Chaos in Memory: A Comprehensive Analysis of Register and Stack Variable Corruption

Andrew James Adiletta

Submitted to the faculty of the WPI ECE department as part of the completion of Master’s of Science in Electrical and Computer Engineering

August 2023
# Contents

1 Introduction  

2 Background  
   2.1 Target Row Refresh (TRR)  
   2.2 Countermeasures in Crypto Libraries  
   2.3 ASLR  

3 Stack Rowhammer  
   3.1 Flipping Bits in the Stack Variables  
      3.1.1 Offline Memory Profiling  
      3.1.2 Online Attack Phase  
      3.1.3 Synchronizing Code Execution with Rowhammer  
      3.1.4 Flipping Bits using Rowhammer  
   3.2 Bypassing Stack ASLR  
      3.2.1 Controlling the Page Offset with ASLR disabled  
      3.2.2 Page Fault Side Channel  
      3.2.3 Remapping Pages Side Channel
Abstract

The past decade has been marked by a multitude of uncovered vulnerabilities within microarchitectures, leading to new attack vectors and spurring an extensive investigation into potential countermeasures. Particularly, the architectural and physical imperfections within DRAM memories have initiated Rowhammer attacks, which provide a pathway to manipulate a victim’s memory space via bit flips. While a considerable body of research has proposed measures to mitigate or nullify the effects of Rowhammer, its full exploitability scope remains underexplored.

This thesis pushes the frontier by exploring an innovative exploitation of Rowhammer attacks. The approach involves inducing faults in a victim process’s stack variables and register values by forcing a task switch. This switch results in the context being stored in the process stack, which, when stored in memory, becomes vulnerable to a Rowhammer attack. Accomplishing such an exploit involves navigating several complex challenges, such as stack pages co-location, ASLR offset randomization, and synchronization. This thesis covers extensive experimentation which resulted in several intriguing findings. Notably, an observed non-random behavior in ASLR offset randomization could potentially facilitate the acceleration of stack co-location.
To illustrate the practical ramifications of these findings, this thesis includes examples of their application, such as bypassing SUDO, SSH authentications, MySQL, and other cryptographic libraries. Thus, this work uncovers a new, potent attack vector, underscoring the necessity for ongoing research into potential vulnerabilities and their countermeasures.
Chapter 1

Introduction

Critical vulnerabilities intrinsic to our computing infrastructure have been unveiled with the rise of attacks like Meltdown [33] and Spectre [29]. Such microarchitectural leakages have served as a foundation for numerous studies to further explore and exploit these inherent flaws [17, 25, 54, 8, 52].

Exploitation of Rowhammer Fault Injection  In contrast to the passive nature of these vulnerabilities, the advent of Rowhammer has equipped potential attackers with a proactive tool for fault injection in DRAM memories [28, 46]. The fundamental premise of Rowhammer is the possibility of inducing bit flips in neighboring rows of a DRAM memory when a single row is accessed repeatedly.

Real-world applications of Rowhammer have demonstrated its potency in live attack scenarios. Examples range from achieving root access via opcode
flipping in the `sudo` binary [18], to breaking `OpenSSH` public-key authentica-
tion [44], to performing a Bellcore attack on a CRT-based RSA implementa-
tion within `WolfSSL` to recover secret keys [56]. The versatility of Rowham-
ner has been further emphasized through its successful remote application
via JavaScript [19, 14], network execution [50, 34], and applicability in cloud
environments [57, 10] and heterogeneous FPGA-CPU platforms [56].

In addition to DRAM, research has also unveiled the susceptibility of
Flash memories to Rowhammer-like cell-to-cell interference [7]. This vulner-
ability, when exploited to target file-system pages, has shown potential for
privilege escalation [31].

**Compromising CPU Internals**  Despite the considerable effort invested
in advancing Rowhammer attacks and countermeasures, a persistent belief
has held that CPU internals are immune to software-based fault injection at-
tacks. More specifically, the SRAM-based registers and caches have typically
been deemed secure from fault injection, barring direct physical manipula-
tions like laser fault injection attacks. Conversely, CPU-external devices such
as DRAM memories are perceived to be highly susceptible to physical tam-
pering, a viewpoint rooted in the early era of Trusted Computing and further
fueled by the advent of Cold-boot attacks [21].

Challenging this long-standing notion, this thesis provides evidence that
CPU internals, including register values, are not invulnerable. Until this
point, Rowhammer attacks have predominantly aimed at corrupting dynam-
ically allocated memory [38] or disk-stored binaries loaded into memory [46].

This research demonstrates that an attacker can manipulate register values to be saved to the stack, and subsequently flushed to memory, thus rendering them susceptible to Rowhammer fault injection. Once faulty values are reloaded into the registers, execution resumes.

However, the program’s stack is not an easy target. Several challenges must be overcome: achieving co-location with stack pages, negating the stack offset randomization imposed by the Address Space Randomization Layer (ASLR), and synchronizing with the vulnerable code. These complexities form the focus of this thesis, offering new insights into the potential vulnerabilities of CPU internals.

The Vulnerability of the Stack Not only flushed register values but also certain vulnerable code sections within the program stack, such as security checks and authentication states, can be exploited. Corruption of these sensitive variables could lead to privilege escalation. It’s noted that crypto libraries often either minimize or entirely avoid the use of dynamic memory and stack, a practice adopted either for compatibility with constrained environments, for safety-critical systems such as embedded or Real-Time Operating Systems (RTOS), or to reduce the exposure of potentially vulnerable internal secrets.

Consider the wolfSSL library, which supports compilation options that preclude the use of dynamic memory. In contrast, the NaCL library ab-
stains from both dynamic memory and variable-size stack allocation. As such, crypto library implementations depend heavily on registers and stack variables. This thesis brings attention to the fact that such variables are not impervious to fault injection, implying that the attack surface of Rowhammer is broader than previously thought.

**Attempts at Rowhammer Security** The considerable risks posed by Rowhammer have sparked numerous attempts at countermeasures, with efforts directed at both detection [24, 9, 58, 22, 42, 20, 4, 12] and mitigation [19, 53, 6]. Regrettably, these strategies have been deemed ineffective as demonstrated in [18].

Moreover, the study [11] revealed that hardware-enabled error checking integrated into many memory devices (ECC) is not impervious to being bypassed. Even the hardware countermeasure known as Target Row Refresh (TRR) fell short of providing full protection, as revealed in recent research [16]. This finding was further supported by [14], which claimed that a staggering 80% of market-available DRAM chips remain vulnerable to Rowhammer attacks.

Adding to the complexity, new forms of hammering such as HalfDouble hammering have emerged, shown to successfully circumvent TRR countermeasures [30]. The persistence of these vulnerabilities in the face of numerous countermeasures underscores the formidable challenge Rowhammer presents.
Thesis Contributions to The Field

This thesis systematically analyzes the threat imposed by Rowhammer fault injections to stack variables and registers which were previously considered secure against Rowhammer. Specifically, it

• Introduces a novel attack to inject faults into register values and to target stack variables;
• Shows how static code/data allocation can be manipulated with bait pages to achieve co-location with the victim’s stack;
• Studies ASLR offset randomization and demonstrate experimental results confirming non-random behavior;
• Introduces new synchronization techniques to enable practical means to target stack and register via Rowhammer;
• Demonstrates attacks on SUDO, OpenSSH, and MySQL;
• Highlights new RSA Bellcore vulnerabilities enabled by the attack vectors discovered in OpenSSL;
• Outlines mitigative coding styles to minimize the attack surface against the newly introduced attack vectors.
Thesis Structure

The remainder of this thesis is organized as follows. Section 2 provides the necessary background information related to the explored attack. Section 3 outlines the threat model associated with the attack. The methods employed for flipping bits in stack variables are discussed in Section 3.1. Section 3.2 exposes the leakage in the page offset and investigates potential means to exploit stack ASLR.

Section 3.3 elaborates on the extension of the attack on stack memory, explaining how CPU register values can be manipulated using Rowhammer. In Section 3.4, an exploration of how relaunching can be utilized to exploit ASLR on systems with fewer flippable locations is presented. Section 4 delves into the experimental evaluation.

Further discussion on the findings and results concerning OpenSSL, sudo, and OpenSSH is found in Section 5. An analysis on RSA Bellcore Attacks on OpenSSL and MySQL is given in Section 5.4. Finally, Section 7 proposes several potential countermeasures against the attack under study.
Chapter 2

Background

Every individual bit of data within DRAM is accommodated in memory cells that make up an array. The circuitry of each memory cell consists of a capacitor and a transistor, facilitating the storage of a single bit. Arranged in rows across one axis of the grid are word lines. Each of these word lines links every cell located on the same row. The memory cell is connected to two bit lines, designated as "+" or "-" bit lines [23], which intersect the word lines at a right angle, establishing connections with cells situated above and below.

The mechanism to hold a memory cell at a high or low voltage is facilitated by a sense amplifier, which causes positive feedback when one bit-line is elevated to a high level and the other is decreased to a low. However, to allow the cell to be written to, disconnection of the sense amplifiers is required.

Illustrated in Figure 2.1 is the DRAM array architecture, which includes
Figure 2.1: DRAM array architecture illustrated with bit line, word line, and DRAM cell illustrated. Simplified to show a total of 21 DRAM cells [2].

The operation of the sense amplifier is key in the function of the Rowhammer exploit, which takes advantage of this mechanism. A sense amplifier is composed of two interconnected inverters placed between the memory cells’ bit lines [45]. These inverters are also referred to as not gates, which are logic gates that invert the input; for instance, an input of one to an inverter results in a zero, and vice versa.

However, the miniaturization of DRAM cells, despite the stabilizing effect of the sense amplifier, has led to reduced noise margins. Consequently, electronic noise can lead to errors in the memory cells [37]. This reduction in size means cells hold less charge, thereby increasing the proportion of noise in relation to the cell’s total charge.

The process of reading a cell involves disconnecting the sense amplifiers
and raising the word-line of the target row. If the targeted cell carries a value of 1, the capacitor within the cell charges the bit line. After reconnection of the sense amplifiers, the outputs of all the sense amplifiers in that row are latched. Subsequently, current flows back up the bit lines to replenish the storage cell. Upon completion of the read operation from the target row, the word line is turned off [35].

Due to the inherent voltage leakage in capacitors over time, DRAM necessitates a "refresh" every 64ms or less, as specified by the Joint Electron Tube Engineering Council (JETEC) that formulates hardware standards for manufacturers [1].

The process of reading a cell can inadvertently cause voltage variations in nearby cells. In DDR3 memory, this phenomenon has been observed, where successive reads lead to voltage fluctuations, potentially causing erroneous data to be written into memory cells during a refresh [27]. DDR, or double data rate, is a memory architecture that enables a single transfer per clock cycle. Utilizing both the rising and falling edges of the clock signal, data transfer can happen at twice the usual rate. For instance, DDR-200 and DDR-400 have an IO bus speed of 100Mhz and 200Mhz respectively, whereas DDR2-800 and DDR3-1600 have IO Bus speeds of 400Mhz and 800Mhz respectively [26]. As memory devices accelerate in speed, more consecutive reads can take place between refresh intervals. This makes high-speed, high-density DDR3 an ideal target for Rowhammer attacks.

Process variation in semiconductor manufacturing has been on the rise
due to the shrinking of transistor sizes. The length, width, and oxide thickness for integrated circuits are variable for the manufacturing process, and can sometimes lead to unpredictable behavior. Generally, to avoid errors designers will run simulations during the validation process to determine if the outputs of a circuit will be within a certain tolerance.

Manufacturing DRAM can be complex, and testing each individual cell in the DRAM array is not feasible, although recent work from [15] has proposed machine learning approaches for analyzing raw trace data and building predictive models for fault detection. The manufacturing process also takes advantage of SVIDs (status variables identification) which monitor things like temperature, pressure, voltage during manufacturing. [15] has proposed models that selectively identify SVIDs that are most correlated with faults, and use them to build predictive models.

However, even with attempted improvements in manufacturing, with increasing DRAM densities the chance for bit flips and reliability failures is increasing. Hence, to retain data every DRAM row has to be continuously refreshed usually with 64 msec intervals. Although refreshing the rows periodically helps preventing the data corruption, Kim et al. [28] showed that frequent access to the neighbor rows causes faster charge leakage, which effectively causes bit flips before the next refresh. This is known as the Rowhammer effect [28]. Seaborn et al. [46] introduced the double-sided Rowhammer flipping the victim cells even faster.

Gruss et al. [18] introduced one-location hammering and achieved root
access with opcode flipping in `sudo` binary in 2018. Gruss et al. [19] and Ridder et al. [14] have shown that Rowhammer can be applied through JavaScript remotely. Tatar et al. [50] and Lip et al. [34] have proved that it can be executed over the network. Rowhammer is also applicable in cloud environments [57, 10] and heterogeneous FPGA-CPU platforms [56]. In 2020, Kwong et al. [32] demonstrated that Rowhammer is not just an integrity problem but also a confidentiality problem.

There have been many efforts on Rowhammer detections [24, 9, 58, 22, 42, 20, 4, 12] and neutralization [19, 53, 6]. Gruss et al. [18] have shown that all of these countermeasures are ineffective. Cojocar et al. [11] in 2019 showed that the ECC countermeasure is not secure either. Another hardware countermeasure Target Row Refresh (TRR) has also been recently bypassed by Frigo et al. [16]. This work was extended by Ridder et al. [14] to attack TRR-enabled DDR4 chips from JavaScript and claim that more than 80% of the DRAM chips in the market are still vulnerable to Rowhammer. Quite recently, hammering beyond adjacent locations was shown [30] to be effective in circumventing TRR mitigations.

## 2.1 Target Row Refresh (TRR)

There have been many efforts to detect Rowhammer by tracking consecutive reads to adjacent rows in a Serial Presence Detect (SPD) chip Intel CPUs deploy a mitigation known as pseudo-TRR or pTRR, which reads the
Maximum Activation Count, or MAC value from the SPD and if reads to consecutive rows reaches the MAC value, the Intel CPU refreshes the row.

A multisided attack works even with TRR enabled, with a strategy based on the Trrespass multisided attack [16].

In Figure 2.2, we see that there are a number of attacker rows and a number of victim rows in the multisided attack. The diagram shows a multisided attack with 4 attacker rows.

2.2 Countermeasures in Crypto Libraries

Physical fault injection attacks are well known among crypto practitioners [5]. Crypto libraries, especially ones designed for embedded platforms, have implemented countermeasures since the early 2000s. For instance, *OpenSSL*
implements error checks in CRT-based exponentiation to thwart Bellcore attacks [5]. Still, fault injection has proven effective in [48] to corrupt Elliptic Curve Parameters in the OpenSSL library. Further, [36] demonstrated a Rowhammer fault injection vulnerability in WolfSSL that resulted in ECDSA key disclosure. The fault was injected during the signing operation with private ECC keys, which occur during a TLS handshake between client and server. WolfSSL addressed this vulnerability by implementing a series of checks during each stage of the signing process to detect if data has been tampered with, and WOLFSSL_CHECK_SIG_FAULTS was released as a security measure[40]. Importantly, these checks that protect dynamic memory operate on the idea that variables in the stack are safe from Rowhammer, which this paper will demonstrate is not the case.

2.3 ASLR

Address-space layout randomization (ASLR) is often used as a primary defense against memory corruption attacks. ASLR arranges the address space of a process randomly to prevent a user from targeting a specific area of code. It is supposed to rearrange the stack, heap, and libraries of an executable in a non-deterministic way. In theory, if an attacker finds a way to corrupt the memory it should not have access to, it should not be able to target any particular area in the process.

ASLR has been shown to be vulnerable in the past, typically through the
Listing 2.1: Page offset randomization for Stack memory in Linux Kernel

```c
unsigned long arch_align_stack(unsigned long sp) {
    if (!(current->personality & ADDR_NO_RANDOMIZE) && randomize_va_space)
        sp -= get_random_int() % 8192;
    return sp & ~0xf;
}
```

use of software-side weak points such as memory disclosure vulnerabilities that reveal run-time addresses [13]. More recent attacks have also shown that ASLR can be broken through the use of EVICT+TIME cache attacks that can derandomize address spaces by correlating cache line addresses with page-table entries [17]. Importantly, these attacks on ASLR do not circumvent stack ASLR, which is implemented in the Linux kernel as shown in Listing 2.1. Stack ASLR should randomize the base address of the stack resulting in random variable offsets as seen in Figure 2.3 to the point a brute force attack randomly flipping bits in the system would be ineffective.
Figure 2.3: Histogram of page offset of a stack variable in stack memory out of 100K trials.
Chapter 3

Stack Rowhammer

We will explain the attack scenarios in detail for each attack target in Section 5. In line with the previous Rowhammer attacks [28, 18, 57, 10, 19], we assume attacker-victim co-location in the same system. Co-location is a common assumption for many micro-architectural side-channel attacks [33, 29, 8, 52, 54]. We assume the operating system works as intended without any compromise in its integrity and the attacker has user privileges throughout the paper. We assume the attacker does not have access to any service that reveals the physical address or DRAM addressing information. Our attack does not require huge-page configuration and works with standard-size pages.
3.1 Flipping Bits in the Stack Variables

For the Rowhammer attack on DDR4 memory, we perform a multisided attack to circumvent TRR protection. We found that a multisided attack with 11 rows was most effective at getting flips on our system and we used \texttt{mfence} to prevent out-of-order execution. It is possible that without \texttt{mfence} CPU optimizations would disrupt the critical order that rows are accessed for the multisided attack which would prevent the attack from working. We found 1M accesses of all the rows were optimal in getting flips and reducing profiling/online time. We also found that doing 100 iterations of 1M accesses along with 100K \texttt{nops} in between also improved the efficacy of the attack in getting flips.

3.1.1 Offline Memory Profiling

Rowhammer requires that rows in DRAM are adjacent to each other physically. We achieve this through the use of the SPOILER and Row Conflict attacks. We use SPOILER \cite{spoiler} because it leaks virtual to physical address translation without the need to read the \texttt{pagemap} file, which would require root access. SPOILER takes advantage of a microarchitecture optimizations speculative loads and store forwarding. For finding addresses that are within the same bank, we use row conflicts \cite{row_conflict}, which is another timing side channel that we exploit to colocate memory for Rowhammer.
Profiling for Contiguous Memory  SPOILER first allocates a large buffer in the memory of the attacker program. The memory from this buffer is distributed throughout the DRAM randomly. Within a window of the memory buffer, SPOILER writes zeros to all the addresses, then times how long it takes to load the first entry in the array. These timings are plotted in Figure 3.1; physical memory dependency requires more cycles to complete and thus would appear as peaks on the graph. For every system, the threshold values of SPOILER need to be adjusted. These threshold values include the timing required to call a memory read an outlier in the dataset (a timing measurement above a certain value is probably the result of a system interrupt or some other event rather than physical continuity), and a timing threshold value to qualify a value as a peak and thus part of the continuous memory buffer.

In our experiments, we generally looked for about 3-5% of our memory allocated to SPOILER to be physically continuous. This means that if we allocated 1024 MBytes of memory to our buffer, we would expect to find around 32-64 bytes of continuous memory. This varies depending on the experiment and the machine the experiment is running on.

Finding Rows in the Same Bank  In addition to finding memory that is physically continuous, the memory must also be in the same bank of the DRAM for Rowhammer to work. We use the row conflict side channel to leak DRAM information which, like SPOILER, does not require root access.
Figure 3.1: Timing peaks found by SPOILER. Equidistant peaks indicate physical continuity in memory.

Rowconflict reads from the first address in the physically continuous memory buffer, then it reads from address $n$ (where $n$ is 1 through the length of the memory buffer) and calculates the time difference between reads. A larger time difference indicates that the row buffer within the DRAM bank needed to be cleared and thus caused a spike in timing. Just like SPOILER, row conflict needs threshold values to be experimentally determined and defined for each machine.

### 3.1.2 Online Attack Phase

**Bait Pages** In order to flip a variable in the stack of a program, the page that the variable is located in needs to be placed in a page that has a flippable bit at the correct page offset. During the profiling stage, a page is determined to have bits that are flippable by Rowhammer, and that page is freed using
munmap. The victim process is then launched and it then fills the last freed page (the page with the flippy bits).

However, suppose we are targeting a specific variable within the process. In that case, there are a number of pages used by the victim process that is irrelevant to our attack and would fill our flippy page before the page with our target variable. Thus, we must release bait pages, which are filled with the victim’s data that is irrelevant to the attack first. We then release our flippy page, which the victim process will place our target variable in. The number of bait pages that need to be released depends on the process and if ASLR is enabled. The pseudo-code for releasing bait pages is given in Listing 3.1.

Listing 3.1: Pseudo code showing how pages can be forced into a specific area in memory using a mapping-unmapping technique

```c
buffer = mmap(baitPages * PAGESIZE)
munmap(flippyPageAddr, PAGESIZE)
for (i = 0; i < bait_pages; i++)
    munmap(&buffer[i*PAGESIZE], PAGESIZE)
```

You can see in Listing 3.1 that we release the bait pages before the flippy page. This is because the Linux Buddy Allocator algorithm that is used to allocate memory to different processes effectively acts like a last-out-first-in system, where the latest pages released to memory are used first.
3.1.3 Synchronizing Code Execution with Rowhammer

Experimental Method  In order for Rowhammer to successfully attack the stack of a program, the variables being attacked need to be loaded into memory at the right time. For experimental purposes, we used signals to make sure that the programs were synchronized. In Linux, signals are often used to report the occurrence of an event – these are often exceptions but do not have to be. In our version of Linux, there are a total of 64 different signals that can be sent to a process. We used the `SIGRTMAX-6` signal to synchronize the attack and victim programs because it was not currently being used by either. The experimental design for signal synchronization works as follows:

1. Attacker process waits to receive a signal from the victim that the target variable has been created and is sitting in the stack
2. The victim process sends the signal that the variable has been created (unblocking the attacker process) and waits for a signal from the attacker process that the attack has completed.
3. The attacker process attacks the victim process, then sends a signal to the victim process to allow it to continue after the attack has completed.

This setup is ideal for an experimental design setup because it guarantees that the victim process will be at the proper point in execution during the attack.
Practical Method  In practice, the victim process cannot be altered to send a signal when it is ready to be attacked and wait for a signal that the attack has finished. Instead, we can use the SIGSTOP signal to stop the program’s execution and create a probabilistic model to determine if the process has stopped in the correct place in the process execution to attack the variables. After the variables have been attacked, the SIGCONT signal can be sent to continue its execution. For an attacker to have permission to send a signal, it must belong to the same session. This is special to the SIGCONT signal.\(^1\)

Attacking Processes During a Blocking Window  The most optimal scenario for an attack is for vulnerable code to have a blocking window where the process is waiting for an event that may be triggered by the attacker. For SUDO, this could be the period where the process is waiting for the attacker to enter a password. The process saves state data to stack while waiting for the user to submit a password. High level examples of synchronizing blocking codes are:

- Password Input
- IP Socket Connections
- Signal Interrupts
- Media Uploads
Looking at figure 3.2, we can see that there is a blocking period during the attack window. This allows for synchronization between the victim and attacker so there is no longer a question of probability of the attacker will launch the Rowhammer attack at the right time. Practically, we can see an example of this in Chapter 6 of synchronization on a real-world TLS handshake.

### 3.1.4 Flipping Bits using Rowhammer

The final step in the workflow after the target variable is loaded into memory is to actually flip the bits in the variable. While the profiling step allowed us to evaluate which bits were flipped in a row, because we do not control the area of memory being flipped, we cannot see which bits specifically were flipped. However, generally, the success of an attack can be determined by checking the new state of the process. For example, if the attack objective was achieved the attacker may bypass password authentication.

---

1. [https://www.sudo.ws/docs/man/1.8.10/sudo.man/](https://www.sudo.ws/docs/man/1.8.10/sudo.man/)
3.2 Bypassing Stack ASLR

Initially, it would seem that ASLR makes running a stack attack difficult. However, profiling a process to determine the number of bait pages required to be released to the system reduces the entropy significantly.

The physical address is split into two parts; the page number and the page offset. The number of total bits in a physical address is calculated as $\log_2(p)$ where $p$ is the total size of the main memory. For a system with 8 GB of main memory, the physical address is $\log_2(8\text{GB})$ or 33 bits. Our operating system was also fragmenting memory into indivisible 4 KB-sized pages, which can be represented as 12 bits. This means that in the physical address of a system with 8 GB main memory and 4KB pages, the first 21 bits represent the page of the physical address, and the last 12 bits represent the offset within the page.

With our bait pages attack, we can effectively remove the entropy of the first 21 bits of randomization by forcing the base address to be placed on a known page around 45% of the time, according to our findings. This leaves the last 12 bits of randomization to deal with. Through experimentation, we noticed that the entropy in the last 12 bits can be further reduced. We found that the last 4 bits of the address always stayed the same. When attacking OpenSSL, for example, we noticed that the last 4 bits always had a value of 0x8. If the variable we are attacking is a 4-byte variable, then there are only four possibilities for the last 4 bits of an address to be potentially vulnerable;
Figure 3.3: The relation between the number of bait pages vs. page offset of a stack variable. When the page offset is large, the number of bait pages is significantly higher (shown in red).

\[ n, n + 1, n + 2, \text{ and } n + 3, \] where \( n \) is the starting address of the variable. This further reduces our ASLR entropy to a mere 8 bits, which can be easily exhausted. We found a relationship between the number of bait pages required to be released by an attacker program to locate a variable in the Rowhammer page correctly and the offset that variable appears within that page. We believe this to be a novel discovery because part of the intention of ASLR is to randomize the page offset.

We found this relationship by unmapping pages in our attacker program and recording their physical address in a list, then in our victim program, determining where our target variable appears in the list, as well as the page offset address of the target variable (the last 12 bits). While this relationship was different for each program, it was clear that there was always a smaller
set of data points where the number of bait pages clearly limited the number of possible page offsets. We created a graph of this relationship in Figure 3.3. We can see from the graph that if 180 bait pages were required to be released to mount the victim variable in vulnerable memory correctly, then the page offset of the said variable would be around 4000. Likewise, if the number of bait pages is 40, it can be assumed that the page offset is going to be somewhere between 0-2500. It should not be possible to find patterns in page offset because ASLR intends the offset to be based on random number generation as the offset is masked by a randomly generated value, as seen in Listing 2.1.

To understand the root cause of this unusual behavior, we investigate the following methods that leak information about the page offset.

3.2.1 Controlling the Page Offset with ASLR disabled

We investigate the dependency between the number of bait pages and the page offset of a stack variable in a more controlled environment. We create the following function where a buffer with a predefined BUFFER_SIZE before integer variable var.

```c
void main(){
    char buffer[BUFFER_SIZE] = {0};
    int var = 0;
}
```
Figure 3.4: The dependency between the number of bait pages (black) and page offset (red) when ASLR is disabled. The page offset of the variable is manually controlled by changing the size of the buffer. The jump in the # bait pages and page offset occurs at the same point.

Note that both the buffer and the variable are stored in the stack. We disable ASLR in the system to make sure we have full control on the page offset of the variable. In Figure 3.4, we vary the BUFFER_SIZE variable from 0 to 4K. Increasing the size of the buffer pushes the variable back in the stack and linearly decreases the page offset. We control the page offset by varying the size of the buffer. We also observe the number of required bait pages has a sudden change together with the page offset of the variable. We speculate this behavior is caused by crossing the page boundaries while increasing the BUFFER_SIZE and results in an increase in the number of total pages consumed by the program. Next, we investigate the same dependency with ASLR enabled.
3.2.2 Page Fault Side Channel

We found that monitoring for page faults gives us a side channel to determine the offset set by ASLR. A page fault will happen when a process requests data from a page in memory that is not currently loaded in DRAM. When the page fault occurs, the page needs to be moved from the swap space in the storage to DRAM. There are two types of page faults; major faults and minor faults. Major faults occur when a page is requested that does not exist in memory and needs to be brought back from the swap space. A minor fault is less performance degrading and occurs when the page is currently in memory and needs to be swapped back out to the disk (usually to free up space in DRAM for other pages).

Looking at Figure 3.5, we can see that if the process receives 275 page faults (marked in red), we can guarantee that the location of the offset in DRAM is going to be somewhere between 200 and 800, which reduces the search space and randomization of the ASLR offset bits by more than a factor of 6. Additionally, if 286 page faults are detected, we know that the offset will generally not be between 200-800, which also reduces the search space.

We are not sure why this side channel exists, but we speculate that the randomized page offset throws page faults which we can monitor using performance monitoring by the attacker. It is important to note that this performance monitoring, e.g., the /usr/bin/time command, do not require special permission to run and thus are practical to use in a real attack.
3.2.3 Remapping Pages Side Channel

One technique we used was page remapping, where we would unmap $n$ pages of our attacker program, launch our victim process, then remap $n$ pages back to our attack program. If we unmapped 500 pages, launched our victim process, then remapped 300 pages back to our attacker program, we would assume that the number of bait pages our victim required was 200 pages. We found a slight correlation in the data, but ultimately it was too noisy to be useful. We speculate this is because remapping pages pulls from unpredictable pools of memory, so the number of pages is not zero-sum.
3.3 Flipping Bits in CPU Registers

3.3.1 Flipping Bits in Stack

The stack is a memory section that software processes use to store values temporarily. The location of the last variable inserted into the stack is saved in the stack pointer registers. In assembly language, the stack can be used freely to store variables. However, higher-level languages and compilers use a convention that is based on the architecture and the operating system. These conventions set rules for converting C code into an assembly code, such as System V i386, System V x86_64, Microsoft x64, and ARM. Each convention uses different registers for function inputs and return variables, and in certain cases the convention also uses the stack to store temporary variables. This makes the variables vulnerable to fault attacks by using Rowhammer on the stack. Now we will summarize the convention to show which situations cause the compiler to use stack for variable storage. Since our setup is focused on Linux, we will focus on System V x86_64. Below, we will discuss two methods to attack the stack variables. The first method discusses a stack attack that occurs naturally by the usage of the stack when a compiler uses the C convention to create executable code. In the second scenario, we discuss a case which the code does not use stack to store variables, but the OS itself uses the stack to store the registers to change the executed process. This way, we can actually force the OS to temporarily store the register values on the stack and launch Rowhammer to inject fault to any register indirectly.
3.3.2 Intel-Ubuntu C convention

The architecture uses 16 64-bit registers which are referred to as \texttt{rax}, \texttt{rbx}, \texttt{rcx}, \texttt{rdx}, \texttt{rbp}, \texttt{rsp}, \texttt{rsi}, \texttt{rdi}, \texttt{r8}-\texttt{r15}. Some of the registers are special purpose registers, e.g. \texttt{rsp}: register stack pointer and others are generic/scratchpad registers. When a C code is compiled and converted into assembly code the following convention is used for functions:

- \texttt{rax} holds the return value of the function
- \texttt{rdi}, \texttt{rsi}, \texttt{rdx}, \texttt{rcx}, \texttt{r8}, \texttt{r9} holds the input parameters of the function. If there are more than 6 input parameters, rest is written into the stack.
- \texttt{rax}, \texttt{rdi}, \texttt{rsi}, \texttt{rdx}, \texttt{rcx}, \texttt{r8-11} are used as scratch registers.
- \texttt{rax}, \texttt{rdi}, \texttt{rsi}, \texttt{rdx}, \texttt{rcx}, \texttt{r8-11} are caller-saved registers. This means that if a routine calls a subroutine, it is the responsibility of the main routine to preserve the values of any relevant registers, as the subroutine is free to modify them. To do this, the calling function can save these values in other registers that will not be changed during the subroutine call or save them on the stack.
- \texttt{rbx}, \texttt{rsp}, \texttt{rbp}, \texttt{r12-15} are callee-saved registers. When a routine makes a subroutine call, it is the responsibility of the subroutine to ensure that the values of the relevant registers remain unchanged after the subroutine call is completed. To achieve this, the subroutine pushes the contents of these registers onto the stack and then restores the original values when it has finished executing by popping them from the stack.
• When a function call has a large number of variable declarations, compilers aim to utilize as many registers as possible to store these values in order to optimize performance. However, when the number of available registers is insufficient to hold all the variables, the compilers will resort to using the stack to store the excess variables.

When the compilers use the above-mentioned convention, the excess variables are stored on the stack if a function has many variables. This makes the variables vulnerable to stack attacks. Our inspection of disassembled code of common libraries shows that these cases are less common as compilers aim to reduce stack usage, but there is still a possibility. Of course, the attack can only be executed if the targeted variable is written to the stack. To enable the stack attack, we can force processes to temporarily store register contents on the stack during the execution of another process. This expands the scope of the attack beyond just variables stored on the stack.

3.3.3 Forcing Register Eviction to Stack

In order to support concurrent processing, the OS schedules each process for a specific amount of time, switching between them as needed. While each OS uses different scheduling protocols, they all typically save the contents and state of CPU (including registers) to a stack in order to allow a process to resume from where it left off when it is reloaded. The location where the CPU state is stored may vary depending on the OS and the type of interrupt that occurs. In some system calls, the contents of some registers are saved
Figure 3.6: We can evict registers to stack by switching contexts which flushes the registers to cache, and then with clflush we can flush the cache to DRAM where data can be flipped with Rowhammer
to the kernel stack associated with the process. However, our experiments have shown that when we interrupt a victim process, many general-purpose registers are saved to the user code stack.

This process switching actually enables a new type of attack, where we can target variables that are stored in registers. As seen in Figure 3.6 even though the variables may not be stored in the stack during the compilation convention (as discussed previously), the victim process can be made to switch processes by running another process or making a system call. This will result in the victim process storing its CPU registers in the stack, making
the variables vulnerable to a Rowhammer attack. By forcing the system to switch processes at the right time, we can target any register and initiate a Rowhammer attack.

3.4 Exploiting Offset Randomization

Although ASLR is built as a security measure to prevent memory attacks, it can be exploited to make the Rowhammer attack more powerful. We propose a technique named relaunching to exploit ASLR for Rowhammer.

The attacker first profiles the memory to find a flippy bit location in memory. In some DRAMs, these flippy locations may be rare. For some Rowhammer enabled attacks, that require a specific bit in the page to be flippy, the attack will become less viable. In our attack, instead, we first find a flippy bit in memory, then perform the following steps:

1. After finding a flippy bit location, the attacker frees memory to the system containing the number of bait pages followed by the flippy page;
2. The attacker launches the victim process which fills the recently deallocated pages;
3. The attacker performs the Rowhammer attack on the victim process (not knowing if the flippy bit aligns with the bits required to be flipped in the victim process);
4. The victim process ends and the attacker process remaps the memory
used by the victim process back to itself and repeats the attack with the
same flippy row.

With this approach, theoretically, the attacker only needs to find a single
flippy bit in the whole system for the attack to work. This is a dramatic
improvement over other Rowhammer attacks where extensive profiling is re-
quired, and often thousands of flips are required before a successful attack.

Relaunching works because ASLR will put the variable into a new location
in the page the next time it runs. This means that rather than looking for
a new flippy bit that might collocate where a flip is needed in the victim
process, the victim can simply be relaunched and ASLR reshuffles the variable
somewhere else, potentially into the location where it can be flipped by the
flippy bits in the page.
Chapter 4

Experimental Results

The experiments are conducted on a system with Ubuntu 20.04.01 LTS with 5.15.0-58-generic Linux kernel installed. The system uses an Intel Core i9-9900K CPU with a Coffee Lake microarchitecture. We used a dynamic clock frequency rather than a static clock frequency to improve the practicality of the attack. End-to-end attack experiments are done on a single DIMM Corsair DDR4 DRAM chip with part number CMU64GX4M4C3200C16 and 16GB capacity. DRAM row refresh period is kept as 64ms which is the default value in most systems. For the experiments on \texttt{sudo}, we use version 1.9.12p1 \footnote{\texttt{sudo} git commit number 3396267291328eccfcb7bf8b1729c77f30216513}, which is the latest \texttt{sudo} version at the time of this work. We use the portable \texttt{OpenSSH} library with version 9.1p1 \footnote{\texttt{OpenSSH} git commit number 0ff46f2ee2ffcc4daf45e679e484da8f6f338c} for SSH experiments. To better accommodate the server environment and reduce the noise caused by desktop applications, we use the OS in console mode.
4.1 Reproducibility of Bit Flips

Until this work, the reproducibility of bit flips induced by Rowhammer was not analyzed in detail. Therefore, it was not known whether each flippy location has equally reproducible or not. As the target size gets closer to a page size, every bit flip found is potentially useful since it will land on the target. However, as the target requires more precision, it is harder to find aligned bit flips; therefore, it is critical to attempt only when we find highly reproducible bit flips. This way, we can put the burden on the offline memory profiling phase and keep the online time as short and accurate as possible. To test the reproducibility of bit flips, we select a 64 MB physically contiguous memory buffer. In DDR3, we apply double-sided Rowhammer and slide the Attacker-Victim-Attacker window by one at every step. Once we finish the buffer, we store the bit flip locations and start the same process from the beginning. We hammer the same memory buffer for 100 times and count the number of flips for each bit location that has flipped at least once. We found 1667 unique flippy bit locations in total. Figure 4.1 illustrates the frequency of bit flips in a heat map. We observe that only a limited portion of found bit flips are actually reproducible, while most of them are not reproducible at all in 100 trials.
Figure 4.1: The comparison of heat maps of bit flips in DDR3 and DDR4 DRAM chips. Darker color illustrates the locations of more reproducible bit flips. The bit flips seen in DDR4 are less reproducible than DDR3.

4.2 Evaluation on Different DRAM Chips

Both offline and online phases of our attack require finding bit locations that are vulnerable to Rowhammer attack. Since the bit flip frequency depends heavily on how flippy a DRAM chip is, we evaluate our attack on different DRAM chips from both DDR3 and DDR4 memory profiles. We have taken 14 DDR3 memory profiles from [49], and we generated the remaining 6 memory profiles on our DRAM chips. In total, we have analyzed 20 DRAM chips. The results are summarized in Table 4.1. In the last column, we can see the probability of finding at least one flip in a 32-bit integer after profiling 0.1% of the total memory. To calculate this probability, we first find $n_{avg}$, the average number of bit flips that land on a 32-bit variable for 256 possible page offsets.
Then, we calculate the probability of having a successful attack with a single flippy page by dividing the average flip count, $n_{avg}$ by the total number of flippy pages, $n_{flippy}$. Finally, for a stealthy attack, we assume we only use 0.1% of the total memory size, $N_{pages}$. The final fault probability is calculated as $p_{fault} = (1 - (1 - n_{avg}/n_{flippy})^{N_{pages}/1000}) \times 100$. While probabilities are over 90% for most DRAM chips, it is important to note that other factors affect the probability of seeing a flip in the target variable of an actual process, including the probability that the process gets loaded into the flippy page in the first place.

### 4.3 Success Rate of Baiting Method

During our experiments, we found that with ASLR enabled, we could successfully locate the target page into the flippy row about 50% of the time. With further engineering efforts, this number can be brought up to 80% [32]. In Section 5, we refer to the ability to locate our target page into the flippy row as our bait-page success rate, as we deallocate a set number of bait pages for the system in the hopes it will force our target variable into a vulnerable place in memory.
<table>
<thead>
<tr>
<th>Brand</th>
<th>Serial Number</th>
<th>Size (GB)</th>
<th>fault (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corsair</td>
<td>CMD16GX3M2A1600C9</td>
<td>16</td>
<td>99.99</td>
</tr>
<tr>
<td>Corsair</td>
<td>CML16GX3M2C1600C9</td>
<td>16</td>
<td>99.99</td>
</tr>
<tr>
<td>Corsair</td>
<td>CML8GX3M2A1600C9W</td>
<td>8</td>
<td>99.99</td>
</tr>
<tr>
<td>Corsair</td>
<td>CMY8GX3M2C1600C9R</td>
<td>8</td>
<td>97.26</td>
</tr>
<tr>
<td>Crucial</td>
<td>BLS2C4G3D1609ES2LX0CEU</td>
<td>8</td>
<td>72.34</td>
</tr>
<tr>
<td>Geil</td>
<td>GPB38GB1866C9DC</td>
<td>8</td>
<td>99.95</td>
</tr>
<tr>
<td>Goodram</td>
<td>GR1333D364L9/8GDC</td>
<td>8</td>
<td>57.47</td>
</tr>
<tr>
<td>GSkill</td>
<td>F3-14900CL8D-8GBXM</td>
<td>8</td>
<td>90.44</td>
</tr>
<tr>
<td>GSkill</td>
<td>F3-19200C10-8GBZHD</td>
<td>8</td>
<td>99.99</td>
</tr>
<tr>
<td>GSkill</td>
<td>F3-14900CL9D-8GBSR</td>
<td>8</td>
<td>88.76</td>
</tr>
<tr>
<td>Hynix</td>
<td>HMT351U6CFR8C-H9</td>
<td>8</td>
<td>99.77</td>
</tr>
<tr>
<td>V7</td>
<td>V73T8GNAJKI</td>
<td>8</td>
<td>45.17</td>
</tr>
<tr>
<td>PNY</td>
<td>MD8GK2D31600NHS-Z</td>
<td>6</td>
<td>92.58</td>
</tr>
<tr>
<td>Integral</td>
<td>IN3T4GZNZBIX</td>
<td>4</td>
<td>79.19</td>
</tr>
<tr>
<td>Samsung</td>
<td>M378B5173QH0</td>
<td>4</td>
<td>69.67</td>
</tr>
<tr>
<td>Samsung</td>
<td>M378B5773DH0</td>
<td>2</td>
<td>99.69</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DDR4</th>
<th>Brand</th>
<th>Serial Number</th>
<th>Size (GB)</th>
<th>fault (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corsair</td>
<td>CMU64GX4M4C3200C16</td>
<td>64</td>
<td>99.99</td>
<td></td>
</tr>
<tr>
<td>Corsair</td>
<td>CMK32GX4M2B3200C16</td>
<td>32</td>
<td>99.98</td>
<td></td>
</tr>
<tr>
<td>GSkill</td>
<td>F4-3600C16D-16GVKC</td>
<td>16</td>
<td>99.99</td>
<td></td>
</tr>
<tr>
<td>Crucial</td>
<td>CT8G4DFD824A.C16FF</td>
<td>8</td>
<td>90.47</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: The probability of flipping at least one bit in a 32-bit integer calculated on 16 different DDR3 chips and 4 DDR4 chips per profile (128 or 256MBs). In our setup, it takes 95 minutes to profile a 128 MB on DDR3 and 480 minutes to profile 256MB on DDR4 chips.

### 4.4 Flipping Bits in CPU Registers

We were able to successfully inject faults into the values stored in registers by first evicting them from the registers, hammering them, and allowing the process to load the values back into the registers. Table 4.2 demonstrates the results of the register attack experiments.

The register attack demonstrated that in 32 minutes, enough pages were profiled to enable an attack to flip a register in a victim process. There was
<table>
<thead>
<tr>
<th>Category</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offline Time</td>
<td>32 mins</td>
</tr>
<tr>
<td>Online Time</td>
<td>5 mins</td>
</tr>
<tr>
<td>Total Flippy Pages</td>
<td>530</td>
</tr>
<tr>
<td>Total Attacks w/ Correct # of Bait pages</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 4.2: Results from the register experiment demonstrating the ability to flip bits in registers of victim processes

a total of 5 minutes of online hammering time on the register values. There was a much smaller bait-page success rate than expected, we suspect it could be improved by more accurately profiling the victim process to determine the number of bait pages required.
Chapter 5

Attacks

Our attacks require finding vulnerabilities in the code, referred to as Rowhammer gadgets. Rowhammer gadgets are pieces of code with security-critical logic that can be corrupted and bypassed by a Rowhammer attack. It generally consists of a stack variable being set to an initial value, then changed depending on the program flow, and being evaluated as being not equal to a certain value. For example, An integer stack variable auth can be defined as equal to zero initially, then after a password check (which would set auth to 1 if entered correctly), check if the variable is not equal to zero.

We would consider this example a Rowhammer gadget because any bit flip in the auth variable would result in it being not equal to 0, thus passing the authentication. It would be better for security-related code to require that code be equal to a certain value rather than check if it is not equal to a certain value.
Listing 5.1: Returns AUTH_SUCCESS if password is correct AUTH_FAILURE otherwise.

```c
// Gadget
int auth = 0;

// password check code
if(auth != 0)
    return AUTH_SUCCESS;
else
    return AUTH_FAILURE;
```

5.1 Attack on OpenSSL Security Checks Stored in Stack

We first experiment with a simple OpenSSL process where we target a security check variable. At the end of the ECDSA sign setup method, a security check determines if a variable called `ret` is not equal to zero. If the variable is equal to zero, it means that a security check failed and a jump occurred past where the variable is set to 1 indicating all security checks passed. A successful security bypass would hammer the security variable in the stack and force it to be 1 regardless of if it made a jump or not. This could potentially be used in conjunction with a Rowhammer attack that targets dynamic memory.

From Table 5.1 we can see that in a total of 3.5 hours, the system experi-
Table 5.1: Results from the OpenSSL ECDSA sign setup method experiment demonstrating the efficacy of the attack.

<table>
<thead>
<tr>
<th>Category</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offline Time</td>
<td>1 hr 45 mins</td>
</tr>
<tr>
<td>Online Time</td>
<td>7 mins</td>
</tr>
<tr>
<td>Total Flippy Pages</td>
<td>686</td>
</tr>
<tr>
<td>Total Attacks w/ Correct # of Bait pages</td>
<td>139</td>
</tr>
</tbody>
</table>

enced two successful attacks disabling the security checks in OpenSSL. This required only 14 minutes of hammering on OpenSSL itself. During profiling, 1372 pages were found to be flippy, and 277 of them were correctly utilized by having the target security variable placed in them during the attack stage, a resulting bait-page success rate of about 20%.

### 5.2 Bypassing SUDO Authentication

`sudo` is a process in Linux-based operating systems that stands for Super User Do. It allows a user to obtain root access to reading, writing, and executing protected files given they enter the correct password. Breaking the functionality of `sudo` is a textbook privilege escalation attack and can be devastating to systems that hide crucial infrastructure behind the root password. The system administrator sets a root password that is stored and hashed on the system, and when a user enters a password, the hashes of the two passwords are compared, and if they match, root access is granted to the user. This is seen in the code sample given in Listing 5.2.
A fault injection attack has been proposed on the `sudo` program before using a different technique [18] that requires a specific bit flip. The researchers found areas in the `sudo` binary where a bit flip could result in an opcode change which could result in privilege escalation. The researchers found a total of 29 bits that could be flipped, resulting in privilege escalation. An opcode flip requires high precision; once a page with flippy bits is found through the Rowhammer profiling stage, a flippy bit needs to be located in the correct position within the page. Flip a bit that is a single bit-distance away from the target will result in a broken `sudo` program and may require up to a system reboot. In contrast, our attack on the Rowhammer gadget code works if any bit in the `matched` variable is flipped, consisting of 4 bytes or 32 bits for the `matched` variable alone.

We generated a model for stopping the program with `SIGSTOP` and deter-
Listing 5.2: Password authentication function in `sudo`. Returns `AUTH_SUCCESS` if password is correct `AUTH_FAILURE` otherwise.

```c
int sudo_passwd_verify (...) {
    char des_pass[9], *epass;
    char *pw_epasswd = auth->data;
    size_t pw_len;
    int matched = 0;
...
    epass = (char *) crypt(pass, pw_epasswd);
    if ( epass != NULL ) {
        if ( HAS_AGEINFO(pw_epasswd, pw_len)
            && strlen(epass) == DESLEN)  
            matched = !strncmp(pw_epasswd, epass, DESLEN);
        else
            matched = !strcmp(pw_epasswd, epass);
    }
    explicit_bzero(des_pass, sizeof(des_pass));
    debug_return_int(matched ? AUTH_SUCCESS : AUTH_FAILURE);
}
```

mining where it stopped in the execution of the `sudo` binary. The probability of `SIGSTOP` stopping the program at the correct point in execution can be increased by increasing the time window that the `matched` variable is created and used in (see Listing 5.2. The window can be increased by changing the way the password is passed to `sudo`. Using the command line argument for `sudo` ”-S“ requests the password through `stdin`, the standard input instead of using a terminal. Passing command line argument ”-A“ executes a helper program (the intention is that it may be a GUI to accept the password) that
passes the password to the program. Either way, the standard C code for
the `strcmp` method that generates the `matched` variable is the same, but the
time that `matched` exists for is different. The difference in attack window
times can be seen in Figure 5.1.

<table>
<thead>
<tr>
<th>Category</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offline Time</td>
<td>1 hr 9 mins</td>
</tr>
<tr>
<td>Online Time</td>
<td>5 mins</td>
</tr>
<tr>
<td>Total Flippy Pages</td>
<td>485</td>
</tr>
<tr>
<td>Total Attacks w/ Correct # of Bait pages</td>
<td>181</td>
</tr>
</tbody>
</table>

Table 5.2: Results from the `sudo` experiment demonstrating the efficacy of the attack

After running the `sudo` experiment for 10 hrs 34 minutes, the system
experienced a total of 11 successful attacks where we gained root access,
as seen in Table 5.2. This amounts to an average of less than an hour of
profiling to see a successful attack. Additionally, we see that the total online
time is less than an hour to see the 11 flips, so a total of 5-6 minutes of
hammering on average on the `sudo` program itself to see a flip. The time
between successful attacks occasionally varied - the system would see 2-3
attacks in a 15-20 minute window of profiling, other times it may take up
to a few hours. We speculate this to be due to where the process is being
placed in memory, as some areas of the DRAM banks may be more flippy
than others due to manufacturing defects. It should also be noted that of the
5334 attacks, there were 1989 attacks where the target variable was placed
correctly in the flippy page. This is a bait-page success rate of about 37%.
It was initially a concern that the Rowhammer would flip too many bits in the stack of the process that it would be unable to finish execution. While we did find that it was flipping bits in other variables other than matched unintentionally, the program still executed successfully, and when matched was flipped we gained root access. Fortunately, stability did not become an issue in our experiments. This attack was done using the experimental method of synchronization outlined in Section 3.1.3. Further work will demonstrate that the synchronization method of Section 3.1.3 (Practical Method) will provide a more practical attack scenario. Regardless, the results of the experiment demonstrate the novel attack on stack can enable privilege escalation by flipping bits in the stack.

5.3 Bypassing OpenSSH Authentication

To demonstrate the extent of the new attack surface that our attack work enables, we implement the attack on SSH protocols. SSH (Secure Shell Protocol) is an application layer protocol that allows secure remote user login, command execution, and other remote network operations such as TCP port forwarding, tunneling, and file transfer. SSH protocol works in a client-server model. Public-key encryption is used for authenticating the client and the server to each other. After the authentication phase, the transferred data is secured using symmetric key encryption schemes, such as AES. Several libraries implement SSH protocol. OpenSSH is one of the most popular
implementations of SSH protocol. Several attacks on OpenSSH have been shown to steal RSA session keys [32].

When the server program starts, it constantly monitors the incoming connections request to port 22 by default. This monitoring is achieved in an infinite while loop. When the server gets a connection request from a client with a given username and password, a chain of functions is called to check if the provided password is correct. Here, we mention the two most important ones that we can use for our attack. The first function is `mm_answer_authpassword`, and the second function is `auth_password` which is called by the first one. We show the truncated versions of these functions in Listing 5.3 and 5.4. Within these two functions, there are two different local variables that carry the information regarding if the user will be authenticated later on.

Listing 5.3: Password authentication function in OpenSSH. Returns 1 if the password is correct and 0 otherwise.

```
int mm_answer_authpassword(...) {
    char * passwd;
    int r, authenticated;
    ...
    authenticated = options.password_authentication && auth_password(ssh, passwd);
    ...
    if ((r=sshbuf_put_u32(m, authenticated)) != 0)
        fatal_fr(r, "assemble");
    ...
    return (authenticated);
}
```
Listing 5.4: Password authentication function in OpenSSH. Tries to authenticate the user using password. Returns true if authentication succeeds.

```c
int auth_password(...) {
    Authctxt *authctxt = ssh->authctxt;
    int result, ok = authctxt->valid;

    if (*password == '\0' &&
        options.permit_empty_passwd == 0)
        return 0;

    result = sys_auth_passwd(ssh, password);
    if (authctxt->force_pwchange)
        auth_restrict_session(ssh);
    return (result && ok);
}
```

In function `mm_answer_authpassword`, authenticated flag is set if the `auth_password` returns 1 in line 5 and then returned. After being returned, the return value is checked if it equals 1. The client is authenticated, and if the condition is met and the SSH session starts. Otherwise, the client is asked to enter the password again. If the correct password is not given in three trials, the client has to send the connection request again. Here, the authenticated flag is stored in the stack memory of the program, and, therefore, in DRAM, and a potential target for our attack. If we flip the least significant bit of this 32-bit integer value after line 5, we see that the client is authenticated regardless of the password value, and remote shell access is given. However, flipping other bits other than the least significant bit results in authentication failure even if the password is correct.
The other target for our attack is the `result` flag in `auth_password` function. It is initialized to 0 in line 3 and set to 1 in line 5 if the password is correct. Note that the `result` flag is given to a `logical and` operation with `ok` flag. `ok` flag is set to 1 if username is valid. Therefore, as long as the `result` is a nonzero value, the return value would be 1. This logic increases the changes of our attack since as long as we flip any bit of the 32 bits of the `result` variable, we can successfully bypass the password authentication.

Table 5.3 shows the results we observed for our attack on the `result` variable in SSH. We observed that over the course of about one and a half hours, we saw two total successful logins into the SSH server without the correct password. This required a total of 11 minutes of online time, for an average of about 5-6 minutes of hammering SSH per successful attack. In order to complete the attack, we found 1025 memory pages in the system with flippy bits. We also saw that of the attacks, 412 out of 1025 released the correct number of bait pages such that the target variable of SSH was placed correctly in the flippy row. This is a bait-page success rate of about 40%. Like the attack on `sudo`, we performed synchronization using the method outlined in Section 3.1.3. Further work would allow for a more practical attack using synchronization from Section 3.1.3 (Practical Method).
Table 5.3: Results from the attack on the result variable in OpenSSH.

<table>
<thead>
<tr>
<th>Category</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offline Time</td>
<td>45 mins</td>
</tr>
<tr>
<td>Online Time</td>
<td>6 mins</td>
</tr>
<tr>
<td>Total Flippy Pages</td>
<td>513</td>
</tr>
<tr>
<td>Total Attacks w/ Correct # of Bait pages</td>
<td>206</td>
</tr>
</tbody>
</table>

5.4 Vulnerability Analysis

**RSA Bellcore Attacks** The early work by Boneh et al. [5] popularly referred to as Bellcore attacks, demonstrated the importance of checking for errors in cryptographic implementations in a CRT-based RSA implementation. The first mitigation against Bellcore attacks on OpenSSL was released in 2001 and is shown in Listing 5.5. The current OpenSSL implementation performs a check operation to find if an error occurred after the fast CRT-based RSA exponentiation. If an error is detected, the code runs a slower (non-CRT based) exponentiation to compute the signature, thus preventing the possibility of initiating the Bellcore attack. The check mechanism involves recomputing the message using the signature and public key. Recomputed message is later subtracted from the original message to check if both are the same message. If the result is zero, it means the messages match and the exponentiation is computed correctly. The zero check function can be seen in line 17 in Listing 5.5.

For a successful attack, the first step is to create a fault in one of the

---

1[https://github.com/openssl/openssl/commit/1777e3fd5eac0e491bb16a0bb37f4b0f298e6486]
partial CRT-based RSA computations. Then, another fault is introduced in
the check mechanism to trick the code into thinking the CRT-based RSA ex-
ponentiation has been calculated correctly\(^2\). This is achieved by launching a
stack attack on the function \(BN\_is\_zero\). When line 17 calls for \(BN\_is\_zero\)
function, the result of the zero check is returned using the \texttt{EAX} register. We
can force the process to halt and put the result on the stack. By using
Rowhammer, we can manipulate the variable once it is on the stack. When
the return value is anything other than zero, the if case will be executed,
giving the appearance that the CRT-based exponentiation was computed
correctly.

**Bypassing MySQL Authentication** MySQL is the most popular open-source
database management system \([47]\), which is commonly used by many organi-
zations and websites in all industries from Defense & Government to Social
Networks including US Navy, NASA, Twitter, Facebook, LinkedIn, and Bank
of America\([39]\).

We found a Rowhammer gadget \(^3\) given in Listing 5.6 in the source code
of MySQL server that is used for authenticating a client with a password.
The password check happens between line 3 and 7 and the result is stored in
\texttt{fast\_auth\_result}. When we simulate a 0 to 1 flip on \texttt{fast\_auth\_result.first}
in line 8, we observe that the client is authenticated even with an incorrect

\(^2\)The probability of both faults going through will be low, however Bellcore requires only one faulty sample to succeed.

\(^3\)https://github.com/mysql/mysql-server
Listing 5.5: Error checking in OpenSSL ModExp

```c
static int rsa_ossl_mod_exp ( BIGNUM *r0 , const
BIGNUM *I, RSA *rsa , BN_CTX *ctx ) {
...
    if ( rsa->e && rsa->n ) {
        if ( rsa->meth->bn_mod_exp ==
BN_mod_exp_mont ) {
            if ( !BN_mod_exp_mont ( vrfy , r0 , rsa->e,
rsa->n , ctx ,
rsa->_method_mod_n ) )
                goto err ;
        } else {
            bn_correct_top ( r0 );
            if ( !rsa->meth->bn_mod_exp ( vrfy , r0 ,
rsa->e , rsa->n , ctx ,
rsa->_method_mod_n ) )
                goto err ;
        }...
    if ( !BN_sub ( vrfy , vrfy , I ) )
        goto err ;
    if ( BN_is_zero ( vrfy ) ) {
        bn_correct_top ( r0 );
        ret = 1 ;
        goto err ; /* not actually error */
    } ... 
}
```

password. Note that, unlike the previous attacks, the target variable requires single-bit precision; hence, the attack is harder to achieve using Rowhammer.
Listing 5.6: Password authentication function in MySQL. Tries to authenticate the client using authorization_id and scramble. The authentication succeeds if fast_auth_result.first is false.

```c
static int caching_sha2_password_authenticate(...) {
    std::pair<bool, bool> fast_auth_result =
        g_caching_sha2_password->fast_authenticate(
            authorization_id,
            reinterpret_cast<unsigned char*>(scramble),
            SCRAMBLE_LENGTH, pkt,
            info->additional_auth_string_length ?
                true : false);
    if (fast_auth_result.first) {
        if (vio->write_packet(vio, (uchar*) &perform_full_authentication, 1))
            return CR_AUTH_HANDSHAKE;
    } else {
        if (vio->write_packet(vio, (uchar*) &fast_auth_success, 1))
            return CR_AUTH_HANDSHAKE;
        if (fast_auth_result.second) {
            const char *username =
                *info->authenticated_as ?
                info->authenticated_as : "";
        }
        return CR_OK;
    }
}
```
Chapter 6

Full Attack on TLS handshake

The effectiveness of this attack can be demonstrated on a TLS handshake without altering the source code at all to provide footholds for synchronization. The attacker is assumed to be colocated with the client and will hammer the victim client’s handshake process forcing it to interpret a faulty signature as valid. This could be in the context of a man in the middle attack, where an attacker is trying to trick a client into thinking a server is the authentic target they are trying to connect to.

In the normal scenario, the client will attempt to connect to the server and will send a ClientHello message to the server. The server will respond with a ServerHello message, which includes the server’s public key and a signature of the handshake. The client will then verify the signature using the server’s public key. If the signature is valid, the client will assume that it is safe to send sensitive information to the server. If the attacker can flip
Figure 6.1: Normal scenario where the client connects to the server, sends a message, receives the message signed by the server, and is able to authenticate the server.

In Figure 6.1, we see the normal scenario where the client connects to the server, sends a message and receives the message signed by the server and is able to authenticate the server. Importantly, the client is vulnerable to the Rowhammer attack during the phase while it is waiting for a response from the server. This connection phase can take time (in the order of milliseconds) and is ultimately controlled by the server, and therefore, the attacker can hammer the client’s memory during this phase.

In Figure 6.2 we can see the attack scenario. The attacker capitalizes on
the fact that the client is vulnerable to Rowhammer during the connection phase. The attacker acts as both the server and is colocated with the client. The attacker responds to the client’s ClientHello with a ServerHello message, which includes the attacker’s public key and a signature of the handshake. The client will then verify the signature using the attacker’s public key. If the attacker can flip a bit in the signature verification process, the client will think the signature is valid and will send sensitive information to the attacker. Theoretically, the attacker can then forward the message to the real server and receive the response. The attacker can then forward the response to the client, and the client will think it is communicating with the real server, otherwise known as a man-in-the-middle attack.

Figure 6.2: Attack scenario where the attacker acts as both the fake server and colocated with the client.
This full attack scenario consists of 3 steps:

- **Step 1:** The client connects to the attacker and sends a `ClientHello` message.
- **Step 2:** The attacker sends a `ServerHello` message to the client, which includes the attacker’s public key and a signature of the handshake.
- **Step 3:** The client will then verify the signature using the attacker’s public key. If the attacker can flip a bit in the signature verification process, the client will think the signature is valid and will send sensitive information to the attacker.

### 6.1 Taking advantage of IP Sockets for Synchronization

This attack does not require any degradation or other synchronization techniques to time the bit flip attack on the client. This is because the attacker is controlling the time that the verification process takes, and thus can simply wait for the bit flip to occur before sending the response to the client.

In Listing 6.1 we see that the client has the ability to verify a signature based on the public key. It keeps the state of the verification process in a variable call `pass`. The `pass` variable is set to 1 if the signature is valid, and 0 otherwise. During the connection phase, the Rowhammer attacker can attack the `pass` variable and flip a bit to make the client think the signature is valid.
In Listing 6.2 we can see that pass is used to verify if the server is authenticated. If pass is not 0, then the server is authenticated. The attacker can flip a bit in the pass variable to make the client think the server is authenticated regardless of the TLS signature verification process executed previously.

Just as with the previous experiments, this full attack was conducted on a system with Ubuntu 20.04.01 LTS with 5.15.0-58-generic Linux kernel installed. The system uses an Intel Core i9-9900K CPU with a Coffee Lake microarchitecture. We used a dynamic clock frequency rather than a static clock frequency to improve the practicality of the attack.

We were able to see a flip in the client after about a half-hour of attacking the client. This meant just over 300 attacks on the client, which is realistic for a Rowhammer attack. The attacker was able to flip a bit in the pass variable, which made the client think the server was authenticated. This is a successful attack on the TLS handshake.
Listing 6.1: Client code that connects to the server and sends a message to be signed. It is vulnerable to the Rowhammer attack during the connection phase.

```c
int pass=0;

// Create client socket
client_fd = socket(AF_INET, SOCK_STREAM, 0);
...

// Connect to server
connect(client_fd, (struct sockaddr *)&server_addr, sizeof(server_addr));

// Send a message to the server
unsigned char message[32] = "This is a message to be signed";
send(client_fd, message, sizeof(message), 0);
...

while ((bytes_received = recv(client_fd, buffer, sizeof(buffer), 0)) > 0)
{
    memcpy(sig_buf + sig_len, buffer, bytes_received);
    sig_len += bytes_received;
}

// Deserialize the signature
const unsigned char *pp = sig_buf;
ECDSA_SIG *signature = d2i_ECDSA_SIG(NULL, &pp, sig_len);
...

// Verify the signature
if (verify_message(message, sizeof(message), signature, ec_key)==SUCCESS)
{
    pass = 1;
}
```
Listing 6.2: Client code that uses the `pass` variable to determine if the signature is valid.

```c
// remove sensitive data from memory
EC_KEY_free(ec_key);
ECDSA_SIG_free(signature);
...
if (pass != 0)
{
    fprintf(stdout, "Server\nAuthenticated\n");
    fflush(stdout);
}
```
Chapter 7

Countermeasures

Tighter, More Precise Logic  This thesis proposes a set of countermeasures that can be used against a Rowhammer attack on the stack of a process. The easiest way to make an attack more difficult is to tighten the logic of the code and avoid using if-not-zero conditionals.

In the first example seen in Listing 7.1, if any single bit in the matched variable is flipped, the first statement becomes true. The Rowhammer attack is not always precise, so checking if matched is not equal to zero allows an attacker to flip any of the 32 bits that make up matched, and the passwords will seem to match. This is a very similar gadget to the one that was found in the sudo program.
Listing 7.1: Loose Logic Susceptable to Rowhammer Attack

```java
if (matched != 0)
    // passwords match
else
    // passwords don’t match
```

Listing 7.2: Tight Logic Less Susceptable to Rowhammer Attack

```java
if (matched == 1)
    // passwords match
else
    // passwords don’t match
```

In contrast, Listing 7.2 is safer because Rowhammer is required to flip only the least significant bit; otherwise, the passwords still will not match. Requiring security-sensitive variables to be stored in registers over stack is not an effective countermeasure because, as seen in Section 3.3.3, registers can be flushed to memory using signal interrupts and can still be flipped. The code that was found in `sudo` and `SSH` has vulnerable code that is susceptible to Rowhammer by changing logic for any flip in the 32-bit variable, while `MySQL` requires a least significant bit flip. Additionally, it can be beneficial to use boolean variables over integers when possible to reduce the target size.

Additionally, the stack Rowhammer attack can further be prevented by requiring specific patterns for security sensitive checks so a single bit flip will not result in a security failure. For our example with the `matched` variable, we could require that the variable be set to a random set of bits that are not all zeros. This would require an attacker to flip all bits in the variable to that specific pattern for authentication which is far more difficult than flipping any single bit. It takes advantage of the fact that rowhammer is a blunt tool
that is often inprecise.

Listing 7.3: Tight Logic Less Suspectable to Rowhammer least significant bit flip required for successful attack

```
if(matched == 1)
    //passwords match
else
    //passwords don’t match
```

Listing 7.4: Strong Rowhammer countermeasure requiring a specific pattern in the matched variable

```
if(matched == 0x69d61fc8)
    //passwords match
else
    //passwords don’t match
```

Consider listing 7.4. This code is far more secure than the previous examples because it requires a specific pattern in the `matched` variable. This pattern is a random set of bits that are not all zeros. In this case, the attacker would need to flip the `matched` variable to `0110100111...`, which includes 17 bit flips in precise locations along the variable. This is more difficult than flipping any single bit in the variable.

**System Changes to Prevent Rowhammer**  One of the most common countermeasures cited is increasing the DRAM refresh rate. Standard DRAM refresh is 64ms, meaning that a Rowhammer attack has 64ms to flip a bit before the row refreshes. So a faster refresh rate will result in a shorter time window for the Rowhammer attack to be performed and result in fewer flips. A faster refresh rate will result in worse performance and more power
consumption, and is not an ideal solution.

One possibility is Hidden Row Activation (HiRA). HiRA is a novel technique proposed in [3] which parallizes row refreshes for DRAM. It allows a row refresh operation to be hidden in the background while a row in the DRAM is being accessed or refreshed in the same bank. It takes advantage of the fact that different rows in the same bank may be connected to different charge restoration circuitry, allowing for concurrent refreshes. By making an effort to reduce the latency of refresh operations, HiRA can reduce the time window for Rowhammer attacks. HiRA claims to be able to concurrently refresh 32% of rows in a DRAM concurrently on 56% of off the shelf DRAM chips. However, despite strides in the direction of more efficient refreshes as a rowhammer mitigation, HiRA is still in its infancy and is not yet available for use in consumer systems.

Additionally, [55] proposes a novel and efficient Rowhammer mitigation by building on existing Probabilistic Adjacent Row Activation (PARA) Rowhammer defences by building Discreet-PARA. Discreet-PARA combines disturbance bin counting, a mechanism for managing refresh operations on rows likely to be corrupted by Rowhammer, and PARA-cache, which is a cache that stores the most recently accessed rows. By tracking accesses and refreshes to rows, Discreet-PARA can detect and mitigate Rowhammer attacks. Researchers were able to optimize these refresh and access tracking mechanisms to reduce the performance overhead from averages from 10.5%-6.6% to 5.3%-2.6%. Still, this mitigation results in overhead that may not be ideal
for consumer systems.

It was initially thought that ECC would be an ample countermeasure to Rowhammer. However, ECC is not a sufficient countermeasure because it can be defeated with \cite{11}. ECC stands for Error Correcting Codes, and its a method of detecting and correcting errors in memory. ECC is a common feature in server memory, but it is generally not available in consumer systems. The authors of \cite{11} demonstrated that ECC can detect and correct a single bit flip, can detect a double bit flip, but cannot detect or correct a triple bit flip. This means that a Rowhammer attack can be performed by flipping three bits in a row, and ECC will not detect or correct the error. ECC as a hardware solution is not a sufficient countermeasure to Rowhammer.

**Machine Learning to Detect Rowhammer Gadgets** Rowhammer gadgets may be an excellent domain for a machine learning algorithm to find and detect vulnerable pieces of code using natural language processing. Similar work has been done using machine learning to detect Spectre gadgets \cite{51}. A dataset of Rowhammer gadgets could be derived from existing code by simulating Rowhammer flips in stack variables and checking if the process experiences a security failure.
Chapter 8

Discussion & Conclusion

Originally thought to be safe areas in to store data, this thesis has shown that the stack and registers are vulnerable to Rowhammer attacks. The stack and registers often hold sensitive program data that if corrupted can lead to privilege escalation, information disclosure, and other security failures.

Faulting the stack is difficult due to the randomization of the stack offset, but this thesis has shown that the stack offset is not as random as previously thought. Additionally, this thesis shows that a probabilistic model can be used to estimate the stack offset and that the stack offset can be brute-forced in a reasonable amount of time. Registers are also vulnerable to Rowhammer attacks, and this thesis has shown that registers can be flushed to memory using signal interrupts and then flipped. This opens an enormous attack service on registers that was previously thought to be safe.

Programs like SUDO, OpenSSH, OpenSSL, and MySQL are vulnerable to
Rowhammer attacks on the stack. This thesis has shown that Rowhammer attacks on the stack and registers are possible and that these attacks can be used to exploit security vulnerabilities in real-world programs.

Lastly, we were able to demonstrate a full attack without tampering with the source code for synchronization by attacking a client-server TLS handshake, and demonstrated this attacks potential is a real world scenario. This attack is a proof of concept that Rowhammer attacks on the stack and registers are possible and can be used to exploit security vulnerabilities in real-world programs.
Bibliography


[6] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn’t touch this: Software-only mitigation


[16] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TR-


[41] OpenAI. Chatgpt. https://www.openai.com/chatgpt/, 2021. Used for revising and restructuring the manuscript to ensure coherence and originality.


[44] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the


