Curious case of Rowhammer: Flipping Secret Exponent Bits using Timing Analysis

Sarani Bhattacharya[1] and Debdeep Mukhopadhyay[1]

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Kharagpur-721302, India
Contact E-mail: {sarani.bhattacharya, debdeep}@cse.iitkgp.ernet.in

Abstract. Rowhammer attacks have exposed a serious vulnerability in modern DRAM chips to induce bit flips in data which is stored in memory. In this paper, we develop a methodology to combine timing analysis to perform the hammering in a controlled manner to create bit flips in cryptographic keys which are stored in memory. The attack would require only user level privilege for Linux kernel versions before 4.0 and is unaware of the memory location of the key. An intelligent combination of timing *Prime + Probe* attack and *row-buffer collision* is shown to induce bit flip faults in a 1024 bit RSA key on modern processors using realistic number of hammering attempts. This demonstrates the feasibility of fault analysis of ciphers using purely software means on commercial x86 architectures, which to the best of our knowledge has not been reported earlier. The attack is also relevant for the newest Linux kernel in a Cross-VM environment where the VMs having root privilege are not denied to access the pagemap.

Keywords: rowhammer, fault attack, *Prime + Probe*, bit flip

1 Introduction

Rowhammer is a term coined for disturbances observed in recent DRAM devices in which repeated row activation causes the DRAM cells to electrically interact among themselves [1–4]. This results in bit flips [2] in DRAM due to discharging of the cells in the adjacent rows. DRAM cells are composed of an access transistor and a capacitor which stores charge to represent a bit. Since capacitors loose their charge with time, DRAM cells require to be refreshed within a fixed interval of time referred to as the refresh interval. DRAM comprises of two dimensional array of cells, where each row of cells have its own wordline and for accessing each row, its respective wordline needs to be activated. Whenever some data is requested, the cells in the corresponding row are copied to a direct-mapped cache termed Row-buffer. If same row is accessed again, then the contents of row-buffer is read without row activation. However, repeatedly activating a row causes cells in adjacent rows to discharge themselves and results in bit flip.

The authors in [2] show that rowhammer vulnerability exists in majority of the recent commodity DRAM chips and is most prevalent for sub $40nm$ memory technology. DRAM process technology over the years have seen continuous reduction in the size of each cell and cell density has also increased to a great extent reducing the cost per bit of memory. Smaller cells hold less charge and this makes them even more vulnerable. Moreover, the increased cell density has a negative impact on DRAM reliability [2] as it introduces electromagnetic coupling effects and leads to such disturbance errors.

In [5], it was first demonstrated that rowhammer not only has issues regarding memory reliability but also are capable of causing serious security breaches. DRAM vulnerability causing bit flips in Native Client (NaCl) program was shown to gain privilege escalation, such that the program control escapes x86 NACL sandbox and acquires the ability to trigger OS syscall. The blog also discusses about how rowhammer can be exploited by user-level programs to gain kernel privileges. In [5], the bit flips are targeted at a page table entry which after flip points to a physical page of the malicious process, thus providing read-write access to cause a kernel privilege escalation. A new approach for triggering rowhammer with x86 non-temporal store instruction is proposed in [6]. The paper discusses an interesting approach by using `libc's memset` and `memcpy` functions and the threats involved in using non-temporal store instructions for triggering rowhammer. A vivid description of the possible attack scenarios exploiting bit flips from rowhammer are also presented in the paper.

A javascript based implementation of the rowhammer attack, Rowhammer.js [7] is implemented using an optimal eviction strategy and claims to exploit rowhammer bug with high accuracy without using `clflush` instruction. Being implemented in javascript, it can induce hardware faults remotely. But, in order to mount a successful fault attack on a system the adversary should have the handle to induce fault in locations such that the effect of that fault is useful in making the attack successful. Another variant of rowhammer exists, termed as double-sided-rowhammer [8]. This variant chooses three adjacent rows at a time, targeting bit blips at the middle row by continuously accessing the outer rows for hammering. The existing implementation [8] is claimed to work in systems having a specific DRAM organization. In all of the existing works, precisely inducing bit flip in the data used by a co-residing process has not been attempted. None of the previous works attempted to demonstrate a practical fault analysis attack using rowhammer. The address mappings to various components of the LLC and DRAM being functions of physical address bits, inducing bit flip in a data residing in an unknown location of DRAM seems to be a challenging task.

In this paper, we illustrate a software driven fault attack on public key exponentiation by inducing a bit flip in the secret exponent. It is well known from [9], theoretically if any fault is induced while public key exponentiation is taking place, then a single faulty signature is enough to leak the secret exponent in an unprotected implementation. However, to inflict the fault using rowhammer on the secret exponent to lead to a usable faulty signature requires further investigation. While [5] was able to successfully induce rowhammer flips in the DRAM to cause a fault in a page table entry, the challenge to induce faults to perform a fault attack on a cipher, requires a better understanding of the location of the secret key in the corresponding row of a bank of the DRAM. More recent developments of rowhammering, like double-sided-rowhammer [8], while increasing the probability of a bit flip, cannot be directly applied to the current scenario, as the row where the secret resides can be in any arbitrary location in the target bank. The chance of the memory location for the secret key lying between the rows of the allocated memory for rowhammer is low. In this scenario a double-sided-rowhammer will reduce the probability of a successful exploitable fault. Hence, it is imperative to ascertain the location of the secret exponent before launching the rowhammer. Our novelty is to combine *Prime + Probe* attack and row-buffer collision, detected again through timing channel, to identify the target bank where the secret resides.

We combine knowledge of reverse engineering of LLC slice and DRAM addressing with timing side-channel to determine the bank in which secret resides. We precisely trigger rowhammer to address in the same bank as the secret. This increases probability of bit flip in the secret exponent and the novelty of our work is that we provide series of steps to improve the controllability of fault induction. The overall idea of the attack involves three major steps. The attacker successfully identifies an *eviction set* which is a set of data elements which maps to the same cache set and slice as that of the secret exponent by timing analysis using *Prime + Probe* methodology. This set essentially behaves as an alternative to `clflush` statement of x86 instruction set. The attacker now observes timing measurements to DRAM access to determine the DRAM bank in which the secret resides in. The variation in timing is observed due to the row-buffer conflict forced by the adversary, inducing bit flips by repeated row activation in the particular bank where the secret is residing. Elements which map to same bank but separate rows are accessed continuously to observe a successful bit flip.

The organization of the paper is as follows:- The following Section 2 provides a brief idea on Cache and DRAM architecture and also the rowhammer bug. In Section 3, we provide the algorithm using timing observations to determine the LLC set, slice and also the DRAM banks in which the secret maps to. Section 4 provides the experimental validation for a successful bit flip in the secret exponent in various steps. Section 5 discusses the existing hardware and software driven countermeasures to the rowhammer problem. Section 6 provides a detailed discussion on the assumptions and limitations of our attack model on bigger range of systems. The final section concludes the work we present here.

2 Preliminaries

In this section, we provide a background on some key-concepts, which include some DRAM details, the rowhammer bug and details of cache architecture which have been subjected to attack.

2.1 Dynamic Random Access Memory

Dynamic Random-Access Memory (DRAM) is a Random-Access Memory in which each unit of charge is stored in a capacitor and is associated with an access transistor, together they constitute a *cell* of DRAM. The DRAM cells are organized in rows and columns. The access transistor is connected to the wordline of a row, which when enabled, connects the capacitor to the bitline of the column and allows reading or writing data to the connected row. The reading

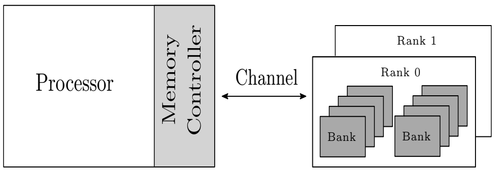


Fig. 1. DRAM Architecture [2]

or writing to cells in a row is done through *row-buffer* which can hold charges for a single row at a time. There are three steps that are performed, when data is requested to be read from a particular row:

Opening Row - The wordline connected to the row is enabled, which allows the capacitors in the entire row to get connected to the bitlines. This results in the charge of each cell to discharge through the bitlines to the row-buffer.

Reading or Writing to cells - The row-buffer data is read or written by the memory controller by accessing respective columns.

Closing Row - The wordline of the respective row is disabled, before some other row is enabled.

DRAM Architecture DRAM is hierarchically composed of Channels, Rank and Banks. The physical link between the memory controller and the DRAM module is termed as *channel.* Inside the channel, the physical memory modules connected to the motherboard are named as *Dual Inline Memory Module (DIMM)* which typically comprises of one or two *ranks* as in Figure 1. Each rank is further comprised of multiple *banks*, as for example, 8 banks exist in a DDR3 rank. Each bank is a two-dimensional collection of cells having typically 2^{14} to 2^{17} rows and a row-buffer. Any row in a particular bank can only be read and written by involving the row-buffer. The latency in DRAM access when two access request concurrently map to same channel, rank, bank but different row is termed as *row-buffer conflict.* The channel, rank, bank and the row index where a data element is going to reside, is decided as functions of physical address of the concerned data element.

The capacitors in each cell of DRAM discharges with time. The capacitor can hold charge upto a specific interval of time before it completely looses its charge. This interval is termed as *retention time*, which is guaranteed to be $64ms$ in DDR3 DRAM specifications [10]. But it is shown, that repeated row activation over a period of time leads to faster discharge of cells in the adjacent rows [2].

2.2 The Rowhammer Bug

Persistent and continuous accesses to DRAM cells, lead the neighboring cells of the accessed cell to electrically interact with each other. The phenomenon of flipping bits in DRAM cells is termed as the *rowhammer* bug [1,2]. As described in Section 2.1, accessing a byte in memory involves transferring data from the row into the bank's row-buffer which also involves discharging the row's cells. Repeated discharging and recharging of the cells of a row can result in leakage of charge in the adjacent rows. If this can be repeated enough times, before automatic refreshes of the adjacent rows (which usually occur every 64ms), this disturbance error in DRAM chip can cause bit flips.

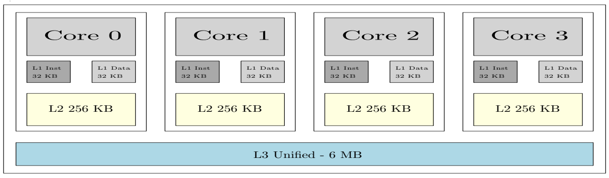


Fig. 2. Cache Architecture in Intel Ivy Bridge [11]

```
Code-hammer
{
  mov (X), %eax  // read from address X
  mov (Y), %ebx  // read from address Y
  clflush (X)    // flush cache for address X
  clflush (Y)    // flush cache for address Y
  jmp Code-hammer
}
```

In [2], a code snippet is provided in which the program generates a read to DRAM on every data access. The *mov* instructions generate access request to DRAM at X and Y locations. The following `clflush` instructions evict the data elements from all levels of cache. However, if X and Y point to different rows in the same bank, code-hammer will cause X and Y's rows to be repeatedly activated. This may lead to disturbance error in the adjacent cells of the accessed rows, resulting in flipping bits in the adjacent row cells.

In the next subsection, we summarize some key concepts of cache architecture in modern processors.

2.3 Cache Memory Architecture

In recent processors, there exists a hierarchy of cache memories where the size of each cache level increases as we move to its higher level, but their access times increases. L3 or Last Level Cache (LLC) is shared across processor cores, takes larger time and is further divided into slices such that it can be accessed by multiple cores concurrently. Figure 2 illustrates the architectural specification for a typical Intel Ivy-Bridge architecture [11]. In Intel architecture, the data residing in any of the lower levels of cache are included in the higher levels as well. Thus are inclusive in nature. On a LLC cache miss, the requested element is brought in the cache from the main memory, and the cache miss penalty is much higher compared to the lower level cache hits.

Requested data are brought from the main memory in the cache in chunks of cache lines. This is typically of 64 Bytes in recent processors. The data requested by the processor is associated with a virtual address from the virtual address space allocated to the running process by the Operating System. The virtual address can be partitioned into two parts: the lower bits in Figure 3 is the offset within the page typically represented by $log_2(page\_size)$ bits, while the remaining upper bits forms the *page_number*. The *page_number* forms an index to *page table* and translates to physical *frame number*. The frame number together with the offset bits constitute the physical address of the element. The translation of virtual to physical addresses are performed at run time, thus physical addresses of each elements are most likely to change from one execution to another.

The physical address bits decide the cache sets and slices in which a data is going to reside. If the cache line size is of b bytes, then least significant $log_2(b)$

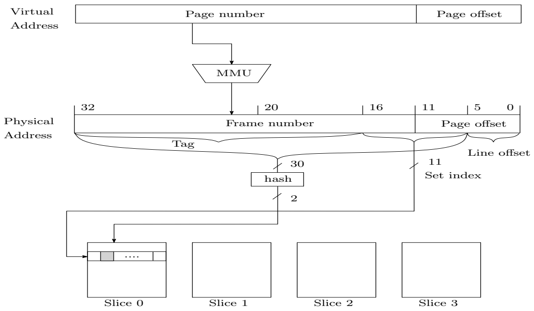


Fig. 3. Cache Memory Indexing [12]

bits of the physical addresses are used as index within the cache line. If the target system is having k processor cores, the LLC is partitioned into k slices each of the slice have c cache sets where each set is m way associative.

The $log_2(k)$ bits following $log_2(b)$ bits for cache line determines the cache set in which the element is going to reside. Because of associativity, m such cache lines having the identical $log_2(k)$ bits reside in the same set. The slice addressing in modern processors is implemented computing a complex Hash function. Recently, there has been works which reverse engineered [12,13] the LLC slice addressing function. Reverse engineering on Intel architectures has been attempted in [13] using timing analysis. The functions differ across different architectures and each of these functions are documented via successful reverse engineering in [12,13].

In the following sections, we use this knowledge of underlying architecture to build an attack model which can cause successful flips in a secret value.

3 Combining Timing Analysis and Rowhammer

In this section we discuss the development of an algorithm to induce bit flip in the secret exponent by combining timing analysis and DRAM vulnerability.

3.1 Attack Model

In this paper, we aim to induce bit fault in the secret exponent of the public key exponentiation algorithm using rowhammer vulnerability of DRAM with increased controllability. The secret resides in some location in the cache memory and also in some unknown location in the main memory. The attacker having user-level privileges in the system, does not have the knowledge of these locations in LLC and DRAM since these location are decided by mapping of physical address bits. The threat model assumed throughout the paper allows adversary to send known ciphertext to the algorithm and observe its decrypted output.

Let us assume that the adversary sends input plaintext to the encryption process and observes the output of the encryption. Thus the adversary gets hold of a valid plaintext-ciphertext pair, which will be used while checking for the bit flips. The adversary has the handle to send ciphertext to the decryption oracle, which decrypts the input and sends back the plaintext. The decryption process constantly polls for its input ciphertexts and sends the plaintext to the requesting process. The adversary aims to reproduce bit flip in the exponent and thus first needs to identify the corresponding bank in DRAM in which the secret exponent resides. Let us assume, that the secret exponent resides in some bank say, bank A. Though the decryption process constantly performs exponentiation with accesses to the secret exponent, but such access requests are usually addressed from the cache memory itself since they result in a cache hit. In this scenario it is difficult for the adversary to determine the bank in which the secret resides because the access request from the decryption process hardly results in main memory access.

According to the DRAM architecture, the channel, rank, bank and row addressing of the data elements depend on the physical address of the data elements. In order to perform rowhammering on the secret exponent, precise knowledge of these parameters need to be acquired, which is impossible for an adversary since the adversary does not have the privilege to obtain the corresponding physical addresses to the secret. This motivates the adversary to incorporate a spy process which probes on the execution of the decryption algorithm and uses timing analysis to successfully identify the channel, rank and even the bank where the secret gets mapped to.

The adversary introduces a spy process which runs everytime, before each decryption is requested. The spy process issues accesses to data elements of the eviction set, which eventually flushes the existing cache lines with its own data requests and fills the cache. Thus during the next request to the decryption process, the access to the secret exponent results in a cache miss and the corresponding access request is addressed from the bank A of main memory. Effectively, a spy process running alternate to the decryption process, makes arbitrary data accesses to ensure that every access request from the decryption process is addressed from the corresponding bank of the main memory.

3.2 Determining the Eviction Set

As mentioned before, the attack model is assumed to be such that the adversary has access to a system where the decryption is running. The decryption algorithm performs exponentiation involving the secret exponent and initially the adversary aims to determine the cache sets in which the secret exponent bits maps to. The adversary is oblivious of the virtual address space used by the decryption engine and thus involves a spy process which uses *Prime + Probe* [11, 14, 15] cache access methodology to identify the target sets. The execution scenario of the decryption and the adversarial processes running concurrently on the same system are depicted in Figure 4. In this context, the spy process targets the Last Level cache (LLC) since it is shared within all cores of the system. The adversary sends input to the decryption engine which performs decryption on request, otherwise remains idle. Figure 4 illustrates the following steps.

1. **Step 1:** The adversary starts the spy process, which initially allocates a set of data elements and consults its own pagemap to obtain the corresponding physical addresses for each element. The kernel allows userspace programs to access their own pagemap (`/proc/self/pagemap`) [1] to examine the page tables and related

[1] (for all Linux kernels before version 4.0, the versions released from early 2015 requires administrator privilege to consult pagemap, refer to Section 6.2)

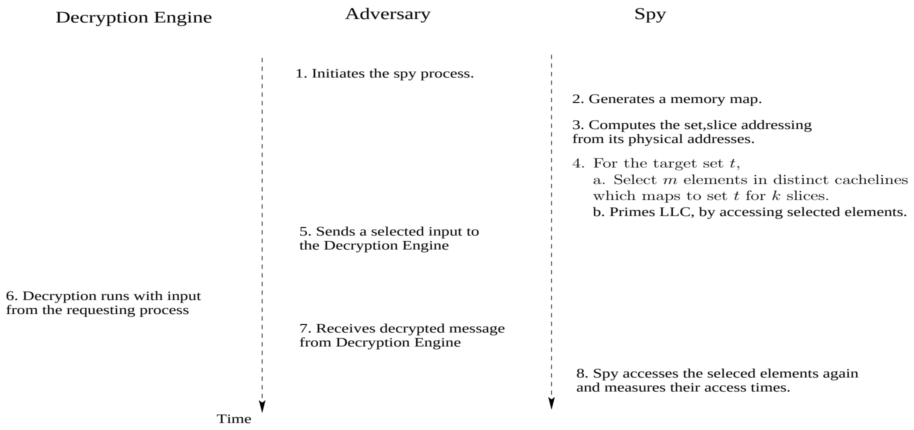

Decryption Engine
Adversary
Spy
1. Initiates the spy process.
2. Generates a memory map.
3. Computes the set,slice addressing from its physical addresses.
4. For the target set t,
a. Select m elements in distinct cachelines which maps to set t for k slices.
b. Primes LLC, by accessing selected elements.
5. Sends a selected input to the Decryption Engine
6. Decryption runs with input from the requesting process
7. Receives decrypted message from Decryption Engine
8. Spy accesses the seleced elements again and measures their access times.
Time

Fig. 4. Steps to Determine Cache sets shared by Secret Exponent

information by reading files in */proc*. The virtual pages are mapped to physical frames and this mapping information is utilized by the spy process in the following steps.

2. **Step 2:** Once the physical addresses are obtained, the Last level cache set number and their corresponding LLC slice mappings are precomputed by the spy with the address mapping functions as explained in Section 2.3. Let us suppose, that the target system is having k processor cores, thus the LLC is partitioned into k slices, each of the slice have c cache sets where each set is m way associative. All elements belonging to the same cache line are fetched at a time.
   - If the cache line size is of b bytes, then least significant $log_2(b)$ bits of the physical addresses are used as index within the cache line.
   - As described in Section 2.3, the $log_2(k)$ bits following these $log_2(b)$ bits determine the cache set in which the element is going to reside.
   - Because of associativity, m such cache lines having identical $log_2(k)$ bits reside in the same set.
   - In modern processors, there exists one more parameter that determines which slice the element belongs to. Computing the Hash function reverse engineered in [12, 13], we can also compute the slice in which a cache set gets mapped. The functions are elaborated in the experimental Section 4.1.

   Thus, at the end of this step, the spy simulates the set number and slice number of each element in its virtual address space.

Repeat the following steps for all of the c sets in the LLC.

3. **Priming Target set t:** The spy primes the target Set t and becomes idle. This is the most crucial step for the entire procedure since the success of correctly determining the cache sets used by spy process entirely depends on how precisely the existing cache lines have been evicted from the cache by the spy in the Prime phase. In order to precisely control the eviction of existing cache lines from set t, a selection algorithm is run by the spy which selects an *eviction set* of $m * k$ elements each belonging to set t from its defined memory map.
   - Thus the selection algorithm selects elements belonging to distinct cache lines for each of the k cache slices where their respective physical addresses maps

to the same set t. These selected data elements constitutes the *eviction set* for the set t.
   - In addition to this, since each set of a slice is m way associative, the selection algorithm selects m such elements corresponding to each k cache slice belonging to set t.
   - The spy process accesses each of these $m * k$ selected memory elements repeatedly in order to ensure that the cache replacement policy has evicted all the previously existing cache lines.

   This essentially ensures that the target set t of all slices is only occupied with elements accessed by the spy.
4. **Decryption Runs:** The adversary sends the chosen ciphertext for decryption and waits till the decryption engine sends back the message. In this decryption process, some of the cache lines in a particular set where the secret maps, gets evicted to accommodate the cache line of the secret exponent.
5. **Probing LLC:** On getting the decrypted output the adversary signals the spy to start probing and timing measurements are noted. In this probing step, the spy process accesses each of the selected m elements (in Prime phase) of eviction set t for all slices and time to access each of these elements are observed.

   The timing measurements will show a variation when the decryption algorithm shares same cache set as the target set t. This is because, after the priming step the adversary allows the decryption process to run. If the cache sets used by the decryption is same as that of the spy, then some of the cache lines previously primed by the spy process gets evicted during the decryption. Thus, when the spy is again allowed to access the same elements, if it takes longer time to access then it is concluded that the cache set has been accessed by the decryption as well. On the other hand, if the cache set has not been used by the decryption, then the time observed during probe phase is less since no elements primed by spy have been evicted in the decryption phase.

Determining the LLC Slice where the Secret Maps The *Prime + Probe* timing analysis elaborated in the previous discussion successfully identifies the LLC set in which the cache line containing the secret exponent resides. Thus at the end of the previous step we obtain an *eviction set* of $m * k$ elements which map to the same set as the secret in all of the k slices. Now, this time the adversary can easily identify the desired LLC slice by iteratively running the same *Prime + Probe* protocol separately for each of the k slices with the selected m elements for that particular slice. The timing observations while probing will show significant variation for a set of m elements which corresponds to the same slice where the secret maps. Thus we further refine the size of *eviction set* from $m * k$ to m elements.

3.3 Determining the DRAM Bank that the Secret Maps

In this section, we describe a timing side channel analysis performed by the adversary to successfully determine the bank of the DRAM in which the secret exponent maps to. In the previous section, a timing analysis is elaborated which finally returns an *eviction set* of m elements which maps to the same set as well as the same slice as the secret exponent. Thus, if the adversary allows the spy and the decryption engine to run in strict alternation, then the decryption engine will always encounter a cache miss for the secret exponent, and the access request shall always result in a main memory access. As described in Section 2.1, DRAMs are primarily partitioned into channel, ranks and banks where each bank access is serviced through a buffer termed as row-buffer. Concurrent accesses to different rows in the same DRAM bank results in row-buffer conflict and automatically leads to higher access time. The functions which decide the channel, rank and bank mapping from the physical addresses are not disclosed by the architecture manufacturers. In some recent works, reverse engineering of these unknown mappings have been targeted. A successful deployment of a high speed covert channel has also been reported [16].

In this paper, we illustrate a timing analysis of accessing separate DRAM banks using this knowledge of reverse engineering and the following steps highlight how this is achieved. In order to exploit timing variation occurring due to the row-buffer collision, accesses requested from the decryption process as well as the adversarial spy process must result in main memory accesses. Intuitively, DRAM access time will increase only if addresses map to the same bank but to a different row. Thus to observe row-buffer conflict between the decryption and adversarial spy the major challenges are:

- To ensure that every access to secret exponent by the decryption process results in LLC cache miss and thus automatically result in main memory access. This is elaborated in the previous subsection, as to how the spy determines the *eviction set* and selectively accesses those elements to evict existing cache lines from the set. Let the spy generates an eviction set C with data elements in distinct cache lines mapping to the same set and slice as the secret.
- This suggests that before each decryption run, the spy has to fill the particular cache set by accessing elements in eviction set C.
- In addition to this, row-buffer conflict and access time delay can only be observed if two independent processes concurrently request data residing in the same bank but in different rows. In order to produce a row-buffer conflict with the secret exponent requested by the spy, the adversary has to produce concurrent access requests to the same bank.

The adversary allows the spy process to `mmap` a chunk of memory and the spy refers to its own pagemap to generate the physical addresses corresponding to each memory element. Following the functions reverse engineered in [16, 17] the spy pre-computes the channel, rank and bank addressing for the corresponding physical addresses. As illustrated in Figure 5, the timing analysis has to be performed by accessing elements from each bank. After each access request by the spy, the elements are flushed deliberately from the cache using `clflush`.

The adversary sends an input to the decryption engine and waits for the output to be received. While it waits for the output, the spy process targets one particular bank, selects a data element which maps to the bank and accesses the data element. This triggers concurrent accesses from the spy and the decryption to the DRAM banks. Repeated timing measurements are observed for each of the DRAM bank accesses by the spy, and this process is iterated for elements from each DRAM bank respectively.

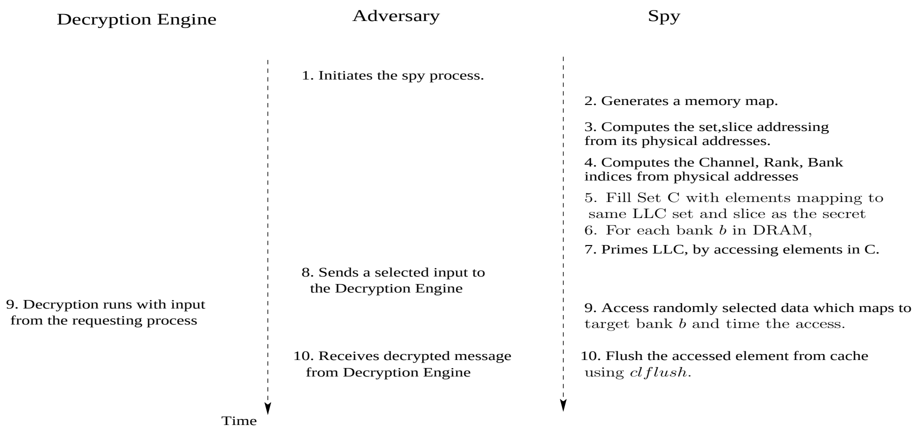

Decryption Engine
Adversary
Spy
1. Initiates the spy process.
2. Generates a memory map.
3. Computes the set,slice addressing from its physical addresses.
4. Computes the Channel, Rank, Bank indices from physical addresses
5. Fill Set C with elements mapping to same LLC set and slice as the secret
6. For each bank b in DRAM,
7. Primes LLC, by accessing elements in C.
8. Sends a selected input to the Decryption Engine
9. Decryption runs with input from the requesting process
9. Access randomly selected data which maps to target bank b and time the access.
10. Receives decrypted message from Decryption Engine
10. Flush the accessed element from cache using $clflush$.
Time

Fig. 5. Steps to determine the DRAM bank in which secret maps to

3.4 Performing Rowhammer in a Controlled Bank

In the previous subsections, we have discussed how the adversary performs timing analysis to determine cache set collision and subsequently use it to determine DRAM bank collisions to identify where the secret data resides. In this section, we aim to induce fault in the secret by repeatedly toggling DRAM rows belonging to the same DRAM bank as the secret.

Inside the DRAM bank, the cells are two-dimensionally aligned in rows and columns. The row index in which any physical address maps is determined by the most significant bits of the physical address. Thus it is absolutely impossible for an adversary to determine the row index of the secret exponent. Thus rowhammer to the secret exponent has to be performed with elements which map to the same DRAM bank as the secret, but on different row indices until and unless the secret exponent is induced with a bit flip.

The original algorithm for rowhammer in [2], can be modified intelligently to achieve this precise bit flip. The algorithm works in following steps:

- A set of addresses are chosen which map to different row but the same bank of DRAM.
- The row indices being a function of the physical address bits are simulated while execution. Elements of random row indices are selected and accessed repeatedly by the adversary to induce bit flips in adjacent rows.
- The detection of bit flip in secret can be done easily, if and only if the output of decryption differs.

The rowhammering attempts required to produce a suitable bit flip on the secret depends on the total number of rows in a bank, since the adversary has no handle to know in which row in the bank the secret exponent is residing. Neither it has handle to place its own `mmap`-ed data deliberately adjacent to the secret, such that it can easily exploit the rowhammer bug. Thus the adversary can only select those elements which belong to the same bank as secret and access them repeatedly to induce bit flips in the secret. To increase the probability of bit flip in the secret exponent, the adversary needs to `mmap` multiple times to generate data which belong to different rows.

4 Experimental Validation for Inducing Bit Flips on Secret

In this section we present the validation of our previous discussion through experiments. Our experiments are framed with the following assumptions:

- We target an 1024 bit RSA implementation using square and multiply as the underlying exponentiation algorithm. We have used the standard `GNU-MP` big integer library (version number `2:5.0.2+dfsg-2`) for our implementation. The adversary sends a chosen ciphertext for decryption which involves an exponentiation using the secret exponent.
- The experimentations are performed considering the address bit mappings of Intel Ivy Bridge micro-architecture. These are the line of processors based on the 22nm manufacturing process. The experiments are performed on Intel Core i5-3470 processor running Ubuntu 12.04 LTS with the kernel version of 3.2.0-79-generic.
- The adversary is assumed to have user-level privileges to the system where decryption process runs. It uses `mmap` to allocate a large chunk of data and accesses its own pagemap (`/proc/self/pagemap`) to get the virtual to physical address mappings. The Linux kernel version for our experimental setup being older than version 4.0, we did not require administrator privileges to perform the entire attack.

4.1 Identifying the Cache Set

The experiments being performed on RSA, the 1024 bit exponent resides in consecutive 1024 bit locations in memory. Considering the cache line size as 64 bytes, 1024 bits of secret maps to 2 cache lines. As described in Section 2.3, 11 bits of physical address from $b_6, b_7, \cdots b_{16}$ refer to the Last Level cache set. Moreover, the papers [12, 13] both talk about reverse engineering of the cache slice selection functions. The authors in paper [13], used *Prime + Probe* methodology to learn the cache slice function, while the authors in [12] monitored events from the performance counters to build the cache slice functions. Though it has been observed that the LLC slice functions reported in these two papers are not same.

In our paper, we devised a *Prime + Probe* based timing observation setup and wished to identify the target cache set and slice which collides with the secret. Thus we were in the lines of [13] and used the function from [13] in our experiments for *Prime + Probe* based timing observations. As illustrated in the following section, the timing observations using functions from [13] can correctly identify the target cache slice where the secret maps to. Reverse engineering of Last Level Cache (LLC) slice for Intel Ivy Bridge Micro-architecture in [13] uses the following function:
$b_{17} \oplus b_{18} \oplus b_{20} \oplus b_{22} \oplus b_{24} \oplus b_{25} \oplus b_{26} \oplus b_{27} \oplus b_{28} \oplus b_{30} \oplus b_{32}$.
But the architectural specification described in [13], documented that their selected system for Ivy Bridge architecture has LLC size of $3MB$. In our experimental setup, instead of $3MB$ we had $6MB$ LLC with is divided among 4 cores. Thus we adopted the functions documented for Haswell, and the function worked

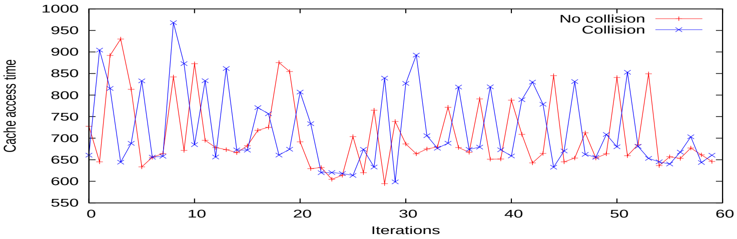


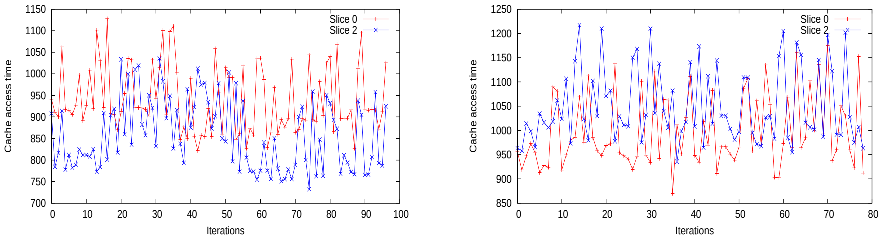
Fig. 6. Timing Observations for Cache set collision

successfully. The functions used for slice selection are:
$h_0 = b_{17} \oplus b_{18} \oplus b_{20} \oplus b_{22} \oplus b_{24} \oplus b_{25} \oplus b_{26} \oplus b_{27} \oplus b_{28} \oplus b_{30} \oplus b_{32}$
$h_1 = b_{18} \oplus b_{19} \oplus b_{21} \oplus b_{23} \oplus b_{25} \oplus b_{27} \oplus b_{29} \oplus b_{30} \oplus b_{31} \oplus b_{32}$
Our host machine has LLC with 12 way associativity and having 4 cache slices each consisting of 2048 sets. The adversary `mmaps` a large chunk of memory, and consults its own pagemap to obtain the physical addresses corresponding to each element in the memory map. Using the equations mentioned above, the adversarial process simulates the cache set and slice in which their respective physical address points to. The experimental setup being 12-way associative, the selection algorithm for each set and slice selects the *eviction set* with 12 elements belonging to distinct cache lines and mapping to the same set for a particular slice. The host machine having 4 LLC slice for 4 cores selects the eviction set having altogether $12 * 4 = 48$ data elements in distinct cache lines mapped to same set and all 4 slices.

In our experimental setup, the adversary runs the *Prime + Probe* cache access methodology over each of the 2048 sets in each of the LLC slices. Each of the 2048 sets are targeted one after another. The cycle starts with priming a set with elements from eviction set and then allowing the decryption to happen and again observing the timings required for accessing selected elements from the set. The timing observations from the probe phase on 2 such LLC sets are illustrated in Figure 6. The sets are chosen such that one of them is having a collision with the secret exponent and the other set does not have any collision. The variation of timing in these two sets is apparent in Figure 6, where the set which observes a collision observes higher access time to the other set. The average access time of the these two sets during the probe phase differs by approximately 80 clock cycles. This implies that the LLC cache set having collision with the secret exponent cache line can be identified from the other sets which does not have collision with the decryption algorithm.

4.2 Alternative Strategy to Determine the Target Cache Set

In the previous subsection, we observed that timing observations obtained by repeating the $m * k$ accesses to the individual set on all k slices is sufficient for identification of the target cache set. Though in [7], it has been stated that only $m * k$ accesses may not be sufficient to guarantee the existing cache lines to be evicted from the cache. In this context, we argue that the cache eviction sets are identified in [7] so that accessing the elements of this set in a predetermined order results in an equivalent effect of `clflush` to induce rowhammer



(a) **Timing Observations during Probe phase when secret maps to slice 0**

(b) **Timing Observations during Probe phase when secret maps to slice 2**

Fig. 7. Timing Observations for LLC slice collision

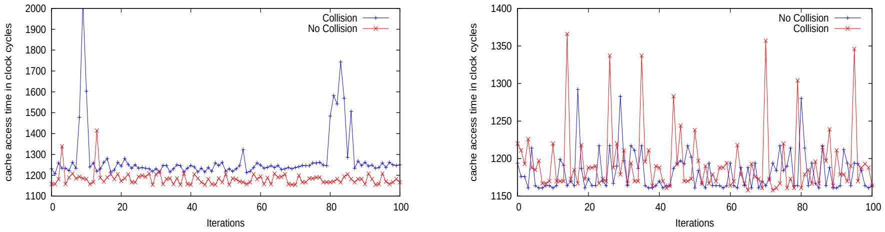
flips. Since to exhibit successful bit flips, hammering of rows needs to satisfy various preconditions, it was crucial for authors in [7] to generate an optimal eviction set.

In our paper the conditions are little less stringent, since we are using `clflush` to induce bit flips. In addition to this, we constrain the hammering to the target bank. The identification of this bank has a series of experiments to be performed. In this scenario, we claimed that accessing the near-optimal eviction set of $m * k$ accesses for each cache set for all k slices repeatedly will result in eviction of secret from the respective cache set and result in DRAM accesses of the secret key. In addition, we have again performed our experiments by implementing the optimal eviction set as described in [7]. The results we obtain in Figure 8a can be compared with Figure 6. The separation of timing and identification of collision set from the non-collision set definitely improves upon accessing the eviction set with parameters defined in [7].

4.3 Identifying the LLC Slice

Once the cache set is identified, the variation from timing observations for different LLC slices leak the information of which LLC slice the secret maps to. In the same experimental setup as in the previous section, we identify the slice in which the actual secret resides, using timing analysis with the slice selection function. Since we have already identified the LLC cache set with which the secret collides, 12 data elements belonging to each slice of the particular set are selected. *Prime + Probe* timing observations are noted for the set of 12 elements for each slice. The slice observing collision with the secret exponent will suffer from cache misses in the probe phase and thus have higher access time to other slices.

We illustrate the timing observations for two scenarios in Figure 7a,b. In Figure 7a, the secret is mapped to LLC slice 0, while in Figure 7b, the secret gets mapped to LLC slice 2. In both of the figures, access time for probing elements for the cache slice for which the secret access collides is observed to be higher than the other cache slice which belongs to the same set but do not observe cache collision. Thus because of collision of accesses of both the processes to the same slice, the spy observed higher probe time for slice 0, than slice 2 in



(a) **Timing Observation of Cache set Collision from optimal eviction set**

(b) **Timing Observation of Cache slice Collision using equations in [12]**

Fig. 8. Timing Observations for LLC set and slice collision

Figure 7a. On the contrary, in a different run, the secret exponent got mapped to LLC slice 2, which in Figure 7b shows higher probe time than slice 0. Thus we can easily figure out the cache slice for the particular set for which both the decryption and the spy process accesses actually collides.

We also extended our experiment with the reverse engineered cache slice functions from [12]. Figure 8b shows the timing observations when we use the slice selection functions for a 4-core processor. The functions [12] are:
$o_0 = b_6 \oplus b_{10} \oplus b_{12} \oplus b_{14} \oplus b_{16} \oplus b_{17} \oplus b_{18} \oplus b_{20} \oplus b_{22} \oplus b_{24} \oplus b_{25} \oplus b_{26} \oplus b_{27} \oplus b_{28} \oplus b_{30} \oplus b_{32} \oplus b_{33}$
$o_1 = b_{07} \oplus b_{11} \oplus b_{13} \oplus b_{15} \oplus b_{17} \oplus b_{19} \oplus b_{20} \oplus b_{21} \oplus b_{22} \oplus b_{23} \oplus b_{24} \oplus b_{26} \oplus b_{28} \oplus b_{29} \oplus b_{31} \oplus b_{33} \oplus b_{34}$
Similar to our previous observations, Figure 8b shows that we were able to identify the target cache slice from the timing observations using cache slice reverse engineering functions from [12].

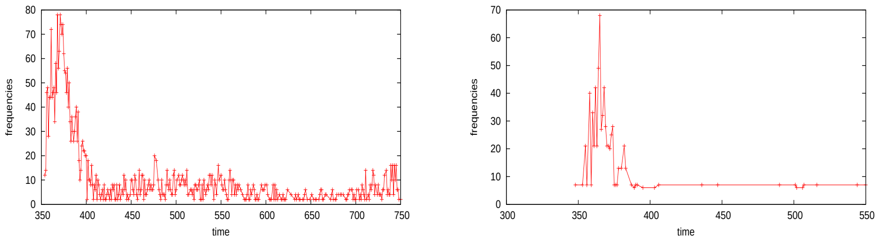
Determining the LLC set and slice in which secret maps, actually gives the control to the adversary to flush the existing cache lines in these locations, and thus everytime the decryption process have to access the main memory. In simple words, accesses made by the adversary to this particular LLC set and slice acts as an alternative to `clflush` instruction being added to the decryption process.

4.4 Identifying the DRAM Bank

From the previous subsections, we identified particularly the LLC set and the slice mappings for the decryption process. Thus if the adversary selects data elements which belong to same set as well slice as to the secret exponent, and alternatively primes the LLC before running each decryption, each time the decryption process will encounter a cache miss and which will eventually get accessed from the main memory. This aids the adversary to identify the respective bank of DRAM, where the secret exponent is mapped. For the 1024-bit RSA exponentiation secret key, the channel, rank and bank mappings of DRAM will be decided by the equations reverse engineered in [16, 17]. In our experimental setup, there exists 2 channel, 1 DIMM per channel, 2 ranks per DIMM, 8 banks per rank and 2^{14} rows per bank.

- The DRAM bank equations for Ivy Bridge [16] is decided by the physical address bits: $ba_0 = b_{14} \oplus b_{18}$, $ba_1 = b_{15} \oplus b_{19}$, $ba_2 = b_{17} \oplus b_{21}$,
- Rank is decided by $r = b_{16} \oplus b_{20}$ and the
- Channel is decided by, $C = b_7 \oplus b_8 \oplus b_9 \oplus b_{12} \oplus b_{13} \oplus b_{18} \oplus b_{19}$.
- The DRAM row index is decided by physical address bits $b_{18}, \cdots, b_{31}$.

In the same experimental setup as previous, the adversary targets each bank at a time and selects elements from the memory map for which the physical addresses map



(a) **Timing Observations in clock cycles for DRAM bank collision**

(b) **Timing Observations in clock cycles of separate DRAM bank access**

Fig. 9. Timing Observations for Row-buffer collision during DRAM bank accesses

to that particular bank. The following process is repeated to obtain significant timing observations:

1. The spy primes the LLC and requests decryption by sending ciphertext.
2. While the spy wait for the decrypted message, it selects an element for the target bank from the memory map, `clflush`'es it from cache, and accesses the element. The `clflush` instruction removes the element from all levels of cache, and the following access to the element is addressed from the respective bank of the DRAM.
3. The time to DRAM bank access is also noted.

It is important to note that, there is no explicit synchronization imposed upon the the two concurrent processes in their software implementation. The decryption and the spy both requests a DRAM bank access. If the target bank matches with the bank in which the secret is mapped, then we expect to have higher access time. Figure 9a, b are the timing observations noted by the spy process while it accesses elements selected from the target bank. Figure 9a refers to the the case where the higher access times are observed due to the row-buffer collision as the bank accessed by the spy is same as the secret mapped bank. While Figure 9b refers to the situation where the elements accessed by the spy are from an arbitrary different bank than the bank where secret maps. In both of the figures, the significant high peak has been observed in timing range of $350-400$ clock cycles. While in Figure 9a, the row-buffer collision is apparent because there are significant number of observations which have timings greater than the region where the peak is observed. Had there been an absolute synchronization of two processes accessing the same DRAM bank, each access to DRAM bank, by either of the two process would have suffered from row-buffer collision. Thus in our scenario, we claim that in majority of cases, though the accesses are addressed from the same bank they seldom result in row-buffer collision, which justifies the peak around $350 - 400$ clock cycles. From this we conclude that detection of row-buffer collision can only be identified over a significant number of timing observations. The DRAM bank which shows such higher access times is identified to be the bank where secret data resides.

4.5 Inducing Bit Flip using Rowhammer

In the previous section, we have illustrated that how the adversary is able to distinguish the bank in which the secret exponent resides. The software implementation of the induction of bit flip is performed by repeated access to the elements of the same bank. The following pseudo-code is used to hammer rows in specific banks After each access to the element it is deliberately flushed from the cache using the `clflush` instruction by the adversary.

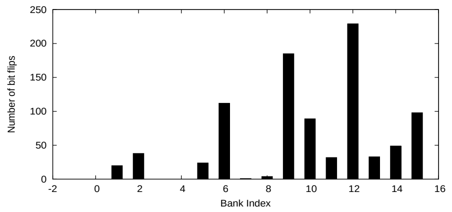


Fig. 10. Number of bit flips observed in all banks of a single DIMM

```
Code-hammer-specific-bank
{
  Select set of 10 data elements mapping to specific bank
  Repeat
  {
    Access all elements of the set
    Clflush each element of the set
  }
  jmp Code-hammer-specific-bank
}
```

A statistic over observations of bit flips in respective banks is reported in Figure 10. The bar graph shows the number of bit flip instances we were able to observe for respective banks of a single Dual In-line Memory Module (DIMM). The bit faults that we have observed in our experiments are bit-reset faults.

The row index of the location of the secret in the DRAM bank is determined by the physical address bits of the secret. Thus this implies that the secret exponent can sit in any of the rows in the target bank. Accordingly, we restricted our hammering attempts in the target bank and we selected random accesses to the target bank which eventually resulted in bit flips. Thus we slightly modified our setup such that the code iteratively runs until and unless the decryption output changes, which signifies that secret exponent bits have been successfully flipped. The fault attack in [9] requires a single faulty signature to retrieve the secret. Thus, bit flip introduced in the secret exponent by the rowhammer in a specific bank can successfully reveal the secret by applying fault analysis techniques in [9]. The probability of bit flip is $1/2^{14}$, since there are 2^{14} rows in a particular bank. Interestingly, the size of the secret key has an effect on the probability of bit flip in the secret exponent. In other words, we can say that the probability of bit flip in the secret exponent will be more if the secret exponent size is larger.

5 Possible Countermeasures

There has been various countermeasures of rowhammer attacks proposed in literature. In [2], seven potential system level mitigation techniques were proposed which range from designing secure DRAM chips, enforcing ECC protection on them, increasing the refresh rate, identification of victim cells and retiring them and refreshing vulnerable rows (for which the adjacent rows are getting accessed frequently). As mentioned in [2], each of these solutions suffers from the trade-off between feasibility, cost, performance, power and reliability. In particular, the solution named as Probabilistic Adjacent Row Activation (PARA) has the least overhead among the solutions proposed in [2]. The memory controller in PARA is modeled such that every time a row closes, the controller decides to refresh its adjacent rows with probability p (typically 1/2). Because of its probabilistic nature, the approach is low overhead as it does not require any complex data structure for counting the number of row activations.

Another hardware level mitigation is reported in [5], where it is mentioned that the LPDDR4 standard for the DRAM incorporated two features for the hardware level mitigation such as Targeted Row Refresh (TRR) and Maximum Activate Count (MAC). Among which, it is reported that TRR technique is getting deployed in the next generation DDR4 memory units [18,19]. TRR incorporates a special module which can track the frequently made row-activations and can selectively refresh the rows adjacent to these aggressor rows. All of the above discussed protections have to be incorporated in hardware, but this does not eliminate the threat from rowhammer attacks since many of the manufacturers refer to these as optional modules.

There are few attempts which provide software level protection from rowhammer attacks. The `clflush` instruction was responsible for removing the target element from the cache and that resulted in DRAM accesses. In order to stop the security breaches from NaCl sandbox escape and privilege escalation [5], Google NaCl sandbox was recently patched to disallow applications from using the `clflush` instruction. The other software level mitigation is to double the refresh rate from 64ms to 32ms by changing the BIOS settings, or alternatively upgrading own BIOS with a patched one. It has been reported in [20], that system manufacturers such as HP, Lenovo and Apple have already considered updating the refresh rate. But both of the techniques such as doubling refresh rate and removing access to `clflush` instruction as a prevention technique has been proved to be ineffective in [20]. The paper illustrates a case study of introducing bit flips inspite of having refresh interval as low as 16ms and the method does not use the `clflush` instruction. The paper also propose an effective, low cost software based protection mechanism named ANVIL. Instead [20] propose a two-step approach which observe the LLC cache miss statistics from the performance counters for a time interval, and examines if the number of cache misses crosses the predetermined threshold. If there are significantly high number of cache misses, then the second phase of evaluation starts, which samples the DRAM accesses of the suspected process and identifies if rows in a particular DRAM bank is getting accessed. If repeated row activation in same bank is detected, then ANVIL performs a selective refresh on the rows which are vulnerable.

6 Further Discussion

The present paper's main focus is to show that targeted faults can be inflicted by rowhammer. As a consequence, we have cited the example of a fault analysis on RSA, which is not protected by countermeasures. One of the objectives of this paper, is to show that fault attacks are serious threats even when triggered using software means. This makes the threat more probable as opposed to a fault injection by hardware means: like voltage fluctuations etc. Thus, this emphasizes more the need for countermeasures, at the software level.

Having said that, even standard libraries like OpenSSL use fault countermeasures, but they are not fully protected against these class of attacks. For example, in Black Hat 2012 [21], a hardware based fault injection was shown to be of threat to OpenSSL based RSA signature schemes. It was reported that the initial signature is verified by the public key exponent, however in case of a fault, another signature is generated and this time it is not verified [21]. The final signature is not verified because it is widely assumed that creating a controlled fault on a PC is impractical. More so, the faults are believed to be accidental computational errors, rather than malicious faults. Hence, the probability of inflicting two successive faults is rather low in normal computations! However, in case of rowhammer, as the fault is created in the key, repeating the process would again result in a wrong signature and thus get released.

Hence, the objective of the current paper is to highlight that inflicting controlled faults are more probable through software techniques than popularly believed, and hence ensuring that verification should be a compulsory step before releasing signatures.

6.1 Assumptions of the Proposed Attack

In our proposed attack, we assumed that the secret decryption exponent resides in a particular location of the DRAM and the decryption oracle continuously polls for input ciphertexts. In addition we also assume that the secret resides in the same location in the DRAM through out the duration of the attack and is not page-swapped by other running processes.

6.2 Limitations and Practicality of our Attack

In this paper, the access to pagemap is assumed to be available at user privilege level since our setup has 3.2.0-79-generic version of Linux kernel. But from early 2015, for versions of kernel 4.0 onwards, the access to this pagemap has been restricted to processes with root privileges. However, the attack would still be relevant in a cross-VM environment as in [22], where the users of the co-located VMs actually have the administrator privilege and can consult the pagemap for the required virtual to physical address translation. In such a scenario, the attacker is assumed to be mounted on a VM which is co-resident to the VM which hosts the decryption oracle. Timing information obtained from *Prime + Probe* methodology in this experimental setup along with the reverse engineering knowledge can be used to precisely induce fault in the secret of the co-resident VM. Our paper primarily focuses on the vulnerability analysis of rowhammer in context to Linux kernels, but the vulnerability may be equally or more relevant in context to other operating systems where the access to a data structure such as pagemap (in context to Linux kernels) is not restricted only to administrator privileges. Moreover, the attack in its original form might be relevant in customized embedded system applications, thus it would be an interesting exercise to ascertain the security impact of rowhammer in such applications.

7 Conclusion

In this paper, we claim to illustrate in steps a combination of timing and fault analysis attack exploiting vulnerability of recent DRAM's disturbance error to induce a bit flip targeting the memory bank shared by the secret. This is a practical fault model and uses *Prime + Probe* cache access attack methodology to narrow down the search space where the adversary is supposed to induce flip. The experimental results illustrate that the timing analysis shows significant variation and leads to the identification of LLC set and slices. In addition row-buffer collision has been exploited to identify the DRAM bank which holds the secret. The worst case complexity of inducing fault by repeated hammering of rows in the specific memory bank typically is same as the number of rows in bank. The proposed attack finds most relevance in cross-VM setup, where the co-located VMs share the same underlying hardware and thus root privileges are usually granted to the attack instance.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments and suggestions. We would also like to thank Prof. Berk Sunar for his insightful feedback and immense support. This research was supported in part by the TCS Research Scholarship Program in collaboration with IIT Kharagpur. This work was also supported in part by the Challenge Grant from IIT Kharagpur, India and Information Security Education Awareness (ISEA), Deity, India.

References

1. Wikipedia. Rowhammer wikipedia page, https://en.wikipedia.org/wiki/Rowhammer, 2016.
2. Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 361–372. IEEE Computer Society, 2014.
3. Rei-Fu Huang, Hao-Yu Yang, Mango Chia-Tso Chao, and Shih-Chin Lin. Alternate hammering test for application-specific drams and an industrial case study. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 1012–1017. ACM, 2012.
4. Dae-Hyun Kim, Prashant J. Nair, and Moinuddin K. Qureshi. Architectural support for mitigating row hammering in DRAM memories. *Computer Architecture Letters*, 14(1):9–12, 2015.
5. Mark Seaborn, Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges,http://googleprojectzero.blogspot.in/2015/03/exploiting-dram-rowhammer-bug-to-gain.html , 2015.
6. Rui Qiao and Mark Seaborn. A new approach for rowhammer attacks. In *HOST, 2016*, 2016.
7. Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. *CoRR*, abs/1507.06955, 2015.
8. Mark Seaborn, Thomas Dullien. Test DRAM for bit flips caused by the rowhammer problem,https://github.com/google/rowhammer-test,2015 , 2015.
9. Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.
10. JEDEC. Standard No. 79-3F. DDR3 SDRAM Specification, 2012.
11. Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 719–732. USENIX Association, 2014.
12. Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In Herbert Bos, Fabian Monrose, and Gregory Blanc, editors, *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*, volume 9404 of *Lecture Notes in Computer Science*, pages 48–65. Springer, 2015.
13. Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*, pages 629–636. IEEE Computer Society, 2015.
14. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In David Pointcheval, editor, *CT-RSA*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

15. Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 605–622. IEEE Computer Society, 2015.
16. Peter Pessl, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Reverse engineering intel DRAM addressing and exploitation. *CoRR*, abs/1511.08756, 2015.
17. Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 191–205. IEEE Computer Society, 2013.
18. JEDEC Solid State Technology Association. Low Power Double Data Rate 4 (LPDDR4), 2015.
19. Micron Inc. DDR4 SDRAM MT40A2G4, MT40A1G8, MT40A512M16 Data sheet, 2015 , 2015.
20. Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd M. Austin. ANVIL: software-based protection against next-generation rowhammer attacks. In Tom Conte and Yuanyuan Zhou, editors, *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 743–755. ACM, 2016.
21. Valeria Bertacco, Armin Alaghi, William Arthur, and Prateek Tandon. Torturing openssl.
22. Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. *IACR Cryptology ePrint Archive*, 2016:596, 2016.