TSINGHUA SCIENCE AND TECHNOLOGY
ISSNll1007-0214 03/10 pp535–545
DOI: 10.26599/TST.2018.9010134
Volume 24, Number 5, October 2019

Memway: In-Memory Waylaying Acceleration for Practical Rowhammer Attacks Against Binaries

Lai Xu, Rongwei Yu*, Lina Wang, and Weijie Liu

Abstract: The Rowhammer bug is a novel micro-architectural security threat, enabling powerful privilege-escalation attacks on various mainstream platforms. It works by actively flipping bits in Dynamic Random Access Memory (DRAM) cells with unprivileged instructions. In order to set up Rowhammer against binaries in the Linux page cache, the Waylaying algorithm has previously been proposed. The Waylaying method stealthily relocates binaries onto exploitable physical addresses without exhausting system memory. However, the proof-of-concept Waylaying algorithm can be easily detected during page cache eviction because of its high disk I/O overhead and long running time. This paper proposes the more advanced Memway algorithm, which improves on Waylaying in terms of both I/O overhead and speed. Running time and disk I/O overhead are reduced by 90% by utilizing Linux *tmpfs* and in-memory swapping to manage eviction files. Furthermore, by combining Memway with the unprivileged *posix_fadvise* API, the binary relocation step is made 100 times faster. Equipped with our Memway+fadvise relocation scheme, we demonstrate practical Rowhammer attacks that take only 15–200 minutes to covertly relocate a victim binary, and less than 3 seconds to flip the target instruction bit.

Key words: Rowhammer bug; Waylaying algorithm; in-memory swapping; page cache eviction

1 Introduction

The Rowhammer bug is a software exploit of the Dynamic Random Access Memory (DRAM) hardware's disturbance error[1], in which the adversary repeatedly accesses ("hammers") adjacent DRAM cells, causing charge loss and flipping certain bits between them[2]. It can bypass most Operating System (OS)-based memory access control techniques because the victim memory is not accessed at all. The Rowhammer attack is considered to be more intrusive than other micro-architectural covert channel attacks such as Meltdown[3], Spectre[4], or Whisper[5], for it is capable of actively modify memory contents, enabling privilege-escalation and escape attacks on various platforms. Since Rowhammer's initial discovery, researchers have tested it on various targets, such as browser sandbox[6], Virtual Machine (VM) page table[7], secret key[8], scripts[9], video buffers[10], etc. Although most of these targets are merely text and data, it is known that attacks against codes and binaries represent a greater threat if they are feasible.

One of the challenges in attacking binaries is how to perform memory manipulation; that is, in order to exploit random bit flips in DRAM memory against codes, the adversary needs dedicated algorithms to move the victim binary onto the target bug location. In 2017, Gruss et al.[11] presented a novel flip-in-the-wall attack against binary programs, while proposing the Waylaying algorithm as the manipulation technique. In the Waylaying algorithm, the operating system is

• Lai Xu, Rongwei Yu, Lina Wang, and Weijie Liu are with the Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China. E-mail: xulai1001@sina.cn; roe-we.yu@whu.edu.cn; lnwang@whu.edu.cn; liuweijie@whu.edu.cn.

∗ To whom correspondence should be addressed.
Manuscript received: 2018-10-10; accepted: 2018-11-10

forced to relocate binary images to random addresses in the page cache. When this process is repeated sufficient number of times, the binary will finally be located at a certain bug address where the adversary can easily launch a one-location Rowhammer attack. The Waylaying algorithm is very stealthy, for it works entirely in free space and does not increase memory usage metrics. Additionally, by putting the attack into Intel Software Guard Extensions (SGX) enclaves, the adversary can avoid performance counter-based detection schemes.

However, the Waylaying algorithm faces shortcomings in practice, related to its low speed and high I/O overhead. Experiments show that it takes $3-10$ seconds to evict and relocate a target binary once (depending on disk throughput), and this random relocation may require repeating 10^4-10^5 times. As a result, the Waylaying process may take too long time to complete. Because the Waylaying algorithm is based on eviction (using disk files to evict page cache contents), its heavy disk I/O overhead can be easily detected outside enclaves. This also means that it takes a long time to start new processes, because the pre-cached code and data files have been totally evicted. Although the original Waylaying algorithm is capable of attacking binary programs, it lacks the flexibility to do so effectively.

Rowhammer, one of the newest micro-architectural threats, is also considered one of the most intrusive. This article proposes Memway, an improved version of the Waylaying algorithm, as a new method for engaging in Rowhammer attacks. Our proposed method has significant advantages over Waylaying, in terms of both speed and resources usage. Our method utilizes the Linux *tmpfs* to hold the eviction file, and uses in-memory swapping to perform eviction. Applying detailed analysis of Linux page cache interfaces, we combine Memway with the *posix_fadvise* API to make highly efficient binary relocations, thereby shrinking the total running time from days to minutes. Our experimental results show that the speed of relocating a target binary is around 100 times faster than Waylaying on average.

In sum, the main contributions of this paper are as follows:

• *Memway* utilizes Linux *tmpfs* to eliminate most of the disk I/O overhead of the original Waylaying algorithm, thereby executing page cache eviction more quickly and with less impact on system performance.

• The Memway+fadvise binary relocation scheme can effectively move a target binary onto an arbitrary memory address. Experimental Rowhammer attack scenarios applying our proposed method run much more quickly, completing in a matter of minutes.

2 Background

2.1 Related works on Rowhammer attacks

DRAM manufacturers discovered the DRAM disturbance error in modern Dual In-line Memory Modules (DIMM) modules prior to 2013, but it remained merely a hardware reliability issue until 2014, when Kim et al.[1] and Seaborn and Dullien[6] demonstrated the Rowhammer exploit. Instead of exploiting software memory utilization vulnerabilities (such as use-after-free[12] and API-tainting[13]), Rowhammer takes advantage of hardware bugs to breach higher-level software stacks, making it a more fundamental and a more difficult one to detect and recover from.

Since Rowhammer's initial discovery, researchers have tested it against various targets. Its potency in circumventing OS access control and software-based memory protection modules are well recognized. Kim et al.[1] gave the first systematic analysis of the Rowhammer bug and carried out the *mov+clflush* primitive. Seaborn and Dullien[6] proposed the first real-world attack scenario of escaping a NaCl sandbox with a double-sided Rowhammer attack. Razavi et al.[8] exploited page deduplication to flip secret keys in neighboring VMs. In this method, the adversary abuses the kernel's same-page merging mechanism to lead the victim to use the bug page instead of its original copy. In reaction to this attack, most public cloud providers have now disabled page deduplication. In another approach, Xiao et al.[7] abused the Xen page table Hypercall interface to copy a Page Directory Table (PDT) to the bug page, then by flipping a certain page directory item, showed that the adversary can use a forged page table to perform a VM escape and gain unlimited memory access. Gruss et al.[11] proposed attacks against binary images in the page cache. The author used Waylaying and Chasing to manipulate the page cache, and presented a new hammering primitive called "one-location hammering", which was more difficult to achieve success with but also more difficult to detect. Cheng et al.[10] exploited the User-after-Kernel (UaK) shared memory to forge attacks, with Memory Ambush provided as the manipulation method

and the kernel's video buffer used to execute a single-sided Rowhammer attack.

On the other hand, Rowhammer mitigation techniques have also aroused the interest of many researchers. Hardware mitigation includes both neutralization and elimination[14] approaches. As for software mitigation, Gruss et al.[11] places the existing methods into 5 categories: static analysis[15], counter-based[16], pattern-based[17], memory abuse prevention[18], and physical proximity prevention[19]. All of these mitigation methods make it harder for an adversary to launch Rowhammer attacks, but only next-generation memory may eliminate the hardware flaw.

2.2 Linux page cache

The Linux page cache is a transparent cache holding code and data files loaded from storage disks[20]. The OS preloads these files in free memory spaces so that processes start more quickly, thus improving overall system performance. There are two key characteristics of the page cache. First, if a process only reads or executes a file in the page cache, this portion of memory will not count towards the process's individual set, and will still be counted as free space. Second, if the memory is nearly full, the OS prefers to evict data files, ahead of binaries, from the page cache to reclaim memory.

The Linux memory subsystem maintains virtual pages in two linked lists: *active_list* and *inactive_list*. When free memory is scarce, the OS will start to swap out inactive pages using certain eviction algorithms (mainly Least Recently Used (LRU)). However, in 4.0 and later kernels, pages dismissed by LRU are not evicted right away. Instead they will be put into *pagevecs*, and the OS will swap *pagevecs* out in batches when convenient. Consequently, userspace or kernel developers can only require a page to be swapped out; they cannot dictate the exact time or sequence for the page to be swapped.

Linux provides a number of APIs and file interfaces for user programs to tweak the page cache. However, when and where the OS puts the cached memory remains transparent to userspace.

2.3 Linux ramdisk

Linux has three kinds of in-memory file systems, namely *tmpfs*, *ramfs*, and *ramdisk*[21]. The most common is *tmpfs*, backing the OS's temp directories. Files in *tmpfs* are copied into the page cache in free memory space, and *tmpfs* memory can be swapped. The size limit of *tmpfs* is half of the physical memory size. On the other hand, *ramfs* does not have a size limit or swapping mechanism, and therefore is not used by the OS by default and must be mounted manually. The *ramdisk* (/dev/ramN) is an earlier block device interface of the memory, the available size of which is very small.

3 In-Memory Acceleration of the Waylaying Algorithm

3.1 Analysis of the vanilla Waylaying algorithm

The Waylaying algorithm proposed by Gruss et al.[11] aims to evict the target binary from the page cache and force the OS to reload it into a different address. The eviction is achieved by memory mapping a large disk file with read and executable privileges, and reading every 4 KB page of the file. As the OS loads this large file into the page cache, the cache will consume all free memory and finally start to evict existing code and data. Free memory metrics are unchanged during the process. A Linux API, *mincore*, can be used to conveniently detect whether the target binary has been evicted from the page cache, although a stealthier approach without API calls is also made available.

We performed a detailed analysis of the speed of the Waylaying algorithm, and found that the eviction time mainly depends on two factors: the amount of free memory and the memory/disk access time. Eviction must be achieved by page swapping and, as mentioned above in Section 2.2, the Linux system uses LRU to swap pages by default. This means that, on every iteration, the Waylaying algorithm must load an evict file larger than the free memory size so as to make the target binary image the least recently used. It is clear, therefore, that the execution time will be longer with a larger amount of free memory space.

Meanwhile, different portions of memory have different access speeds. We tested the memory access time and eviction rate of Waylaying as follows: we used *mmap*() to load a randomized data file into memory, read once from each 4 KB page, and measured the access time. Access is usually uncached, because the file is randomized and has a large size equal to the size of physical memory. We repeated the process several times to count the average eviction success rate. The result is shown in Fig. 1, on which deeper dots represent more frequent occurrences of certain access time.

Explanations to Fig. 1: As long as the evict file is smaller than the available memory size, pages remain

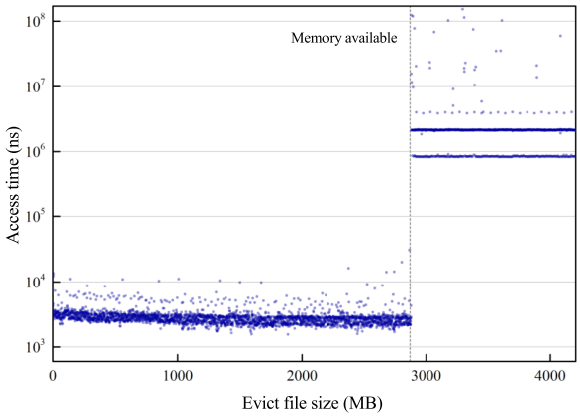


Fig. 1 Access time of different portions of memory.

in memory and the access times are between 10^3 and 10^4 nanoseconds. The periodical tremors in the access time chart may imply extra page table walks for newly loaded memory ranges. The total time for all in-memory access (from 0 to the available memory size) is less than 1 second. However, no eviction occurs during this period.

In contrast, when the available memory size is exceeded, old pages must be swapped out to disk, and the access times rise sharply towards $10^7 - 10^8$ nanoseconds, because 4 KB random reads are extremely slow on Hard Disk Drives (HDDs). At the same time, the eviction rate rises drastically by $+100$ MB of available memory, which is a desirable result.

Combining the above results, we conclude that swapping is essential for eviction, but in a disk swapping scenario, the memory access speed is extremely slow.

3.2 Memway: In-memory swapping with *tmpfs*

To carry out eviction, the vanilla Waylaying algorithm must use disk swapping, which is extremely slow. To increase the speed, our algorithm uses memory swapping instead of disk swapping, and make it work entirely in memory. Accordingly, this section describes the details of Memway, which uses the *tmpfs* ramdisk as a substitute for disk swapping.

Because the OS does not allocate duplicated memory for *mmap*() requests for files in the page cache-based *tmpfs*, simply moving the evict file into *tmpfs* is not sufficient to cause swapping. Thus, additional wrappers are needed to make the OS treat *tmpfs* as a regular disk file.

We can create a File System (FS) within the large temp file, utilizing FS-layer abstraction to shield it from OS-level deduplication. Noting that many Linux file systems, such as *ext4* and *xfs*, also use page cache to improve performance, and therefore are not applicable in our scenario, our algorithm choose *NTFS* to hold the evict file in *tmpfs*; it will be duplicated in memory when being *mmap*()-ed. The *tmpfs*-based evict file hierarchy is shown in Fig. 2.

As the *tmpfs* can fill at most **half** of the physical memory size, and allowing for filesystem metadata overheads, we can create an in-memory evict file that is slightly smaller than half of physical memory. When this file is *mmap*()-ed, the OS will load it into the other half of the page cache, since *NTFS* does not support page caching. These two counterparts of the evict file can fill up the whole free space to achieve eviction. The appropriate evict file size can be calculated as follows:

$$EvictFSSize = MemAvailable \times \frac{1 + OverflowRate}{2 + MetadataRate},$$

$$EvictFileSize = \frac{EvictFSSize}{1 + MetadataRate}.$$

where *MemAvailable* is the current available memory size, *MetadataRate* is the overhead of file-system metadata, and *OverflowRate* is a multiplier of the available memory size to make the file bigger so as to ensure eviction of the page cache. The corresponding evict file generation scheme is described in Algorithm 1.

After the evict file is created, we can use it to invoke page cache eviction. Our proposed *Memway* page cache

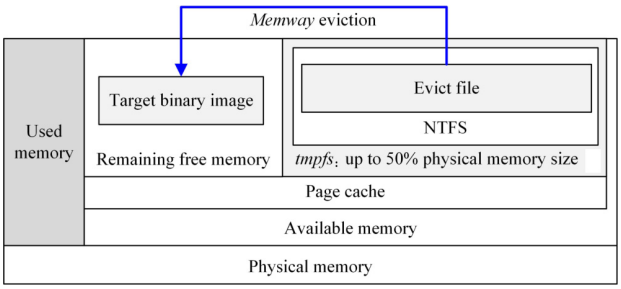


Fig. 2 Evict file hierarchy.

Algorithm 1 In-memory evict file generation scheme

Input: Desired sizes *EvictFSSize* and *EvictFileSize*
Output: Target evict file *E* with desired size
1 Use *fallocate*() to create file *F* in *tmpfs* with size = *EvictFSSize*
2 Use *mkntfs* to create NTFS filesystem in *F*
3 Mount *F* to directory *D* as a loop device
4 Use *fallocate*() to create file *E* with size = *EvictFileSize* inside directory *D*
5 Use *E* as the target evict file.

eviction algorithm is depicted in Fig. 3. In Fig. 3a, we populate half of the available memory with the evict file in *tmpfs*, using the file hierarchy outlined above. The evict file is then *mmap*()-ed into the remaining free space in Fig. 3b, so that the target binary is evicted from the page cache. We repeat the process in Fig. 3c, so that the OS will reload the binary randomly until the target hits one of the exploitable addresses (as defined by Rowhammer templates, explained below in Section 5.2).

One issue with *tmpfs* is that it may be swapped to disk and thus fail to invoke eviction with a given *EvictFileSize*. This can be avoided by disabling swap partitions or by using *ramfs* instead. In practice, cluster servers with large memory pools may in fact disable swap partitions, because their load-balancing system will migrate workloads, avoid memory depletion or swap overheads. This opens up attack surfaces for our approach.

4 Efficient User-Space Page Cache Manipulation with Memway+Fadvise

Because of the *pagevec* mechanism described in Section 2.2, neither the user nor the kernel can directly control when and where to evict the page cache. The Waylaying process must run many times while waiting for the random relocation by the OS to deliver the binary to the target position. This section describes how to combine state-of-the-art page cache eviction schemes with Memway.

4.1 Analysis of page cache eviction interfaces

Section 3 has proposed Memway to achieve fast eviction-based binary relocation. Meanwhile, Linux has interfaces for controlling page caches besides eviction; they are listed in Table 1.

These approaches can be divided into two categories: system call-based and eviction-based. System call-based approaches include the *posix_fadvise* and *fork* methods, which are very fast but unlikely to reach new pages. In contrast, eviction-based Waylaying and Memway methods overhaul the whole page cache so that the OS always reallocates random new pages for the target binary. However, they are extremely slow and inflict a heavy disk load on the system.

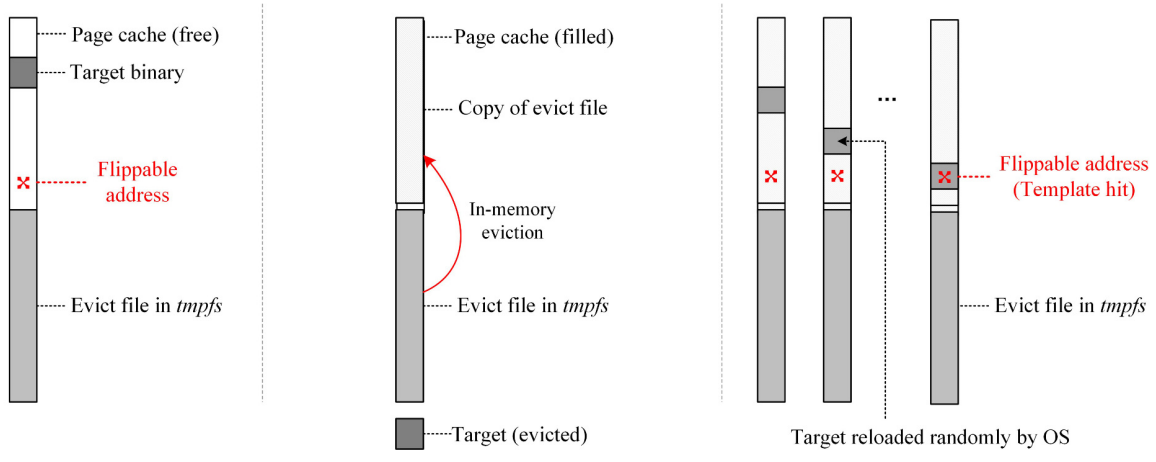
4.2 Combined binary relocation scheme

We can combine these two approaches to achieve both good speed and wide memory coverage. To begin, *posix_fadvise* is used to quickly relocate the target binary, and repeated until it fails to generate a new address in N continuous runs. The *Memway* algorithm is then executed for once to rearrange the whole page cache; because the page cache is thereby overhauled, the target is forced towards a new address and subsequent *posix_fadvise* calls can again generate new addresses.

Our *Memway+fadvise* binary relocation algorithm is described in Algorithm 2. As the *posix_fadvise* API call is significantly faster than pure Waylaying, enumerating over memory pages with our method can be far more

Table 1 Comparison of page cache manipulation interfaces.

Interface	Privileged	Speed (s)	New-page (%)
/proc/vm/drop_caches	Yes	<10	>80
Waylaying/Memway	No	<10	>90
posix_fadvise	No	<0.1	<10
fork Chasing[11]	No	$<10^{-3}$	<1



(a) Evict file fills *tmpfs* (b) Evict file is *mmap*()-ed to evict target binary (c) Repeat until hitting an exploitable template

Fig. 3 Illustration of the page cache eviction process of Memway algorithm.

Algorithm 2 ***Memway+fadvise* relocation scheme**

Input: bt: *BinaryTarget* target binary info (described in Section 5)
T_p: *Template* collection of all hammer templates
Result: The in-memory image of target binary will be moved onto any of the template addresses.

```
repeat_count ← 0, paddr_tried ← {}
mmap bt.binary with read-only access
Access bt.base once to load it into memory
p ← physical address of bt.base read from pagemap
munmap bt.binary, clean up memory
for each t in T_p do
    if (p, bt.offset, bt.value) ==
      (t.base, t.offset, t.value) then
        return;   /* matches found */
    else
        if p not in paddr_tried then
            insert p into paddr_tried
            repeat_count ← 0
        else
            repeat_count += 1
        end
        if repeat_count > threshold then
            repeat_count ← 0
            use Memway to relocate binary
        else
            posix_fadvise(bt.binary, 0, bt.size,
              POSIX_FADV_DONTNEED)
        end
    end
end
goto Step 2
```

efficient, as shown in Section 6.

5 Row-Hammering Binaries in Practice

This section provides details of a practical Rowhammer attack against binaries, with *Memway* and the combined binary relocation scheme proposed in Section 4.

5.1 Preparation: Memory and binary analysis

In order to find bit flips and use them to invoke meaningful alterations in binary code, the preparatory steps of offline memory and binary analysis must be undertaken.

Memory analysis is needed to find out the target machine's Central Processing Unit (CPU) and DIMM models, to infer the physical-to-DRAM address mapping scheme. This scheme is critical because the adversary needs to precisely control the DRAM rows during the attack process. The measuring algorithms are provided in earlier research; e.g., Refs. [7, 22]. This process can be done offline — once the model types are recognized, the adversary can use a similar hardware build to measure the DRAM mappings, which will be applicable on every machine with the same hardware. Another offline job is binary analysis, which involves finding out some critical bytes within the code, where a single bit flip will cause an exploitable change in behavior. For example, one bit flip might turn a *jnz* instruction into a *jz*. This can be done with one of many static binary analysis tools. The flippable target bits in binaries are defined as a 4-ary tuple:

$$\textit{BinaryTarget} \rightarrow (\textit{binary}, \textit{base}, \textit{offset}, \textit{value}),$$

where *binary* is target binary file; *base* and *offset* are target page base address (4k-aligned) and in-page offset; and *value* is the flipped byte value.

5.2 Templating

Because the addresses of potential bug points vary with each DIMM, we need to scan the physical memory of the target machine for as many bug positions as possible. This process is called "templating", and the flippable bytes (bits) are named "Rowhammer templates". Because bug positions are unique and stable on each DIMM unit, templating needs to be done **only once**, and any number of subsequent attacks can make use of the results.

The templating phase is done with existing Rowhammer primitives, such as the Double-sided and/or Single-sided Rowhammer. The scanner allocates a portion of the system memory, hammers each row within, and examines possible bit flips.

We define the template as a tuple. When the scanner finds a flip, it will record the template in data file.

$$\textit{Template} \rightarrow (\textit{base}, \textit{offset}, \textit{value}, p, q, \textit{dir}, f_p, f_r),$$

where *base* and *offset* are victim page base address and in-page offset, locating the target byte; *value* is the flipped byte value, locating the flipped bit within the victim byte; p and q are double-sided hammering physical addresses to invoke bit flips; *dir* is the flipping direction, either 0-1 or 1-0; and f_p and f_r are auxiliary arguments flips-per-page and flips-per-row, respectively. These arguments tell how many flips will occur in the target page/row when hammering p and q.

The adversary then needs to match the *BinaryTarget* tuple obtained during binary analysis with the templates. While the offset should be equal and the value and direction should match, we also need to avoid unexpected alterations in other positions of target

code; therefore, we expect $f_p = f_r = 1$ in a working template.

5.3 Memory manipulation with Memway

After finding appropriate templates, the adversary needs to use memory manipulation to place the target onto one of them. The goal of relocation is to make *BinaryTarget.base*==*Tmpl.base*, which means that the base address of the victim page (rather than the base address of the binary) is put on the target. We use the optimized Memway algorithm to achieve efficient relocation.

As mentioned in the appendix of Ref. [11], the speed of manipulation is mainly related to (1) the physical memory size; (2) the speed of a single relocation operation; (3) the number of exploitable templates; and (4) the number of exploitable positions in the binary. The typical run time of the original *Waylaying* method ranged from $10 - 500$ hours. Memway shrinks this run time to $10 - 200$ minutes, which is an order of magnitude faster.

5.4 Hammering and post-exploit

With proper memory manipulation, it is very easy to launch the successive steps: we acquire address p and q, and hammer the target binary. After the target code bit is flipped, the binary's behavior will change accordingly, and the adversary can exploit this change towards privilege escalation or circumvention.

6 Experiments

6.1 Testbed

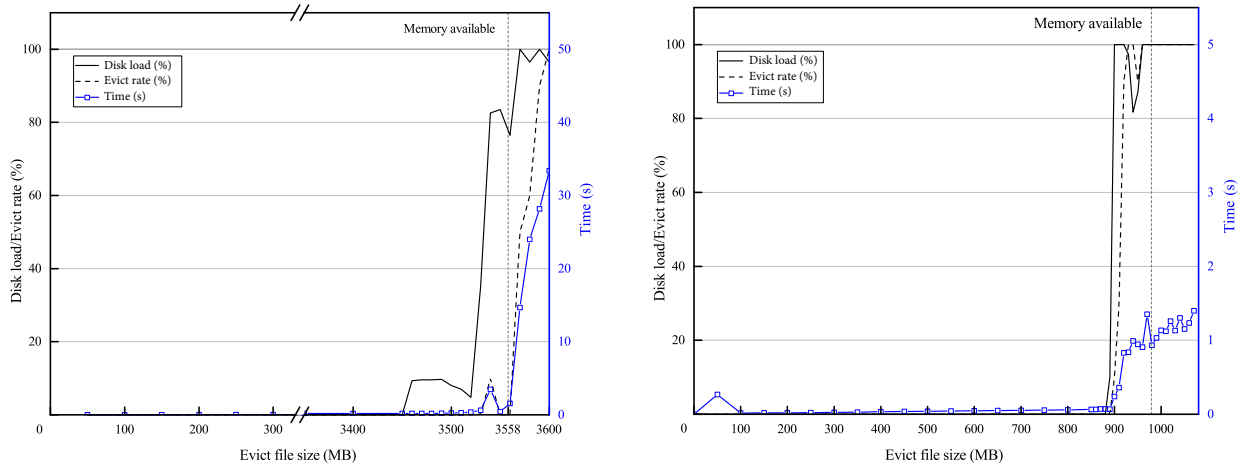
This paper launches experiments on an Intel i5-3470 platform, with a single DDR3-1600 4 GB DIMM module made by Hyundai (Hynix). We use this DRAM module because its row address mapping is straightforward: the higher 15 bits of physical address are directly mapped to row addresses. There are 32 768 rows on this module and the row size (all cells with the same row number) is 128 KB, 32 pages per row.

6.2 Comparison with original Waylaying

The execution time, I/O overhead, and memory consumption of Memway compared with the original Waylaying method are depicted in Fig. 4. The left y-axis shows the percentage of disk load and the target eviction rate, while the right y-axis refers to the execution time of a single eviction.

In the Waylaying scenario, the amount of available memory is 3558 MB. All three indicators rise sharply when the evict file size is larger than 3558 MB, and the running time is more than 10 seconds. Meanwhile, in the Memway scenario, the amount of available memory is 980 MB. This is because the Memway algorithm has utilized maximum *tmpfs* space. The three lines also rise when the evict file size is around 900 MB, however the total execution time is less than 2 seconds.

From the above figures, it is seen that the running time of our algorithm is significantly lower than the original Waylaying. The disk load reaches 100% only when the memory limit is exceeded. This is because the main contribution to load in Memway is the repeated reloading of the same target binary from in-memory *tmpfs*, rather than random 4 KB disk reads. Therefore, the overhead is significantly lower. While our approach costs about **half** of all available space, the system monitor will show the rest as still available (about 35% free space, which is lower than typical alert levels). The



(a) Waylaying (b) Memway

Fig. 4 Disk load/evict rate (%) and evict time (s) over evict file size (MB) for (a) Waylaying and (b) Memway algorithm.

memory consumption is all attributable to *tmpfs*, while the adversary's actual memory cost is negligible.

6.3 Memory coverage of repeated binary relocation

Binary relocation schemes are critical in building attacks against binaries. The memory coverage over time of these schemes is depicted in Fig. 5.

Our Memway+fadvise scheme can obtain a stable increase in unique pages, whereas using *posix_fadvise* or Memway alone is much less effective. Although the Chasing method can quickly enumerate more than 2×10^4 unique pages, it suddenly stops growing at that point because of the exhaustion of *fork*() resources.

We expect that repeatedly running a relocation scheme can reach more new pages rather than used pages. Therefore, we introduce *uniqueness* as a measure of the effectiveness of different schemes. Let $n(t)$ be the number of new pages scanned within t seconds, and $p(t)$ be the number of all pages scanned, then the uniqueness $u(t)$ is their ratio:

$$u(t) = \frac{n(t)}{p(t)}.$$

A high uniqueness rate implies a low repeat rate. The *uniqueness* rate for each of the four schemes is shown in Fig. 6.

Our method (Memway+fadvise) maintains >50% uniqueness; that is, more than 50% of reached pages are new. Chasing and *posix_fadvise* both suffer from low uniqueness (high repeat rate) such that used alone they cannot scan the whole memory. Although Memway (when used alone) has the highest uniqueness rate (for it is constantly overhauling the page cache), it is order

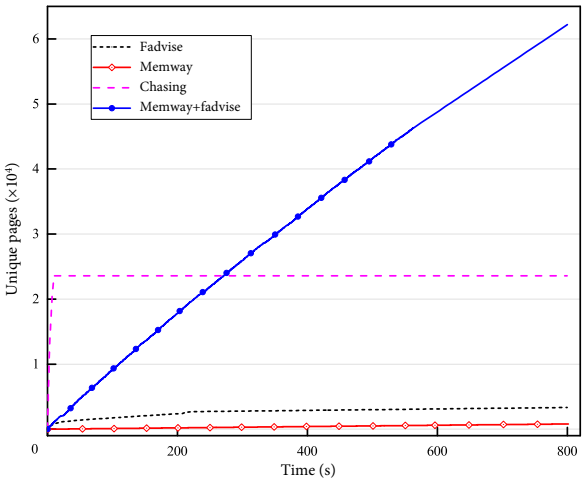


Fig. 5 Memory coverage of different relocation schemes.

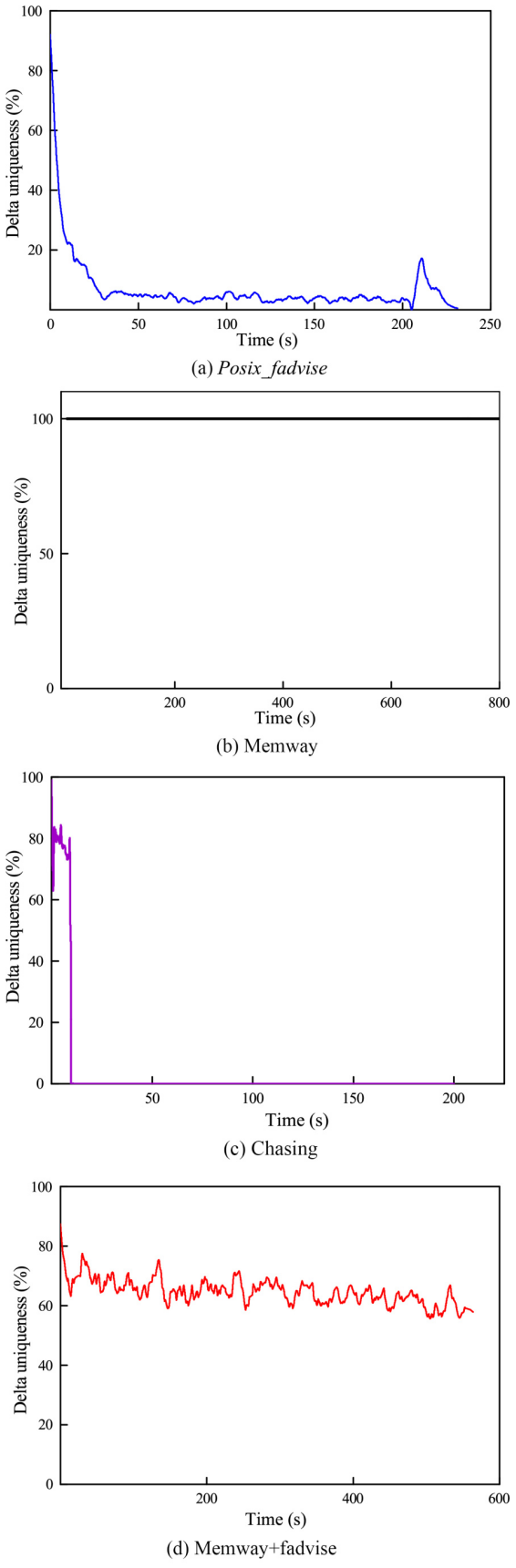


Fig. 6 Unique page rate of various relocation schemes.

of magnitudes slower than other mixed approaches. Therefore, using Memway alone cannot scan a lot of memory in a short period of time.

6.4 Rowhammer against binaries

To demonstrate a real attack scenario, we used our Memway+fadvise relocation scheme to attack an unmodified test program binary. The steps and results are as follows.

Analysis. The test program prints out a string of underlines ("_", 0x5f). Binary analysis shows that the corresponding string field is at offset **0xfaf0**. If any of the underline bytes are flipped to zero by the attack, they can become other printable characters such as "[" (0x5b) or "O" (0x4f).

Templating. We ran the templating process on our test machine for 24 hours and found >50 000 flippable locations. 14 locations have their offsets on **0xaf0**, which can be used as templates for this attack. The templating result of some 512 MB memory is depicted in Fig. 7. Each block represents a complete memory row, which is 128 KB in size, and there are 4096 rows within 512 MB memory range. (Overall row size = 128 KB = 8 bits/cell × 1024 columns × 8 Banks/Chip× 8 Chips/Rank × 2 Ranks/Module. The Bank/Rank data is extracted by CPU-Z and Hynix DIMM specifications.)

The deeper a block is, the more bit flips were found in the corresponding row. In the most vulnerable row, more than 40 flips were detected within the row size of 128 KB.

Manipulation. After templating, we briefly held 2 GB of memory, found the attack pages, and freed all others. Then we used Memway+fadvise to perform memory manipulation. The target binary is placed onto one of the locations within 15 – 100 minutes.

Hammering and post-exploit. Now that we had placed the target binary on site, we first ran it once to check its original behavior. We then Row-hammered 1 000 000 times on the template locations and ran the program again. After hammering, the target byte (**0xfaf0**) was flipped from **0x5f** into **0x4f**, while the disk file remained untouched.

Figure 8 displays the Rowhammer result. After 4752 seconds, 572 236 random relocations were made with Memway+fadvise. The victim page finally reached

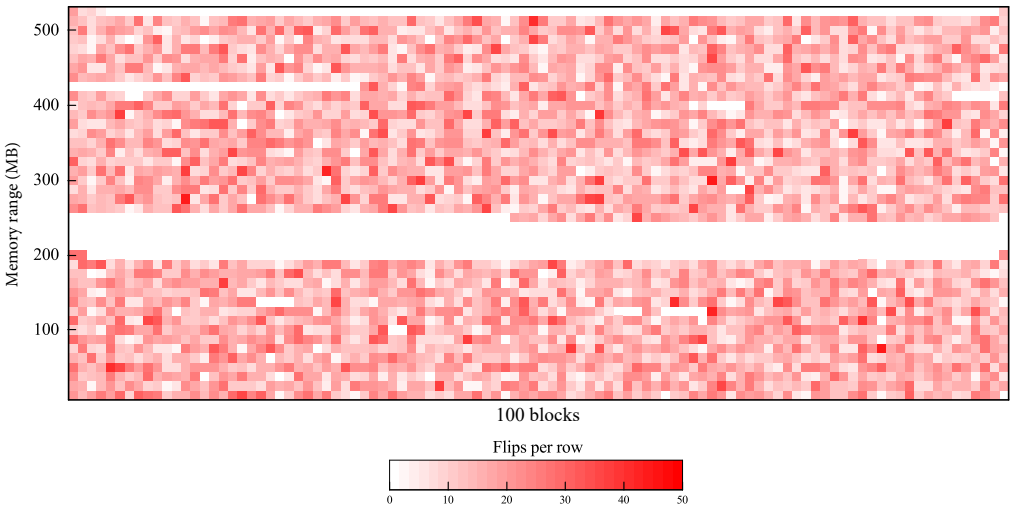


Fig. 7 Flips found within 500 MB range of DDR-3 module. Each block represents a complete memory row, which is 128 KB in size; each horizontal line has 100 blocks.

```
+ step 572233: abb81000 coverage: 1076416k / 4953444 k, uptime: 4752s.
+ step 572234: 660f6000 coverage: 1076420k / 4953444 k, uptime: 4752s.
+ step 572235: 98408000 coverage: 1076420k / 4953444 k, uptime: 4752s.
+ step 572236: 2aa2a000 coverage: 1076420k / 4953444 k, uptime: 4752s.
* Success: target pa=0x2aa2a000, hammer template=0x2aa2a000,2800,0x2aa09000,0x2aa4d000,0xef,0,1,1
- Hold the target...
- Waylaying complete. Time: 4752s.
- Check original program...
_____________________________________________________
* Hammering 1000000 times on 0x2aa09000 and 0x2aa4d000
- Check result
_____________________O_____________________________
- cleanup...
less@less-Lenovo:~/work/libhammer$
```

Fig. 8 Rowhammer result. "O" in result line is flipped by Rowhammer.

0x2aa2a000, and 0x2aa2aaf0 is a bug address which can flip **0x5f**("_") into **0x4f**("O"). After 1 000 000 rounds of Double-sided hammering on 0x2aa09000 and 0x2aa4d000, taking less than 1 second, the target binary string was successfully altered and the result string displayed a letter "O" among the underlines. Because our template satisfies $f_p = f_r = 1$, the other pages of the binary were left untouched. Consequently, the victim binary's in-memory behavior was successfully altered without crashing any part of the binary or the OS.

7 Conclusion and Future Works

The Rowhammer bug is a recent, intrusive micro-architectural threat. The proof-of-concept Waylaying algorithm has some practical shortcomings because of its inefficiency and high disk I/O overhead. This paper presents the Memway algorithm to address these shortcomings. As shown in our experiments, the running time and disk I/O overhead of our method is reduced by 90%, and the binary relocation step is accelerated by 100 times when Memway is further combined with the unprivileged *posix_fadvise* API. More importantly, equipped with the Memway+fadvise relocation scheme, it takes only minutes to covertly relocate a victim binary. Therefore, our proposed methods are practical for Row-hammering attacks against binaries.

Some problems to be solved are (1) building effective attacks on container environments and (2) testing Rowhammer bugs on newer generation memory units.

(1) Popular container virtualization solutions, such as Docker/Moby, use layered file-systems such as aufs to save space and memory, which implies that identical binaries between containers may well be shared in memory. Memway can be used to build Rowhammer attacks against these setups.

(2) The row addressing schemes of DDR4 memory can be easily reverse-engineered, which means that Rowhammer attacks are still feasible on DDR4. Experiments on the Rowhammer vulnerability of newer modules are essential to detect and prevent attacks against next-generation memory units such as DDR5.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Nos. U1836112, U1536204, and 61876134), the Fundamental Research Funds for the Central Universities (No. 2042018kf10281), Foundation of Key Lab of Information Assurance and Technology (No. KJ-17-101), and China Scholarship Council.

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 *Proc. 41st Int. Symp. Computer Architecture*, Minneapolis, MN, USA, 2014, pp. 361–372.

[2] Row hammer, https://en.wikipedia.org/wiki/Rowhammer, 2018.

[3] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, Meltdown: Reading kernel memory from user space, in *Proc. USENIX Security Symp.*, Baltimore, MD, USA, 2018, pp. 973–990.

[4] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al., Spectre attacks: Exploiting speculative execution, https://spectreattack.com/spectre.pdf, 2018.

[5] Z. Y. Wu, Z. Xu, and H. N. Wang, Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud, *IEEE/ACM Trans. Network.*, vol. 23, no. 2, pp. 603–615, 2015.

[6] 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.

[7] Y. Xiao, X. K. Zhang, Y. Q. Zhang, and R. Teodorescu, One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation, in *Proc. 25th USENIX Security Symp.*, Austin, TX, USA, 2016, pp. 19–35.

[8] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, Flip Feng Shui: Hammering a needle in the software stack, in *Proc. 25th USENIX Security Symp.*, Austin, TX, USA, 2016, pp. 1–18.

[9] D. Gruss, C. Maurice, and S. Mangard, Rowhammer.js: A remote software-induced fault attack in JavaScript, https://link.springer.com/chapter/10.1007/978-3-319-40667-1_15, 2016.

[10] Y. Cheng, Z. Zhang, S. Nepal, Still hammerable and exploitable: On the effectiveness of software-only physical kernel isolation, arXiv preprint arXiv: 1802.07060, 2018.

[11] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, and W. Schoechl, Another Flip in the wall of rowhammer defenses, in *Proc. 2018 IEEE Symp. Security and Privacy*, San Francisco, CA, USA, 2018, pp. 245–261.

[12] X. H. Han, S. Wei, J. Y. Ye, Z. Chao, and Z. Y. Ye, Detect use-after-free vulnerabilities in binaries, (in Chinese), *J. Tsinghua Univ.* (*Sci. Technol.*), vol. 57, no. 10, pp. 1022–1029, 2017.

[13] B. J. Cui, F. W. Wang, T. Guo, and B. J. Liu, Research of taint-analysis based API in-memory fuzzing tests, (in

Chinese), *J. Tsinghua Univ*. (*Sci*. *Technol*.), vol. 56, no. 1, pp. 7–13, 2016.

[14] A. Amaya, H. Gomez, and E. Roa, Mitigating row hammer attacks based on dummy cells in DRAM, in *Proc*. *2017 IEEE Int*. *Conf*. *Consumer Electronics*, Las Vegas, NV, USA, 2017, pp. 442–443.

[15] G. Irazoqui, T. Eisenbarth, and B. Sunar, MASCAT: Preventing microarchitectural attacks before distribution, in *Proc*. 8^{th} *ACM Conf*. *Data and Application Security and Privacy*, New York, NY, USA, 2018, pp. 377–388.

[16] J. Corbet, Defending against Rowhammer in the kernel, https://lwn.net/Articles/704920/, 2016.

[17] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, ANVIL: Software-based protection against next-generation rowhammer attacks, in *Proc*. *Twenty-First Int*. *Conf*. *Architectural Support for Programming Languages and Operating Systems*, Atlanta, GA, USA, 2016, pp. 743–755.

[18] 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 *Proc*. *2016 ACM SIGSAC Conf*. *Computer and Communications Security*, New York, NY, USA, 2016, pp. 1675–1689.

[19] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A. R. Sadeghi, Can't touch this: Practical and generic software-only defenses against rowhammer attacks, arXiv preprint arXiv: 1611.08396, 2017.

[20] Page Cache, https://en.wikipedia.org/wiki/Pagecache, 2018.

[21] Tmpfs, https://en.wikipedia.org/wiki/Tmpfs, 2018.

[22] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, DRAMA: Exploiting DRAM addressing for cross-CPU attacks, in *Proc*. *USENIX Security Symp*., Austin, TX, USA, 2016, pp. 565–581.

Lai Xu received the BEng degree from Wuhan University, China, in 2012. He is currently working towards the PhD degree in the School of Cyber Science and Engineering, Wuhan University, China. His main research interests include virtualization security and covert channel analysis.

Rongwei Yu received the BS degree from Northwestern Polytechnical University, China, in 2003, and the MS and PhD degrees from Wuhan University, China, in 2006 and 2010, respectively. He is currently an assistant professor at Wuhan University. He is a member of the China Computer Federation. His main research interests include cloud security and content security.

Lina Wang received the MS and PhD degrees from Northeastern University, China, in 1989 and 2001, respectively. She is currently a professor and PhD supervisor at Wuhan University. She is a senior member of the China Computer Federation. Her main research interests include system security and steganalysis.

Weijie Liu received the BEng and PhD degrees from Wuhan University, China, in 2012 and 2018, respectively. His main research interests include virtualization security and cloud security.