Rowhammer: Problem ohne Lösung

Gruppe 7

Luis Stumpf Lewin Eder

08.05.2022

Inhaltsverzeichnis

Rowhammer: Problem ohne Lösung

1. Abstract

Es ist die ominöse Wolkenwand am Horizont: Rowhammer geistert als gewaltige Sicherheitslücke für Hauptspeicher schon länger durch einschlägige Medien und Fachzeitschriften. Zwischenzeitlich schien das Problem sogar überwunden, die Bedenken mehren sich zuletzt allerdings zunehmend. Im Folgenden wollen wir zeigen, wie Rowhammer funktioniert und darauf aufbauend: Warum es auch in naher Zukunft ein Problem bleiben wird.

2. Grundlagen

2.1 Einleitung: Der Speicher als Fehler Die Schwäche heutiger Speichersysteme gegenüber potenziellen Rowhammer-Angriffen liegt in der Natur des DRAM-Speichers selbst. Die Speicherzellen **sind** quasi der Fehler im System. Wir starten unsere Reise in die Tiefen der Hacker-Theorie also im Kleinen, genauer: im DRAM-Bit.

2.2 Dynamic Random Access Memory - DRAM DRAM hat viele Vorteile. Er ist schnell, kostengünstig und bleibt dabei konzeptionell "simpel", zumindest im Vergleich mit anderem RAM-Speicher. Uns reicht hier eine grobe Idee der Architektur. Die Beschreibung im Text bleibt also bei den Grundlagen.[1]

Um die Probleme der Speicherbausteine gegen Rowhammer-Angriffe zu verstehen, betrachten wir nachfolgend den theoretischen Aufbau einzelner Zellen. DRAM-Bits bestehen in ihrer einfachsten Form nur aus einem Transistor und einem Kondensator (Capacitor), der Information speichert (Siehe Abbildung 1).

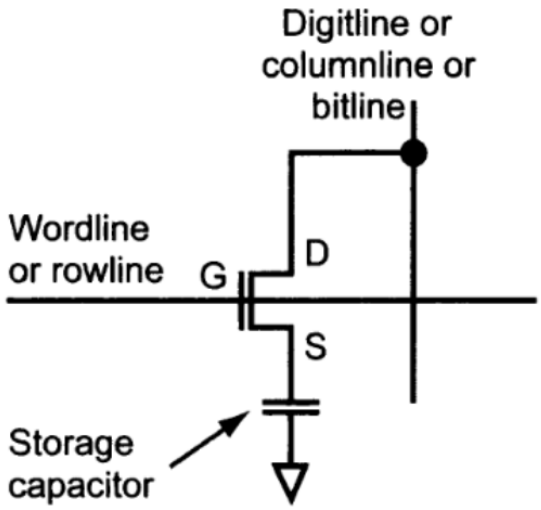

Abbildung 1: Bild aus DRAM Circuit Design [1] S. 10 Fig, 1.12

DRAM ist als eine Matrix solcher Bits aufgebaut (Abb2). Jedes Bit in dieser Speichermatrix wird über eine sogenannte Wort-Leitung und eine Bit-Leitung adressiert - auch Row und Column genannt.

Wenn in ein Bit geschrieben werden soll, muss die Wort-Leitung nur Spannung auf den Gate-Anschluss (Abb1 G) des Transistors geben. Damit wird eine leitende Verbindung von Drain- (Abb1 D) zu Source-Gebieten (Abb1 S) hergestellt und der Kondensator kann über die Bit-Leitung geladen werden. Ganz ähnlich wird auch aus den Kondensatoren gelesen.

Bereits die kleinsten Bausteine enthalten über 1000 Speicherzellen in einer Matrix. In Abbildung zwei wird ein Zugriff in einer solchen Matrix vereinfacht dargestellt.

2.3 Das Problem mit der Grösse DRAM kann extrem kompakt hergestellt werden. Eine Grafik aus dem 2012 veröffentlichten Buch "CHIPS 2020" zeigt, das der durchschnittliche DRAM-Baustein schon vor Jahren aus Billionen von Bits je Quadratzentimeter bestand.[3] Eine so komprimierte Architektur erlaubt Speicherzugriffe im Nanosekundenbereich.

Allerdings bringen Chips mit dieser Kompaktheit und geradezu irren Geschwindigkeit ihre eigenen Herausforderungen mit sich. Kurz: Es gibt auch fehlerhafte Bits, die praktisch nicht zu beseitigen sind.

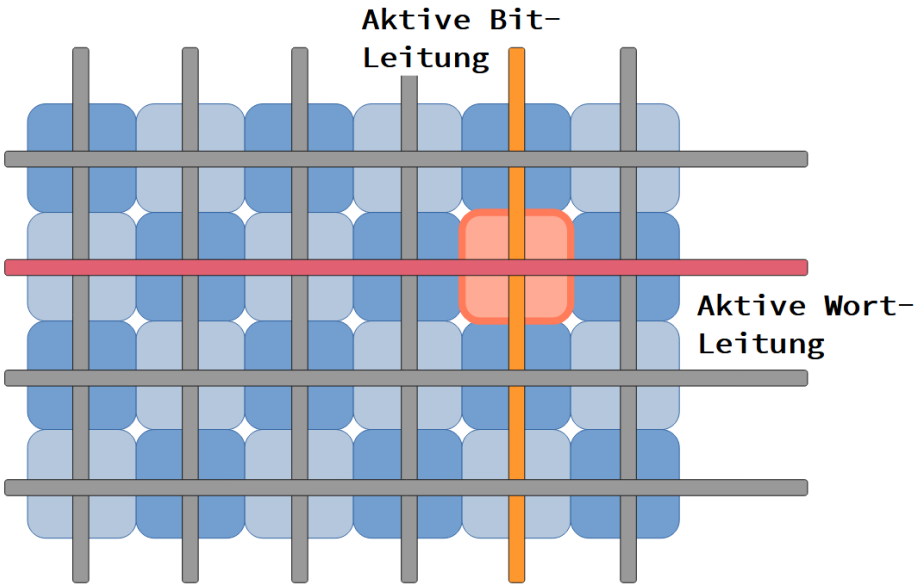

Abbildung 2: vereinfachte Matrix-Darstellung der Bits

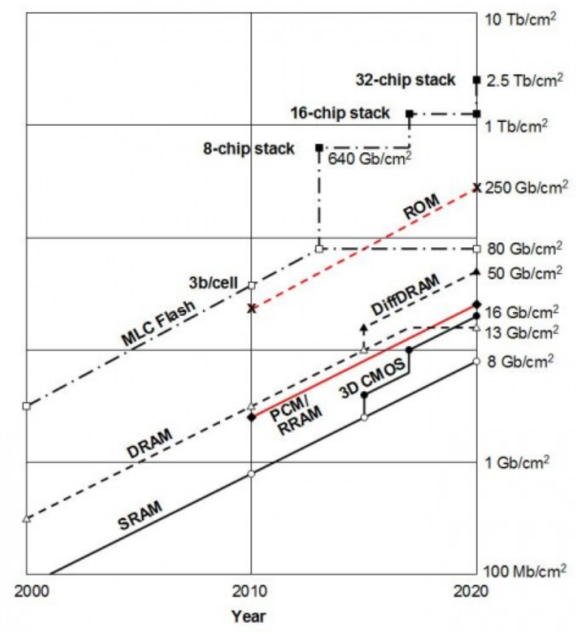

Abbildung 3: *Evolution of CMOS memory density for SRAM, DRAM and Flash memory.* Aus CHIPS 2020 [2],[3]

Diese machen den Chip i. d. R. nicht unbrauchbar, können aber kleine Lecks verursachen, die andere Bits in direkter Nachbarschaft beeinflussen. Im Prinzip können Bits so Ladung austauschen, obwohl sie eigentlich isoliert sein sollten. Und da kommen wir endlich zur Krux: Denn genau diese "schwachen" Bits versucht Rowhammer auszunutzen.[4]

3. Rowhammer

3.1 Die Idee Das Prinzip von Rowhammer ist äußerst einfach. Schwache Bits im Speicher sollen gezielt ausgenutzt werden. Über **schnelle Speicherzugriffe des immer selben Speicherbereichs**, wird im Nanosekundenbereich wieder und wieder die gleiche Wort-Leitung (Row) unter Spannung gesetzt. Der Angreifer "hämmert" sozusagen auf eine Reihe ein. Daher der Name *Row-Hammer*.

Durch dieses Hämmern erhöht sich die Chance, dass ein "schwaches" Bit nachgibt und ein Leck verursacht deutlich.[4] Im schlimmsten Fall löst das entsprechende Bit dann einen Statuswechsel in einer anderen Reihe aus. Etwas konkreter formuliert: Das Leck kann in einem nahe liegenden Bit eine 0 zu einer 1 machen.

3.2 Code Der für das reine "hämmern" nötige Code beschränkt sich für gängige Prozessoren auf ein paar Zeilen x86-Assembler.

```
1 code1a:
2   mov (X), %eax
3   mov (Y), %ebx
4   clflush (X)
5   clflush (Y)
6   mfence
7   jmp code1a
```

Abbildung 4: Beispiel aus *Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors*[5]

In Zeile 1 und 2 werden Registerinhalte in Reihe X und Y gespeichert. Gleich darauf spült oder flushed (invalidiert) das Programm in Zeile 4 und 5 die Reihen wieder[6] und stellt mit mfence in Zeile 6 sicher, dass die Speicherzugriffe in der richtigen Reihenfolge ablaufen. In Zeile 7 wird der Code dann erneut aufgerufen, die Speicherzugriffe wiederholen sich.

Das Speichern und Flushen in zwei unterschiedlichen Reihen ist ein kleiner, aber nötiger Umweg, da gängiger RAM eine Reihe zwischenspeichert, um diese nicht wiederholt aufrufen zum müssen.[5] Ohne den abwechselnden Zugriff auf eine andere Reihe würde das System also nur vordergründig Anfragen an die immer gleiche Zeile richten.

Trotzdem bleibt der Code überraschend einfach. Und fast noch schlimmer: Um diesen Code auszuführen, sind keine Root-Rechte notwendig.

Damit lassen sich Schwachstellen noch nicht gezielt ausnutzen, ein Testprogramm von Google [0] kommt aber bereits mit einigen Hundert Zeilen Code aus. Aktuelle Systeme lassen sich nicht ganz so einfach überlisten. Wichtig ist uns hier einfach zu zeigen, dass Rowhammer auf einem sehr grundlegenden Level agiert. Es nutzt einen Fehler aus, der sich nicht einfach "wegpatchen" lässt.

Abbildung 5 zeigt elementar, dass die Anzahl der anfälligen Bits in Speicherblöcken bis etwa 2012 aufgrund der zunehmenden Dichte immer weiter zugenommen hat. Je näher die Bits also zusammenliegen, umso anfälliger sind sie für Rowhammering. Den Abwärtstrend in späteren Jahren thematisieren wir in einem der nachfolgenden Kapitel.

3.3 Systemschwächen ausnutzen Ab hier beginnt der kompliziertere Teil des Angriffes. Bis jetzt kann unser Prozess nur einzelne Bits zum Kippen bringen, dies geschieht allerdings eher zufällig und bringt im allerschlimmsten Fall vielleicht ein Programm zum Absturz. Um einen wirklich nutzbaren Rowhammer-Angriff anzusetzen, muss sich der Angreifer zum Beispiel in der Paging-Trickkiste bedienen. Wir nehmen hier zu Demonstrationszwecken ein Linux-System, dass so auch von Google-Forschern gehackt wurde.

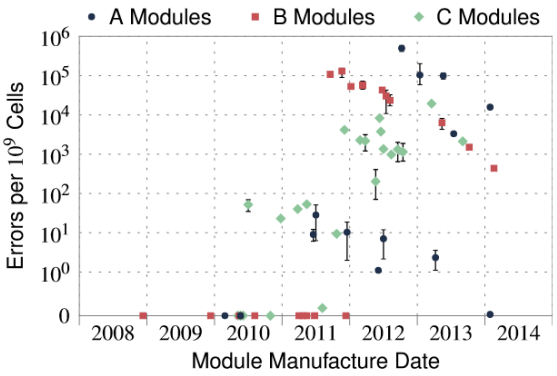

Abbildung 5: *RowHammer error rate vs. manufacturing dates of 129 DRAM modules we tested.* Aus RowHammer: a Retrospektive [4],5

4. Von der Theorie zur Praxis: Angriff auf den Linux-Kernel

4.1 Linux Speicherschutz durch Virtualisierung Der Linux-Kernel hat prinzipiell eine sichere Methode, um Speicher zu verwalten. Das System **virtualisiert** den physikalischen Speicher für Nutzer quasi unsichtbar über Soft- und Hardware. [7] Auch andere Betriebssysteme wie Windows und MAC-OS machen sich diesen Trick zunutze.

Das Betriebssystem teilt den physikalischen Speicher in Seiten (Korrekt eigentlich Seitenrahmen oder auch Seitenkacheln) und kartiert diese in einem separaten Speicherbereich, dem Pagetable. Die Karte für eine solche Seite, im folgenden Pageframe genannt, ist nur für das Betriebssystem sichtbar. Nutzer-Prozessen wird vom Betriebssystem schlicht nicht erlaubt, auf Pageframes zuzugreifen.

So schützt das Betriebssystem sich selbst und Prozesse voreinander. Sie können nur auf für sie speziell kartierte Bereiche zugreifen. Nachfolgend zeigt sich aber: Genau diese Pageframes, also die im Speicher abgelegten Karten, können Linux zum Verhängnis werden.

4.2 Der Pagetable als Achillesverse (Schritt für Schritt-Anleitung) Der Schutz von Speicher gegen bösartige Prozesse basiert im Grunde genommen darauf, dass Prozesse ihren eigenen Pagetable nicht verändern können, weil dieser außerhalb des für sie kartierten Bereiches liegt. Hier schlagen wir Schritt für Schritt eine Bresche in das schöne Sicherheitskonstrukt und beschreiben eine Attacke aus Sicht des Angreifers. Um das Beispiel zu vereinfachen nehmen wir an, dass der physikalische Speicher, von Betriebsystem-Daten abgesehen, leer ist.

Schritt 1:

Wir starten unser bösartiges Programm und erstellen einen geteilten Speicherbereich (Shared-Page), auf welchen wiederholt zugegriffen werden kann.[8]

Schritt 2:

Das Programm allokiert mit dem Systemaufruf mmap() [9] immer wieder denselben Speicher. Es befindet sich also weiter **nur eine Page** im physikalischen Speicher.

Jedes Mapping erhält einen eigenen Pagetable für den geteilten Speicherbereich. Der Speicher wird regelrecht mit Pagetables **zugepflastert**. Diese benötigen zugegeben nicht sonderlich viel Speicher (z.B. 4 kB)[8]. Werden wie in unserem Beispiel allerdings Millionen von Pagetables angelegt, füllt das den Speicher schnell. (Abb. 5)

Schritt 3:

HAMMER-TIME. Jetzt, wo der Speicher praktisch nur noch aus einer Page und Millionen von Pagetables besteht, kommt der große Moment des kleinen Code-Abschnitts in Abbildung 4. Das Programm hämmert im Nanosekundenbereich auf eine Reihe im allokierten Speicher ein. Mit etwas Glück leckt ein Bit. Im benachbarten Pagetable wechselt eine 0 zu einer 1.

Schritt 4:

Dieses eine Bit, kann nun das komplette System kompromittieren. Denn ändert das Bit an der richtigen Stelle seinen Wert, ändert dies die Kartierung des zugehörigen Prozesses. Oder einfacher: Der Pagetable zeigt plötzlich auf einen völlig anderen Bereich im physikalischen Speicher. (Abb. 5)

Und noch besser: Weil der komplette Speicher ja immer noch voller Pagetables ist, zeigt unser neuer Pointer mit großer Wahrscheinlichkeit auf genau so einen Pagetable. Ziel ist jetzt, den Speicherbereich zu finden, der zum attackierten Pagetable gehört. Ist dieser gefunden, sind einem Angreifer eigentlich keine Grenzen gesetzt.[8]

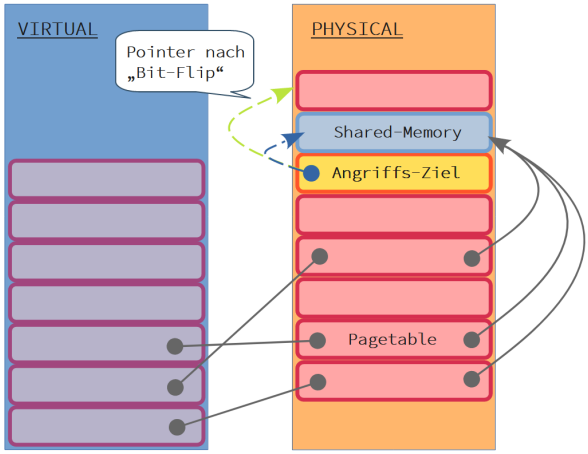

Abbildung 6: *Grobe Visualisierung des physikalischen Speichers*

Schritt 5:

Wir haben Schreib- und Lesezugriff auf einen Pagetable und das im User-Mode. Damit können wir den Pagetable ohne Probleme den kompletten physikalischen Speicher kartieren lassen. Das System gehört praktisch uns.

Die Vorgehensweise ist hier natürlich sehr vereinfacht dargestellt. Die Suche nach schwachen Bits und das Erkennen der richtigen Tables, sowie das korrekte Allokieren und teilweise Freigeben von Speicher brauchen entsprechende Vorbereitung und Fachwissen.

5. Rowhammer in der Realität

Der beschriebene Angriff auf das Linux-System ist für Hacker natürlich nur bedingt nutzbar. Der Angreifer muss lokal ein Programm einschleusen. Allerdings können Angriffe nachgewiesenermaßen sehr vielseitig erfolgen. Wir führen als Beispiel zwei mögliche “Eintritts-Punkte” auf.

5.1 Throwhammer: Eine Rowhammer-Attacke über Netzwerke Ganz allgemein lässt sich Rowhammer nur dann ausnutzen, wenn ein Angreifer mehr oder weniger direkt Prozesse starten und Speicher anfordern kann. Im Gegensatz zu Lese- und Schreibzugriffen im Hauptspeicher sind Netzwerk-Anfragen geradezu lachhaft langsam.

Das heißt allerdings keineswegs, dass Rowhammer-Attacken von einem “Außenstehenden” nicht möglich sind. Rowhammer kann “remote” z.B. gegen Cloud-Services eingesetzt werden und ganz treffend wird diese Methode dann auch **Throw**-Hammer genannt.[15]

Um eine Throwhammer Attacke zu ermöglichen, braucht das Zielsystem eine Netzwerkanbindung mit einer Geschwindigkeit von mindestens 10 Gigabit pro Sekunde. Das ist heutzutage keine Seltenheit mehr.[15] Gerade bei älteren Speicher-Systemen braucht der Angreifer nur einen normalen User-Zugang ohne besondere Rechte.

Auf der Serverseite allokiert der “Hacker” dann einfach einen zusammenhängenden Speicher-Bereich und füllt diesen wiederholt mit Nullen oder Einsen. Währenddessen wird überprüft, ob ein Bit im allokierten Speicher “flippt”. Gibt ein Bit nach, lässt sich so auch in Speicherbereiche schreiben, auf die ein Nutzer keinen Zugriff haben sollte. Allerdings ist hier eine gewisse Kenntnis über das System nötig, da der Nutzer Speicher “intelligent” allokieren und freigeben muss.[15]

Fazit: Sensible Daten nur auf dem eigenen Server, oder auf Servern die nicht “jeden” Kunden akzeptieren.

5.2 Jackhammer: Eine Rowhammer-Attacke über Websites (Rowhammer.js) Auch Attacken auf Heimcomputer sind nicht unmöglich. Hier dient, das bei Programmierern äußerst beliebte Java-Script einmal mehr als Ansetzpunkt für unser virtuelles Brecheisen.

Die Programmiersprache Javascript hat zwar zunächst keine Möglichkeit, Pointer und virtuelle Adressen anzusprechen, kann sich aber sehr große Arrays generieren lassen. Diese werden auf allen aktuellen Google-Chrome- und Firefox-Browsern in einem Linux System als 2-MB-Pages allokiert. Verantwortung trägt hier das Betriebssystem. Die Methode wird bei jeder Scriptsprache so angewendet.[16] Ein bösartiges Skript kann so wiederholt in hoher Frequenz über die Arrays iterieren und damit auch von einer Website aus auf Reihen einhämmern, ergo ein Bit zum Kippen bringen. Ab hier geht ein Angreifer wieder nach dem bereits bekannten Konzept vor.

Der springende Punkt: Ein Angriff startet auf dem Heimcomputer direkt beim Aufrufen einer bösartigen Webseite und ist als solcher auch nicht einfach zu erkennen.

Immerhin können sich auch Heimcomputer-Nutzer denkbar einfach schützen. In allen gängigen Browsern lässt sich Java-Script für unbekannte Webseiten deaktivieren, was wir auch ganz allgemein empfehlen würden.[16]

6. Schutz gegen Rowhammerattacken

Ab hier kommen wir in den Bereich, wo auch Mutmaßungen eine gewisse Rolle spielen. Viele Autoren, die sich jüngst mit Rowhammer befasst haben, bezweifeln, dass sich Systeme aktuell mit vollkommener Sicherheit schützen lassen.[10] Das liegt auch daran, dass ein bekanntes Schutz-Konzept Systeme langsamer macht und deshalb nicht immer praktikabel wäre.

Zudem ist kein bösartiger Rowhammer-Angriff öffentlich bekannt, eine unmittelbare Gefahr und damit auch sofortiger Handlungsbedarf besteht also (scheinbar) nicht. Unsere nachfolgenden Lösungsbeispiele haben wir an die Vorschläge im Bericht: 'Flipping Bits in Memory without accsessing them' von [5] angelehnt.

6.1 ECC (Error-Correction-Code) Die denkbar idealste Lösung für das Rowhammer-Problem wäre, schlicht **bessere Chips** zu bauen. Ohne "schwache" Bits oder mit besserer Kontrolle haben Angreifer keinen Hebel. Theoretisch lässt sich RAM auf fehlerhafte Bits überprüfen, welche dann ersetzt werden können. Dies macht die Herstellung allerdings wesentlich aufwendiger und daher impraktikabel. [5],[11]

Ein gutes Beispiel für eine Lösung ist die bereits in manchen RAM Bausteinen verwendete **ECC-Methode**. Das Error-Corretion-Code-Verfahren verwendet einen Hashwert zur Identifikation von Ein- , Zwei- und teilweise auch Mehr-Bit-Fehlern. Für dieses Verfahren werden 72 statt den üblichen 64 Bits für jede Speicherzelle benötigt.[12] Da aber auch ECC nicht sämtliche Fehler ausmerzt, bietet es keinen optimalen Schutz. Auch-ECC-Systeme wurden bereits nachweislich mit Rowhammering "geknackt".[10]

Immerhin: Ein bisschen mehr Schutz ist besser als gar keiner. Einfache Rowhammer-Angriffe scheitern schon an DDR4-RAM, ein Upgrade schadet also nie.

6.2 Refresh-Rate Noch ein Ansatz für bessern Schutz wäre das **Erhöhen** der sogenannten **Refresh-Rate** innerhalb der Hardware.

In Kürze: D-RAM-Bits verlieren ihre Ladung und müssen daher in sehr kurzen Zeitabständen ausgelesen und wieder geladen werden. D-RAM-Bausteine werden typischerweise alle paar Dutzend Millisekunden aufgefrischt.[14] Der Rowhammer-Angriff **muss innerhalb einer Refresh-Periode abgeschlossen sein**, da die Bits ansonsten wieder ordnungsgemäß geladen sind.

Im Prinzip reicht die Zeit in einem typischen Refresh-Zyklus für mehr als eine Million Speicherinstruktionen. Wird die Refresh-Rate allerdings erhöht, sinkt auch die Chance darauf, dass ein "schwaches" Bit rechtzeitig nachgibt (Abbildung 7).

Problem hier: Eine höhere Refresh-Rate braucht extrem viel Leistung und schlägt deutlich auf die Performance des Systems (mehr Overhead). [5] Das Paper 'An experimental study of DRAM disturbance errors' schlägt hier einen gelegentlichen Teil-Refresh von benachbarten Reihen als Lösung vor. Der Refresh erfolgt per Zufall mit einer geringen Wahrscheinlichkeit. Durch die ständigen Rowhammer-Angriffe, wird der Refresh praktisch immer getriggert. Die Methode nennt sich **probabilistic adjacent row activation** oder kurz PARA. Dafür müsste der Memory-Controller erweitert werden.[13]

7. Fazit: Gelegenheit schafft Diebe

Stand heute ist noch kein gelungener "bösartiger" Hack mittels Rowhammer bekannt. Alle nachweislich funktionstüchtigen Angriffsmethoden wurden von System-Entwicklern und Sicherheitsspezialisten genutzt, um die Schwachpunkte im Hauptspeicher erst nachzuweisen. Da entsprechende Attacken auf älteren

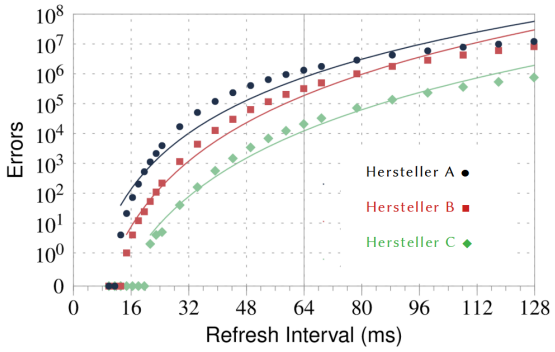

Abbildung 7: *Refresh-Interval vs. Error druch Rowhammer.* Aus RowHammer: a Retrospektive [4],[5] (Die Namen der Hersteller wurden anonymisiert)

Systemen allerdings vollkommen unbemerkt ablaufen können, lässt sich ein Hack auch nicht vollkommen ausschließen. Die Bedenken unter System-Sicherheitsexperten sind zudem real.[10] Gesetz dem Sprichwort im Titel kommen wir zu dem Schluss, dass Chip- und System-Entwickler nicht erst auf den “Dieb” warten sollten.

8. Für Interessierte: (Rowhammer@Home)

Google hat einen Row-Hammer-Test für x86-Computer als Open-Source-Projekt veröffentlicht. Der Test findet nur mit älteren Ram-Riegeln (DDR3) falsche Bits.[0]

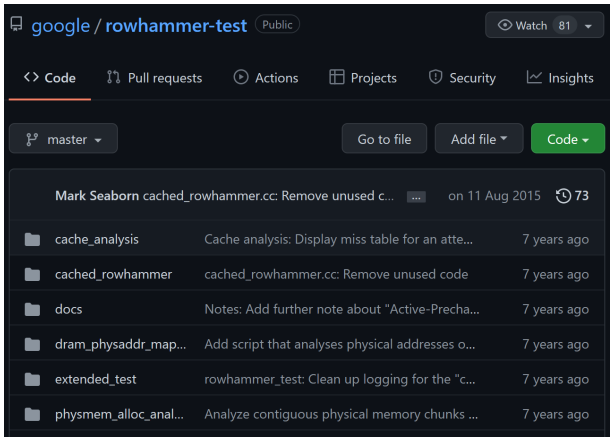

Abbildung 8: Rowhammer Open-Source-Projekt

9. Referenzen

[0] Referenz 0 GitHub - google/rowhammer-test: Test DRAM for bit flips caused by the rowhammer problem

[1] Referenz 1 DRAM Circuit Design: Fundamental and High-Speed Topics - Brent Keeth, R. Jacob Baker, Brian Johnson, Feng Lin - Google Books

[2] Referenz 2 Towards Terabit Memories | Chips 2020

[3] Referenz 3 Chips 2020 | SpringerLink

[4] Referenz 4 [1904.09724] RowHammer: A Retrospective

[5] Referenz 5 Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors | IEEE Conference Publication | IEEE Xplore

[6] Referenz 6 Into the Void: x86 Instruction Set Reference

[7] Referenz 7 Operating Systems: Three Easy Pieces

[8] Referenz 8 Project Zero: Exploiting the DRAM rowhammer bug to gain kernel privileges

[9] Referenz 9 mmap(2) - Linux manual page

[10] Referenz 10 Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks | IEEE Conference Publication | IEEE Xplore

[11] Referenz 11 Nanoscale Memory Repair | SpringerLink

[12] Referenz 12 Understanding and Modeling On-Die Error Correction in Modern DRAM: An Experimental Study Using Real Devices | IEEE Conference Publication | IEEE Xplore

[13] Referenz 13 The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser

[14] Referenz 14 Understanding Reduced-Voltage Operation
in Modern DRAM Devices

[15] Referenz 15 Throwhammer: Rowhammer Attacks over the Network and Defenses

[16] Referenz 16 Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript