Kernel Samepage Merging (KSM)

Funktionsweise und Schwachstellen

MARTIN HECKEL, Hochschule für Angewandte Wissenschaften Hof, Deutschland

Kernel Samepage Merging (KSM) ist ein Mechanismus im Linux Kernel, der das Deduplizieren von Daten im Arbeitsspeicher ermöglicht, indem Speicherseiten („Pages“) mit identischen Inhalten nur einmal im physischen Speicher abgelegt und an den jeweiligen Stellen referenziert werden. Dadurch ist es möglich, den Speicherverbrauch von Systemen signifikant zu reduzieren. KSM betrachtet dabei nur Pages, welche explizit als *merge candidate* deklariert wurden.

Die Vorteile von KSM kommen in virtualisierten Umgebungen (z.B. auf Basis von Kernel Virtual Machine (KVM)) besonders zum Tragen, wenn auf einem Host mehrere Instanzen des gleichen Gastsystems laufen. Da KVM alle innerhalb der Gastsysteme allokieren Pages automatisch als *merge candidate* für KSM deklariert, werden diese Speicherbereiche von KSM für die Deduplizierung berücksichtigt. Entsprechend werden alle Seiten, die innerhalb der Virtual Machines (VMs) allokiert werden, von KSM dedupliziert, sofern ihr Inhalt identisch ist. Wenn auf einem Host mehrere Instanzen des gleichen Gastsystems laufen, sind viele der Speicherbereiche redundant, können also dedupliziert werden. In einem Experiment von RedHat liefen 52 VMs mit Windows XP (jeweils mit 1 GiB virtuellen Hauptspeicher) auf einem Host mit insgesamt 16 GiB physischen Hauptspeicher.

Da die Vorteile von KSM besonders in virtualisierten Umgebungen zum Tragen kommen, stellen diese auch das typsiche Szenario für Angriffe auf KSM dar: Ein Angreifer hat Kontrolle über eine VM *Gast A* und versucht, die Vertraulichkeit, Integrität, oder Verfügbarkeit einer anderen VM *Gast B* zu beeinträchtigen.

In den letzten Jahren wurden einige Angriffe innerhalb dieses Szenario vorgestellt. Im Rahmen dieser Studienarbeit wird neben der Erklärung benötigter Grundlagen konkret auf drei dieser Angriffe eingegangen:

- Information leaks, durch die der Inhalt des Speichers von *Gast B* erraten werden kann
- Covert Channel, durch den Informationen von *Gast B* an *Gast A* übertragen werden können
- Angriff, durch den der Inhalt des Speichers von *Gast B* verändert werden kann

CCS Concepts: • **Security and privacy** → *Side-channel analysis and countermeasures.*

Additional Key Words and Phrases: rowhammer, KSM, FFS

1 EINFÜHRUNG

Das mooresche Gesetz besagt, dass sich die Anzahl der Komponenten auf einem integrierten Schaltkreis (Integrated Circuit (IC)) ungefähr alle zwei Jahre verdoppelt. Diese Verdopplung der Anzahl der Komponenten bedeutet unter Beibehaltung der physischen Größe automatisch auch eine Verdopplung der Integrationsdichte (Anzahl der Komponenten je Flächeneinheit). In den vergangenen Jahrzehnten wurden zunehmend komplexere ICs entwickelt, die sehr stark für den entsprechenden Einsatzzweck optimiert waren: Central Processing Units (CPUs) bekamen mehr Kerne, die Speicherkapazität und Geschwindigkeit von Dynamic Random-Access Memory (DRAM) wurde gesteigert, etc.

Author's address: Martin Heckel, martin.heckel@hof-universiy.de, Hochschule für Angewandte Wissenschaften Hof, Alfons-Goppel-Platz-1, Hof, Bayern, Deutschland, 95028.

Innerhalb dieser Entwicklungen wurde darauf geachtet, dass sich die Komponenten unter „normaler“ Verwendung entsprechend der Spezifikation verhalten. Allerdings wurde vernachlässigt, dass eine „untypische“ Verwendung der Komponenten zu nicht vorgesehenen Zuständen führen kann. In den vergangenen Jahren wurden viele dieser *Hardware Schwachstellen* gefunden und untersucht, beispielsweise Meltdown [13], Spectre [10] und Plundervolt [14]. Rowhammer [9], eine weitere Hardware Schwachstelle, die 2014 gefunden wurde. Rowhammer ermöglicht es einem Angreifer, Bits im Speicher zu *flippen*, d.h. eine 1 zu einer 0 zu ändern bzw. anders herum. Entsprechend ermöglicht es Rowhammer einem Angreifer, in Speicherbereiche, auf die er keinen Schreibzugriff hat, zu schreiben. 2020 wurde von Kwong et al. [12] gezeigt, dass Rowhammer auch zum Lesen von Speicher, auf den kein Lesezugriff besteht, verwendet werden kann.

Aufgrund der technischen Schwierigkeit, die Integrationsdichte von ICs stetig zu erhöhen, werden häufig neben der Verbesserung der Hardware auch Verbesserungen in Software implementiert, um die zur Verfügung stehenden Ressourcen effektiver nutzen zu können. KSM stellt eine dieser Lösungen dar und ermöglicht es, mehrfach vorkommende Speicherseiten („Pages“) zu deduplizieren, d.h. den Inhalt der Page nur einmal im Speicher zu halten und an den anderen Stellen, die eine Page mit identischen Inhalt verwenden, auf diese Stelle zu referenzieren. RedHat hat in einem Experiment demonstriert, dass 52 VMs mit Windows XP (jeweils mit 1 GiB virtuellen Hauptspeicher) auf einem Host mit insgesamt 16 GiB physischen Hauptspeicher laufen können [1].

Neben diesen Vorteilen bringt KSM allerdings auch einige Nachteile mit sich: 2016 wurden einige Angriffe gegen KSM vorgestellt die, u.A. das Raten von Inhalten im Speicher [2], die verdeckte Kommunikation über einen Covert Channel, sowie schreibenden Zugriff auf den Speicher [15] ermöglichen.

Im Rahmen dieser Arbeit sollen diese drei Angriffsmöglichkeiten gegen KSM sowie einige zum Verständnis dieser Angriffsvektoren benötigte Grundlagen erklärt werden.

2 GRUNDLAGEN

In diesem Abschnitt werden einige notwendige Grundlagen vermittelt. Konkret werden dabei der Aufbau und die logische Struktur von DRAM, die Funktionsweise von KSM, sowie Rowhammer erklärt.

2.1 Arbeitsspeicher, DRAM

In heutigen Rechnern werden typischerweise mehrere Arten von Speicher verwendet, beispielsweise Hauptspeicher (normalerweise in Form von DRAM) und Auslagerungsspeicher („Swap“). Unter bestimmten Umständen können Daten vom Hauptspeicher in den Swap verschoben werden. Außerdem laufen üblicherweise mehrere Prozesse gleichzeitig auf einem System. Aus diesem Grund ist es sinnvoll, die Adressen des tatsächlich verwendeten Speichers für

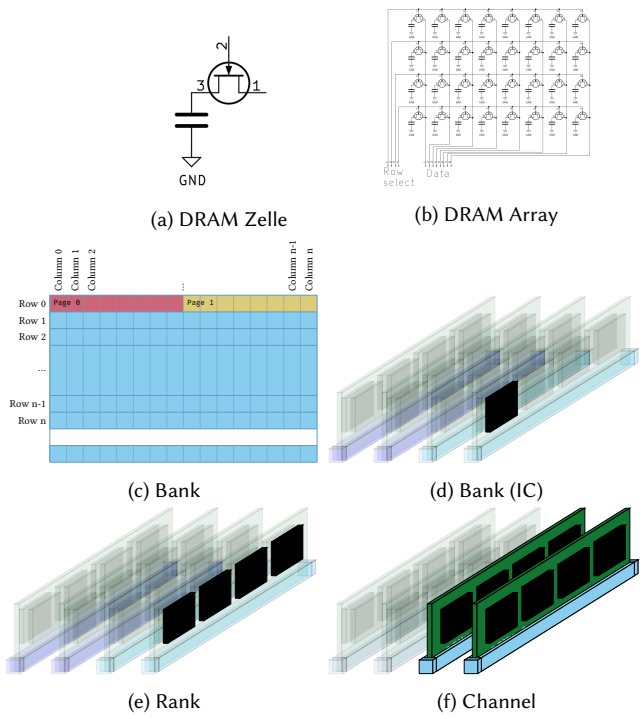
die Prozesse transparent zu verwalten, sodass diese nicht berücksichtigen müssen, an welchen Stellen im Speicher die von ihnen verwendeten Daten liegen.

Dieses Konzept der transparenten Speicherverwaltung wird in der Praxis durch ein Mapping zwischen *virtuellen* und *physischen* Adressen realisiert. Jeder Prozess hat einen eigenen virtuellen Adressraum, in den das Betriebssystem Bereiche (*Pages*) aus dem physischen Adressraum unter Verwendung von *Page Tables* mappt. Da der Prozess nur auf die virtuellen Adressen zugreift, ist das vom Betriebssystem durchgeführte Mapping für den Prozess transparent. Die gemappten Pages haben bei Linux normalerweise eine Größe von 4 KiB. Aufgrund der Tatsache, dass ganze Pages gemappt werden, kann eine Adresse als Summe der Adresse einer Page und einem Offset innerhalb diese Page betrachtet werden. Die Adresse der Page wird dabei durch die Page Tables gemappt (virtuelle Adresse der Page zu physischer Adresse der Page), der Offset innerhalb der Page wird übernommen. Die Page Frame Number (PFN) bezeichnet den Teil der physischen Adresse, der nur die Adresse der Page, allerdings nicht den Offset innerhalb dieser Page beinhaltet.

2.1.1 Physischer Aufbau von DRAM. Physisch besteht DRAM aus Zellen, die aus einem Kondensator und einem Transistor aufgebaut sind (s. Abbildung 1a). Diese Zellen sind wie in Abbildung 1b dargestellt in einer Matrix aus Zeilen und Spalten angeordnet („DRAM Array“), welche wiederum mit einer Verstärkerschaltung und dem Row Buffer eine Bank bildet (s. Abbildung 1c). Der Zugriff auf das DRAM Array erfolgt zeilenweise („Row“) und gepuffert über den Row Buffer, d.h. Daten werden aus dem Array in den Row Buffer geladen, bevor sie über den Speicherbus an die CPU gesendet werden können. Das gleiche gilt auch für schreibende Zugriffe: Die Daten werden in den Row Buffer geschrieben und danach in das Array kopiert. Das Kopieren von Daten aus dem Array in den Row Buffer „zerstört“ die Daten im Array, da die Kondensatoren der Zellen beim Lesen entladen werden. Entsprechend ist es notwendig, den aktuellen Inhalt des Row Buffers in die entsprechende Zeile des Arrays zu kopieren, bevor eine andere Zeile geladen werden kann.

Eine Bank kann physisch auf einem IC auf dem DRAM Modul liegen (s. Abbildung 1d), allerdings sind auch andere Kombinationen möglich (eine Bank auf mehre ICs verteilt oder mehrere Banks auf einem IC). Banks sind wie in Abbildung 1e gezeigt in einem oder mehreren Ranks organisiert, die sich wiederum auf einem Dual In-line Memory Module (DIMM) befinden. Abhängig vom System kann es einen oder mehrere Channels (s. Abbildung 1f) zwischen den DIMMs und der CPU geben. Ein Channel ist dabei ein eigenständiger Bus, wodurch die CPU gleichzeitig mit Modulen auf verschiedenen Channels kommunizieren kann.

Die Zustände der DRAM Zellen entsprechen Ladungszustand des Kondensators, der auf logische Zustände abgebildet wird (z.B. kann der physische Zustand „geladen“ auf den logischen Zustand „1“ und der physische Zustand „entladen“ auf den logischen Zustand „0“ abgebildet werden). Da sich die Kondensatoren durch Leckströme über die Zeit entladen, ist es notwendig, einen geladenen Kondensator wieder vollständig zu laden, bevor dieser nicht mehr von einem entladenen Kondensator unterschieden werden



(a) DRAM Zelle (b) DRAM Array

(c) Bank (d) Bank (IC)

(e) Rank (f) Channel

Fig. 1. Veranschaulichung der Architektur von DRAM (Abbildung aus den Slides zu [7])

kann. In der Praxis wird dieser Prozess durch das periodische Auffrischen („Refresh“) der Speicherzellen realisiert, was typischerweise in Zyklen von 64 ms geschieht.

2.1.2 Virtualisierte Umgebungen. In virtualisierten Umgebungen laufen virtuelle Maschinen nicht direkt auf der Hardware, sondern auf einem Hypervisor. Abhängig von der Virtualisierungslösung wird ein nativer oder ein gehosteter Hypervisor verwendet. Ein nativer Hypervisor läuft dabei direkt auf der Hardware, ein gehosteter Hypervisor läuft als Prozess innerhalb eines Betriebssystems, das auf der Hardware läuft. (s. Abbildung 2).

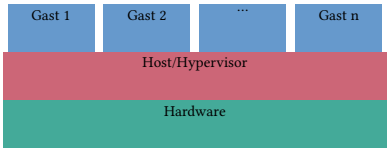


Fig. 2. Darstellung einer virtualisierten Umgebung

Da ein Gast in einer virtualisierten Umgebung keine Information darüber hat, dass die Umgebung virtualisiert ist, verhält sich dieser wie ein normales Betriebssystem: Prozesse innerhalb des Gastsystems verwenden virtuelle Adressen („Guest Virtual Address (GVA)“), die durch die Page Tables des Gasts auf physische Adressen („Guest Physical Address (GPA)“) gemappt werden. Da das Mapping auf Basis von Pages stattfindet, gibt es auch in virtuellen Maschinen ein Äquivalent zu PFNs: Guest Frame Numbers (GFNs).

Eine GFN des Gastsystems wird von QEMU/KVM anschließend auf eine Host Virtual Address (HVA), eine virtuelle Adresse auf dem Hostsystem gemappt. Auf dem Hostsystem gibt es wiederum Page Tables, die die HVAs auf Host Physical Addresss (HPAs) mappen. Wie vorher beschrieben, werden die Teile der HPAs, welche nur die Adressen der Pages, allerdings nicht die Offsets, enthalten, als PFNs bezeichnet.

Die HPAs, welche eine Kombination aus PFN und Offset innerhalb der gemappten Page darstellen werden anschließend vom Memory Controller auf eine physische Stelle im Speicher (bestehend aus Channel, DIMM, Rank, etc.) gemappt (s. Abbildung 3).

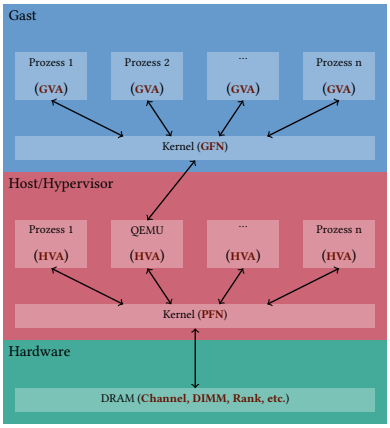


Fig. 3. Adressübersetzungen in virtualisierten Umgebungen (Abbildung aus den Slides zu [7])

2.2 Kernel Samepage Merging (KSM)

Einige Bereiche im Hauptspeicher werden innerhalb verhältnismäßig langer Zeiträume nur lesend verwendet. Teilweise haben mehrere dieser nur lesend verwendeten Speicherbereiche identische Inhalte. Durch das Anpassen der mit Hilfe der Page Tables realisierten Mappings ist es möglich, redundante Speicherbereiche zu deduplizieren, indem diese physisch nur einmal im Speicher gehalten werden und alle Pages mit identischen Inhalt auf diese physische Page referenzieren. KSM ist eine Komponente im Linux Kernel, die diesen Ansatz implementiert.

Durch die Deduplizierung ergibt sich allerdings das Problem, dass alle Schreibzugriffe auf eine dedupliziere virtuelle Page zu einer Änderung der physischen Page und somit zu einer Änderung aller virtuellen Pages führen würde. Um dieses Problem zu lösen wird eine Copy-on-Write (CoW) Policy verwendet, sodass Schreibzugriffe auf deduplizierte Pages zu zusätzlichen Schritten führen: Der Kernel kopiert die physische Page an eine andere Stelle im Speicher und passt die Referenzen an, sodass die virtuelle Page, auf die der Schreibzugriff erfolgen soll auf eine andere physische Page mit identischen Inhalt referenziert. Anschließend findet der Schreibzugriff auf die vorher kopierte Page statt, sodass die geteilte physische Page und damit auch alle anderen virtuellen Pages davon nicht beeinflusst werden.

Generell berücksichtigt KSM nur Pages, die explizit als *merge candidate* deklariert wurden, was sich in der Praxis durch die Verwendung von `MADVISE(2)` realisieren lässt. Da das Deduplizieren von Speicher in virtualisierten Umgebungen besondere Vorteile bringen kann (wenn das gleiche Betriebssystem mehrmals im Speicher liegt), markiert QEMU/KVM automatisch alle Speicherseiten, die alloziert werden, als *merge candidate*.

Intern verwendet KSM zwei rot-schwarz Bäume, um die Pages zu verwalten: Der *Stable tree* enthält Pages, die bereits dedupliziert wurden und der *Unstable tree* enthält Pages, die dedupliziert werden können, sofern eine zweite Page mit identischen Inhalt vorhanden ist. Unter Verwendung dieser Bäume scannt KSM die physischen Pages des Systems periodisch und dedupliziert diese, sofern das möglich ist.

Die Deduplizierung von Pages ist nur sinnvoll, wenn auf diese für einen längeren Zeitraum keine Schreibzugriffe stattfinden, da die Page aufgrund der CoW Policy bei Schreibzugriffen kopiert werden muss, was zu sehr langsamen Schreibzugriffen führt. Entsprechend berücksichtigt KSM nur Pages, auf die seit dem letzten Durchlauf von KSM kein Schreibzugriff stattgefunden hat. Intern wird diese Überprüfung durch die Verwendung des *soft dirty* Bit in den Page Table Entries (PTE) realisiert: KSM setzt das *soft dirty* Bit bei jedem Durchlauf auf 0. Schreibzugriffe auf eine Page führen dazu, dass das Bit auf 1 gesetzt wird. Wenn das *soft dirty* Bit eines PTE also als 0 gelesen wird, fand seit dem letzten Durchlauf kein Schreibzugriff auf die entsprechende Page statt.

Wenn KSM beim Scannen eine Page findet, die als *merge candidate* deklariert wurde und deren *soft dirty* Bit im zugehörigen PTE nicht gesetzt ist, versucht KSM, diese Page zu deduplizieren. Dafür wird im ersten Schritt geprüft, ob eine Page mit identischen Inhalt im Stable tree liegt. Wenn das der Fall ist, wird die neue Page zu den bereits im Stable tree erfassten deduplizieren Pages hinzugefügt. Ist das nicht der Fall, wird geprüft, ob eine Page mit identischen Inhalt im Unstable tree liegt. Trifft das zu, werden die beiden Pages dedupliziert und anschließend aus dem Unstable tree entfernt und zum Stable tree hinzugefügt. Wenn im Unstable tree keine Page mit identischen Inhalt gefunden wurde, wird die Page zum Unstable tree hinzugefügt.

Am Ende jedes Scanzyklus wird der Unstable tree vollständig entfernt. Das ist der Fall, da sonst ein weiterer Schritt zum Bereinigen des Unstable tree notwendig wäre (z.B. wenn sich der Inhalt einer Page geändert hat). Für den Stable tree ist keine Bereinigung notwendig, da dieser nur deduplizierte Pages enthält, für die ohnehin die CoW Policy gilt, die Pages können sich also nicht von KSM unbemerkt verändern. Der Grund dafür, den Unstable tree zu entfernen und nicht zu bereinigen besteht darin, dass das Suchen sowie das Einfügen in einen rot-schwarz Baum im Normalfall in der Komplexitätsklasse $O(\log n)$ liegen. Entsprechend liegen das Verwenden des vollständigen Baums und der Aufbau eines neuen Baums in der gleichen Komplexitätsklasse.

2.3 Rowhammer

Wie in Abschnitt 2.1 (Arbeitsspeicher, DRAM) beschrieben sind DRAM Zellen in einem Array aus Zeilen („Rows“) und Spalten („Columns“) angeordnet. Zugriffe auf das Array sind über den Row Buffer gepuffert und erfolgen immer auf vollständige Rows.

Um höhere Speicherkapazitäten bei ähnlichen Kosten zu erreichen, ist, ähnlich wie bei anderen ICs auch, eine Erhöhung der Integrationsdichte notwendig. Die Konsequenz dieser Erhöhung der Integrationsdichte ist, dass die Speicherzellen in modernen DRAM ICs so nah aneinander liegen, dass häufige Zugriffe auf Rows zu Speicherfehlern in Benachbarten Rows führen. In der Praxis bedeutet das, dass ggf. manche der in benachbarten Rows gespeicherten Bits den Wert von 0 auf 1 oder anders herum ändern.

2.3.1 Historie. Dieser Effekt wurde 2014 im Rahmen einer Veröffentlichung von Kim et al. [9] das erste Mal ausführlicher untersucht. Die Autoren kamen zu dem Ergebnis, dass 110 von 129 getesteten DDR3 DIMMs anfällig für diese Art von Speicherfehlern waren. Bei dem im Rahmen der Veröffentlichung beschriebene Experiment wurde ein Speichercontroller auf Basis eines Field Programmable Gate Array (FPGA) verwendet, um die benötigten Zugriffe auf die Rows durchzuführen.

Ein Jahr später wurde der erste Exploit auf Basis von Rowhammer von Seaborn und Dullien [16] veröffentlicht. Dieser Exploit ermöglichte es einem Angreifer, die Privilegien unter Linux zu erweitern, indem Bits in den Page Tables, welche auch im Hauptspeicher liegen, durch Rowhammer geflippt wurden. Wenn ein Angreifer schreibenden Zugriff auf einen PTE hat, kann er dadurch beliebige Speicherbereiche mit beliebigen Berechtigungen in den eigenen Adressraum mappen, was effektiv darin resultiert, dass der gesamte Hauptspeicher gelesen und geschrieben werden kann. Ab diesem Zeitpunkt war bekannt, dass es sich bei Rowhammer um ein tatsächlich in der Praxis ausnutzbares Problem handelt. Entsprechend wurden danach Mitigationen gegen Rowhammer entwickelt und es begann, wie in vielen anderen Bereichen der IT-Sicherheit auch, ein Wettlauf zwischen Mitigationen und neuen Angriffsmöglichkeiten.

In den darauf folgenden Jahren wurden viele weitere Angriffsmöglichkeiten und Mitigationen entwickelt [3–6, 8, 11, 15, 17, 18].

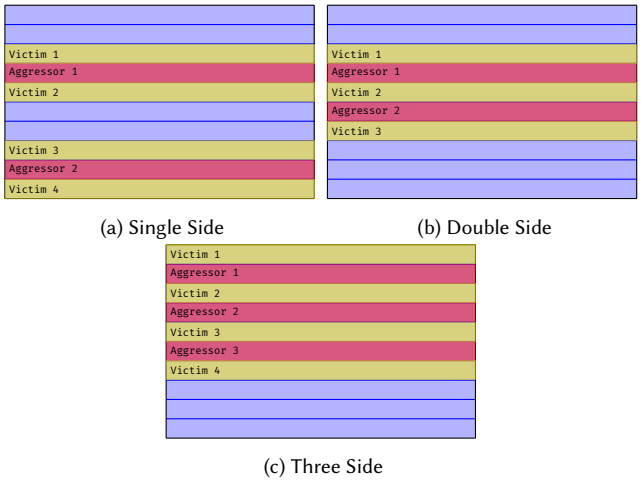
2.3.2 Praktische Umsetzung. Listing 1 zeigt ein Beispiel für die Implementierung von Rowhammer in Assembler. Die Instruktionen in Zeile 2 und 3 laden jeweils den Inhalt von Adresse `X` im Hauptspeicher in das Register `eax` der CPU und den Inhalt von Adresse `Y` im Hauptspeicher in das Register `ebx` der CPU. Anschließend werden die vorher geladenen Variablen in Zeile 4 und 5 unter Verwendung von `clflush` aus dem CPU cache entfernt, sodass sie beim nächsten Zugriff wieder aus dem DRAM geladen werden müssen. Die Instruktion in Zeile 6 führt zu einem Sprung an das Label `hammer` in Zeile 1, wodurch die `mov` und `clflush` Instruktionen in einer Endlosschleife ausgeführt werden.

Die in Listing 1 verwendeten Adressen `X` und `Y` müssen dabei an bestimmten Stellen im physischen Speicher liegen. Die Rows, in denen die Variablen liegen, auf die der lesende Zugriff stattfindet (in dem Beispiel die Rows, die die Adressen `X` und `Y` enthalten) werden

```
1 hammer:
2     mov eax, X
3     mov ebx, Y
4     clflush X
5     clflush Y
6     jmp hammer
```

Listing 1. Beispielcode für Rowhammer (Quellcode von Wikipedia)

als *Aggressor Rows* bezeichnet. Die Rows, in denen nach dem Zugriff möglicherweise Speicherfehler aufgetreten sind, werden *Victim Rows* genannt. In Abhängigkeit der Positionen von Aggressor und Victim Rows, werden verschiedene Muster unterschieden. Abbilding 4 zeigt einige der üblicherweise verwendeten Muster. Da einige Mitigationen auf dem Erkennen von Zugriffsmustern basieren wurde die Verwendung typischer Muster in den vergangenen Jahren verstärkt durch das Generieren von zufälligen Zugriffsmustern abgelöst [3, 4, 8], da diese von den verwendeten Mitigationen häufig nicht als Rowhammer erkannt werden.



(a) Single Side (b) Double Side

(c) Three Side

Fig. 4. Beispiele für Rowhammer-Muster (Abbildung aus den Slides zu [7])

2.3.3 Mitigationen. Da eine Erhöhung der Speicherkapazität eines DRAM IC unter Beibehaltung der physischen Größe bei gleicher Bauart eine Eröhung der Integrationsdichte erfordert, wird die Integrationsdichte der DRAM ICs in Zukunft wahrscheinlich weiter steigen. Laut [4] ist DRAM mit steigender Integrationsdichte anfälliger für Rowhammer, es ist also davon auszugehen, dass die durch Rowhammer induzieren Speicherfehler in Zukunft zunehmen.

Generell können die Mitigationen gegen Rowhammer in zwei Klassen unterteilt werden: Allgemeine und musterbasierte Mitigationen. Zu den allgemeinen Mitigationen zählen u.A. die Verdopplung der Refresh-Rate oder das verwenden von Error Correction Code (ECC) DRAM. Target Row Refresh (TRR) und pseudo TRR (pTRR) sind Beispiele für Mitigationen, die auf Mustererkennung basieren.

Häufig ist mit allgemeinen Mitigationen das Problem verbunden, dass eine zuverlässige Mitigation von Rowhammer einen hohen Overhead mit sich bringt, oder dass Ansätze mit weniger Overhead nicht zuverlässig funktionieren, d.h. die Anzahl der gefunden Speicherfehler je Zeiteinheit sinkt, allerdings treten sie trotz der Mitigation auf.

Die Ansätze, die auf Mustererkennung basieren, reduzieren den Overhead im Gegensatz zu den allgemeinen Ansätzen signifikant, da nicht der gesamte Speicher, sondern nur die Stellen, an denen Rowhammer erkannt wurde, mitigiert werden müssen. Der Nachteil dieser Klasse von Mitigationen besteht in der Erkennung der Zugriffsmuster: Während das Erkennen des Double Side Musters beispielsweise kein Problem darstellt, ist das Erkennen eines automatisch generierten und komplexen Musters kaum möglich.

Aus diesen Gründen ist das Finden von effektiven und zuverlässigen Mitigationen mit akzeptablen Overhead nach wie vor eine offene Forschungsfrage.

3 ANGRIFFE AUF KSM

Wie in den vorherigen Abschnitten erwähnt, bringt die Verwendung von KSM neben dem Vorteil einer deutlich effektiveren Nutzung des zur Verfügung stehenden Hauptspeichers auch einige Nachteile im Bezug auf die Informationssicherheit mit sich. In diesem Abschnitt werden drei mögliche Angriffe auf KSM vorgestellt.

3.1 Information leaks

KSM löst das beim schreibenden Zugriff auf deduplizierte Pages auftretende Problem, dass dieser nicht auf alle deduplizierten Pages, sondern nur auf die Page, auf die der tatsächliche Zugriff stattfindet, angewendet werden darf, durch eine CoW Policy. Dabei wird die Page bei Schreibzugriff an eine andere Stelle im physischen Speicher kopiert, bevor der eigentliche Zugriff stattfindet. Die Konsequenz davon besteht darin, dass ein Schreibzugriff auf eine Page, die dedupliziert ist, deutlich langsamer ist, als auf eine Page, die nicht dedupliziert ist (die deduplizierte Page muss vor dem eigentlichen Zugriff kopiert werden, bei der nicht deduplizierten Page erfolgt der Zugriff direkt).

Wie von Bosman et al. [2] beschrieben, kann dieses Verhalten ausgenutzt werden, um Inhalte im Speicher zu lesen: Ein Angreifer erstellt mehrere Kombinationen einer Page und legt diese im Speicher ab. Anschließend wartet er, bis KSM die Pages dedupliziert hat und misst die Zeit, die zum Schreiben auf diese Pages benötigt wird. Abhängig von der gemessenen Zeit wird unterschieden, ob der Zugriff *schnell* oder *langsam* war. War der Zugriff *schell*, lag die entsprechende Page nicht im Speicher, war er hingegen *langsam*, lag die Page im Speicher.

Da der Angreifer mehrere Pages gleichzeitig im Speicher ablegen kann, können verschiedene Kombinationen parallel getestet werden. Dabei wird Speicherplatz in der Größe einer Page (unter Linux typischerweise 4 KiB) je getesteter Kombination benötigt.

Da QEMU/KVM alle innerhalb der VMs allozierten Pages als *merge candidate* deklariert, sind effektiv alle Pages innerhalb von KVM-basierten VMs von diesem Problem betroffen, sofern KSM auf dem Hostsystem aktiviert ist. Ein kurzer Test einiger gängiger Linux Distributionen[1] zeigt, dass bei keine dieser Distributionen KSM standardmäßig aktiviert ist, allerdings bei allen Distributionen aktiviert werden kann (nicht aus dem Kernel gepatcht wurde).

3.2 Covert Channel

Die Kommunikation zwischen Prozessen und Systemen stellt einen zentralen Bestandteil der heutigen Informationstechnik dar. In vielen Fällen soll diese Kommunikation nicht uneingeschränkt erfolgen, sondern gezielt eingeschränkt werden: Betriebssysteme enthalten Mechanismen zur Interprozesskommunikation, um Kommunikation zwischen Prozessen gezielt zuzulassen oder zu unterbinden, in Netzwerken werden Firewalls verwendet, um manche Arten von Kommunikation zuzulassen und andere zu unterbinden, etc.

Covert Channels stellen Möglichkeiten der Kommunikation dar, die ohne Verwendung der häufig eingeschränkten *typischen* Kommunikationswege Informationen übertragen können. Abhängig vom konkreten Fall kann ein Covert Channel auf eine CPU (z.B. bei Covert Channels auf Basis des CPU Cache) oder ein System (z.B. bei Covert Channels auf Basis der Zugriffszeit auf Rows im DRAM) beschränkt sein.

In Abschnitt 3.1 wurde eine Möglichkeit zum Raten von Inhalten im Hauptspeicher anderer Prozesse (oder VMs) auf dem gleichen Hostsystem vorgestellt, sofern KSM aktiviert und die betreffenden Pages als *merge candidate* deklariert ist. Das in diesem Abschnitt dargestellte Vorgehen zum Missbrauch von KSM als Covert Channel basiert auf dem vorher vorgestellten Ansatz zum Raten von Inhalten im Speicher.

Angenommen, ein Prozess auf einer VM *Gast B* soll mit einem Prozess auf einer anderen VM *Gast A* Kommunizieren, indem der Prozess auf *Gast B* Informationen an den Prozess auf *Gast A* sendet. Beide Prozesse haben den Inhalt einer Page gespeichert, in der die zu übertragende Information durch den Unterschied von einem Bit codiert wird, effektiv gibt es also eine Variante der Page mit einem Wert von 0 und eine Variante der Page mit einem Wert von *1*.

Nun schreibt der Prozess auf *Gast B* die zur Information passende Page in den Speicher (abhängig davon, ob eine 1 oder 0 gesendet werden soll). Anschließend schreibt der Prozess auf *Gast A* beide Varianten der Page in den Speicher und wartet, bis die Page von KSM dedupliziert wurde. Danach schreibt der Prozess auf *Gast A* auf beide Pages und misst die Zugriffszeit. Die Page, auf die der Zugriff *langsam* war, ist die Page, die von dem Prozess auf *Gast B* geschrieben wurde. Entsprechend kann der Prozess auf *Gast A* das gesendete Bit decodieren. In diesem Fall kann ein Bit je Zyklus (Schreiben, auf das Mergen durch KSM warten, lesen) übertragen werden.

Als Erweiterung dieses Ansatzes können die Prozesse den Inhalt mehrerer Pages teilen und diese gleichzeitig verwenden. Dabei ist zu beachten, dass für jedes zu übertragende Bit verschiedene Inhalte von Pages verwendet werden müssen, um zu verhindern, dass KSM die Pages für ein zu übertragendes Bit mit den Pages für ein anderes Bit dedupliziert. Durch die Erweiterung werden $3 \times n$ Pages benötigt, um n Bit in einem Zyklus zu übertragen. Dies ist der Fall,

[1] Alpine Linux, Arch Linux, CentOS, Debian, Fedora, OpenSUSE, Proxmox, Ubuntu, Ubuntu Server

da der Prozess auf *Gast B* je übertragenen Bit eine Page schreiben muss, und der Prozess auf *Gast A* je übertragenen Bit zwei Pages (eine in der Variante für 0 und eine in der Variante für 1) schreiben muss.

3.3 Flip Feng Shui (FFS)

In Abschnitt 3.1 wurde eine Möglichkeit zum Lesen von Speicher eines anderen Prozesses (ggf. in einer anderen VM) auf dem gleichen Host dargestellt. Anschließend wurde in Abschnitt 3.2 ein Covert Channel zum Übertragen von Informationen zwischen Prozessen ohne Verwendung von vorgesehenen Kommunikationskanälen erklärt. In diesem Abschnitt wird ein Ansatz zum Modifizieren des Speichers von anderen Prozessen bzw. VMs auf dem gleichen Host vorgestellt.

Deduplizierte Pages liegen im physischen Speicher an der gleichen Stelle. Aufgrund der CoW Policy führt schreibender Zugriff zum Aufheben des deduplizierten Zustands, wodurch der Zugriff im Anschluss nur auf der Page stattfindet, auf die tatsächlich zugegriffen wurde. Sofern im deduplizierten Zustand allerdings Änderungen am Inhalt der physischen Page ohne das Verwenden einer Schreiboperation stattfinden, betreffen diese Änderungen alle an dieser physischen Stelle deduplizierten Pages.

Der allgemeine Ansatz von Flip Feng Shui (FFS) [15] besteht darin, Pages an einer vom Angreifer ausgewählten Stelle im physischen Speicher zu deduplizieren und anschließend, beispielsweise durch Rowhammer, einen Speicherfehler an dieser Stelle zu provozieren. Dadurch findet kein Schreibzugriff statt und der aufgetretene Speicherfehler betrifft alle an dieser physischen Stelle deduplizierten Pages. Somit ermöglicht dieser Ansatz (mit einigen Einschränkungen) effektiv das Schreiben beliebiger Pages.

Für die praktische Anwendung müssen drei Annahmen erfüllt sein: Einerseits werden Bit Flips als stabil betrachtet, d.h. diese müssen reproduzierbar sein (ein erneutes Durchführen von Rowhammer auf die gleichen Adressen mit den gleichen Inhalten in den Aggressor Rows führt zum Auftreten der gleichen Bit Flips in den Victim Rows). Andererseits müssen der Inhalt der Page, die modifiziert werden soll, sowie der Offset des Bits innerhalb der Page und die Richtung des gewünschten Flips (von 1 auf 0 oder anders herum) bekannt sein. Die dritte Annahme besteht darin, dass die Page zu Beginn des Angriffs nicht im Speicher liegen darf und die Erstellung der Page auf dem Zielsystem durch den Angreifer ausgelöst werden kann. Alternativ ist es auch möglich, dass die Page bereits auf dem Zielsystem vorhanden ist, die physische Adresse der Page allerdings höher ist als die der Pages, die der Angreifer anschließend verwendet. Auch in diesem Fall darf die Page allerdings auch nur einmal im Speicher liegen, d.h. zu Beginn des Angriffs nicht dedupliziert sein.

Generell kann FFS in drei Phasen unterteilt werden: Templating, Mergen und Exploitation. In der Templating Phase werden Pages im Speicher alloziert und nach Bit Flips gesucht, indem Rowhammer auf diesen Bereichen ausgeführt wird. Dabei ist zu beachten, dass KSM aktiv ist, die Pages sollten also zufällige Inhalte haben, um ein Mergen durch KSM zu diesem Zeitpunkt zu verhindern.

Wenn eine Kombination von Aggressor Rows und Victim Rows gefunden wurde, die zu dem gewünschten Bit Flip führt, folgt die zweite Phase. Dabei werden zwei Pages mit dem Inhalt der Page, die angegriffen werden soll, in den Speicher geschrieben, sodass eine der Pages an der in der Templating Phase gefundenen Stelle und die andere Page an einer Stelle mit einer höheren physischen Adresse liegt (um das zu erreichen können in der Praxis Transparent Hugepages (THPs) verwendet werden). Anschließend muss gewartet werden, bis KSM die Seiten dedupliziert hat. Danach kann Erstellung der Page auf dem System, das angegriffen werden soll, ausgelöst werden. Nach erneuten Warten bis KSM die Pages dedupliziert hat liegen nun alle drei Pages an der Speicherstelle, an der vorher der gewünschte Speicherfehler aufgetreten ist.

In der Exploitation Phase wird versucht, den vorher aufgetretenen Speicherfehler durch erneutes Durchführen von Rowhammer mit den gleichen Parametern wie vorher erneut auszulösen. Wenn das gelingt, betrifft der Speicherfehler alle zu diesem Zeitpunkt an dieser Stelle deduplizierten Pages, also auch die Pages des Zielsystems.

4 ZUSAMMENFASSUNG

Im Rahmen dieser Studienarbeit wurden einige Grundlagen zum physischen Aufbau sowie zur logischen Organisation von DRAM aufgezeigt. Außerdem wurde die allgemeine Idee sowie die Funktionsweise von KSM erklärt. Anschließend wurde beschrieben, wie durch Rowhammer Speicherfehler ausgelöst werden können.

Im zweiten Teil wurden drei konkrete Angriffe auf KSM vorgestellt: Ein Information leak, ein Covert Channel und FFS. Diese drei Angriffe ermöglichen es, KSM zum Lesen und Schreiben von Speicherbereiche, auf die eigentlich kein Zugriff bestehen sollte, sowie zur Kommunikation ohne Verwendung eines vorgesehenen Kommunikationskanals zu missbrauchen.

Diese Angriffe zeigen sehr deutlich, wie generell gute Ansätze und Implementierungen aufgrund von nicht bedachten Seiteneffekten (z.B. einer Erhöhung der Zugriffszeit bei Schreibzugriffen auf deduplizierte Pages) zu unvorhergesehenen und mächtigen Angriffsvektoren führen können. Trotz der Einschränkung, dass einige Vorbedingungen bzw. Anforderungen erfüllt sein müssen, lassen sich die dargestellten Angriffe auf normalen Systemen ausführen (FFS wurde als Teil der Präsentation praktisch demonstriert).

In der Praxis führen solche Angriffe dazu, dass KSM effektiv kaum genutzt werden kann, da es zu möglichen Beeinträchtigungen der Vertraulichkeit und Integrität (indirekt auch der Verfügbarkeit) führen kann. Da die Ansätze von KSM besonders im Umfeld von Rechenzentren große Vorteile mit sich bringen, wären effektive und zuverlässige Mitigationen mit akzeptablen Overhead gegen diese Angriffe in der Praxis relevant.

LITERATUR

[1] Linux 2.6.32. URL https://kernelnewbies.org/Linux_2_6_32#Kernel_Samepage_Merging_.28memory_deduplication.29.

[2] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P*, May 2016. URL Paper=https://download.vusec.net/papers/dedup-est-machina_sp16.pdfWeb=https://www.vusec.net/projects/dedup-est-machinaPress=https://goo.gl/ogBXTm. Pwnie Award for Most Innovative Research.

[3] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript. In *Proceedings of the 30th USENIX Security Symposium*, August 2021. URL https://download.vusec.net/papers/smash_sec21.pdf.

[4] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *Proceedings of the IEEE Security & Privacy*, May 2020. URL https://download.vusec.net/papers/trrespass_sp20.pdf.

[5] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *Proceedings of the 13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.

[6] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. *CoRR*, abs/1710.00551, 2017. URL http://arxiv.org/abs/1710.00551.

[7] Martin Heckel. RAEAX – Rowhammer Amplification by Execution of Additional X86 instructions, 2021.

[8] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. BLACKSMITH: Rowhammering in the Frequency Domain. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (IEEE S&P)*, November 2021. URL https://comsec.ethz.ch/wp-content/files/blacksmith_sp22.pdf.

[9] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors. *ACM SIGARCH Computuer Architure News*, 42(3):361–372, June 2014. ISSN 0163-5964. doi: 10.1145/2678373.2665726.

[10] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P 19)*, 2019.

[11] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. ZebRAM: Comprehensive and compatible software protection against rowhammer attacks. In *Proceedings of the 13rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, page 15. URL https://www.usenix.org/conference/osdi18/presentation/konoth.

[12] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 695–711, 2020. doi: 10.1109/SP40000.2020.00020.

[13] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[14] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[15] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *Proceedings of the 25th USENIX Security Symposium*, 06 2016. URL https://download.vusec.net/papers/flip-feng-shui_sec16.pdf.

[16] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. URL https://www.cs.umd.edu/class/fall2019/cmsc818O/papers/rowhammer-kernel.pdf.

[17] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *Proceedings of the 27th USENIX Security Symposium*, July 2018. URL https://download.vusec.net/papers/throwhammer_atc18.pdf. Pwnie Award Nomination for Most Innovative Research.

[18] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proceedings of the 23rd ACM Conference on Computer and Communication Security (CCS)*, Oct 2016. URL https://vvdveen.com/publications/drammer.pdf.