2016 IEEE 35th Symposium on Reliable Distributed Systems

# The Rowhammer Attack Injection Methodology

Keun Soo Yim
*Google, Inc.*
yim@google.com

***Abstract*—This paper presents a systematic methodology to identify and validate security attacks that exploit user influenceable hardware faults (i.e., rowhammer errors). We break down rowhammer attack procedures into nine generalized steps where some steps are designed to increase the attack success probabilities. Our framework can perform those nine operations (e.g., pressuring system memory and spraying landing pages) as well as inject rowhammer errors which are basically modeled as ≥3-bit errors. When one of the injected errors is activated, such can cause control or data flow divergences which can then be caught by a prepared landing page and thus lead to a successful attack. Our experiments conducted against a guest operating system of a typical cloud hypervisor identified multiple reproducible targets for privilege escalation, shell injection, memory and disk corruption, and advanced denial-of-service attacks. Because the presented rowhammer attack injection (RAI) methodology uses error injection and thus statistical sampling, RAI can quantitatively evaluate the modeled rowhammer attack success probabilities of any given target software states.**

***Keywords—Security evaluation methodology; cloud and hypervisor security; quantitative security analysis; and rowhammer error***

## I. INTRODUCTION

If the underlying computer system has some sources of user influenceable reliability errors (UIREs[1]), non-privileged users can exploit such errors and harm the reliability and security of privileged software running on top of the vulnerable hardware. Rowhammer error [1] is a practical example of UIRE, and can occur in multiple DRAM cells of the physically adjacent DRAM rows of an aggressor row if some cells in the aggressor row are repeatedly accessed in a short time interval. Recently, [6] reported two practical techniques to subvert an operating system kernel and a user-space sandbox by exploiting DRAM rowhammer errors. Because an attacker who just compromised one compute node in a target cloud service or infrastructure (e.g., a rootkit installed) can achieve various objectives and expand the control scope using some other mechanisms, UIRE poses new, important security challenges in the management of cloud data centers.

Both rowhammer attackers and defenders want to identify all possible attack target states in the privileged software used in a target computer system. If it is known that a specific state of a privileged software module is vulnerable to a rowhammer attack, an attacker can repeatedly try to corrupt that state by inducing a rowhammer error in the cells which store that target state in the memory. Consequently, knowing more attack target states increases the attack success probability. If a defender can identify such target states before any attackers, the defender can remove the vulnerabilities, make them hard to exploit, or place some detection techniques by rewriting target software or using other system-level mechanisms. In practice, only a small number of engineers (e.g., security experts with experience and domain knowledge) could identify practical attack target states and thus attackers could occasionally win in that race.

This paper presents a methodology, *Rowhammer Attack Injection* (RAI), to identify, validate, and evaluate rowhammer attack target states. RAI tackles two main technical challenges: One is about the cost of and difficulty in reproducing rowhammer errors in a laboratory environment. Validating a rowhammer attack needed at least one defective DRAM chip (or a DRAM module). Yet that could not validate many potentially vulnerable target state because every validation target state had to be stored in one of the defective DRAM cells of the used DRAM chips, while the exact locations and characteristics of defective DRAM cells vary depending on the chip. RAI thus models rowhammer errors in the architecture layer, and using some software mechanisms (e.g., breakpoint) injects the modeled rowhammer errors into certain cells of a logically adjacent row of a target DRAM row. As a result, RAI can emulate rowhammer errors in any selected DRAM row and thus can validate various target states without having to find and use any defective DRAM chips.

The other is to automatically reproduce realistic security attacks (e.g., privilege escalation). According to previous fault injection (FI) studies (e.g., [12]), random errors are likely to cause crash or hang failures even when some of the injected or propagated errors induce some out-of-bound memory accesses, which can potentially lead to realistic security attacks. We find that error injection alone is insufficient to reproduce high-level security attacks and thus model rowhammer attacks using nine generalized steps. Our RAI framework can conduct those nine operations where some (e.g., pressuring memory and spraying landing pages) are specifically designed to increase the chances of, for example, storing a target state in some identified cells and capturing a diverted control or data flow, and to improve the probability of producing possible attacks.

The main contributions of this paper are as follows:

- We develop a quantitative evaluation methodology of the rowhammer attack success probability of any given target software states by modeling multiple, essential operations of general rowhammer attacks, and using an error injection technique. We also present a framework which can assist actual experiments to identify, validate, or evaluate some rowhammer attack targets.

[1] Let us define UIRE as hardware reliability problems that can induce architecture- or software-visible errors where the error occurrence time or location is controlled by non-privileged programs and users.

1060-9857/16 $31.00 © 2016 IEEE
DOI 10.1109/SRDS.2016.10

- Our experiments conducted against a guest OS (operating system) of a state-of-the-art cloud VMM (virtual machine monitor) identified various kinds of guest system states, which if corrupted due to some rowhammer errors, can lead to privilege escalation, shell injection, memory/disk data corruption, or advanced denial-of-service (DoS) attacks. We also present the optimization techniques that improved the attack success ratios in our experiments.

The rest of this paper is organized as follows. §II introduces rowhammer errors. §III reviews the related works. §IV presents the RAI methodology and framework. §V describes the threat model and experimental setup. §VI presents the optimization techniques. §VII presents the experiment result. §VIII concludes this paper.

## II. Background

This section introduces the causes and characteristics of DRAM rowhammer errors, and reviews the existing mitigation techniques. Rowhammer errors got attention from the cloud industry because various DRAM chips shipped from 2011 to 2014 suffered from rowhammer errors and it is still possible to reface such problems in the future. Yet, none of the existing mitigation techniques can completely eliminate the possibility of rowhammer errors.

***(i) Cause of Rowhammer Error.*** A rowhammer error (e.g., bit flip) can occur in one or multiple cells in the DRAM *victim rows* if at least one of the physically adjacent rows (*aggressor row*) is repeatedly accessed (e.g., 139K times [1]) between their two consecutive row refreshes (e.g., 64ms). Processor on-chip caches absorb a large portion of such repeated memory accesses in many applications. Only some applications (e.g., fluid-animate in PARSEC, and cryptography) show more than 139K accesses (e.g., 400K accesses) to a single DRAM row within 64ms due to the inherently high cache miss ratio. A malicious program can use `clflush` (cache line flush) x86 instruction to flush a corresponding cache line after every access to a DRAM word. The program then selects two addresses from the same DRAM bank and alternatively accesses them in order to bypass an SRAM buffer that caches the last accessed row of that bank.

***(ii) Characteristic.*** The unique characteristics identified in [1] clearly show rowhammer errors are different from single- or double-bit soft errors whose models were widely used. That is an aggressor row could disturb >100 cells per row at a time. The reported probability of triple and quadruple bit errors per word was 0.02%, and <0.0002%, respectively, and the reported probability of a bit flip from '1' to '0' was similar to that from '0' to '1' partially because lost charges have to move to other capacitors. With the probability of ~97%, one or two rows were affected by an aggressor row. When the logical address is used, ~90% of the victim rows was the immediate neighbors of their aggressor row. A majority of rows was identified as aggressor rows (e.g., 100%, 99.96%, and 47% in three different chips that suffered from rowhammer defects) according to [1].

***(iii) State-of-the-art Mitigation Techniques.*** *(a) ECC and Scrubbing.* Computers (e.g., mobile and desktop) without ECC are vulnerable to rowhammer errors. In data centers, the widely used SEC-DED ECC corrects single-bit rowhammer errors. A double bit error alone, however, can be severe because system software typically kills the affected software module or halts itself if a double bit error is detected and a machine check exception is raised. 3bit errors and >3bit errors can evade SEC-DED ECC in practice. Not only that, although the probability is low, single or double bit errors accumulated in a word can also evade ECC protection unless the affected word is accessed by a program or scrubber before the errors are accumulated.

*(b) Adjusted DRAM Refresh Rate.* Doubling or increasing the DRAM refresh rate (e.g., every 64ms to 32ms) reduces the time window that a row has to be repeatedly accessed, and thus can significantly reduce the rowhammer error probability. In many CPUs, the DRAM refresh rate (e.g., tREFI parameter) can be controlled by using or updating BIOS. Doubled refresh rate, however, can increase the DRAM power consumption (e.g., ~2%) and slow down applications (e.g., 1-5%) [2] because some operations to a DRAM bank can be slightly delayed if a row in that bank is being refreshed. Such penalties are important criteria for builders and operators of large-scale data centers and can heavily influence their DRAM chip selection.

*(c) Target Row Refresh (TRR).* TRR is another architecture technique which can address rowhammer errors without large power and performance penalties. Ideally, a memory controller needs to measure the activation count of each row and issue a targeted refresh command to the corresponding DRAM bank if the count is above a threshold. DRAM chips that follow the DDR4 standard or its variants actually use its internal row remapping data and refreshes some physically adjacent rows [3]. Pseudo TRR (pTRR) implemented in some earlier memory controllers for DDR3 reduces the cost as compared with TRR that keeps the access count of each row although its coverage is affected. In general, with probabilistic row activation methods [4], refreshing neighbor rows of an accessed row with 0.1% probability showed only ~0.1% performance overhead and $10^{-7}$ error miss probability. Because multiple rows can be affected by rowhammer errors, the coverages of such probabilistic row activation methods would be lower than what was claimed.

*(d) Adaptive Techniques Based on Online Testing.* Using online testing data, adaptive techniques use a high refresh rate to some faulty rows, software remapping to hide faulty rows, or ECC to cover faulty cells [5]. Such techniques often require special hardware and software, and rely heavily on the result of initial online testing, which can miss certain types of hardware faults (e.g., intermittent) and naturally has a tradeoff between testing overhead and coverage. Such online techniques thus also do not prevent or correct rowhammer errors completely.

***(iv) Rowhammer Errors in the Wild.*** The existing detection, correction, and prevention techniques are a kind of statistical error mitigation techniques which work well with the hierarchical fault tolerance model used mainly for the system reliability. However, the remaining small error possibility can still be a serious concern for the system security because intelligent attackers may exploit uncovered, rare errors and subvert victim systems. Various kinds of computer hardware and software are used in cloud data centers. Many of those components are gradually deployed over the years using the latest technologies. As a result, some old devices are not fully equipped with new techniques (e.g., TRR of DDR4). Depending on the usage history and operating conditions (e.g., due to wear out or temperature), some devices can have much higher error rates than the others, while an attacker often need to compromise just one node in a target service or environment (e.g., a rootkit installed in a data center) in order to realize various objectives because the attacker can then expand the control scope using some other mechanisms.

## III. Related Work

***(i) Attacks Based on Random Data Errors.*** At least four pioneering studies suggested that random memory data errors can compromise the security of some critical software systems (e.g., web and FTP servers [7][8], firewalls [9], and JVM [10]). In [10], the errors are injected by placing a light lamp near the memory modules of a target computer, while [7][8][9] assumed soft errors naturally caused by cosmic rays, particle strikes, or other physical sources. The concept of critical state, which if corrupted can expose security vulnerabilities, is studied, and a few such states were identified in [7][9]. While those two early works mainly analyzed control data (e.g., authentication code), [8] showed that a corruption of non-control data (e.g., used to compute the condition of a control flow instruction) can also result in the same. [10] uses many landing pages in order to increase the probability of capturing a random jump because in that work errors were injected randomly by using a light lamp.

***(ii) Attacks Exploiting Rowhammer Errors.*** Some security engineers at Google are the first in presenting the possibility of security attack exploiting DRAM rowhammer errors [6]. One technique is to break the Native Client sandbox for Chrome web browsers. The other is based on a corruption of part of the virtual memory page table (i.e., page table entry, PTE) to directly access any part of the entire memory space. Another work from two European groups also suggested the possibility of JavaScript-based rowhammer attack [11]. Because `clflush` is unavailable for JavaScript, that technique generates specific memory access patterns and causes many cache conflict misses. The scope of that JavaScript-based work, however, did not cover identifications of some specific target states although the targets identified in [6] may be used in that context. That JavaScript-based work focused on presenting a novel idea that it is possible to introduce rowhammer errors on a remote node which an attacker does not have any permission to directly run software, as far as the remote victim system visits a website and downloads a malicious JavaScript program (e.g., injected by an attacker by exploiting an XSS vulnerability).

In general, validating rowhammer attacks on a real system was an expensive process that has been done only by a limited number of security experts who can access to defective DRAM modules and have deep systems knowledge and skills. Not only known rowhammer attack targets, attackers and defenders want to identify other new attack targets. Naturally, there is a race between those two groups because the defenders need to identify and harden target states before any attackers. Thus, we have long wanted to develop some systematic methodology to identify, validate, and even evaluate rowhammer attack targets.

Many previous FI studies (e.g., [12]) on OS kernels, which are the trusted computing base in many systems, showed that faults, randomly injected into the target system memory, likely cause crash or hang failures than data errors in the CPU-based computers. That implies us that the target states of rowhammer attacks must be carefully chosen. Considering various existing mitigation techniques that reduce the error rate, real-world attacks must be conducted adaptively by controlling the state and operational behavior of a target in order to increase the success probability and reduce the time to subvert a target.

## IV. Methodology and Framework

The presented *Rowhammer Attack Injection* (RAI) methodology breaks down rowhammer attack procedures into nine steps. Those nine steps are described in Subsection IV.A. The *Rowhammer Error Injector* (REI) is specifically designed for RAI. REI is a software-implemented fault injector whose fault models capture the key characteristics of the rowhammer errors (Subsection IV.B). REI is part of our RAI attack program that allows an attacker (or an experimenter) to conduct the modeled nine attack steps in a semi-automatic way (Subsection IV.C).

### *A. Modeling Rowhammer Attack Procedures*

We classify rowhammer attacks into two types based on the difference in their procedures. (1) *Whitebox rowhammer attack* (WRA) uses information about the virtual to physical memory mapping of a target system and typically increase the success probability because WRA allows an attacker to aim at specific target states and validate the results of various involved steps. (2) *Blackbox rowhammer attack* (BRA) does not have virtual to physical memory mapping information and thus relies on a good selection of a target state which can be easily replicated many times by non-privileged users. A BRA usually forces a target system to replicate target states many times and sprays landing pages in order to increase the success probability by increasing the chances of corrupting one of the replicated target states and capturing a diverted control flow or corrupted data flow. Both WRA and BRA are modeled using the nine steps:

Let us assume Alice is an attacker. She uses our RAI attack program, capable of processing the commands in Table I.

**Step 1.** *Identify and obtain memory blocks: one with an aggressor address and the other with a victim address.*

To do that, Alice sends the ‘allocate’ and ‘lock’ commands so that the RAI attack program can allocate $k$ memory blocks and lock them in the memory (i.e., avoiding page swap outs). If $k$ is 256 and the block size is 4MB, the total allocated memory size is ~1GB which is ~25% of the entire memory space if the total memory size is 4GB. Alice then uses the ‘profile’ command to find an aggressor row and at least one victim cells in the allocated $k$ blocks (see Fig. 1(1-2)). If no aggressor row is found, Alice can allocate more blocks; retry ‘profile’ using a different initialization mode; or move to another target node.

**Step 2.** *Select the type of attack target states.*

Alice now knows that the current target node has at least one rowhammer attackable cell. Alice thus needs to check the types and versions of software on the target node as well as how they are configured. She manually selects one or multiple potential target states. In order to help her selection, we present *system security vulnerability factor to UIRE* ($S^2VF$). $S^2VF$ is

TABLE I. KEY COMMANDS OF OUR RAI ATTACK PROGRAM.

| Command | Description |
|---|---|
| 'allocate' | Allocates a memory block with a specific size. |
| 'lock' | Locks a memory block to prevent it from being swapped out (using `mlock()`). |
| 'initialize' | Initializes a memory block by setting bits to a given one. |
| 'store' | Stores a payload to a block. |
| 'change' | Changes the permission of a block (using `mprotect()`) for example to make an executable, non-writable block. |
| 'rowhammer' | Induces rowhammer errors using a given aggressor address. |
| 'profile' | Profiles an aggressor row and a victim cell in memory blocks. |
| 'deallocate' | Deallocates a block (using `free()`). |
| 'translate' | Translates a virtual to a physical address only for WRA. |

TABLE II. DEFINITIONS OF ROWHAMMER ATTACK EVALUATION METRICS.

| Metric | Definition |
|---|---|
| $P_{find}$ | Probability of finding an aggressor and a victim row from a target node that has at least one victim cell. |
| $P_{store}$ | Probability of forcing to store a target state using a victim word. |
| $P_{aggressor}$ | Probability of obtaining the permissions to access a word in an *aggressor row* and a word in a *competing row* in the same bank. |
| $P_{toggle}$ | Probability of toggling bit(s) of at least one target state as expected by inducing rowhammer error(s). |
| $P_{landing}$ | Probability of successfully directing the control or data flow to one of the prepared landing locations. 1 if landing is not required. |
| $P_{postproc}$ | Probability of successfully conducting post processing operations. |

defined as $P_{find} \times P_{store} \times P_{aggressor} \times P_{toggle} \times P_{landing} \times P_{postproc}$ (see Table II). The S$^2$VF model indicates if any parameter is significantly low, that state is not a good attack target. *(a)* $P_{store}$ is low if only a small number of target instances exists or can be created or a specific timing is required to corrupt a target state (e.g., dynamically loadable kernel module or kernel stack of a running process). *(b)* $P_{aggressor}$ is for example low for kernel code. In a 32bit x86 Linux, the kernel code and other static segments are stored from the 2nd MB of the physical memory. Because the first 16MB physical memory region is reserved for DMA (e.g., of ISA cards) and is not directly accessible by any user process, $P_{aggressor}$ for kernel code is near zero. *(c)* $P_{toggle}$ is generally lower with a rowhammer fault model than single- or double-bit fault models if the target is code. Let us consider the most common type of x86-64 instruction (`mov`) corrupted as a result of a 1 or 3bit error. For example, `mov [rsp+0x20],rax` (machine code 48894424 20$_{(16)}$) becomes `mov [rax+rbp*1+0x20],rax` (488944$\underline{2}$820$_{(16)}$) with a specific 1bit error and three instructions `pop rax; mov [rsp+riz*4],eax; .byte 0x20` ($\underline{5}$889$\underline{0}$4$\underline{a}$420$_{(16)}$) with a specific 3bit error. Clearly, that 3bit error is more likely to cause a crash than the 1bit error partially because 3bit error has a higher chance to corrupt the opcode than 1bit error. *(d)* $P_{landing}$ indicates that the chance of return-to-libc style exploit is low. The chance of forcing target program to jump to a location where a specific library function (e.g., `exec()`) exists is extremely small if 3bit random bit flips are induced. Also usually specific arguments must be delivered using the stack. That is highly unlikely unless an attacker has an ability to corrupt data in an arbitrary location already. *(e)* $P_{postproc}$ for example indicates that the states of privileged user processes are not a good rowhammer attack target. While the running daemon process count is not controllable by users, an exception is `setuid` programs where a non-privileged user can start many instances of them. However, setting many landing pages is difficult in that case because the virtual address spaces of privileged processes are protected. Based on such analyses, Alice can exclude bad targets.

**Step 3.** *Pressure a target system by allocating nearly all of its available physical memory space.*

Giving a high pressure on the system memory allocators (see Fig. 1(3)) makes them have not many choices when a new memory allocation request is received. Alice needs to check the available physical memory size (e.g., `/proc/meminfo`) and sends the 'allocate' and 'lock' commands to the attack program until the free memory size becomes close to the threshold (e.g., a percentage at `/proc/sys/vm/swappiness`) that can start page swap out operations. Because the pages allocated by the attack program are locked, pages owned by some other processes are evicted if the attack program tries to allocate more than that threshold. Fig. 1(3) depicts the physical memory state after Step 3, and indicates that the allocated blocks occupy a large portion of the zone `HIGHMEM` (above 896MB) and `NORMAL` (from 16 to 896MB) which can be shared by the other processes and kernel dynamic memory. Step 3 is optional if *k* used in Step 1 was big enough. A large *k*, however, increases the time for Step 1 especially when *v* (i.e., the number of searchable aggressor rows) is 0 or extremely small.

**Step 4.** *Record the physical address of at least one victim cell in case of WRA, and deallocate its memory block.*

Alice now sends the 'deallocate' command to free the victim block(s) where each victim block contains at least one identified victim cell. In case of WRA, Alice calculates and records the physical address of each victim cell by using the 'translate' command. That is to check whether the freed victim cells are reused to keep at least one target state if the physical address of each target state is also known (see Step 5). The 'deallocate' command returns an entire memory block (see Fig. 1(4)) instead of a few words that contain the identified victim cell(s) (e.g., by using `mremap()` or `realloc()`). That allows the buddy allocator (i.e., the lowest memory allocator dealing with the physical memory space in the kernel) to free that returned block and allocate that block for a new request sent by Step 5.

**Step 5.** *Force a target system to replicate the target states and store at least one target state using a returned victim cell.*

Alice executes some programs that can force the kernel to create many instances of the selected target states. The actual operations to create many target states depend on the type of the states. It can be a specific system call, which can exercise a specific control flow path in the kernel. For example, if a target state is part of the context of a process, one can create many processes using `fork()` system call.

Kernel data structures, which can be created multiple times upon user requests, are typically managed by the slab allocator. The slab allocator for data structure *D* gets a slab (i.e., a set of physically contiguous memory pages) and splits that slab into sub spaces to individually store each instance of *D*. As more instances of *D* are created, the allocator fills in all the spaces of that first slab and thus allocates another slab using the buddy allocator and repeats the same process. Let us assume that the target state is part of the context of a user process. When Step 5 begins, it is likely that one of the slabs for the process context data structure is partly used. As many user processes are forked

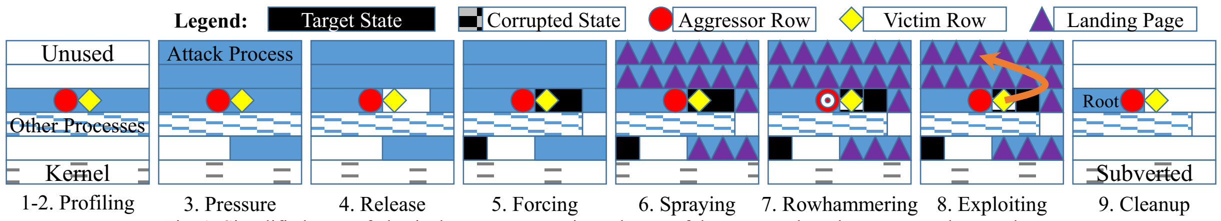

Fig. 1. Simplified state of physical memory space in each step of the presented rowhammer attack procedure.

as part of Step 5, that existing, partially filled slab will soon be full and a new slab will thus be allocated. Because the attack program now occupies nearly all the physical memory space (Step 3) except for the one large memory block, which was just returned to the kernel (Step 4), and some free spaces, the new slab is likely to be allocated in the returned memory block that contains at least one found victim cell. Even when the returned victim block is not immediately reused, the chance becomes higher as more many target states are replicated (see Fig. 1(5)).

**Step 6.** *Spray landing pages to the allocated blocks.*

If an attack requires a landing page (e.g., due to an address or control flow change), Alice needs one or multiple landing pages. She can call the 'store' command using the paths of the produced landing page files so that the attack program can spray them all over the allocated memory blocks (see Fig. 1(6)). There are two types of landing page payloads. The left payload in the below code block is for control flow attacks. It has a large `NOP` (no operation instruction whose machine code is $90_{(16)}$ in x86) sled followed by the malicious code. The right side code is a data payload for example when the corrupted target state is a pointer variable of *X*. It has variables for multiple instances of *X* where some pointer variables of *X* can be used as a knob to introduce memory corruption errors in any chosen locations (see §VII.B).

```
NOP                   | Var A for 1st instance of X
…                     | Var B for 1st instance of X
NOP                   | …
Malicious Code        | Var A for N-th instance of X
```

**Step 7.** *Induce rowhammer errors by rowhammering.*

Alice can now call 'rowhammer' using the identified aggressor row in order to corrupt a target state partly stored in a victim cell. The virtual address of any kernel-side target state is typically unknown to non-privileged process and user unless the target state is returned or directly exposed to a user process. For example, the Linux kernel maintains and exposes such mapping information (e.g., slab caches for each data) to only privileged users via its proc file (e.g., at `/proc/slab info`). If a target state is returned to the attack program, the program can compute its physical address and check whether that target state is allocated to a memory space which is known to have a victim cell. That checking is only for WRA. BRA does not require such checking as it already sprayed the target states.

**Step 8.** *Exploit the corrupted state(s), and if possible and necessary, validate the memory corruption result before that.*

Alice can then perform certain operations which can trigger and exploit the corrupted target state. The triggering operations heavily depend on the target state type. If the target corruption happened as designed, certain symptoms can be observed (e.g., the permission of a certain resource is changed). Otherwise, the requested operations can happen normally. In that case, Alice can go back to one of the previous steps and retry until a target state is compromised successfully. A retry operation using the same parameters can be successful for example because of the use of a user space ASLR (address space layout randomization) and other non-determinism sources in OS kernel and hardware.

**Step 9.** *Conceal and cleanup.*

From this point, it is identical or similar to general security attacks exploiting software vulnerabilities. For example, Alice can access some confidential data, harm the dependability of a subverted system un-recoverably, or install a hidden rootkit to control a target system, monitor the behaviors of other users (e.g., administrators), or gain information about credentials of other users. Alice can then conceal herself from the operators and the other users, and perform cleanup operations to make any future forensic analyses difficult.

## B. *Rowhammer Error Injector (REI)*

***(i) Fault Model.*** Our REI designed for Step 1 and 7 (i.e., the 'profile' and 'rowhammer' operations) use the following fault models. *(a) Flipped-to-'1' Error. N* randomly chosen bits of a target word are toggled to '1' where each of the chosen bits originally stores any value (i.e., either '1' or '0'). If (*N*-1) chosen bits already store '1', the resulting one-bit error is treated as a corrected error by an ECC. *(b) Flipped-to-'0' Error.* Similarly, *N* randomly chosen bits of a word are toggled to '0'. *(c) Multi-Bit Flip Error. N* randomly chosen bits of a target word are toggled regardless of the original values.

***(ii) Tool.*** REI consists of a *user-level driver* and a *kernel-level injector* module. The *driver* is a library linked to and called by a custom attack program, which selects a target virtual address and calls a function of the driver using the selected address as an argument. The driver reads that address in order to validate that the caller has a permission to access the provided virtual address. If the read is successful, it sends: address, process ID, rowhammering mode, and extra optional arguments to the injector module via its proc file. The *injector* converts the virtual address to the physical address, and then deterministically selects three victim cells in a word by using the physical address as the seed of the used random number generator. That is, the generated random numbers are used to select an injection target word from one of the two immediate logical neighbor rows of the derived physical address, and three bits from the selected target word.

The supported rowhammering modes are: (1) *immediate*, to immediately toggle the bits, (2) *delayed*, to toggle the bits after a time interval using a timer, (3) *periodic*, to toggle the bits for *n* times each after a time interval *t*, and (4) *data-type-aware* [12], to inject errors to data of a specific type using the object tracker module that keeps the addresses of the tracked kernel objects. The actual error injection mechanism is slightly different from the general software-implemented fault injectors (SWIFIs) that only use virtual address. The REI injector uses a physical address for each target, and creates a mapping to use a virtual address when corrupting the target memory value.

***(iii) Targeted Fault Injection.*** The key part of RAI is based on the fault injection (FI) based experimental methodology. It can be seen that by using a SWIFI for security validation and evaluation, our RAI methodology dramatically expands the applications of FI-based experiments from the reliability and simple DoS attacks to the sophisticated security attacks. Specifically, not many previous FI works discovered high-level security attacks (e.g., privilege escalation) because the injected random errors led to mainly crash and hang failures and sometimes silent data corruption (SDC) failures (e.g., [12]). In general, FI target state space (e.g., of an OS kernel) is extensively wide. It requires numerous injections until a certain state, which can lead to a security hole, is corrupted but the injection does not cause any immediate crash failures. More importantly, a memory corruption event alone is usually insufficient for successfully performing a non-DoS-style, high-level security attack. A successful attack requires a corruption of a particular state at a particular time, followed by some specific operations (as exemplified in §VII) which were often ignored in the previous FI studies.

In order to overcome such limitations of the blackbox FI experiments, RAI uses the *targeted FI approach* for attack identification and validation. A domain expert manually classifies and analyzes various kinds of potential target states by using the target source code, some profiling tools, and the metrics presented in Table II (as explained in §IV.A Step 2). FI experiments are then selectively conducted in some specific states chosen by an expert. For each target state, the actual FI target instances are carefully chosen by considering the original values and the memory locations for example. That is to reduce the experiment time and validate the feasibility as a practical attack target. Similar to other FI works, the data presented in this paper is all based on conditional probability and captures when one of the used three types of faults occurs.

## C. Framework Implementation

Alice uses the RAI attack program in order to monitor a target system and adaptively choose the operations to perform and their parameters. The following is the four key libraries of the program where some have more than one mechanisms to improve the adaptivity of generated attacks.

*(i)* The contiguous memory allocator (`cmalloc`) for the 'allocate' command first tries to allocate a physically contiguous memory block. It uses `mmap()` to allocate a block using one or multiple huge pages (e.g., 2MB or 1GB per page). If huge pages are not supported by the target system kernel, it creates a file of a requested size and tries to map a memory block to that file using `mmap()`. If a file I/O operation fails or `mmap()` fails, it uses `malloc()` to allocate a virtually contiguous memory block.

*(ii)* The library for 'rowhammer' supports two modes: to perform a rowhammer operation, and to emulate errors in the predefined locations by using REI. To actually perform a rowhammering, this library requires four types of arguments: the address of a word in a target aggressor row (namely, *aggressor address*), the addresses and sizes of allocated blocks, the number of memory accesses to perform on a given aggressor address, and the number of DRAM banks. If the bank size is 1GB, the bank count is *<physical memory size> / 1GB*. It has two sub-execution steps: *(a) Bypass DRAM row buffer.* It randomly selects an address (namely, *competing address*) from the given memory blocks. A competing address is supposed to be in the same memory bank as the aggressor address but in a different row so that when this library alternates accesses to the aggressor and competing addresses, those can bypass the row buffer of a target bank. This library repeats that random selection of a competing address *3 × <bank count>* times and almost guarantees a competing row is chosen correctly at least once (see §VI(i)). *(b) Bypass on-chip caches.* To repeatedly access an aggressor or competing address, this library also needs to bypass all on-chip caches in the processors. One way is using a special instruction (e.g., `clflush` or `clflushopt` in the x86 ISA), which can flush or evict all cache lines for a given memory address. Our current implementation uses an inline assembly routine, `asm("clflush (%0)" :: "r" addr : "memory")` where `addr` is a memory address. Alternatively, one may generate some memory access patterns designed to inflict many cache conflict misses and evict all the cache lines for a given memory address. The efficiencies of various access patterns were studied extensively in the literature (e.g., [11]).

*(iii)* The library for 'profile' initializes the given memory blocks where none should be larger than the DRAM row size. This library uses two bits to configure the initialization mode: one bit for a target block, and the other for all the other blocks. An example is storing '0' to every bit in a target block (i.e., a potential aggressor row to test) and '1's to every bit in all the other blocks. That is to search for victim cells that lose charges and flip the bits to '0'. After the initialization is done, this library calls the 'rowhammer' command with each target block. After each call, this library examines all the other blocks. If at least one victim cell is found (e.g., a cell that stores '0' instead of '1'), this library returns the address of the found aggressor row and the addresses of any found victim cells. Otherwise, this library continues and tests another target block after reinitializing the previous and new target blocks. If no victim cell is found after profiling all the allocated blocks, it returns an error code. The time complexity of this library is O($k^2$) where $k$ is the number of given blocks as this library checks all given blocks other than a target block when searching for victim cells. That is because rows more than 8 rows away from an aggressor row in logical address space can experience some rowhammer errors as explained in §II(iii).

*(iv)* The virtual to physical address translator (V2P) for 'translate' is to obtain the physical frame number (PFN) of a given virtual page and thus is only for WRA. It internally uses the `pagemap` proc files (at `/proc/<pid>/pagemap` where `<pid>` is process ID) in Linux. A `pagemap` proc file exposes the entire PTEs of a process, and each PTE has the actual PFN of a corresponding virtual page. Since Linux kernel v4.0, that proc

file is only readable by super users. While inside the kernel the page tables are directly used to translate between virtual and physical addresses, there is no other straightforward way for a user process to convert between virtual and physical addresses. Yet, some sophisticated attacks (e.g., using side-channels or uninitialized memory [13]) can still use the leaked information from kernel [14] and guess some physical addresses. If even such leak-based approaches are impractical, BRA is an option.

## V. Experimental Setup

***(i) Threat Model.*** Our attack model is as follows. A SaaS (Software as a Service) company uses a PaaS (Platform as a Service)-based cloud service to provide many customized guest VMs to their users. For example, a web hosting company can build web servers and provide non-root shell command line access to the users. If one of the users is an attacker, although he does not have the root permission, he wants to subvert a guest OS and steal the information stored in the guest OS space (e.g., the SaaS credential, infrastructure, or data of other users in the same VM) and possibly expand the attack scope to other VMs managed by the same SaaS provider. ***(ii) Target System.*** Our target system is a QEMU-KVM-style, custom built virtual machine monitor (VMM). While the host OS is a 64-bit Linux kernel, the guest OS which is run inside a VM is a 32-bit Linux kernel v3.4.67. ***(iii) Setup.*** Our experiments used BRA. All our experiments assumed one victim row. If more than one victim rows were exploited, $P_{store}$ (in Table II) would be higher than the derived values. Our experiments also assumed only one word that has three victim cells because less than 3bit errors are at least detected by an SEC-DED ECC. If there were many victim cells in a row, the relevant probabilities ($P_{find}$, $P_{store}$, and $P_{toggle}$) would be generally higher than the derived values, while $P_{landing}$ can be lower than the same (e.g., if a non-target instruction or data, used before exercising a corrupted target state, is also corrupted and leads a failure).

## VI. Optimization

***(i) Validation of 'rowhammer'.*** The actual alignments of banks and rows in a user virtual address space are unknown. To validate the 'rowhammer' library, the average time taken to access aggressor and competing words was measured for 96 randomly chosen competing words. 96 is 3 times the number of banks in the used 32GB memory. As shown as *two clusters* in Fig. 2, the average and standard deviations of the memory access time were 48.7 and 4.36 ms, respectively, if the two words are in the different banks, and 68.5 and 0.94 ms, respectively, if the words are in the same bank. That confirms that a competing row was chosen correctly at least once and the rowhammer operation performed correctly at least once. ~3.4% of the accesses belong to the right cluster, and that is consistent with the 32 banks (i.e., 1/32 = 3.125%). In that experiment, after accessing an address, all corresponding cache lines were flushed. In total, from 100K to 1M memory accesses (see the legend of Fig. 2) happened to each address. When each address was accessed for 500K times and the bank count was 32, the 'rowhammer' execution time was 4.6 seconds on the machine.

***(ii) Optimizing 'profile'.*** It took ~29.2 seconds to profile all the allocated 1GB space excluding the time for 'rowhammer' if $k$ (i.e., the number of blocks) is 256 and the block size is 4MB,

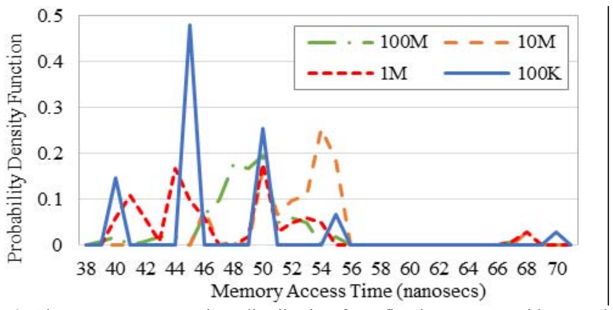

Fig. 2. The memory access time distribution for a fixed aggressor address and randomly chosen competing addresses (legend: total memory access count).

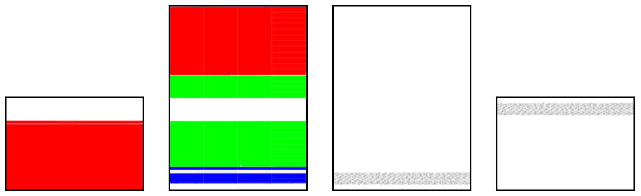

Fig. 3. Virtual address of an attack process (left), physical address after Step 3 (mid-left), physical addresses of kernel stacks (mid-right), and virtual addresses of kernel stacks (right); the bottom row address is 0-4MB.

on the same x86 machine. That is in fact close to the worst-case execution time. In practice, the expected execution time is 1 / ($v$+1) of that measured time if $v$ is the number of searchable aggressor rows in the group of given blocks and the aggressor rows are uniformly distributed in the given blocks. That time reduction is thanks to the used *early termination mechanism* (i.e., stopping after the first aggressor row is found).

***(iii) Using Multiple Processes for Pressure.*** If the physical memory size is larger than 4GB and a 32bit x86 Linux with the PAE mode is used, it is not possible to allocate more than 3GB by using one instance of the attack program because its virtual address space size is 4GB and the 4th GB is exclusively used by the kernel. In that case, *more than one instances of the attack program were created* to allocate nearly all the available physical memory space. Fig. 3(left) shows the virtual address space of an attack process after Step 3, showing that nearly all user address space is allocated. That process allocated ~2.9GB memory and stored the payloads. The physical address space occupied by the three attack processes is shown in Fig. 3(mid left) where green, red, and blue pixels are for process 1, 2, and 3, respectively, and white is for all the other areas. Process 3 evicted many physical pages and got pages in the 1st GB. The total used physical memory size in that experiment was 8GB.

***(iv) Shared Pages for Landing.*** A corrupted control flow (e.g., a random jump) happens in a virtual address space. Thus, the attack program can *reuse a set of physical pages and map many sets of virtual pages to them*. That technique is used if and only if the payloads are location independent (e.g., position independent code). That *shared page mechanism is also useful if multiple attack processes are needed* because processes can then share the physical pages and store the landing payloads.

***(v) Target and Landing Page Address Aware Victim Selection.*** In practice, victim cells are more likely to suffer again from other rowhammer errors than the non-victim cells in the same word. If many victim words are identified, one can evaluate $P_{toggle}$ of each word and use that in the selection of a victim word. For example, *a victim word which, with a high chance, can toggle virtual addresses of target states (e.g., Fig. 3(right)) to one of landing pages (Fig. 3(left)) is a good selection.* Specifically, in that example, if the most significant bit of a word can be flipped to '0', that gives a high $P_{toggle}$. In general, *a multi-bit error has a higher chance to toggle upper bits of a target word* than a single bit error if the probability of corrupting each bit in the target word is uniform. That implies that if a found victim cell is at an upper bit of a word, that can be a good choice for many control flow attacks whose success probabilities heavily depend on whether a corrupted address points to a prepared user space landing page which is relatively far from the kernel.

## VII. Result

### A. Privilege Escalation Attacks

**Target 1.** *Kernel stacks of non-running processes.*

To conduct a code injection attack, an attacker needs to change either: (1) data that decides the control flow or (2) some part of code memory of a target program. While manipulating some control data (e.g., in stack) is practical, inducing an error in the kernel code for a code or shell injection attack is not. The two key registers that determine the execution state of an x86 program are instruction pointer (`EIP` or program counter) and stack pointer (`ESP`). If an attacker can change one of those two register values, the attacker can change the control flow of a target. If those two registers are stored in a processor core, they are not corruptible by a DRAM rowhammer operation. Thus, we focus on cases when they are stored in memory. In x86, an old base pointer (old `EBP` or frame pointer) can store the stack frame pointer of the immediate caller function, and an old `EIP` keeps the return address (i.e., the next instruction of a caller) of the current function. The old `EBP` and old `EIP` values are stored in every process stack. In Linux kernel, two system stacks are kept for a user process: kernel- and user-space stacks. The size of a kernel stack is typically 8KB (i.e., 2 × page size), and the size of a user stack is not fixed as that grows down from the top of a user segment as that is used.

***(i) Forcing to Replicate Targets.*** In the experiment, we created many kernel and user stacks by creating 10,000 user processes on a virtual machine with 8GB guest memory. The kernel stacks of the created 10K processes used 80MB. If 90% of the guest physical memory is allocated by Step 3, ~800MB is free and thus $P_{store}$ is ~80MB/800MB ≈ 0.1. By repeatedly creating 10K processes and triggering them, we often drove the kernel to allocate a kernel stack using the returned victim cell (see Fig. 3 for the created stack locations). Each call stack consisted of: the function arguments, old `EIP`, old `EBP`, local variables stored in the stack before a next function was called, and the thread context. Not entire kernel stack space is used. For example, if the stack depth is 20 functions, ~9.2% of the stack space is used if the average frame size is 36B (e.g., 8B for old `EIP/EBP`, 12B for 3 arguments, and 16B for 4 local variables) and the size of `thread_info` is 32B. If unused part of a kernel stack is corrupted, it has no impact. If on average 3 arguments and 4 local variables are kept per function frame, slightly less than 11.1% of the stack stores the targeted old `EIP`.

TABLE III. Faults in Kernel Stack vs. Failure Probabilities

| Target State | Exploited | Kernel Panic | Kernel Hang | Process Crash | Process SDC | Masked |
|---|---|---|---|---|---|---|
| Old `EIP` | 0\|***50%*** | 67\|***17%*** | 0\|***0%*** | 0\|***0%*** | 17\|***17%*** | 17\|***17%*** |
| Old `EBP` | 0% | 58% | 8% | 8% | 8% | 17% |
| Arguments | 0% | 5% | 2% | 12% | 7% | 74% |
| Local Vars | 0% | 11% | 5% | 5% | 7% | 73% |

* Percentage without landing pages | ***Percentage with valid landing pages***

***(ii) Rowhammering.*** We used REI and injected errors into the kernel stacks (see Table III). In total, 122 3bit flipped to-'1' or to-'0' errors were injected. The guest machine was rebooted after each injection. If an old `EIP` in the kernel stack is corrupted, that process can jump to an arbitrary instruction when that is returning to the user space. If the corrupted `EIP` value points to an instruction in the kernel code memory, the chance of causing a kernel crash failure is relatively high (66.7% with no landing page) because it can jump to the middle of an instruction; some data which the callee requires may not be set; or the kernel stack is not properly constructed for the callee (e.g., different argument count). *When valid landing pages were set, the corrupted `EIP` often pointed to a valid landing page, and led to a successful attack with ~50% probability (see exploited ratio in Table III).* We used two types of errors in that experiment. Notably large variations in the exploited ratio were seen between the two faults (i.e., type and bitmask). Application SDC failures occurred if a corrupted `EIP` points to the original function but a different instruction in the same or even a different basic block. If a fault is injected into an old `EBP`, that can eventually change `EBP` and then `ESP`. A random value, stored in an arbitrary stack pointed by a corrupted `ESP`, is read and used as an old `EIP` value when that corrupted process is scheduled again and tries to return to the user space, showing the high aggregated kernel failure ratio (e.g., 58.3%+8.3%). On the other hand, the faults in the arguments and local variables were more likely masked (73.8% and 72.7%) than those in the special-purpose registers where 'masked' means 'not activated' or 'not manifested'. The experiment used two bitmasks: error bits uniformly distributed in a 32bit word, and error bits only in the low 12 bits. If fllipped-to-'1' errors were used, the second bitmask led to kernel failures 11% more likely than the first one because addresses, which are more error sensitive than the others, had many upper bits preset.

***(iii) Landing.*** When a corrupted `EIP` or `EBP` forces a process to jump from the kernel code to a prepared landing page in a user space, that can lead to a successful code injection attack. In practice, $P_{landing}$ depends heavily on the actual value of a corrupted `EIP`. Fig. 4 visualizes the distances between the original value ($c16dbfa0_{(16)}$ for the address of `schedule()` in a used 32bit x86 machine) and the values corrupted by 1, 2, or 3bit flip errors. There are 32, 496, and 4,960 cases for the 1, 2, and 3bit errors, respectively. If more bits are flipped, the distance becomes farther (see how the slope of the line graph for the 3bit errors changes in Fig. 4). The bar graph shows the probability density function of 3bit errors. That shows that with 36.8% and 21.8% probability, 3bit errors can make a jump farther than 128MB and 512MB, respectively, from the original destination. In 32bit x86 architectures, typically the kernel space is in the 4th gigabyte of every virtual address space and

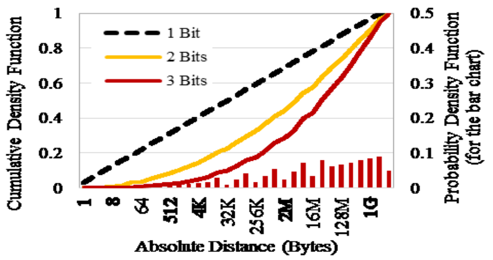

Fig. 4. Density functions vs. Absolute distance between the corrupted and original values when 1, 2, and 3bit errors are injected.

the kernel code is stored in the low part of that 4th gigabyte. Thus, if a jump is made from the kernel to an instruction located more than 512MB away, that jump is likely to a valid landing page of the current user process (e.g., in the first 3GB).

When the control flow of the guest kernel was changed to a user space payload, that payload code was executed using the privileges of the compromised kernel. The used payload consisted of a sequence of `NOP` instructions followed by the absolute jump instructions (`MOV EAX`, <address>; `JMP EAX`). Although those two instructions use a general-purpose register, they do not require a valid system stack (e.g., as compared with when `PUSH` <address>; `RET` are used). The destination of that absolute jump contained: code to escalate the current process to root, and code that leads to a safe return to the user space. *(a) Privilege escalation.* The current attack process must survive after the corruption as the privilege of another process cannot be escalated. Since Linux kernel v2.6.29, no ID value stored in the control block of a process is directly mutable by any other process. Privilege escalation in that experiment was done by changing all the ID fields (e.g., `uid/euid/suid/fsuid`) to 0 (i.e., root) using some existing kernel functions (e.g., `prepare_creds()` and `commit_ creds()`). When the symbol information of those used kernel functions are unknown (e.g., if not possible to read the kernel symbol table at $c1000000_{(16)}$ in a 32bit x86 or at $ffffffff81000000_{(16)}$ in a 64bit x86 machine, or to use `/proc/kallsyms` or a kernel image), the injected code to escalate the privilege has to scan the kernel code memory in order to find certain instruction sequences, and then identify the locations of the used kernel function. While some other techniques can be used to find the function locations, in general that makes $P_{postproc}$ lower in BRA than WRA.

*(2) To return to the user space.* The prepared payload code checked the used kernel stack in order to find an old `EBP` that stores the address of an instruction of `syscall_call()` (e.g., using the symbol information of `syscall_call()`, or the knowledge about the number of function frames pushed to the stack before the injected control flow error). Then the payload code updated the `ESP` register accordingly and executed an `ret` instruction. The target kernel then executed the rest of system call routine and safely returned to the user space. In that case, `EAX` was set to 0 as that contains the error code of the executed system call and is used by the system call interface routine.

The presented attacks do not rely on any direct software vulnerabilities (e.g., buffer overflow or memory corruption) unlike to the conventional *ret2usr*. For non-virtualized native machines, there are software and hardware techniques (i.e., not all yet in the mainline kernel) which can address the *ret2usr* attacks (e.g., PaX and SMAP). While there is a technique [15] that can subvert the existing mitigation techniques if memory mapping (e.g., `pagemap`) is known, our experiments confirmed that guest OSes generally did not yet have all such protections because many existing protection techniques rely on hardware-enforced protection mechanisms and providing such hardware mechanisms to guest OSes increases the design complexities.

## B. *Shell Injection Attacks*

Once `uid` of that attack process became 0 as a result of the fore-described privilege escalation attack, the attack process can fork a root shell (e.g., `execv()`) and accept any other commands. Using the RAI attack program, we then conducted various privileged operations as the root user.

**Target 2.** *A block of homogeneous memory pointers where each pointer points to another object with at least one pointer.*

In theory, if a user can force a target OS kernel to create many instances of a specific data structure, which has many nested pointers, that can lead to privilege escalation, and code and shell injection attacks if a non-leaf-level pointer can be corrupted. Target 2 can be seen as a generalization of the PTE target[2] identified by Seaborn and Dullien [6] but is weaker than the PTE target in some cases for example if the corrupted pointer always has to point to a user-space payload (instead of existing kernel-space data). Thus, in this case, it is important to identify a target state which exists in the kernel space and that is aligned with the implications of [6].

While we were yet unsuccessful in validating an example Target 2 instance which can lead to some high-level attacks, we found some potential targets. If a program calls `io_setup` system call, that internally calls `ioctx_alloc()` kernel function and allocates an instance of the `kioctx` data structure. `kioctx` has `aio_ring_info` which has 8 pointers to 8 different `page` instances. In total, that can create at most 64K `aio_ring_info` instances where all the pointers to `page` occupy 2MB memory space. If the free guest physical memory size is 800MB, the chance of using a victim cell to allocate a `page` pointer becomes as high as 0.25%. In practice, $P_{store}$ can be higher than that depending on the maximum number of user creatable target instances. If an error corrupts a pointer to a `page`, that corrupted pointer can point to another `page` in the page cache. From that point, an attacker may use other system calls which can call some relevant kernel functions in theory.

## C. *Data Corruption to Harm System Integrity (e.g., Disk)*

**Target 3.** *File metadata to induce disk corruption failures.*

***(i) Forcing.*** The attack program accessed many files by reading from and writing some random contents to them. That let the guest kernel to create many instances of `buffer_head`. `buffer_head` is used for extracting block mappings and other

[2] That technique targets PTEs because a user process can create many PTE pages and each PTE page has many nested, homogeneous pointers. It then tries to corrupt the physical address stored in a PTE by inducing rowhammer errors. If many PTE pages are created in advance, a corrupted PTE can point to one of the created PTE pages. Using the corresponding virtual address of the pointed PTE page, one can access other pages and escalate the memory access privilege.

relevant operations as the I/O unit of file systems (e.g., block which is 512 bytes) can be smaller than a page. ***(ii) Rowhammering.*** `buffer_head` has many attributes that include a pointer to a mapped page, a pointer to a data array, and a mapping size variable. If one of those two pointers or the size variable is corrupted, the contents of the corresponding file were highly likely to be corrupted. Depending on the exact use case of a corrupted `buffer_head`, it can also corrupt the file system metadata. For example, flipping the bits of a `b_page` pointer showed a recoverable file system corruption after when the attack program triggered file system sync operation to flush the corrupted in-memory metadata to a storage system and rebooted the target VM. In general, cloud service providers always try to guarantee a certain level of disk data reliability and consistency. If the metadata of a file system is corrupted, it not only requires a scanning and fixing operation which usually takes a relatively long time but also can lose some contents if full recovery is not feasible. Thus, such disk corruption attacks can be a serious threat to many storage cloud operators.

### *D. Advanced Denial-of-Service (or Crash) Attacks*

**Target 4.** *Event and external I/O handlers for timed and externally triggered failures, respectively.*

Two other experiments validated that RAI can be used to conduct advanced DoS attacks. This subsection does not cover immediate crash or hang failures as such simple DoS attacks can be conducted by corrupting multiple randomly chosen kernel states. While kernel hang failures are much more rare than crashes, their impacts are often similar to the crashes from the system reliability point of view if the target system has a high coverage hang detection technique in place. It is also well understood that hang failures are likely to occur if a lock, a loop condition, other equivalent control data is corrupted [12].

***(i) Timed.*** If an attack program corrupts a state which is used only by a periodic event (e.g., a system call that is rarely used but is used by a daily cron job), that can lead to a timed DoS attack. When the attack program corrupted a pointer variable state used by `do_swap_page()`, that led to a timed DoS attack at the time when that function was executed to swap out some pages in order to handle the memory overflow situation emulated by allocating many pages (e.g., peak traffic time in practice). Thus, a state only used under a certain condition can cause a timed DoS attack in general. ***(ii) Externally Controllable.*** If some states are only used if a specific external event is received, those can be the sources of some externally controllable DoS attacks. We corrupted a state that is used to handle a specific type of network packets (e.g., ICMP ping packet). We later sent ping packets to a corrupted target node. That led to a crash failure. In practice, more complex protocols and their sub states can be targeted for the same purpose. Such externally triggerable DoS attacks can make forensic analyses harder than the other attacks because the attacker can leave the target multiple hours or days before the target crashed.

## VIII. Summary and Implication for Mitigation

Based on our RAI methodology, we identified four, new rowhammer attack targets, validated three out of the four, and evaluated one target using a real system by modeling UIREs and the associated attack procedures. We derive the following requirements for mitigation techniques. *(a) High Precision and Possibly Good Recall.* Cloud operators can hardly stop a user even if a user shows abnormal behaviors or symptoms of attacks. That is because user privacy must be respected and normal users can show similar profiles (e.g., in cache miss ratio and maximum access count per DRAM row). If a perfect preventive technique is infeasible, a high precision detection technique would be the second choice because as far as the precision is high, operators can take some actions (e.g., migrating to a node which has better isolation and monitoring). Conversely, a technique with a high recall is unlikely taken if its precision is not high enough because such can significantly increase manual works of operators. (*b*) *High Deployability.* Easy deployability to a large number of nodes is important. Software techniques that do not require rebooting are easier to deploy than techniques that require BIOS update or hardware upgrades. *(c) Backward Compatibility.* It is also important that a technique can work with all the existing, legacy hardware and software platforms because many of them are typically kept for multiple years in production and thus need maintenance.

## 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", *ISCA*, pp. 361-372, 2014.

[2] I. Bhati, M.-T. Chang, Z. Chishti, S.-L. Lu, and B. Jacob, "DRAM Refresh Mechanisms, Penalties, and Trade-Offs," *IEEE Transactions on Computers*, 65(1):108-121, 2016.

[3] K. S. Bains, J. B. Halbert, C. P. Mozak, T. Z. Schoenborn, and Z. Greenfield, "Row Hammer Refresh Command," United States of America Patent Application, No. 13/539,415, January 2nd, 2014.

[4] D. H. Kim, P.J. Nair, and M. K. Qureshi, "Architectural Support for Mitigating Row Hammering in DRAM Memories," *CAL*, 14(1):9-12, 2015.

[5] S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *Proceedings of the ACM SIGMETRICS*, pp. 519-532, 2015.

[6] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," BlackHat (Presentation), 2015.

[7] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer, "An Experimental Study of Security Vulnerabilities Caused by Errors," *DSN*, pp. 421-432, 2001.

[8] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-Control-Data Attacks Are Realistic Threats," *USENIX Security Sym*, 2005.

[9] S. Chen, J. Xu, R. K. Iyer, and K. Whisnant, "Evaluating the Security Threat of Firewall Data Corruption Caused by Transient Errors," *DSN*, pp. 495-504, 2002.

[10] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," *IEEE Sym. Security and Privacy*, pp. 154-165, 2003.

[11] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," *CoRR*, vol. abs/1507.06955, 2015.

[12] K. S. Yim, *From Experiment to Design: Fault Characterization and Detection in Parallel Computer Systems Using Computational Accelerators*, UIUC Computer Science – Ph.D. Dissertation, 2013.

[13] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. Kaashoek, "Linux kernel vulnerabilities: State-of-the art defenses and open problems," *Asia-Pacific Workshop on Systems*, No. 5, 2011.

[14] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization," *USENIX Security Symposium*, pp. 475-490, 2012.

[15] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking Kernel Isolation," *USENIX Security Sym*, pp. 957-972, 2014.