*Computers, Materials & Continua*
DOI:10.32604/cmc.2021.014700
***Article***

Tech Science Press

# Detection Technique of Software-Induced Rowhammer Attacks

**Minkyung Lee[1] and Jin Kwak[2,*]**

[1]ISAA Lab., Department of Computer Engineering, Ajou University, Suwon, 16499, Korea
[2]Department of Cyber Security, Ajou University, Suwon, 16499, Korea
[*]Corresponding Author: Jin Kwak. Email: security@ajou.ac.kr
Received: 10 October 2020; Accepted: 01 November 2020

**Abstract:** Side-channel attacks have recently progressed into software-induced attacks. In particular, a rowhammer attack, which exploits the characteristics of dynamic random access memory (DRAM), can quickly and continuously access the cells as the cell density of DRAM increases, thereby generating a disturbance error affecting the neighboring cells, resulting in bit flips. Although a rowhammer attack is a highly sophisticated attack in which disturbance errors are deliberately generated into data bits, it has been reported that it can be exploited on various platforms such as mobile devices, web browsers, and virtual machines. Furthermore, there have been studies on bypassing the defense measures of DRAM manufacturers and the like to respond to rowhammer attacks. A rowhammer attack can control user access and compromise the integrity of sensitive data with attacks such as a privilege escalation and an alteration of the encryption keys. In an attempt to mitigate a rowhammer attack, various hardware- and software-based mitigation techniques are being studied, but there are limitations in that the research methods do not detect the rowhammer attack in advance, causing overhead or degradation of the system performance. Therefore, in this study, a rowhammer attack detection technique is proposed by extracting common features of rowhammer attack files through a static analysis of rowhammer attack codes.

**Keywords:** Rowhammer attack; static analysis; detecting technique; side-channel attack; bit flip

## 1 Introduction

Side-channel attacks have recently developed into other attack types not only with the data gained through direct access to hardware but also with software-induced hardware data. Among such attacks, a rowhammer attack, a fault attack on dynamic random access memory (DRAM), was reported that can occur in both double data rate (DDR) 3 and DDR4 synchronous dynamic random access memory (SDRAM), which shows the feasibility of a software-induced hardware attack. The first study to exploit the rowhammer attack demonstrated that it was possible to access DRAM using malicious programs in Intel and Advanced RISC Machine (ARM) systems, thereby inducing disturbance errors [1]. The Google Project Zero team demonstrated an attack in which the programs in a user level x86-64 Linux environment gained kernel privileges. The research

also proved that a rowhammer attack can be executed with a native client (NaCI) program and that the system calls of the host operating system (OS) can be directly called by acquiring permission to escape from the x86-64 sandbox of the NaCI. In addition, a rowhammer attack was exploited in x86-64 Linux processes and the rowhammer escalated the privilege to access all physical memory [2]. Moreover, a study was conducted to take over a server that was vulnerable to a rowhammer attack using a web browser with JavaScript codes. The study provided a proof-of-concept that web-browser-based rowhammer attacks are possible [3]. It has also been reported that it is possible to trigger Drammer attacks, one of the attack types that exploits a rowhammer by using a malicious application at the user level on a mobile device. It does not require user privileges or software vulnerabilities [4]. In addition, a study was conducted on the throwhammer attack, which can trigger and exploit rowhammer bit flips directly from a remote machine that is connected to a remote direct memory access (RDMA) network. This is accomplished by sending only network packets. The study demonstrated how a malicious attacker can exploit rowhammer bit flips to execute codes on a remote key-value server application [5].

As described above, rowhammer attacks have been proven to be exploited in various platforms, and can damage the system memory and system itself, or grant full control to an attacker. In response to such a variety of rowhammer attacks, efforts are being made to improve DRAM chips so that they are no longer vulnerable, or to correct the errors caused by the rowhammer using an error correcting code (ECC). Alternatively, rowhammer-prone cells can be remapped or retired by increasing the refresh rate or through a static or dynamic post-manufacturing analysis. In addition, hammered cells during runtime can be identified and refreshed [6]. However, although numerous studies have been conducted to monitor DRAM cells to detect a rowhammer or mitigate an attack, no studies have been conducted to detect a rowhammer in advance. Meanwhile, the number of software-induced hardware attacks has increased in recent years. Thus, there is an increasing need for studies on the detection of software-induced rowhammer attacks before they occur.

In this study, we propose a technique to detect rowhammer attacks in advance using static analysis. Section 2 describes the DRAM architecture, which is the target of a rowhammer attack. It defines the mechanism of a rowhammer attack as well as techniques to mitigate such an incident. Further, the IDA Python module used in the proposed technique for detecting rowhammer attacks is also described. In Section 3, as the proposed mechanism of this study, we conduct static analysis of rowhammer attacks and propose a technique for their detection. In Section 4, experimental results and analysis of the proposed techniques for detecting a rowhammer attack are provided. Finally, concluding remarks are given in Section 5.

## 2 Related Works

### *2.1 Analysis of DRAM for Rowhammer Mechanism*

A rowhammer attack exploits the structure of the cells adjacent to the DRAM. The increasing cell density of DRAM is a result of technological advancements; it can enable repeated access to rows, causing bit flips in neighboring memory rows [1]. A bit flip is a phenomenon in which a repeatedly accessed memory row generates an electrical disturbance and affects adjacent rows. As such, the hammering process can cause bit flips that affect the memory rows and consequently change the bits in memory. This section describes the DRAM architecture, which enables bit flips, and the mechanism of a rowhammer attack.

*2.1.1 Dynamic Random Access Memory (DRAM) Architecture*

In a computing environment, the main memory subsystem consists of DRAM and a memory controller. As shown in Fig. 1, the DRAM and processor are interconnected by channels and are controlled by the memory controller [1–4,7]. The main components of the DRAM are as follows:

- Channel: As a data transfer pathway between DRAM and the memory controller, a single channel has a bandwidth of 64-bits.
- Rank: Each rank consists of multiple DRAM banks that share an internal bus for data reading and writing. In general, DRAM is composed of two ranks consisting of eight banks.
- Memory Controller: A memory controller can interact with DRAM ranks by time-multiplexing the I/O bus of the channel with a rank. As the I/O bus is shared, a memory controller serializes the accesses to different ranks of the same channel. The functions performed by the memory controllers are as follows:
  - —Convert a process or system request into a memory system command
  - —Generate a timing sequence by determining the priority of the command of the memory system
  - —Send commands such as read/write/refresh to the DRAM chip
  - —Retrieve or store data for the processor or system I/O
- Chip: A memory chip consists of DRAM banks and is an integrated circuit that can store data.
- Bank: As a component of a memory chip, a bank shares all internal data and a command bus within the chip. The bank is composed of circuits in which a capacitor stores electricity, and a transistor, which is a semiconductor device, acts as an amplifier. They are combined in the form of rows and columns.
- Command Bus: A command bus is used to transmit and receive data between a DRAM memory chip and a memory controller.

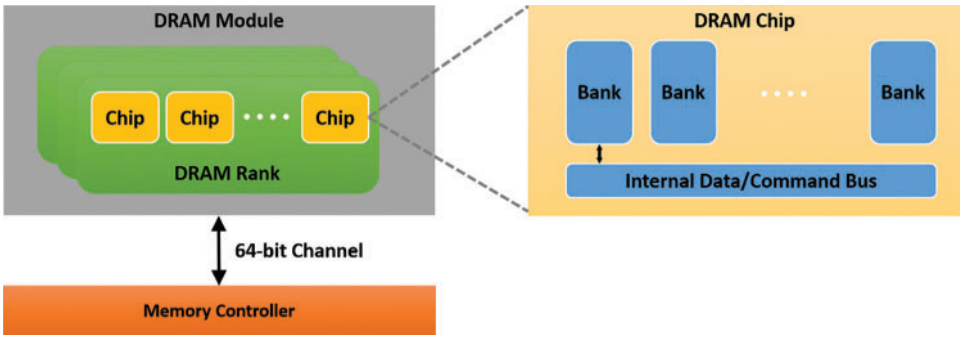



**Figure 1:** A type of DRAM-based system

As shown in Fig. 2, a DRAM chip consists of a wordline, which is a horizontal row, and a bitline which is a vertical column, with cells organized in rows and columns. In other words, DRAM chips are composed of a two-dimensional array of cells, and each cell has a transistor and a capacitor that stores data. When the capacitor is charged, the data are stored as a 1, and when it is discharged, the data is stored as a 0. As such, DRAM records the data by storing

charges in the capacitor. Because the capacitor leaks charge over time, DRAM requires a periodic refresh operation to restore its electrical charge in the cells.

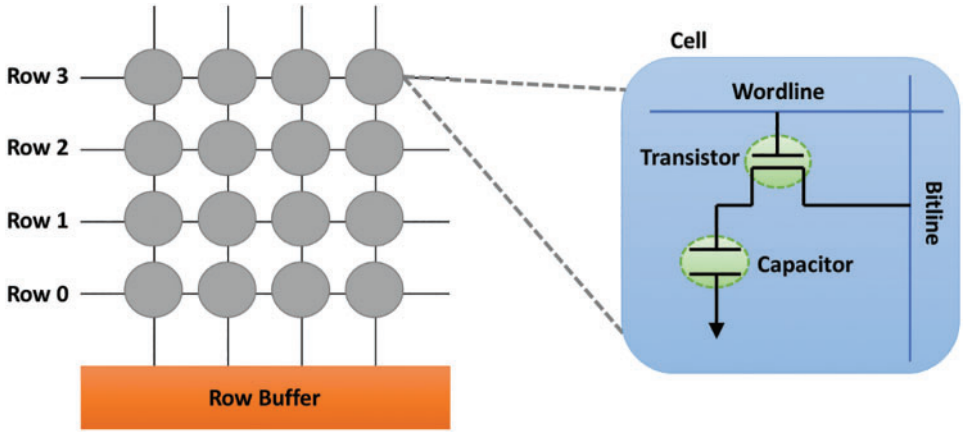



**Figure 2:** Architecture of DRAM chips

*2.1.2 Disturbance Error*

As technologies have advanced, the cell density of DRAM has increased and the cells are no longer isolated, causing them to electrically interact with each other [7]. Therefore, a refresh operation to restore charge in a DRAM cell caused by charge leakages affects the neighboring cells, causing disturbance errors. Recently, an attack called a rowhammer attack has been reported that applies a bit flip attack by deliberately generating such disturbance errors [1]. If disturbance errors can be induced, the rowhammers can be exploited in system-level attacks to escalate privileges, leak confidential data, and cause a denial of service. Accordingly, hardware manufacturers have recently adopted a target row refresh (TRR) to protect against rowhammer attacks and reduce the vulnerability of DRAM chips. In addition, memory controller and system manufacturers have implemented countermeasures such as increasing the refresh rate.

### *2.2 Mechanism of Rowhammer Attack*

The mechanism of a rowhammer attack involves deliberately generating a disturbance error in DRAM, as described ealier. Tab. 1 displays the assembly codes for generating disturbance errors. The bits that cause multiple flips are identified through these codes [1].

**Table 1:** Rowhammer attack code

| Code: | | |
|---|---|---|
| 1 | mov(X), %eax | //read values of address X and Y |
| 2 | mov(Y), %ebx | |
| 3 | clflush(X) | //evict data in the cache |
| 4 | clflush(Y) | |
| 5 | jmp code | //repeat above stage |

The steps involved in the rowhammer attack mechanism are as follows:

**Step 1:** Two mov instructions (codes 1 and 2 in Tab. 1) read data from the DRAM at addresses X and Y and load the data into the register and cache.

**Step 2:** Two clflush instructions (codes 3 and 4 in Tab. 1) evict the data that were loaded in the cache, enabling data to be read directly from DRAM rather than from the cache.

**Step 3:** Finally, the iteration of these instructions allows repeated hammering, a process of memory reading from DRAM, and thereby enables bit flips to be applied by causing disturbance errors.

To generate a disturbance error by accessing addresses X and Y as above, addresses X and Y must be mapped to the same bank and to different rows simultaneously. To deliberately generate a disturbance error by accessing the desired address, the address in the adjacent row of the same bank must be identified in advance. Accordingly, the virtual address needs to be mapped to the physical address. Therefore, a predictable method or a probabilistic method is used to identify the corresponding addresses. It is also necessary to bypass the cache because a quick activation of the rows in each bank of DRAM is essential to create the bit flips. To bypass the cache, it needs to clear the cache line using the clflush instruction. Consequently, the codes in Tab. 1 can be injected into other programs or used to intercept the system control by utilizing a bit flip attack.

### *2.3 Analysis of Mitigating Rowhammer Attacks*

Recently, studies have been conducted to mitigate rowhammer attacks by monitoring memory alteration using additional cells. In this section, previous studies on the mitigation of rowhammer attacks are described.

#### *2.3.1 Mitigating Techniques on Hardware*

The hardware-based techniques used for the mitigation of rowhammer attacks are as follows:

- CRA and PRA: Counter-based row activation (CRA) and probabilistic row activation (PRA) have been proposed to mitigate rowhammer attacks. CRA employs a counter to calculate the number of activations of each row, and if the corresponding counter exceeds the hammering threshold, a dummy activation is actively transmitted to refresh the data. PRA is used to reduce the overhead that is incurred during CRA, which facilitates the generation of a dummy activation probabilistically for all memory accesses. CRA and PRA require an additional counter to count the number of hammerings of the victim rows [8]. For example, if there is an 8 GB memory system with one million rows, 2 MB are required for the total counter size. Because this requires additional memory consumption, mitigation techniques using CRA and PRA can degrade the performance.
- GuardION: GuardION defends against direct memory access (DMA)-based attacks in an ARM environment. This method enables buffers to be physically isolated by adding two empty rows called guard rows. It eliminates the possibility of a bit flip because the victim rows can be arranged more than one row away from the attacked rows [9]. However, GuardION consumes additional DRAM memory of the user's device to add the guard rows, which can affect the device performance.
- ECC memory: To defend against rowhammer attacks, error correcting code (ECC) memory is introduced in each rank of DRAM. This method adds a parity bit for detecting errors occurring in data bits and a control bit for transmitting whether an error has occurred.

ECC memory was used to detect and correct errors that could occur owing to the external environment of DRAM, and the introduction of ECC memory became an obstacle for exploitation during a rowhammer attack. However, it has been reported that an attack called ECCploit can be used to conduct a rowhammer attack even in ECC memory-added DRAM [10].

- TRR: In response to rowhammer attacks, a target row refresh (TRR) has been implemented in a DDR4 model to refresh adjacent rows when an access is attempted beyond the threshold value. However, even in TRR-applied DRAM, rowhammer attacks can be reactivated by enabling bit flip attacks through new hammering patterns [11].

The aforementioned methods that use hardware to mitigate rowhammer attacks require additional hardware resources to detect or monitor errors, which can degrade the performance of existing devices. Furthermore, the rowhammer may be reactivated despite the application of rowhammer attack mitigation techniques to DRAM. Tab. 2 summarizes the hardware-based techniques used for mitigating rowhammer attacks.

**Table 2:** Analysis of mitigating rowhammer attacks based on hardware

| Mitigating technique | Description | Disadvantages |
|---|---|---|
| CRA and PRA | Counters are added to calculate the number of activations of each row to apply a data refresh based on the threshold of the number of hammerings | Degradation of memory performance from memory overhead and additional counters |
| GuardION | Guard rows are added to reduce the possibility of rowhammer occurrence through physical isolation of buffers | Degradation of memory performance from the use of memory resources in DRAM |
| ECC memory | A parity bit that detects errors in a data bit is added to detect and correct memory errors in DRAM | Degradation of memory performance from additional bit adoption and calculation, and the possibility of bypassing through ECCploit attack |
| TRR (target row refresh) | The number of row activations is calculated, and if the corresponding count exceeds the threshold, a data refresh is used | Possibility of bypassing through new hammering patterns such as TRRespass |

### *2.3.2 Mitigating Techniques Based on Software*

The software-based techniques used for mitigation of rowhammer attacks are as follows:

- ANVIL: ANVIL, a software-based method for mitigating rowhammers, detects rowhammer attacks by tracking the access location of DRAM using an existing hardware performance counter. Subsequently, victim rows within the vicinity are selectively refreshed to prevent hammering operations on the victim rows that are frequently accessed through rowhammer attacks [12]. However, because this method uses existing hardware counters to dynamically detect and respond to attacks, an additional system overhead may occur, which also requires modification in the Linux kernel [13].

- Technique using skewed hash tree: The occurrence of a bit flip is detected through a secure hash algorithm-3 (SHA-3) Keccack hash function-based dynamic integrity tree structure and a sliding window [14]. This has been proposed to generate minimal overhead and achieve cost efficiency, but requires a memory controller (MC) that stores the root hash and is not at risk of alteration.
- Technique using deep learning: Another study adopted a convolutional neural network (CNN) model, which is a deep learning model, to analyze the access patterns of DRAM in order to predict the rows where rowhammer attacks can occur in advance [13]. However, it requires exploiting various rowhammers in advance for analysis and training of the access patterns of DRAM.

As such, the techniques that use software to mitigate rowhammer attacks require the process of monitoring DRAM to adjust the refresh rate or predicting the rows in which an attack may occur by learning the DRAM access patterns. Tab. 3 below summarizes software-based techniques for mitigating rowhammer attacks.

**Table 3:** Analysis of mitigating rowhammer attacks based on software

| Mitigating technique | Description | Disadvantage |
|---|---|---|
| ANVIL | A rowhammer attack can be detected through the location of rows frequently accessed by DRAM using the existing hardware performance counters | System overhead may be inccured and a modification of the Linux kernel is required |
| Technique using skewed hash tree | Bit flips can be detected through a dynamic integrity tree structure and sliding window | Memory controller is needed to store the root hash and the safety of the MC should be ensured |
| Technique using deep learning | Rows, where a rowhammer attack could occur, can be predicted by learning the access patterns of DRAM | The training of patterns exploiting various rowhammers is required for deep learning |

### *2.4 IDA Python*

The interactive disassembler (IDA) is most widely known as a disassembler and is a tool that can be used to perform static analysis of binaries [15]. IDA Python is an extension of IDA, which uses the Python script to help with binary analysis. This study uses IDA Python to propose a static analysis and detection technique for rowhammer attack files, and the main modules used are as shown in Tab. 4.

**Table 4:** Module of IDA python

| Module of IDA python | Input | Output |
|---|---|---|
| FindText() | Address to start retrieval, string to retrieve, etc. | When the string is found, the address containing the corresponding string |
| GetFunctionName() | Specific address | The name of the function that the input address belongs to |
| GetDisasm() | Specific address | The assembly code that corresponds to the address received as the input |

## 3 Proposed Detection Technique of Rowhammer Attacks

This section analyzes the common features of various rowhammer attack files through a static analysis on such files that use x86-64 instructions. Based on this, we propose a technique for detecting rowhammer attacks.

### *3.1 Premise of the Dataset*

Files that trigger the rowhammers used for actual attacks are limited because of the difficulty of conducting an attack in a real-world environment. Furthermore, the number of proof-of-concept rowhammer files that verify the operability in a variable environment is limited. Among them, the attack codes of the rowhammer files using an x86-64 instruction are also limited. Currently, there are six reported attack codes [2,3,16].

### *3.2 Static Analysis for Detecting Rowhammer Attacks*

In this study, a static analysis was used to detect a rowhammer attack. The analysis was conducted to analyze the attack files without direct execution and facilitates the search for suspicious features and codes [17–19]. Accordingly, the number of times that the opcodes and application programming interfaces (APIs) were used was applied to analyze the features through static analysis based on rowhammer attack files using x86-64 instructions.

Fig. 3 shows the assembly codes of an actual rowhammer attack file. In an actual attack file, mov and clflush instructions access the same addresses to reload the cells of DRAM, and the number of iterations of the corresponding code can be analyzed. Tab. 5 shows the analysis results for the number of opcodes and APIs used in each attack file. Here, for mov and clflush, we analyzed the number of times the opcodes and API were used to access the same memory. In addition, for "open," we used the number of times the API accessed "/proc/self/pagemap."

As shown in Tab. 5, the common opcodes and APIs used in all attack files are mov, clflush, mmap, and open ("/proc/self/pagemap"), which enable hammering attacks. Furthermore, the opcodes and APIs used in each attack file are as follows:

- mov and clflush: Mov and clflush used in the static analysis proposed in this study deal with memory access, as shown in Tab. 1. Here, mov is an opcode used to access the input address, and clflush, an x86 instruction, is used to remove the cache line containing the address received as an input. Therefore, the re-load operation can be executed iteratively in the memory row of DRAM if the same memory is accessed repeatedly through the mov and the cache line is removed through clflush. In other words, they can affect the neighboring cells of DRAM to induce a bit flip, which can intentionally cause a disturbance error [1].

- mmap: Mmap requests the kernel to perform mapping according to the given memory length at the starting position of the given object [20]. Mapping is performed without an empty memory for a rowhammer attack through the corresponding API.
- cupid: Cpuid informs hardware information such as the model, manufacturer, and processor of the user device [20]. It is used to exploit rowhammers that operates under certain environments.
- rdtscp: Rdtscp is a command that returns the TSC value of the process. It is a serialization command that is executed sequentially. Thus, it does not facilitate parallel execution [20]. The corresponding command is used to measure the memory access time during a rowhammer attack.
- mfence: Mfence is a command that prevents the parallel execution of memory loading and storing of commandsexecution when consecutive memory commands are executed [20]. Thus, it ensures that the data are completely flushed.
- open: The open function for /proc/self/pagemap is used in the proposed technique. Attack files use the pagemap interface to find attack rows for each victim row [9].

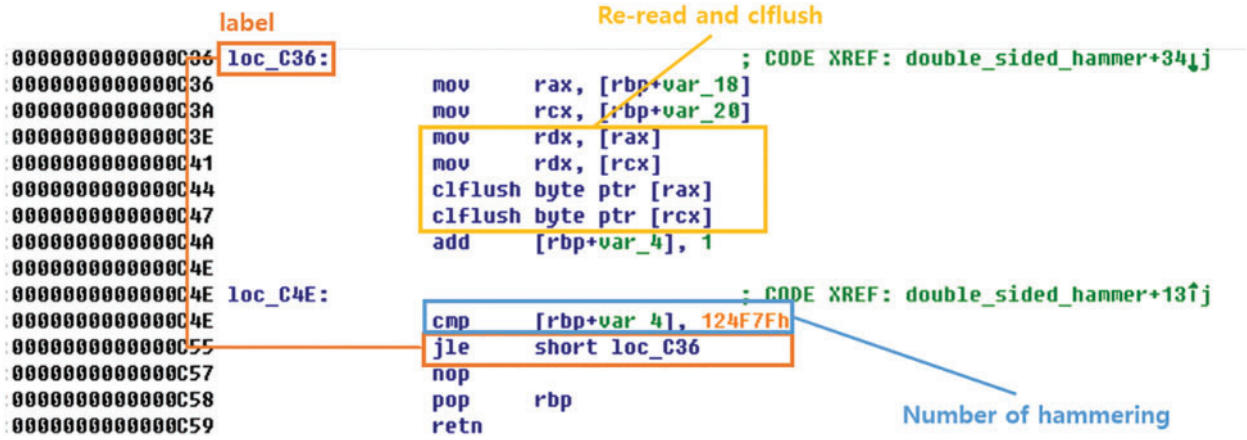



**Figure 3:** Assembly code of a rowhammer attack

**Table 5:** Number of opcodes and APIs from rowhammer attack file

| Attack file | mov | clflush | mmap | cpuid | rdtscp | mfence | open |
|---|---|---|---|---|---|---|---|
| Double side rowhammer | 1,024,000 | 1,024,000 | 1 | – | – | – | 2 |
| Run_rohammer | 1,200,000 | 1,200,000 | 1 | – | 1 | 2 | 1 |
| Pinpoint rowhammer | 1,200,000 | 1,200,000 | 1 | – | – | 1 | 1 |
| Rowhammer-haswsell | 1,000,000 | 1,000,000 | 1 | 1 | 1 | – | 1 |
| Rowhammer-ivy | 1,000,000 | 1,000,000 | 1 | 1 | 1 | – | 1 |
| Rowhammer-sandy | 1,000,000 | 1,000,000 | 1 | 1 | 1 | – | 1 |

### *3.3 Detection Technique of Rowhammer Attacks*

The technique proposed in this paper, shown in Fig. 4, is based on the number of opcodes and APIs, that are analyzed in the rowhammer attack files. It consists of mov, clflush, mmap, and open ("/proc/self/pagemap"), which enable hammering.

**Step 1** Open("/proc/self/pagemap"):

If the input file uses the open API and simultaneously accesses "/proc/self/pagemap" from the open API, it is determined that the input file uses pagemap. Because the use of pagemap alone cannot determine the rowhammer, the presence of a rowhammer is not determined in this step. If the input file access the pagemap, there is a possibility that it is searching the victim rows to conduct a rowhammer attack: hence, the process proceeds to the subsequent step (Step 2). This can be implemented as shown in Tab. 6, which can be used to determine whether the corresponding attack file uses "/proc/self/pagemap" through FindText, and whether the corresponding string is called from the open API through the GetFunctionName module.

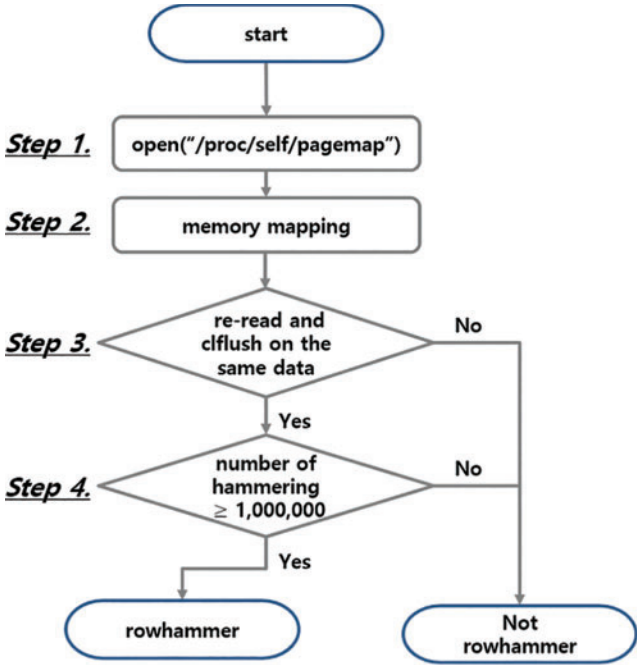



**Figure 4:** Flow chart for detecting rowhammer

**Step 2** Memory Mapping:

If the input file applies memory mapping using the mmap API function, it is determined that the input file conducts memory mapping. The mmap function requests the kernel to map the object pointed by the file descriptor into memory, which can be used not only in rowhammers but also in various executable files. Accordingly, because the use of nmap alone cannot determine the rowhammer, the presence of rowhammer is not determined in this step. Because there is a possibility of a pre-memory mapping step for executing the rowhammer if the input file applies memory mapping, the process proceeds to the subsequent step (Step 3). This can be implemented

as shown in Tab. 7, which determines whether mmap is included in the list of functions used in the corresponding attack file by using the GetFunctionName module.

**Table 6:** Pseudo code of Step 1

```
Find_pagemap:
1       WHILE start address < end address:
2         start address ← idc.FindText("/proc/self/pagemap")
3         IF start address does not exist, "/proc/self/pagemap" cannot be found
4         ELSE IF start address is called in "open," RETURN TRUE
5         start address ← next instruction address
6       RETURN FALSE
```

**Table 7:** Pseudo code of Step 2

```
Find_mmap:
1   FOR all functions used in input file:
2     func_name ← GetFunctionName(func)
3     IF func_name is "mmap," RETURN TRUE
5     ELSE, RETURN FALSE
```

**Step 3** Re-read and clflush on the same address:

If a memory read operation is conducted using the mov opcode for the same address, and the cache line is evicted by using the clflush opcode, this process is determined as a re-read operation. If it is determined as a re-read operation, the process proceeds to Step 4 to analyze the number of hammerings. Also, if it is decidedthat the re-read operation is not applied. it determines that the given file is a file other than a rowhammer file. This process is implemented as shown in Tab. 8. If the mov command is used in the same memory register in an adjacent location after confirming that clflush is used through the FindText module, it is determined as a re-read operation for hammering.

**Step 4** Number of hammerings $\geq 1,000,000$:

Step 4 is conducted to analyze the number of re-read operations in Step 3. In Step 4, the file is determined as a rowhammer file if the number of hammerings in which the re-read operation is repeated is 1,000,000 or more. If the number is less than 1,000,000, it is determined that the given file is not a rowhammer file. The threshold is set to 1,000,000 because the number of re-reads using mov and clflush in the read rowhammer file is 1,000,000 according to the static analysis of the rowhammer file, as shown in Tab. 5. Step 4 can be implemented as shown in Tab. 9, and the label of the basic block that operates the hammering of Step 3 is obtained through the Flowchart module. If the corresponding label uses a jump command such as jmp, the use is determined to be repeated, and the number of iterations through the cmp command before the jump command is checked.

**Table 8:** Pseudo code of Step 3

```
Find_hammering:
1 while start address < end address:
2   start address ← idc.FindText("clflush")
3    IF start address does not exist, "clflush" cannot be found
4    ELSE
5     mov address ← idc.FindText("mov")
6     IF the corresponding address does not exist, a re-read is not conducted
7     ELSE
8      IF any command corresponding to mov address accesses a register such as clflush,
9       go to number_hammering
10     ELSE, re-read is not performed
11     start address ← next instruction address
12 RETURN
```

**Table 9:** Pseudo code of Step 4

```
Number_hammering:
1       Label ← Label of basic block in which mov and clflush are located
2       IF a label is called in a jump statement
3         cmp address ← idc.FindText(cmp statement prior to a jump statement)
4         IF the comparison number of "cmp" is 1,000,000 or more
5          RETURN it is a rowhammer
6         ELSE
7          RETURN it is not a rowhammer
```

## 4 Analysis of Proposed Detection Technique of Software-Induced Rowhammer Attacks

- Malicious File: any malicious software executable file, other than rowhammer attack file
- Normal File: any normal software executable file, other than rowhammer attack file

To evaluate the method proposed in this study, test sets were established, as shown in Tab. 10, including six attack files exploiting a rowhammer, six malicious files excluding a rowhammer, and six normal files such as a chrome-sandbox. A rowhammer attack file is an attack file that uses an x86-64 instruction, the proof-of-concept code of which was obtained from GitHub. Subsequently, the corresponding code was compiled to obtain a total of six files. Malicious files without a rowhammer were also obtained from GitHub, and these files included a spectre attack, which is a software-induced hardware attack. A spectre attack was included in the input file because clflush, which is used in rowhammer exploits, was applied owing to the existence of Flush + Reload attack [21]. Application programs that can be typically obtained were used as normal files. For all input files, files using the same x86 instruction as the selected rowhammer attack file were used. Tab. 10 shows the files used to analyze the proposed detection method.

**Table 10:** Input for analysis of proposed detection technique

| Type of file | Index | Name of file |
|---|---|---|
| A. Rowhammer attack file | 1 | Double side rowhammer |
| | 2 | Run_rohammer |
| | 3 | Pinpoint rowhammer |
| | 4 | Rowhammer-haswsell |
| | 5 | Rowhammer-ivy |
| | 6 | Rowhammer-sandy |
| B. Malicious file | 1 | Spectre |
| | 2 | Eggshell |
| | 3 | Big_file_writer |
| | 4 | Fork_bomb |
| | 5 | Mem_killer |
| | 6 | Sleeper |
| C. Normal file | 1 | Apt |
| | 2 | Basename |
| | 3 | Chrome-sandbox |
| | 4 | Dpkg |
| | 5 | gettext |
| | 6 | lsattr |

### *4.1 Computation Environment*

Tab. 11 shows the computation environment applied for evaluating the proposed technique.

**Table 11:** Computation environment

| Operating system | Windows 10 pro |
|---|---|
| CPU | Intel® Xen® W-2123 |
| RAM | 64 GB |
| IDA Pro | version 6.8.150423 (64-bits) |
| Python | version 3.8 |

### *4.2 Detecting Simulation Results*

When the results of the rowhammer attack files were analyzed through the proposed detection technique of the rowhammer attack described in Section 3.3, the following results were derived. When the rowhammer detection technique proposed in this study was applied, six actual rowhammer attack files, which corresponded to true positives, were detected as rowhammer files, and 12 rowhammer attack files, which corresponded to true negatives, were detected as non-rowhammer files, as shown in Tab. 12. Furthermore, none of the results was detected as a false positive or false negative, indicating that false detections and non-detections did not occur in the rowhammer detection technique. Thus, the comparison with the input of the evaluation confirmed that all input values were normally detected.

**Table 12:** Confusion matrix of evaluation

| Confusion matrix | | True condition | |
|---|---|---|---|
| | | Condition positive | Condition negative |
| Predicted condition | Predicted condition positive | 6 | 0 |
| | Predicted condition negative | 0 | 12 |

*4.2.1 Condition Positive—Predicted Condition Positive (True Positive)*

According to the results of detecting the actual rowhammer files (A.1–A.6 in Tab. 10) as a rowhammer file, the results for six input rowhammer files are identified as shown in Fig. 5.

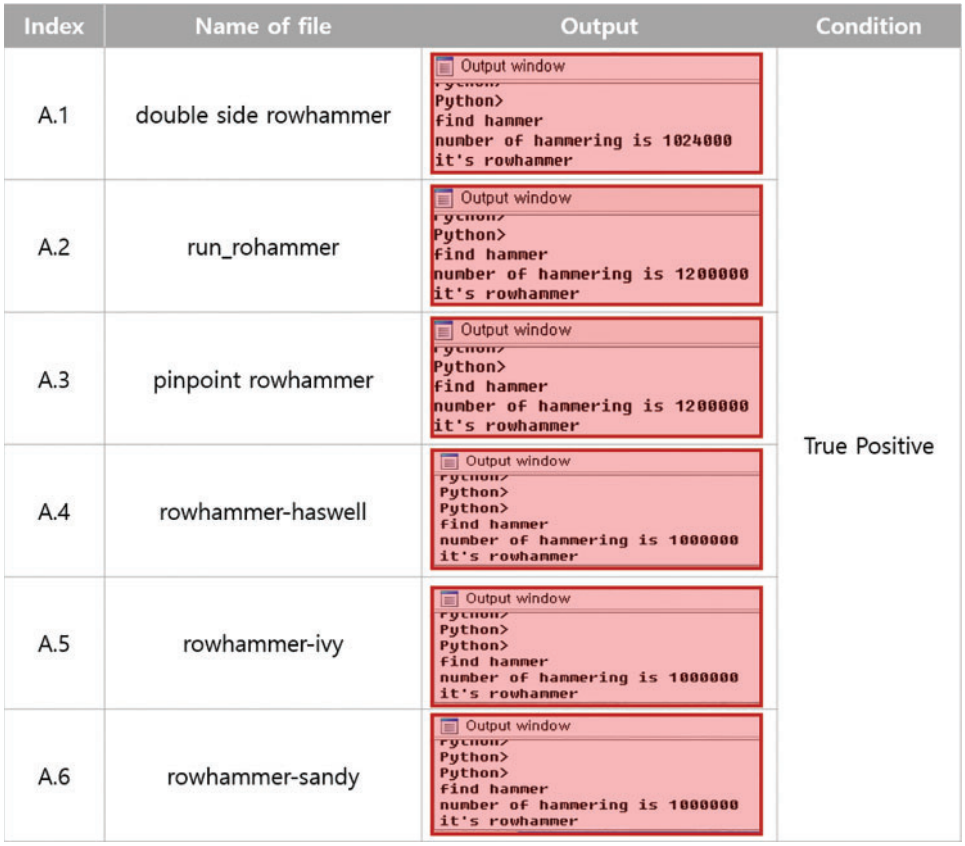



**Figure 5:** Results of applying the proposed technique to the rowhammer files (A.1∼A.6)

As shown, all rowhammer attack files executed in the x86-64 instruction were accurately detected. All rowhammer attack files were confirmed as using the API that accessed the pagemap and conducted memory mapping through open ("/proc/self/pagemap"). Furthermore, by identifying the mov and clflush commands that access the same register in all attack files, it was possible to determine whether the re-read operation was performed. Although the number of re-read hammerings was different for each attack file, the detection was possible because the number of hammerings was determined to be 1,000,000 or more.

*4.2.2 Condition Negative—Predicted Condition Negative (True Negative)*

According to the results of detecting non-rowhammer files (B.1–B.6, C.1–C.6 in Tab. 10) as non-rowhammer files, the normal files, or malware files other than the rowhammer files, were not detected as rowhammer attack files because of the missing process of accessing pagemap used to generate a rowhammer attack, such as in Figs. 6 and 7. Moreover, the results of analyzing the number of hammerings for the corresponding files confirmed that the hammering process for executing the bit flip was not applied.


**Figure 6:** Results of applying the proposed technique to the malicious normal files (B.1∼B.6)

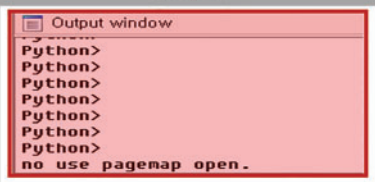
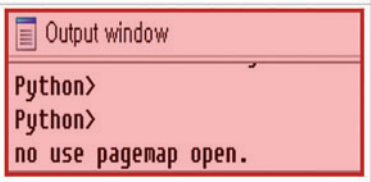
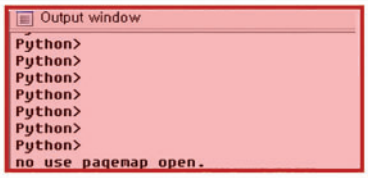
| Index | Name of file | Output | Condition |
|---|---|---|---|
| C.1 | apt |  | True Negative |
| C.2 | basename |  | |
| C.3 | chrome-sandbox |  | |
| C.4 | dpkg |  | |
| C.5 | gettext |  | |
| C.6 | lsattr |  | |



**Figure 7:** Results of applying the proposed technique to the normal files (C.1∼C.6)

- Condition Positive—Predicted Condition Positive: Result of detecting an actual rowhammer file as a rowhammer file (true positive)
- Condition Negative—Predicted Condition Positive: Result of detecting an actual non-rowhammer file as a rowhammer file (false positive)
- Condition Positive—Predicted Condition Negative: Result of detecting an actual rowhammer file as a non-rowhammer file (false negative)
- Condition Negative—Predicted Condition Negative: Result of detecting an actual non-rowhammer file as a non-rowhammer file (true negative).

Tab. 12 shows the results of deriving the confusion matrix through the input values provided in Tab. 10 for evaluation. No results were confirmed to be Condition Negative—Predicted Condition positive (false positive) or Condition Negative—Predicted Condition Negative (false negative), indicating that the technique proposed in this study can successfully detect a rowhammer attack.

## 5 Conclusion

According to recent studies, next-generation DRAM chips are vulnerable to rowhammer attacks, which can be carried out on various platforms (e.g., Android, ARM, and Linux) and environments (e.g., JavaScript, and network packets), which can be used in various attacks such as kernel privilege escalation, full control, and encryption key tampering. Thus, to mitigate rowhammer attacks, various studies are being conducted, such as adjusting the refresh rate of the DRAM chip, or adding a guard row by monitoring the number of row activations. However, these techniques require continuous monitoring, which can cause overhead or additional hardware resources, resulting in performance degradation. Although efforts are being made to cope with rowhammer attacks, despite the performance degradation and overhead, it is known that it is extremely difficult to mitigate these attacks while generating less overhead in terms of efficiency [6]. Moreover, it is known that the selective refresh-based rowhammer mitigation techniques have difficulty in establishing an optimal refresh rate [22]. In addition, DRAM and memory controller companies are applying countermeasures to devices to cope with rowhammer attacks. Nevertheless, attacks such as ECCploit and TRRespass have appeared to be able to bypass the countermeasures. As process technology scales have continued to increase, it has become more difficult to mitigate rowhammer attacks. Thus, it is necessary to cope with rowhammer attacks through research detecting rowhammers in advance without inducing overhead.

The proposed technique applies a static detection method for rowhammers using an x86-64 instruction, which is the first study on detecting rowhammers in advance because rowhammers can be exploited in attacks such as a privilege escalation and encryption key alteration. In this paper, we applied six datasets to simulate the proposed technique. Datasets that trigger the rowhammers used for actual attacks are limited because of the difficulty of conducting an attack in a real-world environment. Furthermore, the number of rowhammer files that were collected was limited to the number of proof-of-concept files that proved the method operation in a variable environment. Among them, the attack codes of the rowhammer files using an x86-64 instruction were also limited.

If the proposed static-analysis-based detection technique is applied based on an opcode and API analysis using various platforms such as Android and ARM, which have been proven to be exploitable by rowhammers, the method will help with the classification of existing malicious codes as well as a pre-detection of rowhammer attacks operating in various environments. Furthermore, to cope with various software-induced hardware fault attacks as well as rowhammer attacks, studies on detection prior to attack file exploitation of file vulnerabilities are needed, as are studies on adding hardware resources or hardware monitoring.

**Funding Statement:** This work was supported by a National Research Foundation of Korea (NRF) Grant funded by the Korean government (MSIT) (No. NRF-2017R1E1A1A01075110).

**Conflicts of Interest:** The authors declare that they have no conflicts of interest to report regarding the present study.

## References


[1] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee *et al.,* "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *Proc. ACM/IEEE 41st Int. Sym. on Computer Architecture*, Minneapolis, MN, USA, pp. 361–372, 2014.

[2] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," 2015. [Online]. Available: https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[3] D. Gruss, C. Maurice and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in JavaScript," in *Proc. International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, San Sebastián, Spain, vol. 9721, pp. 300–321, 2016.

[4] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice *et al.,* "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, Vienna, Austria, pp. 1675–1689, 2016.

[5] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos *et al.,* "Throwhammer: Rowhammer attacks over the network and defenses," in *Proc. USENIX Annual Technical Conf.*, Boston, MA, USA, pp. 213–226, 2018.

[6] O. Mutlu and J. S. Kim, "RowHammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2019.

[7] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson *et al.,* "Improving DRAM performance by parallelizing refreshes with accesses," in *Proc. IEEE 20th Int. Sym. on High Performance Computer Architecture*, Orlando, FL, USA, pp. 356–367, 2014.

[8] D. H. Kim, P. J. Nair and M. K. Qureshi, "Architectual support for mitigating row hammering in DRAM memories," *IEEE Computer Architecutre Letters*, vol. 14, no. 1, pp. 9–12, 2015.

[9] V. Van Der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna *et al.,* "GuardION: Practical mitigation of DMA-based rowhammer attacks on ARM," in *Proc. Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, Saclay, France, vol. 10885, pp. 92–113, 2018.

[10] L. Cojocar, K. Razavi, C. Giuffrida and H. Bos, "Exploiting correcting codes: On the effectiveness of ECC memory against rowhammer attacks," in *Proc. IEEE Sym. on Security and Privacy*, San Francisco, CA, USA, pp. 55–71, 2019.

[11] P. Frigo, E. Vannacci, H. Hassan, V. Van Der Veen, O. Mutlu *et al.,* "TRRespass: Exploiting the many sides of target row refresh," in *Proc. IEEE Sym. on Security and Privacy*, San Francisco, CA, USA, pp. 747–762, 2020.

[12] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks *et al.,* "ANVIL: Software-based protection against next-generation rowhammer attacks," *ACM SIGPAN Notices*, vol. 51, no. 4, pp. 743–755, 2016.

[13] A. Chakraborty, M. Alam and D. Mukhopadhyay, "Deep learning based diagnostics for rowhammer protection of DRAM chips," in *Proc. IEEE 28th Asian Test Sym.*, Kolkata, India, pp. 86–91, 2019.

[14] S. Vig, S. Bhattacharya, S. K. Lam and D. Mukhopadhyay, "Rapid detection of RowHammer attacks using dynamic skewed hash tree," in *Proc. Int. Workshop on Hardware and Architectural Support for Security and Privacy*, New York, NY, USA, pp. 1–8, 2018.

[15] C. Eagle, *The IDA pro book*, 2nd ed., San Francisco, USA: No Starch Press, 2011.

[16] S. Ji, Y. Ko, S. Oh and J. Kim, "Pinpoint rowhammer: Suppressing unwanted bit flips on rowhammer attacks," in *Proc. ACM Asia Conf. on Computer and Communications Security*, New York, NY, USA, pp. 549–560, 2019.

[17] S. Anwar, M. F. Zolkipli, V. Mezhuyev and Z. Inayat, "A smart framework for mobile botnet detection using static analysis," *KSII Transactions on Internet and Information Systems*, vol. 14, no. 6, pp. 2591–2611, 2020.

[18] J. Hwang, J. Kwak and T. Lee, "Fast k-NN based malware analysis in a massive malware environment," *KSII Transactions on Internet and Information Systems*, vol. 13, no. 12, pp. 6145–6158, 2019.

[19] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, USA, pp. 156–168, 2001.

[20] U. Degenbaev, "Formal specification of the x86 instruction set architecture," Ph.D. dissertation, University of Saarland, Germany, 2012.

[21] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss *et al.,* "Spectre attacks: Exploiting speculative execution," in *Proc. IEEE Sym. on Security and Privacy*, San Francisco, CA, USA, pp. 1–19, 2019.

[22] J. S. Kim, M. Patel, A. G. Yagikci, H. Hassan, R. Azizi *et al.,* "Revisiting rowHammer: An experimental analysis of modern DRAM devices and mitigation techniques," in *Proc. ACM/IEEE 47th Annual Int. Sym. on Computer Architecture (ISCA)*, Valencia, Spain, pp. 638–651, 2020.