nonetheless, it may be something to look further upon in the nearer future.

4.7 Evaluation

We have discussed how we can control the bits sent over the lanes to the DIMM. The lanes which we can control are the data lanes where we can send data on both flanks of each clock only seldom discontinued by the Row Refreshes, which theoretically can be delayed. To send the data we use non-temporal instructions to skip the cache while writing to the memory. Hereby we are able to write arbitrarily periodic 8 bit patterns to the memory. If we want to use longer patterns we identified the problem, that the WRITE requests can and will be reordered at random. That makes in unfeasible to use WRITE requests to induce errors. It is stated in [Int20a] that the WRITE and READ request reordering is generally not used on modern processors, but this does specifically not apply to non-temporal instructions, which can be freely reordered. We were only able to verify that

READ requests are indeed not reordered. Therefore, our tests remain inconclusive if it is possible to induce errors by sending arbitrary data over the memory bus.

5 Conclusions

5.1 Summary

At the beginning of this thesis we have seen that the memory of a system was always a vulnerable point. There have been many kinds of attacks like Cold Boot Attacks and the Rowhammer Bug. Even on DDR4 new ways to compromise it were found. The newest one was TRRespass to side step the target row refresh, a feature introduced to eliminate Rowhammer bit-flips. We have discussed how the Memory Scrambler worked on DDR3 and that the same methods no longer work here. Also, we analysed the by Intel stated information regarding the Memory Scrambler and tried to verify which of the obfuscation features may be used. By disabling the Memory Scrambler inside the BIOS and writing data specifically to hugepages we have found a reliable way to get many pairs of scramble codes and address pairs. On this scramble code we were able to improve the equations. Herby it is possible to identify scramble codes easier or to reconstruct the scramble code with fewer bits. But we were not able to fully reconstruct the Memory Scrambler. In the last part we tried to send data as efficiently as possible to the memory. For this we used non-temporal operations. We have seen that we can write arbitrary 8 bit patterns to the memory, but if we use other patterns, we arrived at the problem, that our WRITE requests can be reordered. This makes the process of inducing errors via non-temporal operations unfeasible. It remains to be seen if the same holds for READ requests, which should not be reordered. We also discussed, that ECC would make any such attack much more difficult and that CRC would make the attack with our knowledge impossible.

5.2 Discussion and open problems

With our methods we were able to slightly improve the existing test for the scramble code. But we are not able to fully reverse engineer the functionality of the Memory Scrambler and recover the values of the LFSR. Therefore, we assume that a non-linear substitution or another unknown obfuscation method is used. To solve the non-linear substitution more sample data and more efficient methods are necessary.
Regarding the transfer of data we were not able to fully analyse, if it's possible to induce errors by reading from the memory. We have shown, that it is unfeasible by using WRITE

requests, because we either get undesired interruptions or the desired patterns are only sent with low probability. But it remains to be shown if it is possible to induce errors with READ requests. The restrictions for WRITE requests do not hold. Therefore, it is possible to send arbitrary patterns. Nonetheless, it is more difficult to check for errors in the only temporary data streams. In the future any possible induction of errors via the data lanes could be made even more difficult if CRC would become commonly used or is introduced not only on WRITE, but also on READ requests.

References

[Ee20] Ee Loon Teoh, Eng Hun Ooi, Christopher Mozak, Brian McFarlane. Lower-power scrambling with improved signal integrity. https://patentimages.storage.googleapis.com/bb/2e/72/b5554494c8e3e7/US20160188523A1.pdf, June 2016 accessed 27. Oct, 2020. accessed 14. October 2020.

[Ma16] Mark Lanteigne. How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware. http://www.thirdio.com/rowhammer, March 2016. accessed 8. October 2020.

[Alb12] Martin R. Albrecht. The M4RIE library for dense linear algebra over small fields with even characteristic. In *International Symposium on Symbolic and Algebraic Computation, ISSAC'12, Grenoble, France - July 22 - 25, 2012*, pages 28–34. ACM, 2012.

[Bar09] Gregory V. Bard. *The Method of Four Russians*, pages 133–158. Springer US, Boston, MA, 2009.

[BGF16] Johannes Bauer, Michael Gruhn, and Felix Freiling. Lest we forget: Cold-boot attacks on scrambled DDR3 memory. *Digital Investigation*, 16:S65–S74, 03 2016.

[Cok17] Russel Coker. Hugepages. https://wiki.debian.org/Hugepages, 19.06.2017. accessed 20. October 2020.

[Con20] GCC Contributors. Using Vector Instructions through Built-in Functions — GCC Online Documentation. https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html, accessed 13. Oct, 2020.

[Dev13] Advanced Micro Devices. BIOS and Kernel Developer's Guide(BKDG)for AMD Family 15h Models 00h-0Fh Processors, 2013. accessed 20. October 2020.

[Dre07] Ulrich Drepper. What Every Programmer Should Know About Memory. https://www.akkadia.org/drepper/cpumemory.pdf, 21.11.2007.

[FVH+20] 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, 2020.

[GLS+18] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. Another flip in the wall of rowhammer defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261, May 2018.

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, page 279–299, Berlin, Heidelberg, 2016. Springer-Verlag.

[HSH+09] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, May 2009.

[IIES14] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-vm attack on aes. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, pages 299–319, Cham, 2014. Springer International Publishing.

[Inc18] Micron Technology Inc. Ddr4 sdram datasheet, 2018. accessed 20. October 2020.

[Int20a] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3, Chapter 8, Section 8.2.2 "Memory Ordering in P6 and More Recent Processor Families". `https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html`, 27.05.2020.

[Int20b] Intel. Overview: Intrinsics for Intel® Advanced Vector Extensions 2 (Intel® AVX2) Instructions — Intel® C++ Compiler 19.1 Developer Guide and Reference. `https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions-2/overview-`

intrinsics-for-intel-advanced-vector-extensions-2-intel-avx2-instructions.html, accessed 13. Oct, 2020.

[JED05] JEDEC Solid State Technology Association. Double Data Rate (DDR) SDRAM Specification. http://cs.ecs.baylor.edu/~maurer/CSI5338/JEDEC79R2.pdf, May 2005. accessed 13. August 2020.

[JNW07] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[KDK+14] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, June 2014.

[LJF+19] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia che Tsai, and Raluca A. Popa. An off-chip attack on hardware enclaves via the memory bus. *ArXiv*, abs/1912.01701, 2019.

[LLJ18] J. Lucas, S. Lal, and B. Juurlink. Optimal dc/ac data bus inversion coding. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1063–1068, March 2018.

[MMK11] Praveen Mosalikanti, Christopher Mozak, and Nasser Kurd. High performance DDR architecture in Intel® Core™ processors using 32nm CMOS high-K metal-gate process. pages 1–4, 04 2011.

[Per05] Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[PGM+16] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, Austin, TX, August 2016. USENIX Association.

[Sea20a] M. Seaborn. Exploiting the dram rowhammer bug to gain kernel privileges. https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf, March 2015 accessed 20. Jan, 2020. accessed 20. October 2020.

[Sea20b] M. Seaborn. How physical addresses map to rows an banks in dram. `http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html`, May 2015 accessed 20. Jan, 2020. accessed 20. October 2020.

[SJC+17] W. Shin, J. Jang, J. Choi, J. Suh, and L. Kim. Bank-group level parallelism. *IEEE Transactions on Computers*, 66(08):1428–1434, aug 2017.

[WJ05] David Tawei Wang and Bruce L. Jacob. *Modern Dram Memory Systems: Performance Analysis and Scheduling Algorithm*. PhD thesis, USA, 2005. AAI3178628.

[YADA17] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin. Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 313–324, Feb 2017.

[YF14] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 719–732, USA, 2014. USENIX Association.

Appendices

A Equations for LFSR matrix

As follows are the columns of a LFSR matrix of size 5:

- $(c_0, c_1, c_2, c_3, c_4)$
- $(c_0 + c_1, c_0 \cdot c_1 + c_2, c_0 \cdot c_2 + c_3, c_0 \cdot c_3 + c_4, c_0 \cdot c_4)$
- $(c_0 + c_2, c_0 \cdot c_1 + c_0 \cdot c_2 + c_1 + c_3, c_0 \cdot c_2 + c_0 \cdot c_3 + c_1 \cdot c_2 + c_4, c_0 \cdot c_3 + c_0 \cdot c_4 + c_1 \cdot c_3, c_0 \cdot c_4 + c_1 \cdot c_4)$
- $(c_0 \cdot c_1 + c_0 + c_1 + c_3, c_0 \cdot c_1 + c_0 \cdot c_2 + c_0 \cdot c_3 + c_4, c_0 \cdot c_2 + c_0 \cdot c_3 + c_0 \cdot c_4 + c_1 \cdot c_3 + c_2, c_0 \cdot c_3 + c_0 \cdot c_4 + c_1 \cdot c_4 + c_2 \cdot c_3, c_0 \cdot c_4 + c_2 \cdot c_4)$
- $(c_0 \cdot c_1 + c_0 \cdot c_2 + c_0 + c_4, c_0 \cdot c_2 + c_0 \cdot c_3 + c_0 \cdot c_4 + c_1 + c_2, c_0 \cdot c_1 \cdot c_2 + c_0 \cdot c_2 + c_0 \cdot c_3 + c_0 \cdot c_4 + c_1 \cdot c_2 + c_1 \cdot c_4, c_0 \cdot c_1 \cdot c_3 + c_0 \cdot c_3 + c_0 \cdot c_4 + c_1 \cdot c_3 + c_2 \cdot c_4 + c_3, c_0 \cdot c_1 \cdot c_4 + c_0 \cdot c_4 + c_1 \cdot c_4 + c_3 \cdot c_4)$

B Pin Layout

Adapted from [Inc18].

Pin	Symbol	Pin	Symbol	Pin	Symbol	Pin	Symbol	Pin	Symbol
1	NC	2	V_{SS}	3	DQ4	4	V_{SS}	5	DQ0
6	V_{SS}	7	DQS9_t	8	DQS09_c	9	V_{SS}	10	DQ6
11	V_{SS}	12	DQ2	13	V_{SS}	14	DQ12	15	V_{SS}
16	DQ8	17	V_{SS}	18	DQS10_t	19	DQS10_c	10	V_{SS}
21	DQ14	22	V_{SS}	23	DQ10	24	V_{SS}	25	DQ20
26	V_{SS}	27	DQ18	28	V_{SS}	29	DQS11_t	29	DQS11_c
31	V_{SS}	32	DQ22	33	V_{SS}	34	DQ18	35	V_{SS}
36	DQ28	37	V_{SS}	38	DQ24	39	V_{SS}	40	DQS12_t
41	DQS12_c	42	V_{SS}	43	DQ30	44	V_{SS}	45	DQ26
46	V_{SS}	47	CB4	48	V_{SS}	49	CB0	50	V_{SS}
51	DQS17_t	52	DQS17_c	53	V_{SS}	54	CB6	55	V_{SS}
56	CB2	57	V_{SS}	58	RESET_n	59	V_{DD}	60	CKE0
61	V_{DD}	62	ACT_n	63	BG0	64	V_{DD}	65	A12/BC_n
66	A9	67	V_{DD}	68	A8	69	A6	70	V_{DD}
71	A3	72	A1	73	V_{DD}	74	CK0_t	75	CK0_c
76	V_{DD}	77	V_{TT}	78	EVENT_n	79	A0	80	V_{DD}
81	BA0	82	RAS_n/A16	83	V_{DD}	84	CS0_n	85	V_{DD}
86	CAS_n/A15	87	ODT0	88	V_{DD}	89	CS1_n/NC	90	V_{DD}
91	ODT1/NC	92	V_{DD}	93	CS2_n/C0	94	V_{SS}	95	DQ36
96	V_{SS}	97	DQ32	98	V_{SS}	99	DQS13_t	100	DQS13_c
101	V_{SS}	102	DQ38	103	V_{SS}	104	DQ34	105	V_{SS}
106	DQ44	107	V_{SS}	108	DQ40	109	V_{SS}	110	DQS14_t
111	DQS14_c	112	V_{SS}	113	DQ46	114	V_{SS}	115	DQ42
116	V_{SS}	117	DQ52	118	V_{SS}	119	DQ48	120	V_{SS}
121	DQS15_t	122	DQS15_c	123	V_{SS}	124	DQ54	125	V_{SS}
126	DQ50	127	V_{SS}	128	DQ60	129	V_{SS}	130	DQ56
131	V_{SS}	132	DQS16_t	133	DQS16_c	134	V_{SS}	135	DQ62
136	V_{SS}	137	DQ58	138	V_{SS}	139	SA0	140	SA1
141	SCL	142	V_{pp}	143	V_{pp}	144	NC	145	NC

Pin	Symbol	Pin	Symbol	Pin	Symbol	Pin	Symbol	Pin	Symbol
146	V_{REFCA}	147	V_{SS}	148	DQ5	149	V_{SS}	150	DQ01
151	V_{SS}	152	DQS0_c	153	DQS0_t	154	V_{SS}	155	DQ7
156	V_{SS}	157	D32	158	V_{SS}	159	DQ13	160	V_{SS}
161	DQ9	162	V_{SS}	163	DQS1_c	164	DQS1_t	165	V_{SS}
166	DQ15	167	V_{SS}	168	DQ11	169	V_{SS}	170	DQ21
171	V_{SS}	172	DQ17	173	V_{SS}	174	DQS2_c	175	DQS2_t
176	V_{SS}	177	DQ23	178	V_{SS}	179	DQ19	180	V_{SS}
181	DQ29	182	V_{SS}	183	DQ25	184	V_{SS}	185	DQS3_c
186	DQS3_t	187	V_{SS}	188	DQ31	189	V_{SS}	190	DQ27
191	V_{SS}	192	CB5	193	V_{SS}	194	CB1	195	V_{SS}
196	DQS8_c	197	DQS8_t	198	V_{SS}	199	CB7	200	V_{SS}
201	CB3	202	V_{SS}	203	CKE1/NC	204	V_{DD}	205	NC
206	V_{DD}	207	BG1	208	ALERT_n	209	V_{DD}	210	A11
211	A7	212	V_{DD}	213	A5	214	A4	215	V_{DD}
216	A2	217	V_{DD}	218	CK1_c	219	CK1_t	220	V_{DD}
221	V_{TT}	222	PARITY	223	V_{DD}	224	BA1	225	A10/AP
226	V_{DD}	227	NC	228	WE_n/A14	229	V_{DD}	230	NC
231	V_{DD}	232	A13	233	V_{DD}	234	A17	235	NC/C2
236	V_{DD}	237	CS3_n/C1,NC	238	SA2	239	V_{SS}	240	DQ37
241	V_{SS}	242	DQ33	243	V_{SS}	244	DQS4_c	245	DQS4_t
246	V_{SS}	247	DQ39	248	V_{SS}	249	DQ35	250	V_{SS}
251	DQ45	252	V_{SS}	253	DQ41	254	V_{SS}	255	DQS5_c
256	DQS5_t	257	V_{SS}	258	DQ47	259	V_{SS}	260	DQ43
261	V_{SS}	262	DQ53	263	V_{SS}	264	DQ49	265	V_{SS}
266	DQS6_c	267	DQS6_t	268	V_{SS}	269	DQ55	270	V_{SS}
271	DQ51	272	V_{SS}	273	DQ61	274	V_{SS}	275	DQ57
276	V_{SS}	277	DQS7_c	278	DQS7_t	279	V_{SS}	280	DQ63
281	V_{SS}	282	DQ59	283	V_{SS}	284	V_{DDSPD}	285	SDA
286	V_{pp}	287	V_{pp}	288	V_{pp}				

C Pin Descriptions

Adapted from [Inc18].

Symbol	Description
Ax	Address Inputs. Row addresses for Activate and column addresses for READ/WRITE. Some of this pins have additional functions
A10/AP	Auto precharge. Determines if an automatic precharge should be performed.
A12/BC_n	Burst Chop.
ACT_n	Command input.
BAx	Bank address.
BGx	Bank group address.
Cx	Chip ID.
CKx_t/c	Clock.
CKEx	Clock enable.
CSx_n	Chip select.
ODTx	On-die termination.
PARITY	Parity for command and address.
RAS_n/A16 CAS_n/A15 WE_n/A14	Command inputs
SAx	Serial address inputs.
SCL	Serial clock for temperature sensor.
DQx, CBx	Data and check bit
DM_n,DBI_n,TDQS_t	Input data mask and data bus inversion.
SDA	Serial Data.
DQS_t,DQS_c	Data strobe.
ALERT_n	Alert output.
EVENT_n	Temperature event.
TDQS_t,TDQS_c	Termination Data Strobe
V_{DD}	Module power supply.
V_{pp}	DRAM activating power supply.
V_{REFCA}	Reference voltage.

Symbol	Description
V_{SS}	Ground.
V_{TT}	Power supply for termination.
V_{DDSPD}	Power supply for I^2C.
RFU	Reserved for future use.
NC	Not connected.
NF	No function.

D SDRAM Timing Parameters

Adapted from [JNW07].

Parameter	Description
t_{AL}	Additive Latency allows a WRITE/READ command to be used immediately after an ACTIVATE command, instead it is held inside the device for the t_{AL} duration.
t_{BURST}	The time period needed for one burst of data.
t_{CCD_L}	Bank accesses to a different bank in the same bank group requires t_{CCD_L} delay between commands.
t_{CCD_S}	Bank accesses to a different bank in a different bank group requires t_{CCD_S} delay between commands.
t_{CL}	Column-Address-Strobe Latency is the delay between the internal READ and the start of the output.
t_{CWL}	Column-Address-Strobe Latency is the delay between the internal WRITE and the start of the output.
t_{DQSCK}	Describes the position of the the data strobe relative to the Clock.
t_{FAW}	Four Activate Window describes the time frame in which at most four ACTIVATE commands can be used.
t_{MRD}	Describes the minimal time required to be complete between two different Mode Register Set commands.
t_{MOD}	Describes the minimal time required between a MRS command and a non MRS command.
t_{OST}	Describes the time necessary to switch ODT control from rank ot rank
t_{RAS}	Row Access Strobe. Describes the time frame after an ACTIVATE command while no PRECHARGE can be issued.
t_{RC}	Row cycle. $t_{RC} = t_{RAS} + t_{RP}$
t_{RCD}	Row address to column address delay time, the time necessary between the transmission of the row address and the data being at the sense amplifiers.
t_{REFI}	The average time between two REFRESH commands.
t_{RFC}	Delay between REFRESH and the next command.
t_{RP}	Describes the time necessary to precharge and close a row and to open the next one.

Parameter	Description
t_{RRD_L}	Describes the delay necessary between two ACTIVATE commands to banks of the same bank groups.
t_{RRD_S}	Describes the delay necessary between two ACTIVATE commands to banks of different bank groups.
t_{RTP}	Describes the interval between a READ and a PRECHARGE command.
t_{RTRS}	Describes the interval necessary to switch ranks.
t_{WL}	Write Latency. $t_{WL} = t_{CWL} + t_{AL}$
t_{WR}	Write Recovery, time interval between the end of a WRITE burst and the PRECHARGE.
t_{WTR}	Write to Read delay, describes the time between the end of a WRITE burst and the start of a READ.