Embedded systems security

Hardware architecture/microarchitecture attacks

Prof. Stefano Di Carlo

RowHammer

LAB EXERCISES

Lecture credits:

Nicolas Abril

Massimiliano Di Todaro

Matteo Isoldi

Diego Porto

Co-funded by the European Union

La Région Auvergne-Rhône-Alpes

AGENCE ERASMUS+ FRANCE · EDUCATION FORMATION

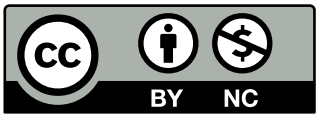
Erasmus+

Acknoledgments

This material was initially developed as part of an assignment for the Operating Systems for embedded systems course delivered at Politecnico di Torino by Prof. Stefano Di Carlo during the academic year 2022/20023.

Credits for the preparation of this material go to:

- Nicolas Abril (`https://github.com/developedby`)
- Massimiliano Di Todaro (`https://github.com/bOhYee`)
- Matteo Isoldi (`https://github.com/mditodaro`)
- Diego Porto (`https://github.com/akhre`)



1 Introduction

The board used to perform the RowHammer attack is the Raspberry Pi 3B+. It ships with the **BCM2837B0** chip which contains an ARM Cortex A53 processor based on an Arm-V8 architecture.

It is suggested to install and use the 64-bit version of RaspbianOS to see the effects of memory corruption correctly, but any 64-bit distribution of Linux can be used.

You can also build a 64-bit version of Embedded Linux through Yocto, but be aware of the configuration needed to compile the OS for the Raspberry board correctly. If you choose this path and during the boot process, you receive some errors or the board doesn't produce any graphical output to your screen, it could be related to the type of image created by the bitbake program.

In this thread you can find what lines have to be changed in the configuration files to get a bootable OS image.

2 Board initialization

First of all, you need to download the software ***"Raspberry Pi Imager"*** from the manufacturer's site on your PC: https://www.raspberrypi.com/software/.

Pick the SD card containing the OS and mount it on the PC. Open *Raspberry Pi Imager* and choose the OS:

Raspberry Pi OS (other) → *Raspberry Pi OS (64-bit)*

Now, you need to choose the correct SD card mounted on the system and then click *Write* to start the process; it will take some time, depending on the quality of the SD card.

When the process has finished, you can put the SD card in the Raspberry Pi 3B+ and plug in the power supply. If there are no errors, the Raspberry should power up and, after a while, show the initial configuration page. In this phase, you need to choose a username and a password.

Note: If you have connected the Raspberry Pi to the LAN, skip to the next point.

Only if the Raspberry is not connected via LAN, you need to configure the WIFI settings running the command:

```
$  sudo raspi-config
```

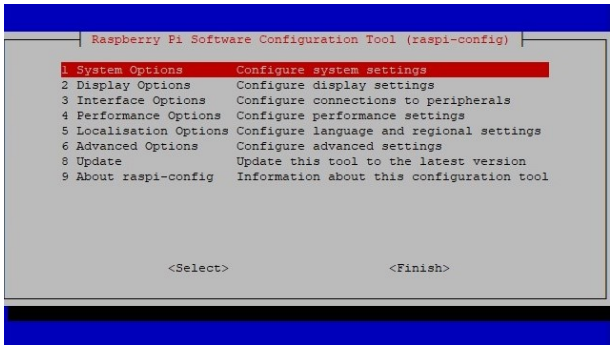


Figure 1: Raspberry Pi Wifi Configuration

After that, you must select *System Options* and then *S1 Wireless LAN*. Now insert the correct information about your WIFI SSID and password.

At this point, the board should be initialized and connected to the internet.

3 Program installation

To run the code, first, you need to download it from the repository and compile it with GCC. For these reasons, you need to install both git and gcc with the following command:

```
$  sudo apt install git gcc
```

Then, you need to clone the repository with the following line:

```
$  git clone https://github.com/developedby/rowhammer_rpi3.git
```

And compile the code:

```
$  gcc -o run_rowhammer rowhammer.c
```

To run the code, execute the following command:

```
$  ./run_rowhammer
```

4 Code

The code is based on several blocks, each with its functionality.

To make it work correctly, you need to set three different parameters:

- *pattern*: the initial values stored in the memory block we want to hammer. A certain bit pattern in the block results in significantly higher disturbance error rates. In this laboratory, there will be only two different patterns.

    1. All 0s: all memory cells are in the discharged state.
    2. All 1s: all memory cells are in the charged state.

- *hammer_type*: it allows you to select which technique to use to perform the attack. In this laboratory, you will use two different techniques:

    1. One-sided hammering: we hammer only a single row performing accesses on it with store operations. We need to perform a load operation between different stores on a second "decoy" row. It is necessary to access this second row because of the presence of the row buffer.
    2. Double-sided hammering: we hammer two rows surrounding one specific victim row.

- *mode*: used to select the specific action to remove/invalidate data from the cache. The code offers three different flavors:

    1. DC CVAC: Cleans data cache by virtual address to PoC.
    2. DC CIVAC: Cleans and invalidates data cache by virtual address to PoC.
    3. DC ZVA: Zeroes data cache by address

5 Time to practice

Follow these exercises to verify your knowledge about this threat and evaluate yourself. How many points, given your answers, would you get?

1. Exercise (2 points): With blast radius, we refer to the physical distance between an aggressor and a victim row. For example, a blast radius of 1 corresponds to the case when

the aggressor and victim rows are adjacent. Given the following memory disposition and location of aggressor rows, describe which rows are likely to be susceptible to bit flips in the following cases:

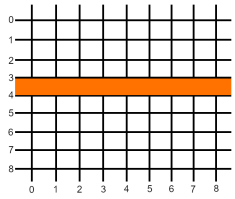


Figure 2: Third-row aggressor row: one-sided hammering with a blast radius of 2

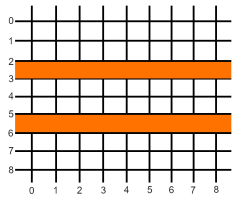
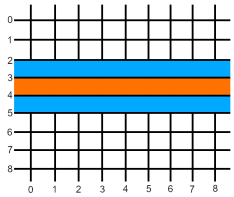


Figure 3: Second and fifth aggressor row: double-sided hammering with a blast radius of 3.

2. Exercise (3 points): You did learn during the lesson what RowHammer is and how much of a threat it represents to the current state of the memory. Explain briefly why the cache interferes with row hammering and how it can be bypassed to target the main memory directly.

3. Exercise (5 points): There are many more cache invalidation instructions than what is used in the code you are provided with. Here are reported all of the 64-bit instructions supported by the Arm-V8 architecture: try to find two more cache invalidation instructions that can be used to implement a double-sided RowHammer, based on the code you already have.

4. Exercise (8 points): With "half-double" hammering, we refer to the practice of harnessing victim rows to further expand the aggressor row influence due to their high-frequency bit flips (more on this practice can be found here). Given the following situation, can you describe what happens when row three gets hammered while rows two and four are accessed simultaneously?



After understanding how this attack pattern works, try to implement it on the source code provided using the three different approaches (**DC CVAC**, **DC CIVAC**, and **DC ZVA**). For simplicity, use one of the rows found by the *find_candidate* function as the aggressor row. The other one is the first victim that gets accessed a few times.

5. Exercise (12 points): In the Arm-V7 architecture, cache invalidation instructions are absent (the architecture supports only 32-bit instructions, while these are 64-bit operations). Even so, attacks on the memory can still be performed: using this library (or one of your choice) for cleaning the cache try to implement one of the approaches already defined and compare the results to see how many bit flips you get.

6 Exercise solutions

1. For the first figure, rows 1, 2, 4, and 5 are most likely to suffer some bit flips. In the second figure, considering a blast radius of 3, every row will differ after hammering.

2. To make a RowHammer attack work properly, the system needs to access the DRAM with high frequency. For this reason, a cache memory could interfere significantly with the flow of the attack by reducing DRAM accesses. To avoid using the cache and directly access the DRAM, different techniques are used:

   (a) Flushing the cache: based on native x86 clflush instruction. It invalidates and cleans from every level of the cache hierarchy in the cache coherence domain the cache line that contains the address specified with the memory operand. Since this is based on a Flush and Reload loop, the attack is performed on the memory and cache.

   (b) Non-temporal access: using operations that don't follow standard cache-coherency rules. If there is memory access, there is no need to cache the data because the memory controller assumes that that data won't be used anymore.

(c) Cache eviction: evict data from the cache by doing multiple loads that go to the same set of the desired evicted data.

3. Other instructions that can be used to bypass cache on an Arm-V8 architecture are **DC CVAP**, **DC CVADP**, and **DC CVAU**. Be aware that they invalidate data on cache lines based on different conditions than that used by the already mentioned instructions.

4. Repeatedly hammering row 3 may cause some bit flips in rows 2 and 4. Still, more importantly, it will prime them, making the probability of flipping 1 and 5 much higher when 2 and 4 are accessed, with a frequency much lower than that of row 3.

5. Using *libflush*, we need first to set up the library objects as explained in the repository, first include with `#include <libflush/libflush.h>`, then initialize the library with `libflush_init(&libflush_session, NULL);`. Then, in the hammering part, instead of the inline assembly, we access the row and then use the library to evict the two used addresses from the cache: `*(unsigned long*)addr1 = attacker_bit;` and `libflush_evict(libflush_session, (void*)addr1)`.
   For machines with access to cache cleaning instructions, we could also use Linux's `cacheflush()` or GCC's `__builtin___clear_cache()` in a similar way to clear the cache while still maintaining some portability.