GbHammer: Malicious Inter-process Page Sharing by Hammering Global Bits in Page Table Entries

Keigo Yoshioka*, Soramichi Akiyama†
*Shibuya Junior & Senior High School, Tokyo, Japan
†College of Information Science and Engineering, Ritsumeikan University, Osaka, Japan
Email: s-akym@fc.ritsumei.ac.jp

***Abstract*—RowHammer is a vulnerability inside DRAM chips where an attacker repeatedly accesses a DRAM row to flip bits in the nearby rows without directly accessing them. Several studies have found that flipping bits in the address part inside a page table entry (PTE) leads to serious security risks such as privilege escalation. However, the risk of management bits in a PTE being flipped by RowHammer has not yet been discussed as far as we know. In this paper, we point out a new vulnerability called *GbHammer* that allows an attacker to maliciously share a physical memory page with a victim by hammering the global bit in a PTE. GbHammer not only creates a shared page but also enables the attacker to (1) make the victim's process execute arbitrary binary and (2) snoop on the victim's secret data through the shared page. We demonstrate the two exploits on a real Linux kernel running on a cycle-accurate CPU simulator. We also discuss possible mitigation measures for GbHammer and the risk of GbHammer in non-x86 ISAs.**

***Index Terms*—RowHammer, Page Table, Global Bit**

I. Introduction

RowHammer is a vulnerability inside DRAM chips where an attacker repeatedly accesses a DRAM row to flip bits in the nearby rows without directly accessing them [1]. It can result in serious outcomes such as OS-level privilege escalation [2], secret data leakage [3], Denial-of-Service [4], breaking VMM-level memory isolation [5], and confusing AI models [6].

Despite many RowHammer-based exploits found, the risk of management bits in page table entries (PTEs) being hammered is yet to be discussed. To this end, we demonstrate two new exploits when the global bit of a PTE is flipped by RowHammer, which we refer to as *GbHammer*. GbHammer forces the CPU to use the same address translation information between the attacker and the victim, resulting in a physical memory page maliciously shared by them. Through the shared page, the attacker can make the victim execute binary code that the attacker crafts and snoop on secret data of the victim.

The contributions of this paper are as follows:

1) We are the first to discuss the risk of management bits in PTEs being hammered as far as we know.
2) We propose two new exploits based on GbHammer, and demonstrate that they indeed work using a cycle-accurate CPU simulator gem5 and a real Linux kernel.
3) We investigate the specifications of ISAs other than x86_64, namely ARMv7 and RISC-V, and discuss that the same exploits can be executed on them.
4) We discuss possible mitigation measures of GbHammer and the challenges in achieving these measures.

II. Preliminaries

A page table entry (PTE) refers to one line of a page table that consists of a mapping from a virtual address to a physical address and some additional management bits. Among these management bits, a PTE in x86 has one called a *global bit* [7]. A PTE with the global bit enabled is called a *global PTE*. A global PTE indicates that the address translation information represented by that PTE may be used among different processes. It is beneficial for some special cases where multiple processes use the same address translation information. For instance, the kernel address space is often mapped to the same virtual address ranges of different processes.

On Intel processors, a global PTE is associated with a *global TLB entry*. TLB (Translation Look-aside Buffer) caches address translation information that is recently used and the cached information is referred to as TLB entries. It greatly improves CPU performance because accessing the TLB is faster compared to a normal page table walk that requires multiple DRAM accesses. When the address translation information within a global PTE is cached to the TLB, the TLB entry also becomes global. This means that the CPU may use the same TLB entry for different processes from the one that has created the entry. More concretely, the Intel manual [7] says that a *processor may use a global TLB entry to translate a linear address, even if the TLB entry is associated with a PCID different from the current PCID*. Here, PCID (Process-Context Identifier) is a value assigned to each TLB entry and it is unique to the process that created the entry. PCID prevents the TLB from being completely flushed in a context switch.

III. New Vulnerability: GbHammer

A. Threat Model

In this paper, we assume that an attacker can login to the same physical machine as the victim and can execute programs with a user privilege. Specifically, an attacker can

1) login to the same machine as the victim by using measures such as an SSH client,
2) prepare a program on that machine by either writing code and compiling it or transferring an already compiled binary by using measures such as SCP,
3) execute the prepared program with a user privilege on any core including the one on which the victim's process is executed, and

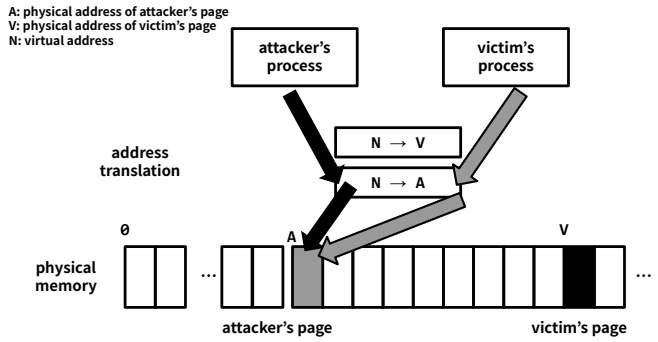

Fig. 1. Malicious Page Sharing by GbHammer

4) access the executable file that the victim invokes as a process (the intention is explained in Section III-C).

B. Malicious Page Sharing

GbHammer is an attack that allows an attacker's process to maliciously share physical memory pages with a victim's process. This attack is achieved by flipping a global bit of a PTE by RowHammer. This makes the same address translation information used by both the attacker's and the victim's processes, resulting in a shared physical page among them.

Fig. 1 illustrates the situation where a physical page is maliciously shared by GbHammer. Here, `N` represents a virtual address that both the attacker's and victim's processes use. Note that using the same virtual address itself is completely normal and allows no malicious data access thanks to the isolation of virtual address spaces. `A` and `V` represent the physical addresses that are mapped from `N` in the virtual address spaces of the attacker's process and the victim's process, respectively. These mappings are shown in the figure as the *address translation* labeled with `N`→`A` and `N`→`V`, respectively. The black arrows from the attacker's process to the physical page at address `A` show that memory accesses by the attacker's process go to the attacker's page as expected. On the other hand, the grey arrows from the victim's process show that memory accesses from the victim's process are maliciously forwarded to the **attacker's** page.

In summary, GbHammer allows an attacker to create a shared page as follows:

1) The attacker's process flips the global bit of the PTE that maps `N` to `A` in their own address space using RowHammer. This makes this PTE global.
2) The attacker's process issues a memory access to virtual address `N`. This creates a global TLB entry that maps `N` to `A`.
3) The attacker waits until the victim's process accesses virtual address `N`. Because the CPU has a global TLB entry that maps `N` to `A`, the access by the victim's process ends up going to the physical address `A` instead of `V`.
4) This means that the physical page starting from physical address `A` is now shared among the attacker's process and the victim's process.

C. Steps of GbHammer

Here we elaborate on the steps of GbHammer in more detail.

- **Step (1)**: The attacker acquires a virtual address `N` that the victim's process uses. This is necessary because GbHammer only works when the attacker's and the victim's processes use the same virtual address.
- **Step (2)**: The attacker creates a mapping from `N` to any physical address in the attacker's virtual address space.
- **Step (3)**: The attacker flips the global bit in the PTE that maps `N` in the attacker's virtual address space by RowHammer.

Step (1): The attacker's process conducts the following two procedures. Note that the procedures described here only work for the Arbitrary Binary Execution exploit based on GbHammer (explained in Section III-D), and how to achieve Step (1) for the Data Snooping exploit is future work.

First, the attacker acquires the offset of binary code that the victim's process executes. This is done by disassembling the binary that is executed by the victim's process. Although the attacker only has a normal user privilege on the target machine, they can still achieve this when the victim executes a pre-compiled binary downloaded from software repositories (e.g., by the `apt` command). This is a common assumption in some ROP-style attacks [8], [9].

Second, the attacker bypasses ASLR (Address Space Layout Randomization) to acquire the starting virtual address on which the code of the victim's process is placed. By combining this address and the offset acquired in the previous paragraph, the attacker can know the virtual address `N` of an arbitrary piece of code that the victim executes. To bypass ASLR, the attacker can leverage existing techniques such as detecting collisions in the BTB (Branch Target Buffer) [10].

Step (2): There are two methods to achieve Step (2). The first method is to use the `mmap` systemcall. POSIX.1-2001 defines the prototype of `mmap` as follows [11].

```
void *mmap(void *addr, size_t length,
int prot, int flags, int fd, off_t offset);
```

`addr` is a hint to the OS for the starting address of the created mapping. The attacker can specify `N` to this argument and the OS usually respects it when creating a mapping.

The second method to achieve Step (2) is to specify `N` as an address of functions and global variables inside an ELF binary. An ELF binary file contains virtual addresses on which sections are loaded, and this address is used as-is when the binary is compiled with the `-static` option of gcc.

Step (3): The attacker's process conducts the following five procedures. First, the attacker allocates a large and continuous memory region R with `mmap`. The returned memory region is also continuous on the physical memory due to the characteristics of the buddy allocator of Linux [3], [12]. A corner case scenario where memory is fragmented is also

discussed in [3]. Second, the attacker hammers pages in R to find a *target page*. A target page is a page that has a bit vulnerable to RowHammer at the bit position which would be interpreted as global bits when the page is used as a page table page. For example, when `N` is `0x20000`, any page whose 32712 ($= 64 \times 511 + 8$)$^{\text{th}}$ bit from the least significant bit is vulnerable can be a target page. Note that the address translation information for virtual address `0x20000` is placed in the first PTE out of 512 PTEs (64 bits each) stored in a page table page. Third, the attacker returns the target page to the OS by `munmap`. Fourth, the attacker creates a mapping from virtual address `N` using `mmap` with the `MAP_POPULATE` flag. Because the OS reuses the just-returned page as the page table page for the newly created mapping [13], this makes the target page to be used to store the PTE that maps virtual address `N` to a physical page. Finally, the attacker uses other pages in R to hammer the target page until the global bit flips. A bit that has flipped in the previous hammering is likely to flip again due to the strong locality of bit-flips.

We make two important notes here. **First**, targeting a management bit of a PTE with RowHammer is almost equivalent to targeting the address part (the only difference is the bit position to hammer), which is already done in a body of work [2], [5], [12], [13]. Therefore, we believe that targeting a global bit is feasible although our experiments focus on the outcomes of it. **Second**, separating user- and kernel-space pages in the physical memory to prevent the OS from reusing a target page as a page table page can be invalidated by PTHammer [2]. As far as we know, there is no solution yet to PTHammer except for activation-tracking mechanisms [14], [15] that require additional hardware.

D. Exploits based on GbHammer

GbHammer enables two new exploits: arbitrary binary execution and data snooping.

1) Arbitrary Binary Execution: The attacker's process can make the victim's process execute an arbitrary binary as illustrated in Figure 2. To do this, the attacker's process maliciously creates a share page that is mapped from virtual address `N` where a part of the binary code of the victim's process resides in the victim's address space. Then, the attacker's process stores the binary that it wants the victim's process to execute to the created shared page. Third, the attacker executes the binary by itself to create a global **iTLB** entry. When the victim's process tries to execute the code at virtual address `N` later on, it will mistakenly execute the binary stored by the attacker.

2) Data Snooping: The attacker's process can snoop on data written by the victim's process. To do this, the attacker's process maliciously creates a share page that is mapped from virtual address `N` to which the victim's process stores secret data. Then, the attacker's process accesses address `N` to create a global **dTLB** entry. The attacker waits until the victim's process mistakenly stores its data on the shared page by writing it to address `N`. After that, the attacker simply reads the data from the shared page.

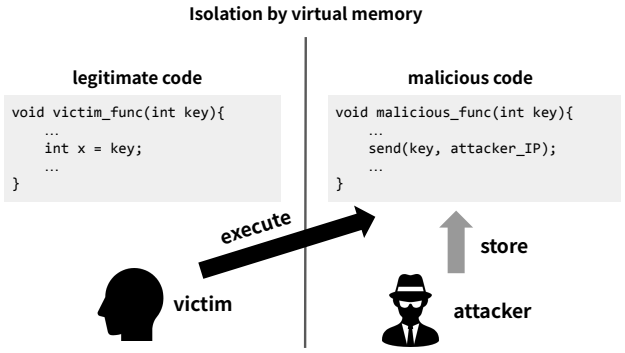

Fig. 2. Binary Execution Exploit

TABLE I
EXPERIMENT ENVIRONMENT

Software	Version
Simulated OS	Ubuntu 18.04.2 (Linux kernel 5.4.49)
gem5	23.0.0.1
gcc	9.4.0

TABLE II
CONFIGURATION PARAMETERS OF GEM5

Software	Version
Simulated ISA	x86_64
CPU	AtomicSimpleCPU, 1 core
Memory	Atomic
TLB	64 entries each for iTLB and dTLB

IV. EXPERIMENTAL RESULTS

A. Reproducing GbHammer with gem5

We reproduce GbHammer on a cycle-accurate CPU simulator gem5 [16] and its Full System mode that can run an entire OS to assess the risk under realistic settings. We use the latest Linux kernel known to work on it [17] and other software as shown in Table I. We employ the `AtomicSimpleCPU` model that does not simulate the detailed timing of each instruction to achieve fast simulation (tens of minutes to boot Ubuntu). The use of this simple CPU model should not affect the validity of the experimental results because the detailed timings such as pipeline stalls are not relevant to this work. The simulated CPU has a single core to ensure that the attacker's and the victim's processes are executed on the same core and use the same TLB. For the memory model, we also employ a simple one named `Atomic`. The use of this simple memory model should neither affect the validity of the experimental results because we do not reproduce RowHammer inside the simulated memory unlike Hammulator [18] does. Other configuration parameters of gem5 are shown in Table II.

Reproducing GbHammer follows the procedures below.

1) We fix the target virtual address (i.e., `N`) to `0x20000`. This avoids the necessity of achieving Step (1) and allows us to focus on the outcomes of GbHammer.

```
int f() {
    return 1;
}

void main() {
    sleep(5); // GbHammer happens here
    printf("This is victim.\n");
    printf("Expected output: 1\n");
    printf("Actual output: %d\n", f());
}
```

Fig. 3. Victim Code for the Binary Execution Experiment

2) Processes of the attacker and the victim are invoked on the simulated OS controlled via Telnet. We explain how these processes are implemented later.
3) When a TLB entry is created for the target virtual address, the entry is forcefully set as global. We implement this in the source code of gem5, namely `src/arch/x86/tlb.cc`. The `insert` function is modified to set the global bit to 1 when the virtual address of a new TLB entry is `0x20000`.

The choice of using gem5 to reproduce GbHammer comes from the reliability. Although it is possible to reproduce RowHammer (and thus GbHammer) on a real machine, reproducing it reliably requires much engineering effort such as reverse-engineering the DRAM address mapping embedded in the memory controller using existing methods [19], [20]. We use gem5 to reliably reproduce GbHammer to focus more on the outcome and the risk of it, rather than on how to make it happen which is based on well-established building blocks.

B. Setup: Arbitrary Binary Execution

In this experiment, the source code of the victim's process defines a function `f` that returns 1. The victim's process sleeps for 5 seconds, calls `f`, and then prints the result returned by `f`. Fig. 3 shows the source code of the victim's process. The sleep duration (5 s) can be any value as long as there is enough time for the attacker to do its job between the invocation of the victim's process and the call of `f`. The attacker's process, meanwhile, creates a shared page whose virtual address starts from `0x20000` by GbHammer, writes to the shared page a binary blob compiled from a function that returns **2**, and then executes the binary blob as a function. This creates a **global** iTLB entry (because of our modification to gem5) that translates the virtual address `0x20000` into the physical address of the attacker's page that contains the binary blob.

The victim's function `f` is placed at virtual address `0x20000` with the help of a linker script we craft. A linker script is passed to `gcc` and specifies how functions and global variables are placed in the virtual address space of a process. This enables us to place `f` at a specific virtual address and also separate it from other functions and variables. On the other hand, in a real attack scenario, the attacker must rewrite an entire page that may contain not only `f` but also other data so that the victim's process does not simply crash.

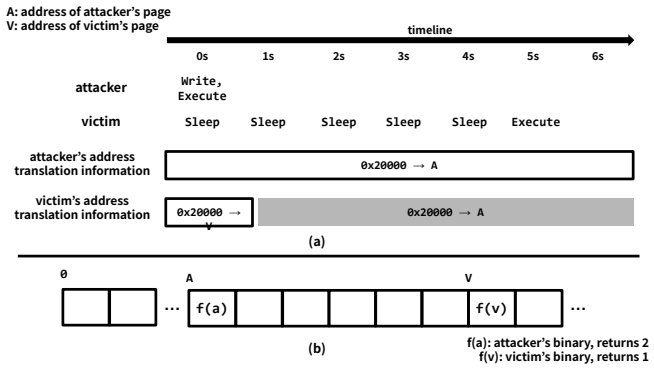

Fig. 4. Timeline of the Binary Execution Exploit

Figure 4 shows the timeline of this experiment. Here, `A` and `V` refer to the physical addresses that are translated from the virtual address `0x20000` in the address spaces of the attacker and the victim, respectively. The first and second rows (labeled as *attacker* and *victim*, respectively) show the operations executed by the two processes. The third and fourth rows show the address translation information that the two processes would use at a particular point in time. The victim's process can properly translate `0x20000` to `V` before the attacker conducts GbHammer. However, after `t = 1s`, the victim's process mistakenly translates `0x20000` into `A`, where the malicious binary blob has been written by the attacker. This is because the global iTLB entry created by the attacker's process is used by the victim's process as well.

C. Setup: Data Snooping

In this experiment, the victim's process first allocates a memory region starting from address `0x20000`, sleeps for 5 seconds, and then writes a secret string "This is victim's data" to the region. The sleep duration (5 s) can be any value as long as there is enough time for the attacker to do its job between the memory allocation and data writing by the victim's process. The attacker's process, meanwhile, creates a shared page whose virtual address starts from `0x20000` by GbHammer, reads the data from it every second, and outputs the data concatenated with a string "This is attacker". A **global** dTLB entry is created when the attacker's process reads the shared page for the first time because of our modification to gem5. The victim's process mistakenly translates the virtual address `0x20000` using this dTLB entry and the secret string is maliciously stored on the shared page.

D. Results

1) Binary Execution: Fig. 5 shows the output from the victim's process we observed on the terminal. The first and second lines are pre-defined strings and they are properly output. The value after "Actual output:" in the third line is the return value of what the victim's process thinks is `f`.

We confirmed that GbHammer allowed the attacker to make the victim's process execute the binary that the attacker prepared. After the victim's process executed what it thinks

```
This is victim.
Expected output: 1
Actual output: 2
```

Fig. 5. Result of Binary Execution Experiment

```
This is attacker
This is attacker
This is attacker
This is attacker
This is attacker
This is attacker
This is attacker This is victim's data
This is attacker This is victim's data
This is attacker This is victim's data
This is attacker
```

Fig. 6. Result of Data Snooping Experiment

is function `f`, it printed **2** instead of 1 as shown in Fig. 5. This means the binary that the attacker had written to the maliciously shared page was executed instead of `f`.

2) Data Snooping: Fig. 6 shows the output from the attacker's process we observed on the terminal. From the first to the fifth lines (from `t = 0s` to `t = 5s`), the attacker's process printed "This is attacker" and nothing after it. From the seventh to ninth lines (from `t = 6s` to `t = 8s`), the secret string that the victim had written to a memory region that it had allocated was printed by the attacker.

We confirmed that the GbHammer allowed the attacker to snoop on the victim's data. After the victim's process had written the secret string to the memory region that it had allocated, the attacker's process successfully printed that data to the terminal. This means that the victim was forced to write the secret data to the maliciously shared page and the attacker could read the data from it.

V. Possible Mitigation Measures

A. Modifying mmap

The OS can ignore the virtual address given as an argument of `mmap` to make Step (2) of GbHammer harder to achieve. Because the virtual address of an allocated memory region does not matter for ordinary programs, this change should not affect many use-cases of `mmap`. For example, when a program maps a file into memory using `mmap`, the same source code of the program should work for any virtual address used as the starting address of the mapping.

There are two use cases of `mmap` we are aware of that require a newly allocated memory region to be placed at a specific virtual address. Except for these cases (and others that we are not aware of if any), it is safe to simply ignore the virtual address specified as the starting address of the mapping. The two use cases are as follows:

1) **Shared library loading**: Linux programs specify virtual addresses to `mmap` when they load shared libraries (i.e., `.so` files). This is because a shared library contains different sections that must be loaded with different protection modes (e.g., read-only, executable-but-not-writable) but must also be contiguous to each other. One way of achieving this without specifying the starting addresses of mappings would be to create a new system call that accepts multiple protection modes and offsets inside a mapping to use each protection mode. For example, a program can use this new system call to map `lib.so` to any address, and make it read-only from offset `0` to offset `1024` but executable-but-not-writable from offset `1024`.
2) **Container live migration**: migrating a container from one host machine to another requires memory allocation in the destination host with the virtual addresses specified. This is because the layout of the virtual address spaces of the processes that the container consists of must be replicated from the source to the destination host. Otherwise, every value in the container's memory and registers that represents an address must be rewritten to a new value. This is impractical because there is no decisive method to know if a value (e.g., `0x20000`) is an address or something else.

B. Modifying the Loader

Specifying virtual addresses of sections in an ELF binary when it is loaded is another method to achieve Step (2) of GbHammer. The OS can ignore this as long as the ELF binary is compiled as position-independent code (PIC). PIC enables sections of an ELF binary to be loaded on any virtual address by placing fake addresses in the binary that are rewritten to actual ones at load time.

This measure is applicable as long as the source code of the victim's program is available and compatible with a modern compiler. Because creating a PIC binary requires compiling it from the source code, it cannot be applied to existing non-PIC programs whose source code are either closed or lost.

C. Ignoring the Global Bits

Enabling the global bit of a PTE does not guarantee that the address translation information is used by other processes, as the Intel manual says *a processor may use* [7] global TLB entries. The PGE bit (bit 7) in a control register named CR4 decides whether the global bit of a TLB entry is respected or not (described in Section 4.10.2.4 "Global Page" of [7]). Thus, the risk of GbHammer is completely eliminated if this flag is disabled. However, it is important to assess the effectiveness of this measure by considering the overhead incurred by completely disabling the global bits. As far as we know, this overhead is not well studied under the combination of modern processor architectures and modern applications.

VI. Discussion

A. GbHammer in ARMv7 and RISC-V

GbHammer and the exploits based on it are also applicable to the ARMv7 and RISC-V ISAs. A PTE of ARMv7 has nG bit (non Global bit) which functions conversely to the global

bit in x86 [21]. Just like on x86, an attacker can share a single page with the victim with each successful GbHammer attempt on an ARMv7 processor. This is because the nG bit exists only in the last level of the hierarchy of address translation information where the physical page number is stored.

The hierarchy of address translation information in RISC-V is different from that in x86 and ARMv7. Unlike them, every level of the hierarchy in RISC-V has the same management bits including G bits, meaning that every level can be global. Specifically, the manual [22] says in its Section 10.3.1 that *for non-leaf PTEs, the global setting implies that all mappings in the subsequent levels of the page table are global*. Note that every level of address translation information is called a page table in RISC-V (unlike page table, page directory, etc. in x86). The risk of GbHammer in RISC-V could be much larger than that of x86 or ARMv7, because a much larger address range is maliciously shared at once.

B. Relaxed Constraints on the Victim's Process

We let the victim's process sleep for some amount of time (5 seconds specifically in our current experiments) so that the attacker's process can execute GbHammer during this period. This constraint could be a large hurdle for the attacker because there is no straightforward way to let other processes sleep with a user privilege. Instead, an attacker could target a process that repeatedly accesses the same virtual address (e.g., calling the same function over and over) in a loop to relax this constraint. This effectively gives the attacker enough time to execute GbHammer to the address that the victim accesses. In this scenario, the attacker needs to invalidate the non-global TLB entry created by the victim's process by using measures such as constructing eviction sets for the TLB [2].

Our experimental setup needs improvement to conduct experiments for this scenario. In the current setup, we set the global bit of a TLB entry when its corresponding virtual address is `0x20000`. This happens no matter which process creates the TLB entry because our modified gem5 does not distinguish the process that creates a TLB entry. Thus, a TLB entry is mistakenly set as global when the victim's process first creates one inside a loop, resulting in a diversion from what would happen when a real GbHammer attack is executed.

VII. Related Work

GbHammer and the exploits based on it are novel in three aspects. First, GbHammer reveals the risk of management bits in page table entries being hammered, while previous studies focus on hammering the address parts. Zhang *et al.* [2] show that an attacker can access protected memory regions such as the kernel space by maliciously overwriting address translation information on a page table entry. Yuan *et al.* [5] demonstrate that a bit flip induced by RowHammer attack can bypass memory isolation forced by virtualization. Second, the arbitrary binary execution exploit is advanced in its uniqueness. Although many exploits based on RowHammer have been reported [2]–[6], we are the first to achieve arbitrary code execution as far as we know. Finally, the data snooping exploit could be advanced in its reliability and read speed. A previously known exploit that enables reading data is RAMBleed [3]. Because RAMBleed uses RowHammer to guess the data bits stored in the aggressor row by observing bit-flips incurred to the victim row, it only achieves *0.31 bits/second at an accuracy rate of 82%* [3]. Our data snooping exploit can read data with much faster speed and higher accuracy once a shared page is successfully created because a data read from the shared page is merely a normal memory copy (from the shared page to either a register or another memory page).

VIII. Conclusion and Future Work

While various exploits based on RowHammer have been discovered, this research focuses on the risk of the global in a page table entry (PTE) being hammered. We found that two new exploits, arbitrary binary execution and data snooping, are possible by flipping the global bit with RowHammer. We also demonstrated that they actually work for a real Linux kernel on a cycle-accurate CPU simulator. Our future work includes observing the behavior of GbHammer in real machines and reproducing GbHammer in RISC-V to prove that it indeed has a larger risk than in x86 and ARMv7.

Acknowledgment

This work was supported by JST Global Science Campus Experts in Information Science, and JST, PRESTO Grant Number JPMJPR22P1, Japan. We thank the anonymous reviewers for their valuable feedback to improve this paper.

References

[1] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *International Symposium on Computer Architecture (ISCA)*, 2014, pp. 361–372.

[2] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses," in *International Symposium on Microarchitecture (MICRO)*, 2020, pp. 28–41.

[3] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading bits in memory without accessing them," in *IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 695–711.

[4] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-Bomb: Locking down the processor via rowhammer attack," in *Workshop on System Software for Trusted Execution (SysTEX)*, 2017, pp. 1–6.

[5] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation," in *USENIX Security Symposium*, 2016, pp. 19–35.

[6] S. Li, X. Wang, M. Xue, H. Zhu, Z. Zhanga, Y. Gao, W. Wu, and X. S. Shen, "Yes, one-bit-flip matters! universal DNN model inference depletion with runtime code fault injection," in *USENIX Security Symposium*, 2024, pp. 1–16.

[7] Intel, "Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3A," https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html, 2023.

[8] P. Muntean, R. Viehoever, Z. Lin, G. Tan, J. Grossklags, and C. Eckert, "iTOP: Automating counterfeit object-oriented programming attacks," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021, p. 162–176.

[9] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 745–762.

[10] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[11] Linux manual page, "mmap(2)," https://www.man7.org/linux/man-pages/man2/mmap.2.html, 2023.

[12] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *ACM Conference on Computer and Communications Security (CCS)*, 2016, p. 1675–1689.

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

[14] N. Bostanci, I. E. Yuksel, A. Olgun, K. Kanellopoulos, Y. C. Tugrul, G. Yaglikci, M. Sadrosadati, and O. Mutlu, "CoMeT: Count-min sketch-based row tracking to mitigate rowhammer with low cost," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2024, pp. 593 – 612.

[15] A. Saxena and M. Qureshi, "Start: Scalable tracking for any rowhammer threshold," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 578–592.

[16] gem5, "The gem5 simulator," https://www.gem5.org/, 2024.

[17] Ayaz Akram, "Tutorial: Run Full System Linux Boot Tests," https://gem5art.readthedocs.io/en/latest/tutorials/boot-tutorial.html, 2019.

[18] F. Thomas, L. Gerlach, and M. Schwarz, "Hammulator: Simulate now – exploit later," in *Third Workshop on DRAM Security (DRAMSec)*, 2023, pp. 1–7.

[19] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for Cross-CPU attacks," in *USENIX Security Symposium*, 2016, pp. 565–581.

[20] C. Helm, S. Akiyama, and K. Taura, "Reliable reverse engineering of intel dram addressing using performance counters," in *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020, pp. 1–8.

[21] ARM, "ARM architecture reference manual ARM v7-A and ARM v7-R edition," https://developer.arm.com/documentation/ddi0406/b, 2008.

[22] RISC-V, "The RISC-V instruction set manual volume II: Privileged architecture," Version 20240411, 2024.