Drammer
Deterministic Rowhammer Attacks on Mobile Platforms


Vrije Universiteit Amsterdam          UC Santa Barbara
Graz University of Technology

Presented by David Enderlin
ETH Zürich, 17 October 2019
Background, Problems & Goal
Rowhammer

- Flip bits in adjacent memory rows by “hammering” [1]

Current Rowhammer Exploits

- Current exploits are either
  - Probabilistic [2]
  - Rely on special memory management features [3,4]

- Probabilistic attacks are especially problematic

- Only target x86

- There was doubt whether Rowhammer is even possible on ARM

Rowhammer Exploits in General

- Triggering the Rowhammer bug is different than using it

- We need three things:

  1. Physical Memory Addressing
     - To attack a specific row we have to know which rows are next to it

  2. Fast Uncached Memory Access
     - Hammering fast enough to trigger the Rowhammer bug

  3. Physical Memory Massaging
     - Some way to get the sensitive data into the attacked row
Goal

Show that Rowhammer is possible on ARM/Android

Implement the first deterministic Rowhammer-based Android root exploit

- Without requiring special memory management features
- Without requiring any permissions
Novelty, Key Ideas and Attack Overview
The paper makes two important contributions:

1. **Phys Feng Shui**: A generic technique for deterministic Rowhammer exploitation
   - Using *commodity features* offered by the OS
   - Abusing the predictable behavior of the memory allocator

2. Using this technique to implement an Android root exploit: **Drammer**
RowhARMer

- Rowhammer on mobile devices is possible!

Device: LG Nexus 5 (Android 6.0.1)

# Observed bit flips vs. # NOP instructions

- # Observed bit flips
- median access time per read (ns)

Graph showing the relationship between the number of NOP instructions and the number of observed bit flips and median access time per read.
How To Exploit Rowhammer

1. Physical memory addressing

2. Fast uncached memory access

3. Physical memory massaging
How To Exploit Rowhammer

1. Physical memory addressing

2. Fast uncached memory access

3. Physical memory massaging
1. Physical Memory Addressing

- Physical memory layout is unknown to userspace

- Problem: We need to know the mapping from virtual to physical memory pages to exploit Rowhammer

- Current methods in x86:
  - Pagemap interface
  - Huge pages
2. Fast Uncached Memory Access

- Prerequisite to trigger the Rowhammer bug

- Problems:
  - Memory controller might not be fast enough
  - CPU cache masks out all memory reads after the first

- We need to bypass the cache somehow

- Current methods in x86:
  - Explicit cache flush using clflush
  - Cache eviction sets
  - Non-temporal access instructions (e.g. MOVNTI, MOVNTDQA)
DMA Buffer Management

- Modern (mobile) devices have many different hardware components:
  - e.g. GPU, Display Controller, Camera, Sensors, ...

- OS needs to provide direct memory access (DMA) to support efficient memory sharing between components

- Most devices perform DMA operations on contiguous physical memory pages

- Without DMA the CPU would have to stall for all memory accesses from all hardware components
DMA provides all we need

- DMA bypasses the cache ✓
- DMA gives us physically contiguous memory ✓
  - This provides us with at least relative physical memory addressing
- On Android: ION memory allocator
How To Exploit Rowhammer

1. Physical memory addressing

2. Fast uncached memory access

3. Physical memory massaging
3. Physical Memory Massaging

- Trick the victim into using a memory cell that is vulnerable to Rowhammer

- Victim should store security-sensitive data (e.g. page table) into vulnerable cell

- Current methods in x86:
  - Page-table spraying
  - Memory deduplication
  - MMU paravirtualization
1. Allocate “everything”

2. Free a page which is vulnerable

3. The victim has to use the vulnerable page for its data
## x86 vs. ARM

<table>
<thead>
<tr>
<th></th>
<th>x86 Platforms</th>
<th>ARMv7/ARMv8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Physical Memory Addressing</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pagemap interface</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>Huge pages</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td><strong>Fast Uncached Memory Access</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Explicit cache flush</td>
<td>●</td>
<td>○ / ○</td>
</tr>
<tr>
<td>Cache eviction sets</td>
<td>●</td>
<td>- / -</td>
</tr>
<tr>
<td>Non-temporal access instructions</td>
<td>●</td>
<td>- / ○</td>
</tr>
<tr>
<td><strong>Physical Memory Massaging</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page-table spraying</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>Memory deduplication</td>
<td>●</td>
<td>-</td>
</tr>
<tr>
<td>MMU paravirtualization</td>
<td>●</td>
<td>-</td>
</tr>
</tbody>
</table>

●: Available in unprivileged mode  ○: Available in privileged mode  ○: Not practical enough
Attack Overview

1. **Memory Templating**
   Scan memory for useful bit flips

2. **Land sensitive data**
   Store a page table on a vulnerable location

3. **Reproduce the bit flip**
   Modify the page table to get root access
Mechanisms
Attack Procedure in Detail

1. Probe DRAM row size
2. Phys Feng Shui
3. Hammering the page-table
4. Exploiting
Attack Procedure in Detail

1. Probe DRAM row size

2. Phys Feng Shui

3. Hammering the page-table

4. Exploiting
Probing DRAM Row Size

- We have to know the DRAM row size to apply Rowhammer
- Two page reads from the same bank are slower than from different banks
Probing DRAM Row Size in Practice

Device: LG Nexus 5

The diagram shows the relationship between page 1 and page 2, with the x-axis representing page 1 and the y-axis representing page 2. The color scale on the right indicates the time per read (in ns), with darker shades representing longer times and lighter shades representing shorter times.
Attack Procedure in Detail

1. Probe DRAM row size
2. Phys Feng Shui
3. Hammering the page-table
4. Exploiting
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks

Physical Memory:

```
16 * 4 KB pages = 64 KB rows
```
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks

<table>
<thead>
<tr>
<th>Physical Memory:</th>
<th>Allocate: 64 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
</tr>
</tbody>
</table>

16 * 4 KB pages = 64 KB rows
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks

Physical Memory:

Allocate: 64 KB

16 * 4 KB pages = 64 KB rows
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks

Physical Memory:  
Allocate: 8 KB

16 * 4 KB pages = 64 KB rows
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks

Physical Memory:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Allocate: 8 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>

16 * 4 KB pages = 64 KB rows
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks

Physical Memory:  Allocate: 8 KB

<table>
<thead>
<tr>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
</tr>
<tr>
<td>16</td>
</tr>
</tbody>
</table>

16 * 4 KB pages = 64 KB rows
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks

Physical Memory:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>

Allocate: 8 KB

16 * 4 KB pages = 64 KB rows
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks

Physical Memory: Allocate: 32 KB

256

64

8 8 16 32

16 * 4 KB pages = 64 KB rows
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks

Physical Memory:

<table>
<thead>
<tr>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>32</td>
</tr>
</tbody>
</table>

16 * 4 KB pages = 64 KB rows
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks

Physical Memory: Free: 8 KB

256
64
16 16 32

16 * 4 KB pages = 64 KB rows
Exploit predictable behavior of the Linux Buddy Allocator

- Split smallest chunk until it fits the requested allocation
- On free: Merge chunks back into bigger chunks

Physical Memory:

- 256
- 64
- 32

Free: 8 KB

16 * 4 KB pages = 64 KB rows
Scan memory for vulnerable rows and keep track of which bits flipped in which row.

![Memory Rows]

- Red cells represent rows with flipped bits.
- Green cells represent rows without flipped bits.
- The diagram illustrates how memory rows can be scanned to identify potential vulnerabilities.
Scan memory for vulnerable rows and keep track of which bits flipped in which row

| 00000000000000000000000000000000000000000000000 |
| 11111111111111111111111111111111111111111111111 |
| 00000000000000000000000000000000000000000000000 |

39
Scan memory for vulnerable rows and keep track of which bits flipped in which row
Scan memory for vulnerable rows and keep track of which bits flipped in which row
Scan memory for vulnerable rows and keep track of which bits flipped in which row
Scan memory for vulnerable rows and keep track of which bits flipped in which row.
Scan memory for vulnerable rows and keep track of which bits flipped in which row
Scan memory for vulnerable rows and keep track of which bits flipped in which row
Phys Feng Shui

- L-Chunks: Largest possible contiguous chunk = 64 KB
- M-Chunks: Row Size = 16 KB
- S-Chunks: Page Size = 4 KB
Phys Feng Shui

- Assume we have an exploitable bit-flip in the red location
- Trick the OS to place a page table in that location
Trick the OS to place a page table in the red location

A 1-to-0 flip in the n-th offset bit causes the PTE to point to a location $2^n$ pages before.
Phys Feng Shui - Steps

1. **Exhaust(L) + Template(L)**
2. Exhaust(M)
3. Free(L*)
4. Exhaust(M)
5. Free(M*) + FreeAll(L)
6. Land(S)
7. Padding(S)
8. Map(M)
Allocate as many L-Chunks as possible
Phys Feng Shui – Exhaust(L) + Template(L)

- Allocate as many L-Chunks as possible
- Scan rows in L-Chunks for vulnerable rows (Templating)
Phys Feng Shui – Exhaust(M)

- Allocate as many M-Chunks as possible
Phys Feng Shui – Free(L*)

- Free the L-Chunk with the vulnerable row
Phys Feng Shui – Free(L*) + Exhaust(M)

- Free the L-Chunk with the vulnerable row
- Allocate as many M-Chunks as possible

```
M | M* | M | M
M |     | M | M
M |     |   |   
L |     | M | M
L |     |   |   
M |     |   |   
L |     |   |   
M |     |   |   
```
Phys Feng Shui – Free($M^*$) + FreeAll($L$)

- Free the M-Chunk with the vulnerable row
Phys Feng Shui – Free(M*) + FreeAll(L)

- Free the M-Chunk with the vulnerable row
- Free all remaining L-Chunks
Phys Feng Shui – Land(S)

- Allocate S-Chunks until they land in the vulnerable region
  - We can use `/proc/zone-info` and `/proc/pagetypeinfo` to determine when we reach the vulnerable region
Insert some padding so that the next allocated page-table will be placed in the vulnerable page
Phys Feng Shui – Map(M)

- Force another page-table allocation
- Map the PTE with a bit flip at offset bit $n$ to a location $2^n$ pages away from the PT

![Diagram of page allocation and mapping]
Phys Feng Shui – Map(M)

- Force another page-table allocation
- Map the PTE with a bit flip at offset bit $n$ to a location $2^n$ pages away from the PT

![Diagram showing page table allocation]

- Map the vulnerable PTE to $M'$ which is $2 = 2^1$ pages away

![Diagram showing vulnerable PTE mapping]

- A 1-to-0 flip in the 2\textsuperscript{nd} offset bit of the PTE would result in the PTE mapping to the PT itself
Attack Procedure in Detail

1. Probe DRAM row size
2. Phys Feng Shui
3. Hammering the page-table
4. Exploiting
Hammering

- Hammer until we reproduce the bit-flip from the templating stage

- Our PTE now points to the PT itself and we can effectively access the whole memory including kernel pages.
Attack Procedure in Detail

1. Probe DRAM row size

2. Phys Feng Shui

3. Hammering the page-table

4. Exploiting
Exploitation

1. Fill PT with PTE’s to kernel memory

2. Search for the security context of our own process stored in a struct cred

3. Overwrite our uid and gid to get root privileges
Methodology and Evaluation
Methodology

- Only Android devices were tested

- Architectures:
  - ARMv7
  - ARMv8

- DRAM types:
  - LPDDR2/3/4

- Metrics:
  - Time until first bit-flip
  - Number of bit-flips
  - Number of exploitable bit-flips
## Analysis

<table>
<thead>
<tr>
<th>Device</th>
<th>Hardware Details</th>
<th>Analysis Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><em>SoC</em></td>
<td>DRAM</td>
</tr>
<tr>
<td>Nexus 51</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nexus 52</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nexus 53</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nexus 54</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nexus 55</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nexus 56</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nexus 57</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nexus 59</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nexus 510</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nexus 511</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nexus 512</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nexus 513</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nexus 514</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nexus 515</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nexus 517</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>Galaxy S5</td>
<td>MSM8974 ²</td>
<td>2 GB</td>
</tr>
<tr>
<td>OnePlus One</td>
<td>MSM8974 ²</td>
<td>3 GB</td>
</tr>
<tr>
<td>OnePlus Two</td>
<td>MSM8974 ²</td>
<td>3 GB</td>
</tr>
<tr>
<td>Moto G2013</td>
<td>MSM8226</td>
<td>1 GB</td>
</tr>
<tr>
<td>Moto G2014</td>
<td>MSM8226</td>
<td>1 GB</td>
</tr>
<tr>
<td>Nexus 4</td>
<td>APQ8064</td>
<td>2 GB*</td>
</tr>
<tr>
<td>Nexus 5x</td>
<td>MSM8992</td>
<td>2 GB</td>
</tr>
<tr>
<td>Galaxy S6</td>
<td>Exynos87420</td>
<td>3 GB</td>
</tr>
<tr>
<td>K3 Note</td>
<td>MT6752</td>
<td>2 GB</td>
</tr>
<tr>
<td>Mi 4i</td>
<td>MSM8939</td>
<td>2 GB</td>
</tr>
<tr>
<td>Desire 510</td>
<td>MSM8916</td>
<td>1 GB</td>
</tr>
<tr>
<td>G4</td>
<td>MSM8992</td>
<td>3 GB</td>
</tr>
</tbody>
</table>

¹MSM8974AA  ²MSM8974AC  *LPDDR2  ⁰LPDDR4
Analysis Summary

- 80% of ARMv7 devices vulnerable
- 16% of ARMv8 devices vulnerable
  - Seems more robust
- The same device can sometimes be vulnerable and sometimes not
  - 20% of Nexus 5 devices were not vulnerable
- Time until first bit flip can vary greatly
- Percentage of exploitable bit-flips always around 7%
- LPDDR2/3 is vulnerable
- LPDDR4 maybe vulnerable (only 1 device tested)
Mitigation
Software Mitigation

- Disallowing `clflush` and non-temporal access instructions
- Disallowing `pagemap` interface
- ANVIL [5]
  - Detect Rowhammer attack by observing cache misses

Hardware Mitigation

- Increase refresh rate
  - Needs 8x refresh rate for complete mitigation

- ECC Memory

- Target Row Refresh
  - LPDDR4 supports this

- PARA\textsuperscript{[1]} & ARMOR\textsuperscript{[7]}

\textsuperscript{[1]} Y. Kim, et al. Flipping Bits in Memory Without Accessing Them, ISCA ‘14
\textsuperscript{[7]} M. Ghasempour, et al. ARMOR: A Run-Time Memory Hot-Row Detector, 2015
Drammer Mitigation

- Restricting the DMA interface

- Isolate DMA-able memory from other regions
  - We can currently allocate memory in low memory regions used for kernel and page tables

- Introduce per-process memory limits

- All mitigations that prevent bit-flips are effective
Summary
Summary

- First effort to show that Rowhammer is possible on a platform other than x86

- Implemented a deterministic Rowhammer attack that grants root privileges using DMA and Phys Feng Shui
  - Even without using special OS features
  - Shown by implementing it on ARM/Android

- Many devices are vulnerable
  - If there are bit-flips, the device is vulnerable
Strengths
Strengths

- Novel and elegant solution to exploiting Rowhammer
- Does not rely on special OS features
- It is hard to mitigate if Rowhammer is possible on the device
- Well structured paper
- Most of it is well explained
Weaknesses
Weaknesses

- Assumes that bit-flips are always reproducible
- Not well tested on ARMv8
- Not tested outside of Android
- Some parts are not good explained
- Paper proposed some mitigation options which are not useful (Flikker\textsuperscript{[8]}, RAPID\textsuperscript{[9]})

\textsuperscript{[8]} S. Liu, et al. Flikker: Saving DRAM Refresh-power through Critical Data Partitioning, ASPLOS ’11
\textsuperscript{[9]} R. K. Venkatesan, et al. Retention-aware placement in DRAM (RAPID), HPCA ’06
Related Work
Related Work

- **ARMageddon** [10]
  - Demonstrated cache eviction on ARM

- **DRAMA** [11]
  - Demonstrated that reverse engineering can reduce search time for Rowhammer bit flips

- **Android ION Hazard: the Curse of Customizable Memory Management** [12]
  - Shows security flaws of Android ION memory allocator

---

Takeaways
Takeaways

- Prior x86 Rowhammer exploitation methods cannot be used on ARM/Android

- ARM Memory controllers are fast enough to do Rowhammer

- Drammer is a novel deterministic method to exploit the Rowhammer bug

- Bypasses defenses like ANVIL using DMA

- No easy software fix

- Simple and effective
Open Discussion
Open Discussion

- Thoughts on the previous ideas to exploit Rowhammer?
- Will the problem stay relevant even with recent efforts of mitigation?
- Can you think of any additional mitigation for this attack?
- Could you think of other applications for this attack?
Can this be applied to iOS?
  - If we can use DMA from userspace, probably

It could be used to root your Android Phone with a simple app for your own use
  - No bootloader unlocking needed