**Rowhammer Final Project Report**  
Darasimi Ogunbinu-Peters  
CptS 426 – Spring 2025

**Introduction**

The Rowhammer vulnerability exploits the physical properties of DRAM memory, where repeatedly accessing (or "hammering") specific rows can cause adjacent rows to experience unintended bit flips. This project aims to study Rowhammer attacks, attempt to reproduce bit flips, and analyze the effectiveness of various mitigation techniques.

**Attempted Approach**

Initially, I attempted to clone and compile several public Rowhammer test programs, including Google's rowhammer-test and CMU-SAFARI's Rowhammer repository. However, due to limitations in the WSL2 environment — specifically missing kernel headers (sys/io.h) and lack of direct DRAM access — these projects could not compile or execute properly.  
As a result, I proceeded to implement a C program that meets the assignment requirements: allocating memory, initializing values, hammering memory using CLFLUSH, and testing for bit flips.

**Program Design**

A custom C program (rowhammer\_final.c) was created to:

* Allocate a 256MB memory buffer using mmap()
* Initialize the buffer with a user-defined pattern (0xFF or 0x00)
* Hammer two adjacent memory addresses repeatedly
* Use the CLFLUSH instruction to flush cache lines
* Check for bit flips after hammering
* Accept three command-line arguments:
  + Initial pattern (init\_value\_hex)
  + Hammer count (hammer\_count)
  + Number of tests (test\_count)

**Flowchart**

**Program Flowchart:**

Start

↓

Parse command-line arguments (init\_value, hammer\_count, test\_count)

↓

Allocate 256MB memory buffer (mmap)

↓

Initialize memory with init\_value

↓

For each test:

- Randomly select two adjacent addresses

- Hammer the addresses hammer\_count times

- Flush addresses using CLFLUSH

- Check if bit flip occurred

↓

Free memory (munmap)

↓

End

**Answers to Assignment Questions**

**1. Does the program use CLFLUSH?**

✅ Yes. The program uses the \_mm\_clflush intrinsic from <x86intrin.h> to flush cache lines and force direct DRAM access.

**2. What is CLFLUSH? Why was it included in the instruction set?**

CLFLUSH is a low-level CPU instruction that invalidates and flushes a cache line to main memory (DRAM). It forces subsequent accesses to fetch from RAM instead of cache.  
**Purpose:** It ensures data coherency and supports low-level memory operations. Manufacturers included CLFLUSH to help developers control cache behavior, especially in applications requiring real-time or consistent memory views.

**3. Does the program allocate a large buffer? How?**

✅ Yes. The program allocates 256MB using mmap() with MAP\_ANONYMOUS and MAP\_PRIVATE flags. This directly maps virtual memory to the process without any file backing.

**4. How does the program initialize the memory with a known value?**

✅ The program uses memset() to fill the entire buffer with a user-specified initial value (0xFF or 0x00) before hammering.

**5. How does the program carry out the attack?**

✅ For each test:

* Randomly selects two adjacent memory addresses
* Accesses them alternately and repeatedly (hammering)
* Flushes them with CLFLUSH between accesses
* After hammering, checks if bit values have changed

**Results and Observations**

* **Environment:** WSL2 on an HP laptop (Windows Subsystem for Linux 2)
* **DRAM Type:** Not directly identifiable, assumed LPDDR4 or newer (based on device specs)
* **Findings:** No bit flips observed across multiple runs.
* **Theory:** Bit flips were not detected likely because:
  + WSL2 virtualizes memory and shields direct DRAM access.
  + Modern DRAM modules have built-in Rowhammer mitigation techniques like TRR (Target Row Refresh).

**Screenshots / Code used**

#define \_GNU\_SOURCE

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <string.h>

#include <x86intrin.h>

#include <unistd.h>

#include <sys/mman.h>

#include <time.h>

#define BUFFER\_SIZE (256 \* 1024 \* 1024) // 256MB

void flush(void\* addr) {

    \_mm\_clflush(addr);

}

int main(int argc, char\* argv[]) {

    if (argc != 4) {

        printf("Usage: %s <init\_value\_hex> <hammer\_count> <test\_count>\n", argv[0]);

        return 1;

    }

    uint8\_t init\_value = (uint8\_t)strtol(argv[1], NULL, 16);

    int hammer\_count = atoi(argv[2]);

    int test\_count = atoi(argv[3]);

    printf("[\*] Allocating %dMB of memory...\n", BUFFER\_SIZE / (1024 \* 1024));

    uint8\_t\* buffer = mmap(NULL, BUFFER\_SIZE, PROT\_READ | PROT\_WRITE,

                           MAP\_ANONYMOUS | MAP\_PRIVATE, -1, 0);

    if (buffer == MAP\_FAILED) {

        perror("mmap failed");

        return 1;

    }

    printf("[\*] Initializing buffer with 0x%02X...\n", init\_value);

    memset(buffer, init\_value, BUFFER\_SIZE);

    srand(time(NULL));

    for (int t = 0; t < test\_count; ++t) {

        size\_t offset = (rand() % (BUFFER\_SIZE - 128));

        volatile uint8\_t\* a = buffer + offset;

        volatile uint8\_t\* b = buffer + offset + 64;

        printf("[\*] Test %d: hammering addresses %p and %p...\n", t + 1, a, b);

        for (int i = 0; i < hammer\_count; ++i) {

            \*a;

            \*b;

            flush((void\*)a);

            flush((void\*)b);

        }

        // Check for flips

        if (a[0] != init\_value || b[0] != init\_value) {

            printf("[!!] Bit flip detected at offset %zu\n", offset);

            printf("     a[0] = 0x%02X, b[0] = 0x%02X\n", a[0], b[0]);

        } else {

            printf("[OK] No bit flip at this location.\n");

        }

    }

    munmap(buffer, BUFFER\_SIZE);

    return 0;

}

A computer screen shot of a blue screen

AI-generated content may be incorrect.

**mmap() vs malloc()**

| **Function** | **Description** |
| --- | --- |
| mmap() | Maps memory directly into the process' address space. Offers control over memory behavior, alignment, and page properties. Suitable for low-level memory manipulation like Rowhammer. |
| malloc() | Allocates heap memory managed by the C runtime. Does not provide control over mapping or memory properties. Simpler but less powerful for direct hardware attacks. |

**Mitigation Techniques for Rowhammer**

* **Increased Refresh Rates:** Refresh memory rows more often to reduce vulnerability windows.
* **ECC Memory:** Use Error-Correcting Code (ECC) RAM to detect and correct bit flips.
* **TRR (Target Row Refresh):** Automatically refresh adjacent rows when suspicious activity is detected.
* **Rowhammer-aware Memory Controllers:** Implement hardware-level defenses against repeated access patterns.

**Conclusion**

Although real-world bit flips were not observed in this environment, the experiment successfully simulated the hammering behavior and tested DRAM cells for vulnerability. The limitations of using WSL2 instead of bare-metal Linux systems were noted and documented. Through this project, I gained valuable hands-on experience with DRAM vulnerabilities, low-level memory manipulation, and the challenges of real-world exploitation.