Analyzing the Resilience of Modern Smartphones Against Fault Injection Attacks

Nourdin Ait el Mehdi

Analyzing the Resilience of Modern Smartphones Against Fault Injection Attacks

by

Nourdin Ait el Mehdi

to obtain the degree of Master of Science
at the Delft University of Technology,

Student number:	4276825	
Thesis number:	Q&CE-CE-MS-2019-11	
Project duration:	February, 2018 – August, 2018	
Thesis committee:	Prof. Dr. Ir. S. Hamdioui ,	TU Delft, supervisor
	Dr. Ir. M. Taouil	TU Delft, daily supervisor
	Dr. Z. Erkin	TU Delft
	Ir. N. Timmers,	Riscure B.V.

Part of this thesis has been removed due to confidentiality.

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Acknowledgments

I would not have been able to finish this thesis on my own, therefore I want to thank all the people that have helped me.

First, I want to thank my supervisors Prof. Dr. Ir. Said Hamdioui and Dr. Ir. Mottaqiallah Taouil from Delft University of Technology (NL). I appreciate their help during the project and all the useful comments with respect to writing this thesis.

Second, I want to thank Niek Timmers and Albert Spruyt from Riscure. I appreciate their guidance during the project. Thank you for making time for me to have weekly meetings and always being open to discussions. Further more I would like to thank everybody at Riscure for giving me a good time there. Everyone was always very friendly and ready to help whenever needed.

Last but not least, I want to thank my parents for their emotional support and motivation, I wouldn't have made it this far without them.

Nourdin Ait el Mehdi
Delft, june 2019

Contents

List of Figures

List of Tables

List of Acronyms

ADB Android Debug Bridge

ADD Addition

ART Android Run Time

APK Application Package

AOT Ahead of Time Compilation

BEQ Branch if Equal

CPU Central Processing Unit

DAC Discretionary Access Control

Dram Dynamic Random Access Memory

EM Electromagnetic

EL Exception Level

FI Fault Injection

JIT Just in Time Compilation

JVM Java Virtual Machine

LIFO Last in First Out

MAC Mandatory Access Control

PC Program Counter

PL Privilege Level

SELinux Security Enhanced Linux

Sram Static Random Access Memory

STR Store

SP Stack Pointer

TEE Trusted Execution Environment

Abstract

The costs and damages that result from cyber security weaknesses are increasing year by year. The cybercrime economy has grown to acquire $600 billion in profits every year, nearly one percent of the global GDP according to a study made by The Center for Strategic and International Studies (CSIS), in collaboration with McAfee. This is likely to increase in the upcoming years especially with the growing IoT and smart devices market, mainly because security is often (partially) ignored during the design phase due to its additional costs. As a result, devices become vulnerable to attacks. In 2017, Accenture, a consulting services company, analyzed the security of 30 mobile banking applications and found out that all apps contained at least one known security vulnerability.

For many years there have been advances in software security. It gets harder and harder to attack software, due to many countermeasures developed over the years. As a result, attackers have shifted their focus to hardware attacks. Exploitable vulnerabilities found in hardware most often require hardware revisions. In contrast to software which can be updated easily. There are many forms of hardware attacks; examples are IP piracy, hardware Trojan, side-channel analysis and fault injection (FI). Recent advances in these attacks made some of them accessible to the public, which worsens the problem. One of the popular hardware attacks is FI, which is able to temporarily alter the functionality of the hardware during run time.

This thesis will focus on analyzing the resilience of smartphones against FI attacks, to further improve the security of smartphones, by researching the possibility of using FI to brute force the lock screen of an Android phone. The attack target is a high frequency package on package chip that has not been attacked before in the literature as far as we know. Based on the target design ad properties of the available FI techniques, electromagnetic fault injection (EMFI) was chosen. Using this technique, a characterization of the target chip was performed to verify its vulnerability to EMFI. To be able to use the acquired knowledge from the characterization a FI model was made, i.e., this model is based on the activity during a glitch and the glitch location. The FI model describes that faults can be injected during the transfer of instructions from the DRAM to the cache. A second characterization was performed using the FI model while running the Android OS, which showed that Android doesn't have any defenses against FI. Before attacking the lock screen, a simulation of the attack was performed, which proved the possibility of successfully performing this attack. Unfortunately, the right time to successfully glitch the lock screen was not found due to the large search space, unreliable trigger and possibly the usage of a non-optimal probe position on top of the chip. Nevertheless, it was demonstrated that the lock screen can be attacked by simulating the effect of glitching certain specific instructions and it was proven that PoP chips with a high clock frequency are vulnerable to EMFI.

1 Introduction

This section introduces the importance, related work and main contributions of the topic of this thesis. First Section 1.1 explains the need for security analysis and why attackers are shifting towards hardware attacks. Second Section 1.2 gives an introduction into cyber security. Thereafter Section 1.3 focuses on a popular hardware attack, i.e. fault injection and discusses the state of the art in fault injection attacks. Subsequently Section 1.4 summarizes the contributions of this thesis. Finally, Section 1.5 provides the outline of the thesis.

1.1. Motivation

Security is often (partially) ignored during the design phase due to its additional costs, which makes designs vulnerable to attacks [8]. The impact of such attacks is gigantic as demonstrated by the examples that happened in the recent years below:

- In January 2017, a hospital in the US got attacked with so called ransomware [9]. The files of the hospital including patient records were encrypted by a malware virus. The attackers demanded four bitcoins ($55.000) for the decryption keys.
- Malvertising are ads on the internet that contain malicious code. This costed ad networks 1.13 billion dollars in 2018 [10].
- Meltdown and Specter are two attacks for which almost all devices containing a CPU are vulnerable. They make it possible for an attacker to steal data from protected memory [11].
- Secret keys of the Nintendo switch were obtained by using fault injection [12]. The ARM7 CPU which is used to prepare the system for the main CPU was glitched by using voltage glitching.
- In 2014 the hartbleedbug was exposed. It was a bug in the OpenSSL cryptography library. This bug made it possible to steal people's private information from servers like yahoo [13] [14].

This problem worsens, as the number of connected devices will double in the next five years as shown in Figure 1.1. The tendency is to connect all kinds of devices such as cars, lights, etc. This network of connected "things" is called an IoT network. The growth is stimulated by projects like Android GO. Android is an open source operating system developed by Google. In December 2018, Android occupied 74.92% of the worldwide market share [15]. Android GO is a light weight version of Android that will be ran on lower end devices. These devices are getting into the 100 Euro price range, which makes them affordable for everyone, including people in developing countries. However, these IoT devices are typically designed by companies that don't have a strong background in security and lack the financial means to properly secure their device [16]. As a result, it becomes easier to attack such devices, which lowers the threshold for cyber criminals.

Digital devices (such as smartphones) are nowadays also being used for high-risk tasks such as banking applications or the storage of private information, without users knowing if these devices are even secure. Many people even replace their banking cards with smartphones, as their usage is convenient and practical. Smartphones contain a lot of privacy sensitive information, that could harm someone when leaked into the wrong

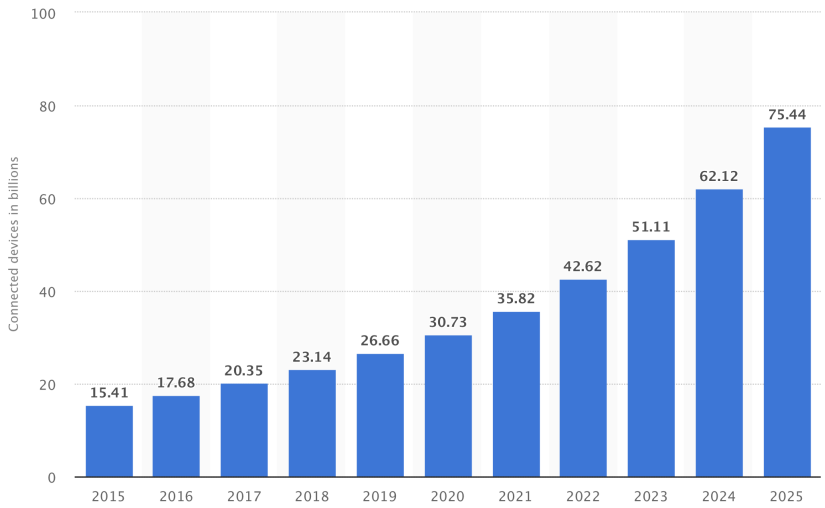

Figure 1.1: IOT devices connected world wide[1]

hands. This new shift gives attackers additional motivation to attack such devices, which is a major concern in a connected world. AccentureConsulting and NowSecure inc. analyzed 30 mobile banking applications and concluded that all of them contained at least one known software security vulnerability, which ideally shows that they aren't fully secure [17].

To counter an attack, a defender has to protect all the possible entrances to a system. As systems can be attacked in multiple different ways, therefore multiple countermeasures are needed. A system can be roughly divided into three components: network, hardware and software. The network handles all the communication to the outside world; this includes wired and wireless communication. The hardware contains all physical components of a device, while the software consists of a set of instructions executed on hardware. Many countermeasures have been developed to protect the system from software and network attacks.
Lately, attackers are shifting their focus to hardware attacks. Software attack countermeasures have been developed for many years now and are relatively secure. There is a constant cat and mouse game taking place between attackers and defenders. Eventually, attackers often find a way around the countermeasures and each countermeasure can be seen as a delay for newer attacks. The focus has been limited on hardware, as a consequence hardware is becoming the main focus of attacks. While software is often updated when a vulnerability is found, this is typically not the case for hardware. Sometimes the only solution, as in the case for Meltdown and Spectre attacks, is to stop using parts of the hardware [18]. This however, has a severe impact on the performance. A vulnerability found in the hardware needs a hardware revision and therefore, devices that are already in the field will remain vulnerable for the rest of their lifetime. This demands the need for hardware security, which is currently highly underdeveloped as a result of software being the main security front.

The most dangerous hardware attacks are the ones that are successful and accessible to a large public. In the hardware domain, two of such attacks exist. The first is side channel analysis; it is an attack that gathers information leaked from the physical implementation of a device. Leakage information could exist of: power [19], acoustic [20], electromagnetic waves [21], etc. The second is fault injection (FI), which is a technique used to alter the code execution flow of a device or change its data. Multiple methods exist to perform fault injection. The most popular methods are [22]: creating glitches in the clock or power supply lines, or glitching using electromagnetic (EM) waves [23] or lasers. Some FI techniques like lasers are only accessible to large businesses or nation state actors, due to the pricey equipment needed. Nevertheless, more and more professional tools to perform fault injection are being sold by companies like Riscure [24] and NewAE [25]. During this thesis, NewAE announced their latest product which is the chipshouter [26]. This device opens up electromagnetic fault injection to the public. Such products lower the threshold for hackers.

Based on the previously mentioned arguments this thesis will focus on analyzing the resilience of smartphones against fault injection attacks. Understanding these attacks is the key to developing new countermeasures. Without appropriate countermeasures, the attacks will have an even bigger effect and become more sophisticated. The next section will first present an introduction into cyber security.

1.2. Cyber Security

Cyber security is the protection of a system, service or data from unauthorized modifications, access, disclosure and interruption [27]. In this thesis, the focus will be on systems. A system can be divided into multiple components; they are access, node and inter-node communication as illustrated in Figure 1.2. The access represents doors to enter the rest of the system. There are many ways to access a system. For example through its I/O ports, scan chains and jtag, etc. All these access ports must be secured properly. The nodes provide the functionality and services of the system. A couple of examples of nodes are: CPU, storage, network controllers etc. There are many different protection mechanisms depending on the node type and its usage. A storage device could for example be equipped with light sensors that remove all the data when the storage is opened. Inter-node communication is as the name already suggests the communication that takes place in the network between the nodes. This network can be wired and or wireless. A way to protect the communication between nodes could be to encrypt the data.

To protect a system, a defender needs to know the security risks and cost of each part of the system versus the security gain. Every security method brings more cost and complicates the system, which is sometimes not desirable and unnecessary. For example, encrypting the infrared signal from a tv remote would just unnecessarily increase the cost due to low risk. Therefore, defenders have to examine each part of the system and decide based on that use case, which security features to include. To facilitate this decision, security criteria can be used. They are explained next.

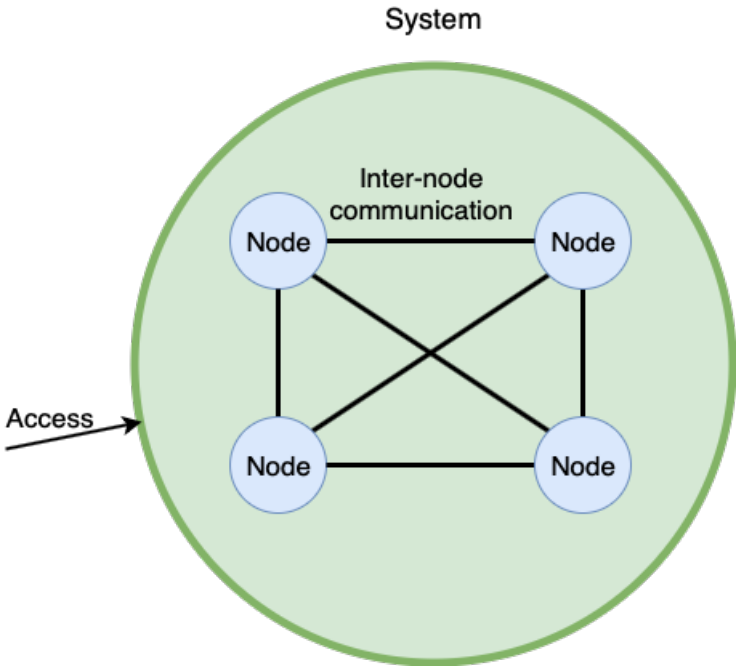

Figure 1.2: System Components

Security Criteria

The security of a system can be evaluated by using three criteria:

- **Confidentiality:** Unauthorized users should not have access to private and sensitive information [28].
- **Integrity:** Unauthorized users should not be able to apply modifications to data [28].
- **Availability:** The system, service, or data should be available when needed [28].

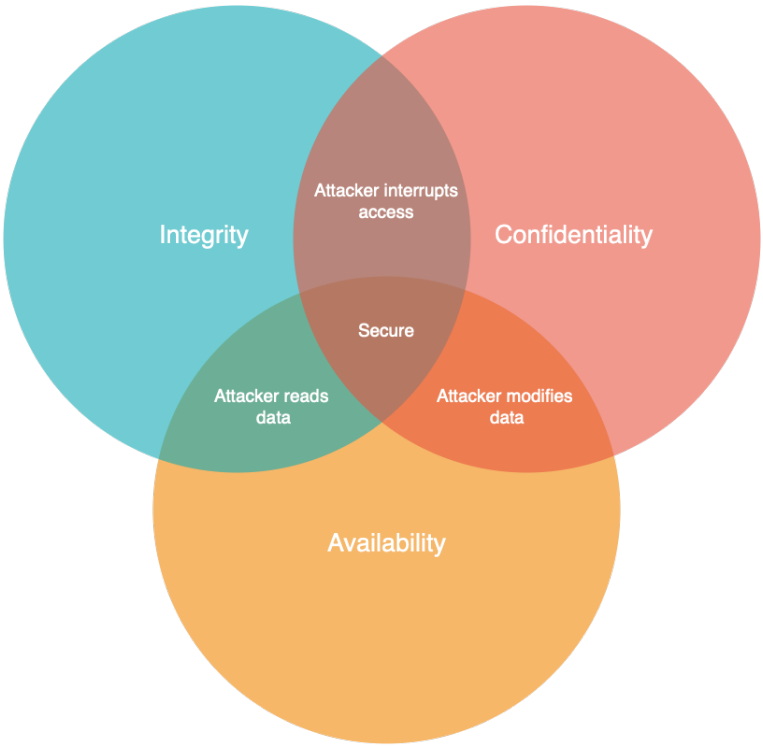

Figure 1.3: CIA Triad

The importance of these criteria depends on the application. For a banking application, confidentiality and integrity are more important then availability. However for a public website availability could be more important. Attacking the lock screen of a mobile phone by using FI breaches the confidentiality and indirectly also the integrity and availability, due to the sensitive information on the phone. The next section will go into the state of the art in fault injection.

1.3. State of the Art

Smartphone security analysis has always been focused on the flaws in the operating system or applications. Like the work presented in [29] [30] [31], they mostly focus on the abuse of permissions in Android. Currently, as far as we known, there is no focus on the security of the under laying hardware.

This thesis analyzes the security of smartphones from a hardware point of view by using fault injection. Figure 1.4 illustrates the most used FI techniques, which can be semi-invasive or non-invasive. Semi-invasive attacks require the removal of the package during the attack without changing the chips functionality. Non-invasive attacks do not require physical changes to the device.

Fault injection was originally used as a method to check the reliability of chips. In 1970 researches noticed that radioactive particles influenced chips [22]. Aerospace groups like NASA became concerned about the effect of charged particles on their airborne systems. The first fault injection techniques were mostly based on software simulations. Later on pins of the chips would be forced to a zero, one or floating state to mimic the impact of radiation [32].

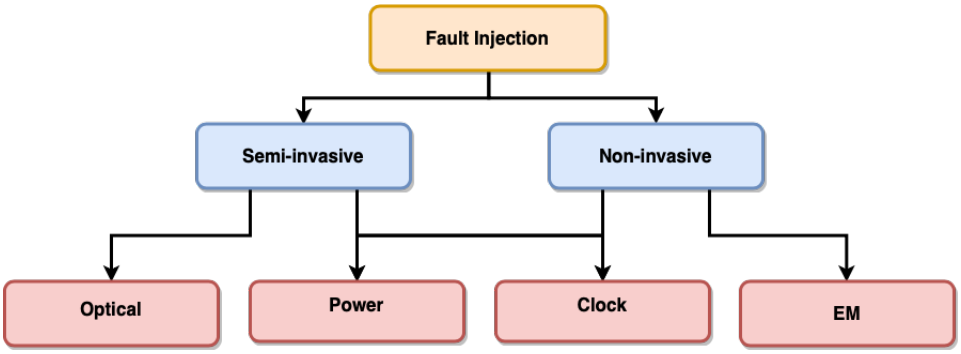

Figure 1.4: Fault Injection Techniques

In 1991 ion radiation and voltage line glitching were used to inject faults into an 8-bit chip. Both techniques resulted in different errors. Tests showed that most errors generated by the ion radiation were seen in the address bus and control flow errors were mostly induced by the voltage line glitching [33].

In 1997 it was theoretically shown that cryptography schemes could be attacked by inducing transient faults [34]. These faults can be described by single bit flips in the registers. Six years later voltage glitching was successfully applied against smartcards running RSA chinese remainder theorem (CRT) when all countermeasures were turned off [35].

Optical fault injection attacks were introduced in 2003 [36]. An eight dollar laser was used to glitch a decapped chip. Individual bits in the SRAM could be set or reset at will. They also mentioned that their technique was successful in influencing jump instructions. An attacker could utilize this to bypass for example password checks.

RSA CRT was again attacked in 2007 with EM waves [37]. The authors used a spark-gap generator to generate an EM field that would influence an 8-bit chip. This attack is harder to control then the laser as the EM waves impacted logic around the targeted SRAM. For example, the program flow was effected and some of the flash memory cells were not writable anymore for several hours.

In 2013 a group of researchers set out to understand more in depth what was happening during an EMFI attack [23]. The target was a 32-bit ARM Cortex-M3 running at 56MHz. They induced bit flips in the chip by using a small coil on top of the chip. The bit flips were observed using JTAG. Tests showed that the EM pulse width and the voltage used to charge the capacitors to create the EM pulses had a different impact, depending on the executed instructions.

In 2017 the Badfet project tried to attack a 1GHz embedded multi core system from ARM by using EMFI [38]. Their goal was to break the secure boot. The authors concluded based on their literature study that using EMFI on chips that have a frequency higher than a 100 MHz was not possible. As a consequence they targeted the DRAM running at a frequency of a 100 MHz and the Flash running at 40 MHz. The boot code was successfully altered in the DRAM. As a result the universal bootloader would encounter a fault and jump into a debug state at start up. This can be used by an attacker to load a binary that could exploit a vulnerability in the system [38].

In 2017 it was shown that voltage glitching can be used to alter complex code [39]. The authors managed to gain privileges in the Linux OS. This attack showed that attackers don't have to rely on programming bugs anymore, due to the code flow altering capabilities of FI. This attack was performed on the ARM Cortex A9 which operates on a 1 GHz frequency. In 2016 the same authors showed that it's possible to control the program counter on an 32-bit ARM processor using voltage glitching [40]. They managed to insert controlled data into the program counter of an ARM chip running at a frequency up to 1 GHz.

Also the secure boot defense mechanism of an Android phone was bypassed using fault injection in 2017 [41]. The attackers managed to induce faults using a laser beam in an ARM Cortex A9 running at 1.4 GHz. Although this chip is used in a package-on-package (PoP) configuration, the DRAM on top was removed so that the CPU can be exposed to fault injection. Due to this the phone is not able to boot into Android anymore. This shows that even high speed chips are vulnerable to fault injection attacks. Note that the success rate of attacking such chips is highly dependent on the fault injection technique used.

The above developments clearly shows that fault injection attacks become more and more sophisticated.

The last successful known fault injection attack on a Android smartphone targeted the secure boot defense mechanism. The next logical step for an attacker would be to target the Android OS itself or parts thereof. This thesis analyzes the possibility of using fault injection against the Android OS running on a recently released chip, that is used in flag ship Android phones, which are still available on the market today. By having insight into these attacks, better countermeasures can be developed.

1.4. Contributions

The goal of this thesis is to analyze the resilience of smartphones against EMFI. The objective is to brute force the Android lock screen by removing the timeout security feature. To achieve this several tasks have been performed. First we determined if it was possible to influence the working of the chip using EMFI. Upfront it was very uncertain if PoP chips with a high clock frequency are vulnerable to EMFI. Therefore a characterization of the chip was performed while running a custom program without the OS. Second, a fault injection model was created based on this characterization; the FI model explains what is happening during a glitch and how to utilize its effect to be able to successfully perform an attack. Third, a second characterization was performed using the FI model, while running the Android OS. This characterization was to determine if the Android OS has any defenses against EMFI attacks. Furthermore, glitching a program running on Android could result in different glitch behaviour in comparison to running without an OS, due to the OS services running. Fourth, an FI simulation was performed on the Android lock screen, to determine the possibility of successfully attacking it by successfully glitching a certain instruction. Finally, the Android lock screen was attacked by using EMFI to demonstrate the danger of this attack. The main contributions of this thesis are:

- **We demonstrated that the Android OS has no defenses against fault injection.**
  Comparing the two characterization phases, i.e. one where an application is run without OS versus one with OS, we observed the same output results. Hence Android has no defenses against fault injection.

- **We discovered multiple weaknesses in the Android lock screen source code.**
  The Android lock screen (gatekeeper) source code was reviewed to identify possible attack targets. This revealed five possible vulnerabilities (instructions) that could result in a successful brute force attack. The attack on two of these five vulnerabilities resulted in a lock screen that was not logging the failed attempts anymore. Hence, the lock screen could be brute forced. This was tested by manually adjusting one of the target instructions in the binaries of the lock screen.

- **We successfully demonstrated that PoP chips even with a high clock frequency are vulnerable to non-invasive EMFI attacks.**
  Both characterizations showed that this chip is vulnerable to EMFI. Multiple different test cases were used to make these characterizations. The characterizations showed that the DRAM on top could be glitched. As the instructions are stored in DRAM it's possible to modify the program flow. The benefit of glitching the DRAM with respect to the CPU is that only one successful glitch is needed. After that the glitched instructions will stay in the CPU cache as long as they are not replaced. This massively increases the success rate.

- **We created a fully automated test setup.**
  A fully automated test setup was build using Riscure's tools. The setup was able to emulate user input and command the attack target based on its feedback.

1.5. Outline

The remainder of thesis is divided into four chapters. The next two chapters give the necessary background and an overview of the affiliated topics to this thesis. The chapter there after contains the core of this thesis and describes the methodology and implementation of the attack. The last chapter concludes this thesis. The list below contains a more in depth description of all the chapters.

Chapter 2 classifies hardware attacks and gives an overview of known hardware attacks and possible countermeasures.

Chapter 3 dives into the Android operating system and the hardware it is running on. This hardware will be the attack target.

Chapter 4 is the core of this thesis and describes the attack in detail.

Chapter 5 concludes the project by summarizing the thesis and proposing future work.

2

An Overview of Hardware Attacks

In contrast to software attacks, where attackers typically can use ready made tools, hardware attacks require specific knowledge and tools. This chapter focuses on different hardware attacks. Section 2.1 explains how hardware attacks can be classified. We distinguish between four major attack classes, which are IP piracy, hardware Trojans, side channel analysis, and fault injection attacks. Sections 2.2 to 2.5 describe the different hardware attacks and countermeasures in more depth, respectively.

2.1. Classification of Hardware Attacks

Figure 2.1 illustrates the hardware attack classification that will be used as the red line of this section. The classification is based on four metrics which are related to the target, technique, phase and location of the attack. The attack target specifies what is attacked and can be subdivided into three main categories: intellectual property (IP), data and functionality.

The second metric, the attack technique, specifies how the attack is performed, it can be subdivided into three main categories: invasive, semi-invasive and non-invasive. Invasive techniques require in-reversible alterations of the design and depackaging of the chip. Semi-invasive techniques require reversible modifications for example by removing the package only. Non-invasive techniques keep the package intact and don't need any modifications.

The third metric, the phase, specifies when the attack is performed. The attack phase can be subdivided into three main categories: design, manufacturing and field. The design phase is the time period when the IC is designed, starting from a specification until the masks are produced. Manufacturing is the phase in which the design is manufactured and tested. After that, the chips are sent to the customers and the devices are in the field. Chips in the field can be obtained by attackers. Therefore, they have all the freedom to attack the device when it's in their possession.

The last metric is the location; it specifies where the attack takes place. The where can be subdivided in: access, node and inter-node communication.

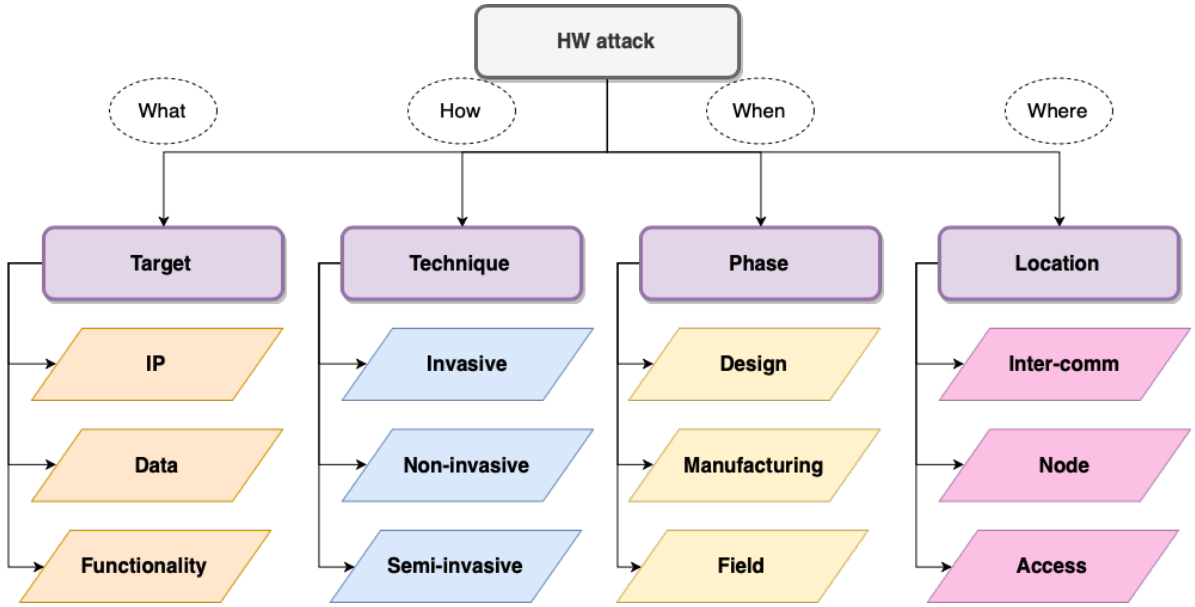

Figure 2.1: Hardware Attack Classification

2.2. IP Piracy

IP piracy involves the illegitimate use of IPs of others, such as distribution of illegal copies [42]. IP's are inventions that are protected by law. However not all countries comply to these laws, as it can be a lucrative business to not follow them. This facilitates IP piracy attacks. Figure 2.2 summarizes the different IP piracy attacks. Note that the location (which is always the node), is not shown to keep the figure concise. The attacks are grouped based on invasive and non-invasive attacks. Both are further explored together with their countermeasures in the following subsections.

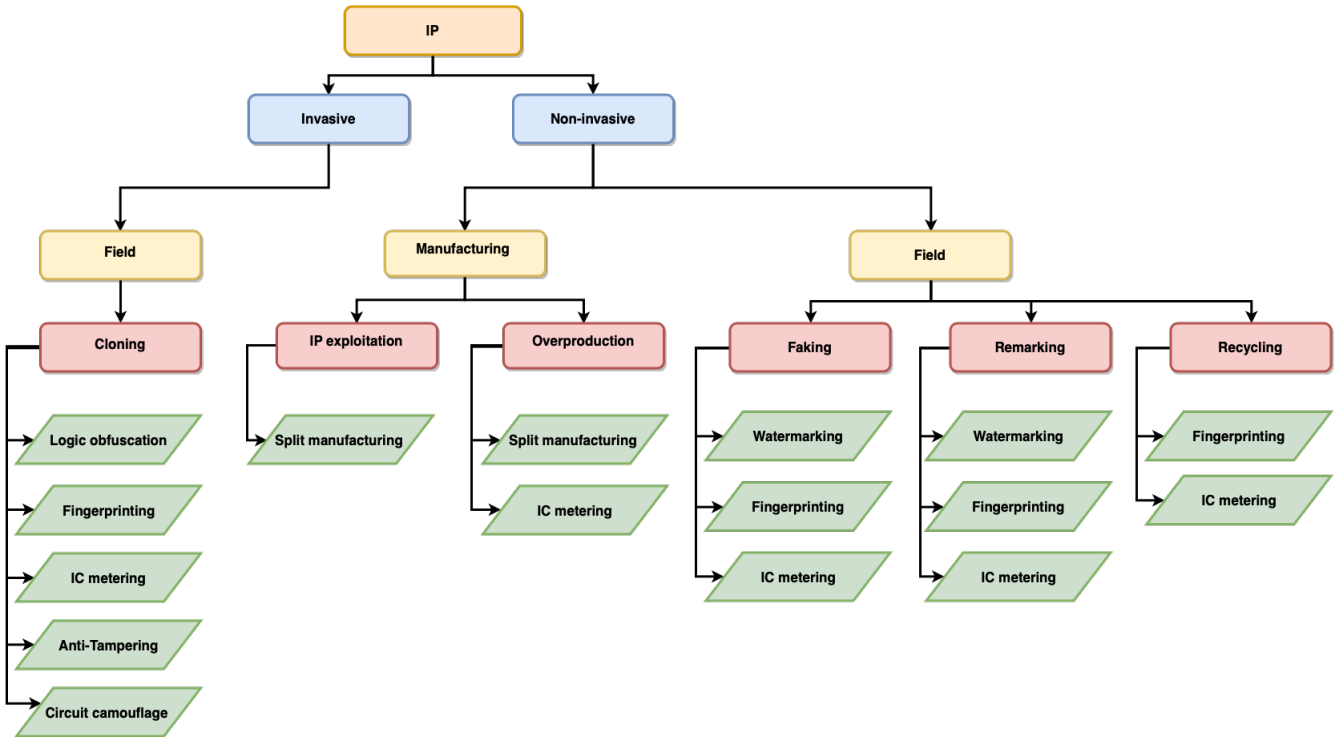

Figure 2.2: IP Piracy

2.2.1. Invasive Attacks & Countermeasures

Invasive IP piracy attacks are only accessible to organizations which have a lot of funding and possess sophisticated knowledge. As Figure 2.2 shows, there exists only one invasive attack which is cloning. This attack only takes place in the field.

- **Cloning:** An attacker illegitimately obtains a design and manufactures it. For example, a chip can be reverse engineered to retrieve the net list. This could give the attacker the ability to sell the same chip for a cheaper price.

There are many countermeasures that can be applied against cloning. They are explained next:

- **Circuit camouflage:** The layout of gates are changed without affecting the functionality. This makes it harder to distinguish between gates as they look similar [43].
- **Logic obfuscation:** Additional so called key gates are added to the design [44]. The design only functions when the correct key-bits are applied to these gates. The key can be stored inside the chip using special protected memory. Even if an attacker manages to clone a device, he or she cannot use it due to these additional key gates. The attacker needs more effort to identify them.
- **Finger printing:** A signature of the designer is placed in the chip to assure its originality without effecting it's functionality [45]. This mark should be hard to remove and easy to read out [46]. As all chips have a unique mark, cloned chips can be identified.
- **IC metering:** Additional logic used to remotely enable and disable chips during and after manufacturing [47]. IC metering can prevent cloned chips from being activated.
- **Anti-tampering:** Extra hardware is added to detect malicious behavior [48]. For example, upon detection of tampering the chip can be destroyed [49].

2.2.2. Non-invasive Attacks & Countermeasures

Non-invasive IP piracy attacks are much cheaper and can be even applied by individuals. Non-invasive attacks can be applied during the manufacturing phase and in the field. The non-invasive IP piracy attacks are:

- **IP exploitation:** The IP is exploited by stealing valuable information from it and for example selling it to third parties.
- **Overproduction:** In this attack, the manufacturer produces more chips then agreed upon. These chips can for example be illegally distributed on the black market.
- **Faking:** An attacker manufactures a product and sells it under the name of a different manufacturer. For example, an attacker makes his own SD cards and sells them under a different brand name.
- **Remarking:** The product of another manufacturer is relabeled (remarked) to be sold as their own.
- **Recycling:** A used or defective product is resold as new by repackaging or relabeling the product name.

There are many countermeasures that can be applied against non-invasive attacks. They are explained next:

- **Split manufacturing:** In split manufacturing, two different semiconductor companies are involved in the fabrication process, e.g. by splitting the Front-end-of-line (FEOL) and back-end-of-line (BEOL). This avoids IP exploitation and overproduction as none of the foundries will have a complete design [50].
- **IC metering:** As IC metering can activate/deactivate individual chips, they help against most non-invasive attacks as well, such as overproduction, faking, remarking and recycling.
- **Watermarking:** Like fingerprinting, watermarking uses a mark to identify chips. However, in watermarking all chips from the same assembly line will be marked the same [45]. Therefore, watermarking is less powerful than fingerprinting as can be used to identify fake chips and remarked chips.
- **Fingerprinting:** As fingerprinting is able to identify devices uniquely, it helps not only preventing faking and remarking attacks but also recycling attacks.

2.3. Hardware Trojan

A hardware Trojan is the malicious modification of hardware. A third party can modify the hardware's functionality or target its data. This modification can have different goals like espionage or failure during deployment in the field. A hardware Trojan is an invasive or semi-invasive attack that changes the initial hardware design permanently or reversibly. During the design phase the attack alters the silicon irreversibly, which makes it an invasive attack. During manufacturing it's possible to change the silicone and or add extra hardware. Adding extra hardware makes it a semi-invasive attack considering that its a separate component that can be disabled. In the field it's still possible that an attacker adds hardware to the device, which is also semi-invasive. A hardware Trojan can target any of the system components.

2.3.1. Attacks & Countermeasures

There is no evidence of hardware Trojans really being used. Nonetheless the danger is very real, which is why there is many research going on about hardware Trojans. In 2017 the Syrian radar system failed to warn of an incoming air strike. The rumors are that a backdoor installed in the systems chips is responsible for this [2]. The authors of [2] separated the hardware Trojans into analog and digital Trojans. An analog Trojan is activated by an analog event like temperature, delay or device aging. A digital Trojan is activated by Boolean logic. The activation triggers a payload that could: activate a side channel, alter functionality or ex-filtrate data. Figure 2.3 shows the structure of a hardware Trojan.

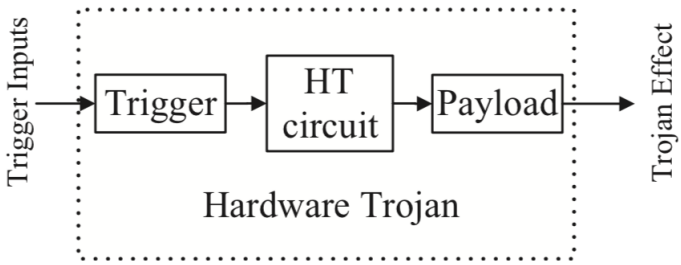

Figure 2.3: Hardware Trojan Structure [2]

There are no of the shelf counter measures that can be added to counter hardware Trojans. Ideally there should be three groups of counter measures: detection, diagnoses and prevention [2]. Detection can be done by reverse engineering the IC. This is a complex invasive technique that consumes a lot of time. Diagnoses is used to find the types, locations and triggers of the hardware Trojan, such that it's possible to remove or mask the hardware Trojan from the circuit. There is no complete or fully reliable technique to prevent hardware Trojans. The used techniques are logic obfuscation, layout filler and split manufacturing. Only layout filler was not explained before. It makes sure that the empty places on the IC are filled such that there is no space for a hardware Trojan. This techniques should decrease the chance on a hardware Trojan.

2.4. Side Channel Analysis

Side channel analysis gathers information leaked from the physical implementation of a device. This information can contain secret data like for example device keys. Multiple methods exist to perform side channel analysis by monitoring: power consumption, heat dissipation, time, acoustic, electromagnetic radiation etc. Known side channel analyses attack methods are summarized in Figure 2.4.

2.4.1. Semi-invasive Attacks & Countermeasures

The only semi-invasive side channel attack is optical analysis. It requires decapping the chip and it's also possible that the substrate has to be thinned.

- **Optical analysis:** The photons released by switching transistors are captured [51]. This technique can focus on a particular part of the chip. The number of photons released and from which part, can be used to determine what the chip is doing.

There are many countermeasures that can be applied against side channel attacks. They are explained next:

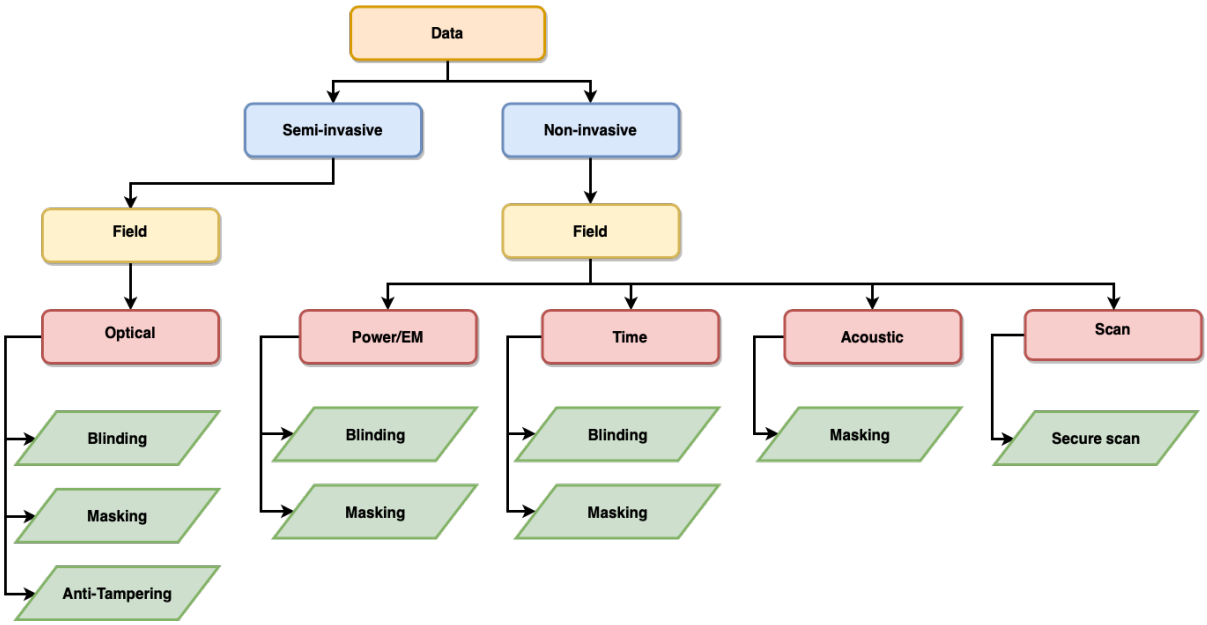

Figure 2.4: Side Channel Attack

- **Blinding:** Is a technique that tries to hide the leaked information by making sure that the leak is constant. For example, the power that is consumed by the chip is always more or less the same [52]. The disadvantage is that the chip will use more power even when it's not needed.

- **Masking:** This technique tries to make the leaked information random. Using the same key twice will result in different power measurements. This is done by for example xoring random data to the secret key before starting the encryption process. This way all parts of the computation will use masked data [53].

- **Anti-tampering:** Extra hardware is added to detect malicious behavior [48]. For example, upon detection of tampering the chip can be destroyed [49].

2.4.2. Non-invasive Attacks & Countermeasures

Side channel analysis attacks are most often non-invasive. The methods below are all performed in a non-invasive way, which doesn't leave a trace on the target. It was even shown that it's possible to record the audio of a cryptography operation of a laptop with a normal mobile phone to obtain the secret key [20].

- **Power analysis:** Monitors the power that a device consumes during operation. The power consumption is linked to what instructions and data a device is using [54]. Executing multiplication instructions consumes far more energy than executing shift operations. This information can be used to obtain the secret key.

- **EM analysis:** It's the same idea as power analysis. The biggest difference is that the EM radiation of the chip is monitored instead of the power line [55].

- **Time analysis:** The time needed for particular operations/computations is measured [56]. This attack happens also often in software. The password that someone just entered is still visible in the cache because the entered key strokes are saved there. By measuring the time needed for a character to for example appear on the screen the attacker can find out if a specific character was already stored in the cache. Based on this an attacker can limit the search space for the password to a couple of characters.

- **Acoustic analysis:** Electronic components vibrate which creates a high pitch noise. This noise can be used in the same context as power analysis [20].

- **Scan chain:** Extra hardware is added to the design of a chip to make it possible to test it. This hardware is called the scan chain. This technique makes it possible to read out or set bits in the chip. The downside of this technique is that it can be abused. An attacker could use the scan chain to extract sensitive information [57].

The previously mentioned countermeasures are all also used for the non-invasive side channel attacks. Only one countermeasure for scan attacks is left:

- **Blinding:** Can also be used for non-invasive attacks. The amount of noise, EM radiation or time between instructions is kept constant.
- **Masking:** Can also be used for non-invasive attacks by randomizing the amount of noise, EM radiation or time between instructions.
- **Secure scan:** The scan chain is adjusted to only work under particular circumstances. This makes it possible to restrict the use of the scan chain during or after critical operations. For example, the chip will first have to be set in test mode which will fully reset the chip and wrap the non-volatile data, before the scan chain can be used. This prevents the leakage of secret information [58].

2.5. Fault injection

Fault injection (FI) is a technique that induces faults in a system to alter its intended behavior by changing its environmental conditions. There are many different fault injection methods [59]. Fault injection is most often a very time dependent attack, therefore the attack technique must be easy controllable. Figure 2.5 summarizes fault injection attacks, however the four most used techniques are: power, clock, laser and EM.
FI started of as a method to induce faults in cryptography operations as seen in Section 1.3. This made it possible to obtain confidential information. Nowadays fault injection is often used to influence executing code. The instructions of a program can be influenced in three ways: instruction corruption, instruction "skipping" and data corruption. Instruction corruption means that an instruction is changed and this change has an effect on the program flow, which was not programmed in before. Instruction “skipping” follows the same principle as instruction corruption. The only difference is that the program flow is not changed. The instruction changes into an instruction that doesn’t have an unintended effect on the functionality of the program. For example, a branch instruction can be "skipped" which will result in the code jumping into the first case after the if. This is seen as normal behavior for the code considering that it was already a possibility that was programmed in. Data corruption is as the name already explains the change of data as a result of fault injection.
There is no full protection against fault injection. Multiple methods exist that try to counter or just make it harder to perform FI attacks. Some of these methods will be elaborated on below.

2.5.1. Semi-invasive Attacks & Countermeasures

Figure 2.5 shows the FI attacks and their countermeasures. Optical and BBI attacks are semi-invasive due to the need for a de-capped chip. Power and clock attacks can be semi-invasive or non-invasive based on if modifications to the device are needed.

- **Optical:** Optical fault injection makes use of the photo electric effect [59]. Light induces photons into the silicon which results in a transient voltage. This voltage propagates through the circuit and induces bit flips. Multiple light sources can be used however the easiest controllable source is a laser. This attack is mostly only performed by attackers with a lot of funds considering that expensive equipment is needed depending on the attack target.
- **Body biasing injection:** Body biasing injection (BBI) is an upcoming semi-invasive technique. A high voltage is injected into the substrate of the chip by probing with a needle [60]. The needle is connected to a capacitor bank that discharges quickly into the needle. This can induce bit flips in the chip when placed on the right spot.
- **Power:** Influences the propagation delay [61]. A change in the supply voltage triggers a wrong bit flip due to propagation delay changes or not having enough power to change the bit value. This attack can be non-invasive or semi-invasive.

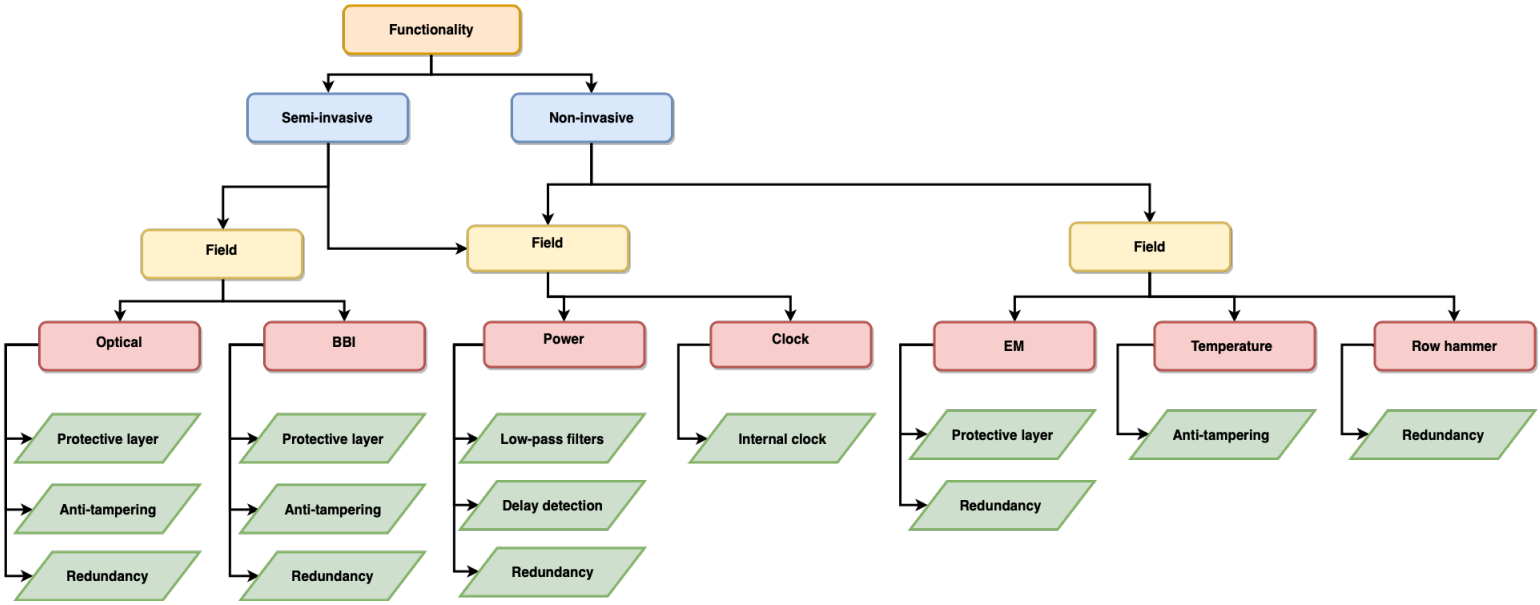

Figure 2.5: Fault Injection Attack

- **Clock:** This thesis focuses on synchronized circuits. These circuits use a clock to synchronize every action. For example, there are 2 flipflops which have some distance between each other. On every rising clock edge, the input of the flipflop is sampled and stored. However this change of the input doesn't happen instantaneously. There is some time between the change of the input of the first flipflop and the second flipflop. This delay is called the propagation delay. For a signal to be sampled correctly it has to be stable before the rising clock edge. Clock glitching abuses this requirement by changing the clock for small periods. Considering the propagation delay, the signal will not be stable yet when the clock period is made smaller, which results in the wrong value being sampled [59]. This attack can be non-invasive and semi-invasive.

There are many countermeasures that can be applied against semi-invasive attacks. They are explained next:

- **Protective layer:** A reflective layer can be added to block a laser from penetrating. Another option could be to add a extra layer of metal to block the BBI needle from touching the substrate.
- **Anti-tampering:** Extra hardware is added to detect malicious behavior [48]. For example, light sensors can be added that will delete the secret keys when the package is opened.
- **Redundancy:** The same calculation is performed multiple times in parallel by copying the same hardware [62]. After the calculations the results are compared. A deviation from the average value may conclude that an FI attack was performed.
- **Low-pass filters:** Low pass filters try to filter out the abrupt change in the voltage, which could counter the effect of power glitching. The problem with this technique is that the low pass filter is often placed outside of the chip, which makes it possible to remove the filter.
- **Delay detection:** A circuit is added that checks the IC for delays [61]. A delay could mean that a fault injection occurred. The problem with such a circuit is that signals naturally vary in their delay. Due to this a good margin has to be chosen to distinguish between a natural delay and one caused by FI.
- **Internal clock:** The clock is nowadays most often generated inside of the chip. This makes conventional clock glitching a lot harder.

2.5.2. Non-invasive Attacks & Countermeasures

Power and clock glitching can be both semi-invasive and non-invasive depending on if the attacker has to perform some modifications. For example, the attacker might need to de-solder the capacitors of the voltage line, which else might filter out the glitch. Or the clock oscillator could be replaced. The other not yet mentioned FI attacks don't require any modifications and are therefore non-invasive.

- **Electromagnetic:** Electromagnetic fault injection is a non-invasive technique that induces EM waves into a chip [38]. There are two possible types of EM waves that can be used: harmonic and transient. Harmonic EM waves are continuously induced EM waves. These waves will influence the chip continuously which makes it hard to reproduce a successful glitch. The second type are transient EM waves which only exist for a small period. These waves are far more controllable than the harmonic waves, which makes a successful glitch repeatable.
  The source of the EM waves is a small coil that is placed on top of the chip. A bank of capacitors is quickly emptied into the coil, which will create transient EM waves that could induce a voltage in the chip. A big advantage of this technique is that the attack target doesn't need to be modified, even the enclosure might not need to be opened.

- **Temperature:** Chips have a certain operating temperature range. Outside of this range the chip might fail or show strange behavior. This is a non-invasive technique that is very hard to control, due to this it's not often used.

- **Row hammer:** Row hammer is a fault injection technique that doesn't need external physical tools like the previous FI techniques. This attack uses a flaw in the design of DRAM chips [63]. DRAM chips are built out of multiple rows which contain the data. If a row's voltage line is toggled very fast, then some nearby rows will leak their charge. This leaking happens at a faster rate than the recharge rate. As a result the other rows might lose some data in the form of bit flips. These bit flips are the injected faults.

The previously mentioned countermeasures can also be used against the non-invasive attacks.

- **Protective layer:** A protective layer made of a wire mesh can be used against EM attacks. This layers acts like an EM shield which blocks the EM waves from penetrating the IC [64] .

- **Redundancy:** Can be used against multiple fault injection attacks regardless of the attack being semi-invasive or non-invasive. Defeating this protection needs the creation of multiple similar faults at the exact same time in different parts of the hardware, which is a very hard task.

- **Anti-tampering:** Extra hardware is added to detect malicious behavior [48]. For example, temperature sensors can be added to shutdown the chip when a certain temperature is reached.

3

Case study: Android

Android is the most used mobile operating system in the world as seen in the introduction and therefore evaluating its security features is important. In this chapter Section 3.1 first dives into the architecture of Android. Thereafter Section 3.2 elaborates the security features build into Android.

3.1. Android Architecture

The Android operating system, simply referred to as Android from now onwards, consists of a stack of software components. These components can be roughly sub divided into: Linux kernel, hardware abstraction layer (HAL), native libraries, Android runtime, application framework and the applications as can be seen from Figure 3.1. The Linux kernel contains all the drivers for the hardware it's running on. These drivers are connected to the rest of the software stack through the hardware abstraction layer. This layer is just an interface to make it possible for manufactures to easily add a new driver to their device without having to modify the higher levels. The Android runtime makes use of the Linux kernel functionality through the HAL. Next to the Android runtime the native libraries can be seen. These are the libraries which are generally in Linux. Above the Android runtime/native libraries the Android framework is seen, which contains all the core libraries that are specific to Android. These libraries implement specific functionality that can be used by the applications. The applications layer is the layer that the user directly interacts with, which provides for example: the browser, Email client, media player etc. Each component is described further in a separate subsection.

3.1.1. Linux Kernel

Android is built upon the famous Linux kernel. The Linux kernel is the abstraction layer between the hardware and software. It contains all the necessary drivers for the hardware, networking, memory management and process scheduling. The Linux kernel used for Android was specifically adjusted for Android. One of the most noticeable differences is that in Linux each user has its own process and memory space. In Android however, each application is seen as a user and therefore has its own process and memory space. This concept is the foundation for one of Androids main security features named application sand boxing; which will be further described in Section 3.2.1. Additional functionality added to the Linux kernel is listed below [65] [66].

- **Android Shared Memory (Ashmem):** a file-based shared memory system.
- **Binder:** an inter-process communication (IPC) and remote procedure call (RPC) system.
- **Logger:** a high-speed in-kernel logging mechanism optimized for writes.
- **Paranoid Networking:** a mechanism to restrict network I/O to certain processes.
- **Physical Memory (pmem):** a driver for mapping large chunks of physical memory into user-space.
- **Viking Killer:** a replacement OOM killer that implements Android's "kill least recently used process" logic under low memory conditions.
- **Wakelocks:** Android's unique power management solution, in which the default state of the device is sleep and explicit action is required (via a wakelock) to prevent that.

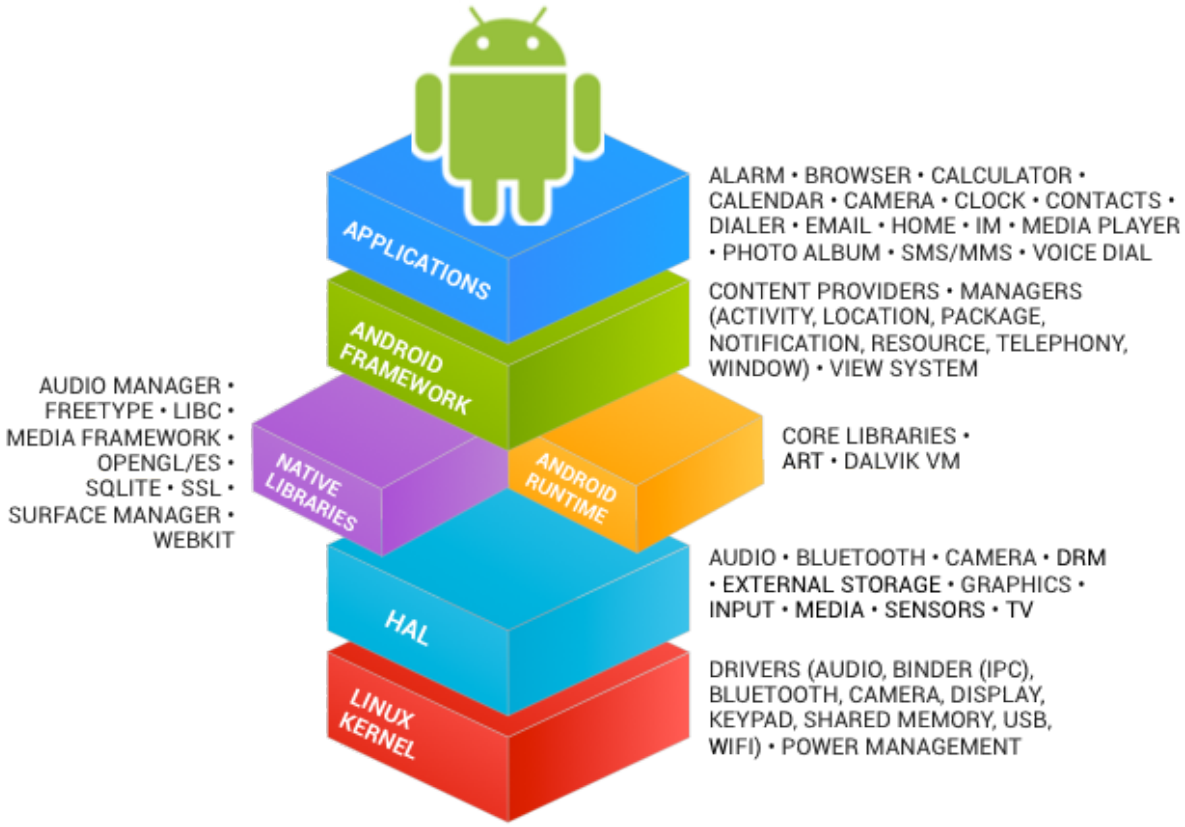

Figure 3.1: Android Software Stack

3.1.2. Hardware Abstraction Layer

The layers above the Linux kernel need someway to make use of the kernel's functionality without having to change the upper levels. The HAL contains standard interfaces to connect drivers to the higher levels without having to modify these levels. This makes it possible to run Android on all kinds of hardware combinations. There is further not much to say about it.

3.1.3. Android Runtime

The HAL is used by the Android runtime to communicate with the Linux kernel in a standardized fashion. Android run time (ART), which is the successor of the previously known Dalvik, is a Java Virtual Machine(JVM) specifically tuned for Android. It has been in use since Android 5.0 (Lollipop) released in 2014. Each application uses it's own ART copy to prevent the whole system from crashing, when one process crashes. The benefit of using a JVM is that application developers only have to develop the source code once. The source code is compiled into Java byte-code which will be changed into machine specific code by the JVM and hence, being independent from the machine. An in dept explanation of the JVM is out of the scope of this thesis and for more details we refer interested readers to [67].
The first JVM for Android called Dalvik was not able to directly translate Java byte-code into machine code. The Java byte-code had to be first translated into Dalvik byte-code and stored in so called dex files. These dex files were stored inside system Java libraries and application packages (APK). Dex files are optimized for systems with limited memory and computing power, which was back then the average specification of an Android device.

The biggest difference between the general JVM and Dalvik/ART, is that the JVM uses a stack based architecture and Dalvik/ART a register based architecture. A stack based architecture manages the memory by using a stack. The stack stores all the variables and results in a last in first out manner (LIFO). For example, the ADD instruction uses two variables and returns one result. The two variables need to be popped from the stack and the result needs to be pushed after the ADD instruction is done, as can be seen in listing 3.1.

```
POP 5
POP 4
ADD 5, 4, res
PUSH res
```

Listing 3.1: Stack based addition

The listing above shows four instructions. Three of these instructions the pop and push are there to interact with the stack. The advantage here is that the stack pointer can be used to address the variables. Therefore it's not needed to know at exactly what address the data is stored.

In a register based architecture the memory is managed by a number of registers. Therefore the address of the variable must be known to use it. The advantage in this case can be seen in listing 3.2.

```
ADD R1, R2, R3
```

Listing 3.2: Register based addition

The listing above shows only one instruction that uses three registers (R1, R2, R3), which reduces the execution time. In this case it's not needed to explicitly interact with the memory three times, which is very important for the performance constrained devices that Android runs on. Also, a result can be stored for a longer time without recalculating it when needed. The disadvantage of using the register based architecture is that instructions will be bigger. This is due to the address of the register which has to be explicitly added to the instruction. This is not the case for stack based architectures as they use the stack pointer.

ART is able to run most of the applications that were made for Dalvik. However Dalvik has some techniques that do not work on ART. The biggest difference between ART and Dalvik, is that ART uses ahead of time (AOT) compilation and Dalvik uses just in time (JIT) compilation.
JIT means that applications are compiled (dex to machine code) during runtime. This technique was used back in the day when Android devices had a small amount of memory. The advantage of JIT is that it doesn't have to compile the dex files at install time. This reduces the amount of memory needed to store the applications. On the other hand JIT introduces more overhead on the CPU side which has to compile dex files on the run.
Nowadays flash memory has become a lot cheaper which is one of the reasons for replacing Dalvik with ART. AOT compilation compiles the whole application to machine code at install time. This technique should remove the extra load on the CPU introduced by JIT compilation. As a result the runtime performance of the application should increase.
Having this ART does not mean that Android is only able to run Java code. Other languages like C can be compiled into machine code as well and executed directly on the device. For more information about ART and Dalvik see [68].

3.1.4. Native Libraries

The native libraries are not specific to Android. These libraries are part of Linux or can be added to Linux. Like LIBC which is a standard library that contains functionality used in C code. OpenGL which is a library used to draw (3D) graphics on the screen. Webkit which is used to render browser pages etc.

3.1.5. Android Framework

The Android framework contains all the functionality that is needed by application developers, which are not part of the standard Java libraries. Applications are build out of 4 components: activities, services, broadcast receivers and content providers, which are all based on the Android framework.
An activity represents a single screen with a user interface [69]. The Android framework makes it possible to have multiple activities which all have a different task. For example, a music player app can have an activity for browsing through the music files and an activity for playing music, these activities can be used separately from each other. As a result applications can make use of particular activities from other applications. However they need the right permissions for this, which will be seen in Section 3.2.3.
A service is a task that is running without an user interface. The user may or may not be aware of this running task depending on what the task is doing. Services are used by activities to perform tasks without blocking the user interface. For example, synchronizing emails is a service which will run unnoticed (background service), playing music is a service that can be ran without an user interface however this is noticed by the user (foreground service).

A broadcast receiver is the part of an application that makes it possible to interact on events without being active. These events come from other applications or the system. The broadcast receiver will trigger the right activity of an application when it receives a request for this.
A content provider makes it possible to share data between applications. An SQL database is an example of a content provider. Applications all have there own process and memory space as seen in Section 3.1.1. This is the reason for using a separate component, which makes it possible to access the data of another application, if given the right permissions.

3.1.6. Applications

Applications can be divide into two groups: systems applications and user applications. System applications are pre-installed by the manufacturer and most often cannot be deleted. This is partly true because there are ways which may void the warranty of the device, to delete these applications. The system applications are installed in a different place then the user applications and also have different permissions. These permissions change based on the capabilities and needs of the application.
User applications are installed by the user him self. These applications are stored in a different place and with less permissions. The main source for these applications is the Google Play Store, which is an online marketplace for applications.

3.2. Android Security

Android has multiple security features which are elaborated in this section. Some features are specific to Android and others are inherited from Linux as will be seen. First is the application sandbox which is Android's main security feature, that makes sure that applications can't make use of another application's process or data. This security feature is strengthen by the use of security enhanced Linux which expends the granularity of the application sandbox. However an application can be given the ability to make use of services or data outside of its sandbox if given specific permissions, by using inter process communication (IPC). Android has a specific IPC implementation called Binder. Furthermore Android uses encryption to keep the data in the device safe from extraction by unauthorized persons.

3.2.1. Application Sandbox

The application sandbox as it is called is Androids main security feature implemented at kernel level [70]. From now on the application sandbox will be referred to by sandbox. Applications in Android all have their own process and memory space as mentioned in Section 3.1.1. As a result applications that don't have explicit permission can't make use of another applications process or data. Figure 3.2 shows the difference between using a sandbox and without. This limiting of the reach of applications is based on discretionary access control (DAC).

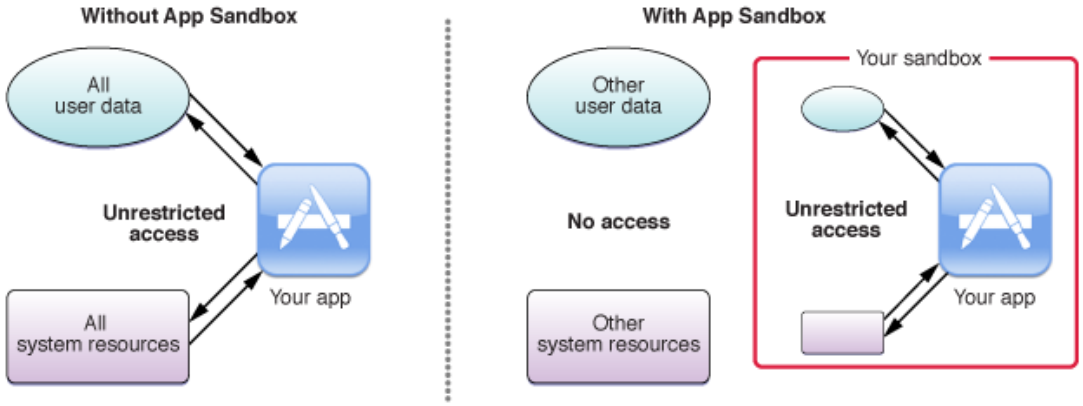

Figure 3.2: Application Sandbox [3]

The sandbox doesn't fully protect the user against malicious applications. It only limits their reach to their own process and memory space. Which is not fully true depending on the permissions that the application has.

The application sandbox also makes Android more stable. If an application crashes it most of the time only stops it's own process. This prevents the whole system from crashing.

DAC

In Linux every resource, service or data has a list containing all the user ID's (UID) and group ID's (GID) that are allowed to make use of it. These lists are maintained by the user who is the owner of this resource, service or data. Based on this UID and GID Linux decides if a user has permission to read, write or execute. In linux all users have there own UID and can be part of a GID. In Android the principle stays the same, however every application has a UID and GID. These values are assigned at install time by the kernel. This concept makes it possible for Android to know what permissions an application has.

There are a couple of problems with DAC [71] [72]. Linux and also Android have one UID which is called the superuser or root. This UID gives access to the whole system without restrictions, which forms a single point of failure. An attacker that is able to become the superuser has compromised more or less the whole system, due to the fact that all the permissions are only based on the UID and GID. Android users at first don't have access to the superuser. However they are free to so called root there devices to become the superuser. This is at the price of risking to lose there warranty on the device. Figure 3.3 illustrates gaining root access.
A second problem is that DAC doesn't protect the flow of information. An user that has access to some data can give any other user access to it. Considering that if an user has read access, nothing prevents him from copying this data. The data can be copied to a place where the other user does have access rights.
A third problem is that an user or an application ran by the user, has the ability to change permissions. This way a malicious application could give access to unauthorized applications or users. Furthermore a user could be tricked into changing the permissions him self.

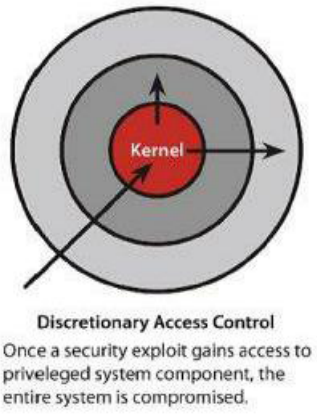

Figure 3.3: Gaining root access in DAC [4]

3.2.2. SELinux

Security enhanced Linux (SELinux) is an enhancement for the Linux kernel [73] [74] [70]. It's designed to solve most of the problems that DAC has. SELinux uses mandatory access control (MAC) which is based on the bell-lapadula model.

Bell-lapadula

The bell-lapadula model is based on the military level based access control [75]. This model works with subjects and objects. Subjects are in the case of Android processes/applications. Objects are: resources, services or data. There are four possible interactions between subjects and objects:

- Subjects can only read the object but not modify them. (Read only)
- Subjects are able to write the object but can't read it. (Append)

- Subjects are able to execute the object but are not able to directly read or write it. (Execute)
- Subjects can both read and write the object. (Read and write)

In the military level based access control subjects and objects have 4 classifications: unclassified, classified, secret and top secret. Every subject is classified by the system admin. Subjects on there turn classify their objects. The operating system will use these classifications to decide if a subject has permission to access an object. There are 2 basic rules:

1. No subject has read access to an object with a higher classification level then the subject. (simple security policy)
2. No subject has permission to write to an object with a lower classification level. (star property)

Rule 2 has an exception for so called trusted subjects. They are able to write to an object with a lower classification level. Subjects are being labeled trusted by for example the system admin. A variation on rule 2 is the strong star property. This property says that subjects can only write to objects of the same classification level. For more information about the bell-lapadula model see [75].

MAC

MAC adds an extra layer of security by adding labels to the subjects and objects. In DAC the access to an object is only based on it's UID and GID as seen in Section 3.2.1. The labels for the subjects have the form of "user : role : type : classification level (optional)". The user can be every possible account in the system. Role shows which role this user has for example system admin or marketing. A system admin will have the ability to install and delete programs. The user with role marketing will only be able to run certain programs. Type adds even more flexibility. It's possible to make a browser type that contains all browsers or make a separate type for every browser. The subject will get access to this type however this label doesn't indicate what the subject can do with it. The classification level is optional. It adds the military classification levels. These labels make it possible to have more fine grained privileges. Therefore instead of giving a particular user root access to use one resource, more rules can be added to give this user access without compromising security.
Policies are the labels for the objects which are loaded in at the start of the OS. They are written by for example the system admin. A policy has the form of "role : type : class : access rights". Role and type are the same as explained earlier. Class is the object and access rights tells the system what the subject is allowed to do with it.
When a user tries to interact with an object, first the UID and GID are checked. If this check passes then the label will be checked by using the policy list. The user will get access when he has the right role and is allowed to access objects of that type. A notification will be send to the user when a check fails. Figure 3.4 illustrates the difference between MAC and DAC.
SELinux logs all failures such that a system admin could notice malicious behaviour. There are two SELinux modes: permissive and enforcing. Permissive mode has SELinux partly turned off. Failures are logged however access is not denied based on the labels. This mode can also be seen as a debug mode which is also it's main purpose. Enforcing mode also logs all failures however it also denies access based on the labels.

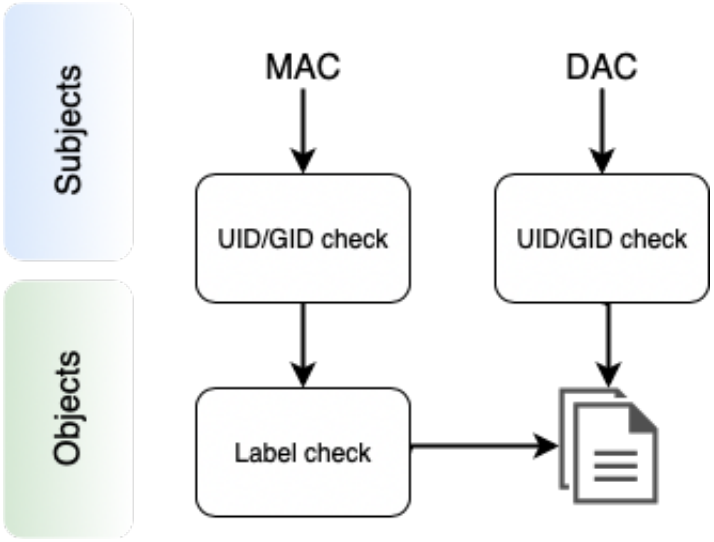

Figure 3.4: MAC vs DAC

Mac upgrades the security of DAC significantly. In DAC a user is the owner of an object and can give any one permission to use it [72]. In MAC there are restrictions for the owner of an object. He is not able to give anyone permission to a resource that he owns just because he wants it. MAC makes sure that the user has to comply to the system wide-policies to make these decisions.
Even processes are limited in MAC. A process in DAC that is running as root is able to change it's UID at will. This way the process can pose as any user. Also a process not running as root can change it's UID if he is able to authenticate himself. MAC also assigns labels to processes. These labels cannot be changed by the process. Even the root user is limited in MAC due to the labels and system-wide policies, which protects the system from malicious applications that try to get root access (see Fig 3.5).

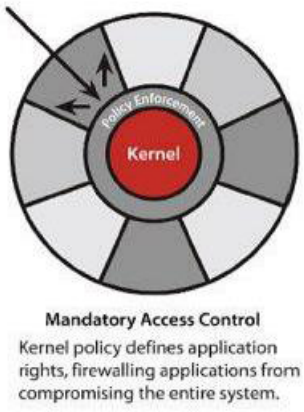

Figure 3.5: Gaining root access in MAC [4]

The data copying problem in DAC can not fully be solved with MAC. It can only limit the reach of the data by adding the classification levels. This way a user is not able to leak data to users with a lower classification level, which follows the star-property of the bell-lapadula model.

3.2.3. Permissions

Applications can only make use of the resources, services and data in their sandbox [76] [77]. After here resources will be used to address resources, services and data. However there are ways to make use of resources outside of their sandbox by asking permissions. These permissions are access rights that can be obtained by

asking the user. System applications most of the time have more permissions then user applications. System applications come directly from the OEM and therefore are trusted by the OEM, who gives them special permissions. The user is most of the time not able to decline these permissions.
User installed applications are not trusted and need to ask for permissions. These permissions are shown at install time to the user who has to accept all of them to install the application. Users could only revoke permissions by deleting the application. This changed after the release of Android 6. Users are now able to accept and decline permissions at will. The user will not be asked to accept the permissions at install time. Permissions are asked for when the application tries to make use of a resource. A window will appear asking the user to accept or decline a permission. An error message will be send to the user, whenever the application tries to make use of a resource without permission.

Manifest file

Permissions are stored in an applications manifest file [78]. This file contains all the information like the needs, configurations and components of an application. Developers will have to put all the needed permissions into this file. Android will check the file at run time and grant permissions, if they were accepted by the user.
Not all resources are available to every application. Some system resources are only for the system applications. It is also possible that resources from other applications are only available to applications of the same developer. Certain parts of the system don't have API's to directly access them for security reasons, like the sim card.
Developers are able to make their own permissions based on their application. This way they can give other applications access to their application resources. These custom permissions also need to be stored in the manifest file.

Package manager

The package manager keeps track of all the permissions [79] [80]. It maps the build in permissions to GID's which are added to the application. Applications can have multiple GID's. The custom permissions can only be used if the application that grants these permissions is installed before the applications that uses them. Every time that the application tries to use a resource it has to ask the package manager again for permission. Permissions are part of a group. If one permission in the group is granted then all the permissions are granted. For example an application could ask permission to read the contacts. This permission is part of the contacts group, therefore granting the permission to read also grants the permission to write contacts.

Types

There are four types of permissions: normal, dangerous, signed and signed or system. Normal permissions do not directly impact the users privacy and system operations. These permissions will be granted by Android without consulting the user. It's possible that an application uses permissions that in a new Android version, doesn't have the normal type anymore. In this case Android will still grant the permission considering that the application was developed for an older Android version. Otherwise there is a risk that many applications will stop working. Dangerous permissions can impact the users privacy or system operations. Therefore the application has to ask permission first to for example use the camera. Signed permissions are only available to applications signed with the same key. This most of the time means that they come from the same developer. Every developer must sign his application before it can be installed by the users. This is a rule Google came up with to establish trust between users, developers and the Google Play Store [81]. The last type is signed or system. Permissions that belong to this type can be used by system applications or applications signed with the same key.

3.2.4. IPC

IPC stands for inter process communication, which is a way for applications to make use of processes and data outside of their own sandbox. Android doesn't have all of the IPC possibilities that the Linux kernel has. A special IPC was developed for Android based on OpenBinder which was started by Be Inc. and later finished by Palm Inc. [82]. The OpenBinder version for Android has fully rewritten user-space and driver code to fulfill the needs for Android. Figure 3.6 shows the IPC framework for Android, its parts will be elaborated on below.

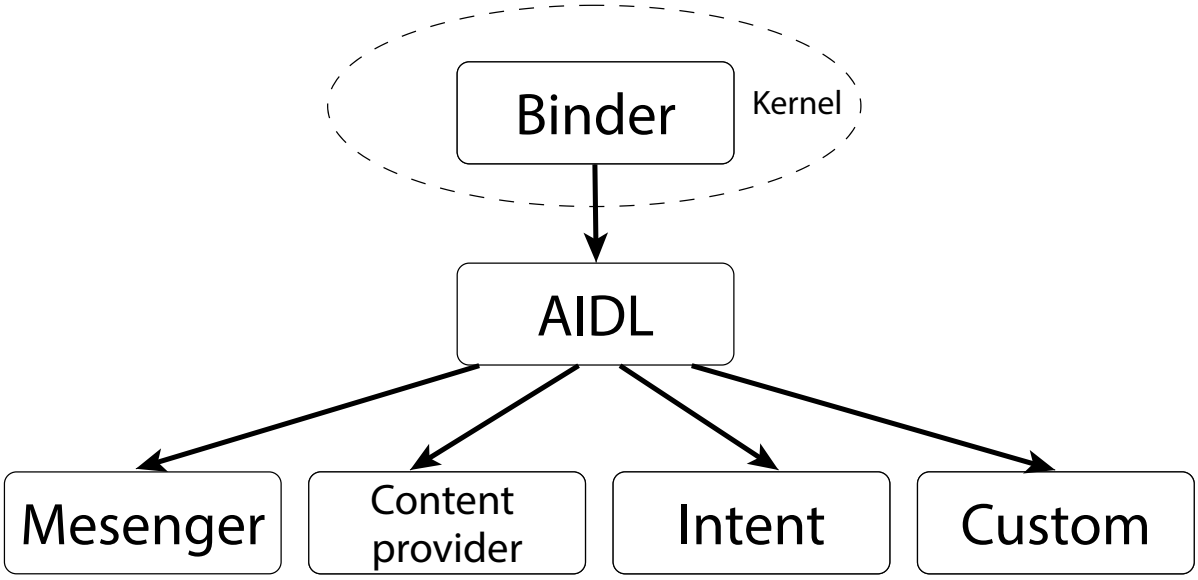

Figure 3.6: Inter Process Communication

Binder

Binder is the IPC driver that resides inside the kernel [76] [83] [84]. All the higher level API's have binder as there base. An application makes a call to binder that it wants to communicate with another application. Applications don't talk to each other directly due to security reasons which will be clear soon. Binder is in control of a part of the memory of every process and is the only one who can write there. The application is only able to read from that part of the memory. The message from the caller is written inside the memory of the callee by binder. Binder adds the effective UID of the application and the process ID (PID) to this message. The callee has to check these values by using internal-logic and system wide information. Based on this check the callee decides to accept or decline this message. When the callee is done with the message it signals binder.
Binder protects against privilege escalation, which is gaining higher level access without permission, by adding the UID and PID in the kernel. This means that applications don't have the ability to spoof these values and gain more rights then they have.

AIDL

The Android Interface Description Language (AIDL) is as the name describes a way to define your own interface between a client and a service [85] [86]. It decomposes objects into primitives such that the underlying code can understand it. Android already has three implementations: intents, messengers and content providers. However in the case that an application needs to be able to communicate concurrently with multiple applications and therefore needs multi-threading. Than AIDL needs to be used instead of the already existing interfaces.

Higher-level API

There are three higher level API's available for IPC as can be seen in Figure 3.6. The fourth one says custom because developers are able to make there own IPC using AIDL.
Messenger is a two-way interface for sending data. Both processes are able to receive and send data to each other [87] [88]. Content provider is an interface to use the data from application A in the process of application B [89].
Intents are the mostly used API. It's able to carry data and commands that will be executed by the callee. An intent has four main inputs and is able to carry extra key value pairs and flags [90]. First is the component name to which the intent needs to be sent. A component is a part of an application and specifying its name is optional as will be seen later. Second is the action that the component has to perform. Third is the data that the component has to use for this action. Fourth is the category to which the intent needs to be sent. The

category contains extra information to identify the right component.
Intents can be used to overcome the complexity of using resources directly. You don't need to ask permissions for using a resource and don't need to build an activity to use it. An intent will be used to ask an application that has permission, to use the resource. The result will be returned to the caller application.
There are two kinds of intents: explicit and implicit intents. Explicit intents have a component name. This means that the developer knows before hand which application needs to perform this action. Implicit intents don't have a component name. This means that Android has to search for an application, which is capable of performing this action based on the category. Developers need to add special intent fields to there applications manifest file, to let Android know which intents it is able to handle. The user is asked to choose which application to use if more than one capable application is found. Android immediately sends the intent to an application if it is the only capable application.
Implicit intents pose a security risk and should not be used, considering that a malicious application could get the intent and miss use it.

3.2.5. Encryption

Phones contain a lot of privacy sensitive data. Losing a phone can be very risky for an users privacy. Encryption is a way to protect data from unauthorized users. Only the user with the key is able to decrypt the data. Encrypted data looks like garbage to an user without the key. Android has support for multiple encryption schemes.
Android started file system encryption with Android 3.0 [91]. Android 5.0 added full-disk encryption of the user data partition by using one key. Android 7.0 added file based encryption, which added the possibility to encrypt multiple files with different keys. This way only a part of the data is compromised when a key is compromised.

TEE

The Trusted Execution Environment (TEE) is a separate part of the phone [92] [93] [94]. It uses a separate OS that runs along side the main OS. The TEE OS can run on a separate micro chip or share the CPU with the main OS. The TEE has more access rights than the main OS. Even after all the security features the main OS is still not trusted. The TEE OS is not open to the public depending on the OEM and build with the focus purely on security, it is able to use protected parts of the hardware like: fuses, certain parts of the ram and memory. The TEE is used for all security sensitive operations like: mobile payment, cryptography operations, key storage, data encryption etc. All the components of the TEE share a secret key to authenticate each others messages. This key is the HMAC key and is randomly derived at every system reboot.
There are multiple different TEE OS's possible. It depends on the OEM which one it chooses: Samsung has his own version called Samsung Knox, ARM has Trustzone and Android originally has trusty OS, however this OS can be replaced by any TEE OS the OEM wants.

Keystore

The keystore is a place in memory where keys are stored encrypted. Android 1.6 introduced the key store for VPN and WIFI keys [95] [96]. Later on Android 4.0 added the capability to store arbitrary keys. Some phones even have a separate secure storage chip only for keys. The key (master key) for performing this encryption is based on a user chosen password, which will be seen in Section 3.2.5. Next to the user chosen password also a key that is unique to the phones hardware (hardware backed key) will be used if available.
Applications can store their keys in two ways. The first way is by making use of the so called keychain. The keys in the keychain are system-wide available. However applications might want to keep their keys private, in that case they can make use of the keystore which is private to every application.
The key store prevents the extraction of keys from the device. Applications never use the keys in their own process. An application has to make use of another process or hardware when it wants to do cryptography operations with the keys. An example is using the TEE as seen in Section 3.2.5. Keymaster is the module that is running in the TEE that will do all the cryptography operations without leaking the keys to Android. This way an attacker is never able to steal the keys from a device even when he compromises an application. However he will be able to use the keys of the compromised application. Android 6.0 protects against this scenario by letting applications define who can use their keys. Keys can only be used after the user has validated him self to the application. It's also possible to make the keys available for only a certain time after validation.

Lockscreen

The lock screen is the first screen seen after boot [5]. A user that hasn't setup a lock screen password will just have a slider. However most users use a password, pin, pattern, fingerprint or facial recognition to unlock their phone. Password is the strongest protection due to having the highest entropy. Pin is an at least 4 digit code which is the second best option. Third is a pattern code which consists of multiple dots that are connected in a particular pattern. This pattern represents a byte sequence that will be hashed and compared to the saved hash on the device. Also the password and pin are saved as a hash on the device.
The pattern is seen as the most weak option of this three. The user draws this pattern on the screen using his finger which leaves a trace on the screen. This trace can be seen under the right light conditions or with special camera's.
Android also has the capability to use bio-metric authentication by using a camera for facial recognition or a fingerprint sensor. Using facial recognition is not advised considering that someone who looks like the user might be able to unlock the phone. On the other hand the fingerprint sensor is seen as a very secure option.
All these lock screen unlock options can be used to obtain a master key that will unlock the keys in the keystore. Gatekeeper is the service that enrolls and verifies the password, pin or pattern. After here we will just mention password instead of all three. During the enrollment of a new password, a 64 bit secure ID (SID) is created. This SID is derived by using a pseudo random number generator. The password and SID are cryptographically bound to each other. All the keys that will be made after obtaining the SID will be bounded to it. Therefore all the bounded keys will be lost without the SID. To enroll a new password the user first has to submit his old password, considering that only then the SID can be transferred to the new password. This protects the device against an attacker that has the ability to enroll a new password. Without the old password all the previously derived keys will be lost. This keeps the user's data secure after a compromise.

The Gatekeeper service consists out of three main components: gatekeeper daemon (gatekeeperd), gatekeeper HAL and gatekeeper TEE. Gatekeeperd contains the platform independent logic. Gatekeeper HAL is an interface between the hardware and gatekeeperd, which has to be implemented by the OEM. Gatekeeper TEE is the counterpart of gatekeeperd inside the TEE. It performs the authentication of the device password, pin or pattern. Gatekeeper TEE is also platform dependent and has to be implemented by the OEM. Figure 3.7 gives an overview of the below mentioned steps needed to authenticate the provided password.

1. The input from the authentication method is send to gatekeeperd. In case of a biometric input it will be send to the right biometric deamon. Biometric inputs are out the scope of this thesis.

2. Gatekeeperd will hash the input and send it to gatekeeper TEE. Inside the TEE the provided password hash will be verified. On success it will send back a signed authentication token. The authentication token mainly contains the SID, a timestamp and a HMAC.

3. The authentication token will be send to the keystore service by gatekeeperd.

4. The keystore service will send the authentication token to keymaster in the TEE. Keymaster will verify the authentication token by using the pre-shared HMAC key. Based on this verification keymaster will release the needed keys.

Gatekeeper will also monitor the amount of failed unlock attempts. The user will be blocked from trying to unlock the phone for a particular time, if there are too many failures. The time that the user has to wait can range from a couple of seconds to days depending on the amount of failures. This protects the lockscreen against brute force attacks. It's also possible that the user has forgotten his unlock code. In that case the user can make use of a back-up code, which was registered to the device during the time of registering the unlock code or login with a google account linked to the phone. If this both doesn't work then the only option is to factory reset the device.

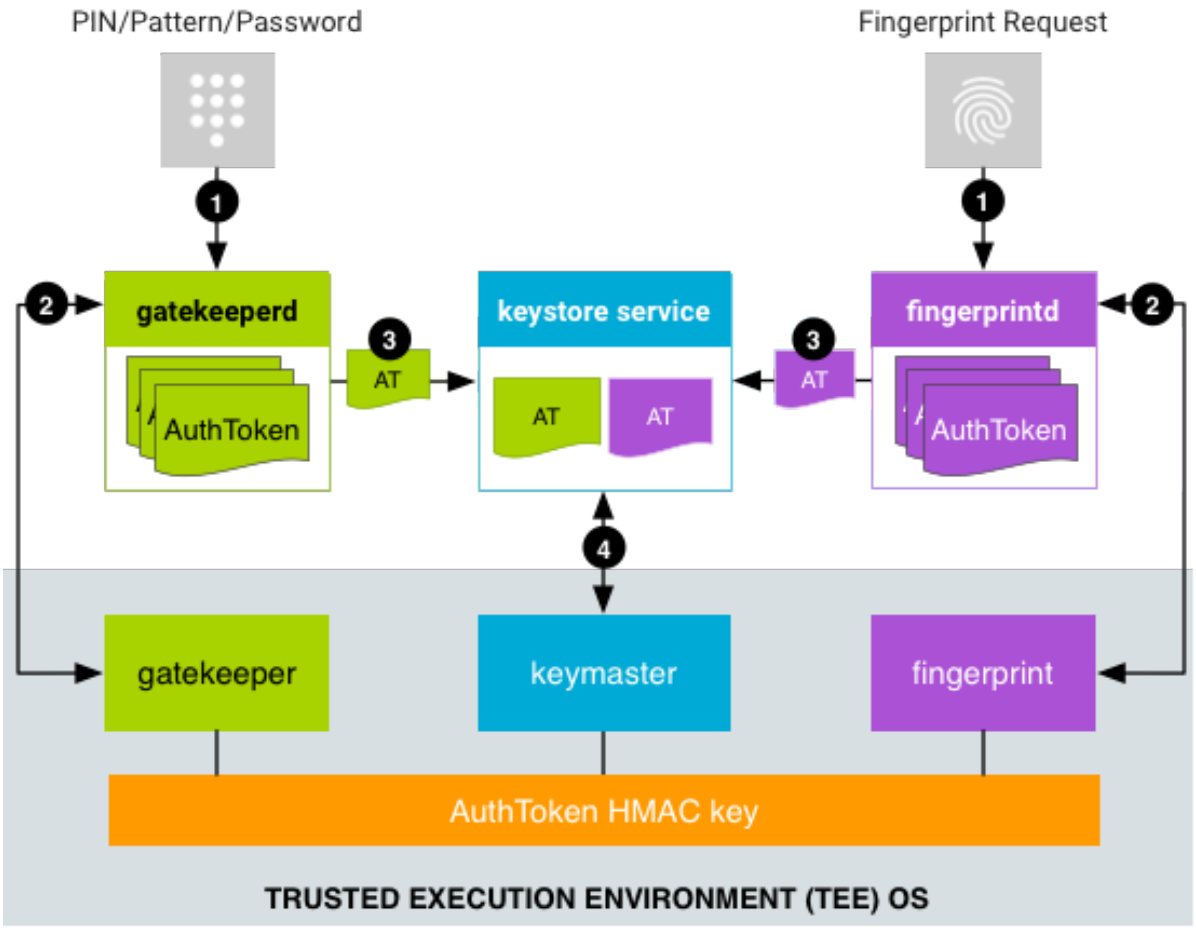

Figure 3.7: Authentication Flow [5]

3.3. Hardware Architecture

Removed due to confidentiality

4

Attack Implementation

This chapter explains the whole process of attacking the target. Section 4.1 gives a short overview of the project plan. Section 4.2 describes the process of choosing a suitable FI method. Section 4.3 performs a characterization of the target to verify its vulnerability to EMFI. The results of the characterization are used in Section 4.4 to create a FI model. Section 4.5 uses the FI model to perform a characterization while running the Android OS, to determine if Android has any defenses against EMFI. Section 4.6 performs a simulation of the EMFI attack on the lock screen. The simulation is used to determine in a short period of time, if the lock screen could be attacked by using FI. After that the actual real life attack is performed in Section 4.7. As last Section 4.8 discuses the obtained results.

4.1. Plan

This thesis focuses on analyzing the security of smartphones against fault injection attacks. The Android lockscreen will be attacked to achieve an infinite amount of unlock attempts, which will result in a possible brute force attack that takes 10^x tries for a 'x' number pin code. Most people use a four digits pin code, which are only 10.000 possible combinations. The popularity of the four digits pin code comes from the fact that people are used to four digits pin codes for unlocking their sim-card or using their banking card.
Previously, fault injection has been used to bypass pin code checks by for example: glitching the branch instruction that checks if the pin code was correct. This attack mainly rests on the way the programmer defined the branch structure. If the pin code true case comes before the false case, then simply said only the branch instruction has to be skipped to unlock the device. The Android lock screen will probably not be so easy to glitch, considering the much higher complexity of the code used to authenticate the user, as seen in Section 3.2.5.
The table below gives a short overview of the plan for this project. The steps will be elaborated on in the next sections.

1	Choose FI method
2	Target characterization
3	Define FI model
4	Android characterization
5	Simulate lockscreen attack
6	Attack lockscreen

Table 4.1: Project Plan

4.2. FI Method

First a fault injection method had to be chosen which suits the characteristics of the target. Voltage glitching is the easiest and probably the cheapest option for an attacker. However, the full documentation of the board was not available. Consequently the power lines had to be traced by reverse engineering the board. This is a time consuming task due to the many capacitors at the back of the board. Even after finding the power lines

it would still be a risky task to remove the capacitors, due to their placement close together and very small size. These boards are very expensive therefore we decided to look further.

Laser and BBI were also two possible options however both needed a de-caped chip. De-capping the chip was not possible at the front side due to the PoP configuration and also not from the backside considering the many capacitors placed there. Furthermore a laser station is very expensive and not open to the public, which limits the number of people that can use it. The more people that are able to use a certain FI method the more dangerous it is.

In this thesis we want to use a FI method that is open to the public. EMFI is an attack method that is becoming open to the public due to devices like the Chipshouter as seen in Section 1.1. However, there are at least two difficulties for using this method on the target. The first is that the attack target is a PoP chip which has the CPU at the bottom. The DRAM on top could block the EM waves from reaching the CPU like an EM shield [64]. The second difficulty is that the chip is running on 1,8 GHz for the small cores and 2,4 GHz for the large cores. Based on the explanation of the ARM big-little architecture, it could be said that it's highly likely that the 1.8 GHz cores, are most of the time the only active cores. 1.8 GHz translates to a clock with a period of 1.8ns. For simplicity it can be said that an instruction takes up one period. The EMFI device creates an EM field for 20ns. This roughly translates to 11 instructions having passed during the firing of the EMFI device. This could mean that the EMFI device doesn't have enough time to effect an instruction. Nonetheless, EMFI became the preferred choice considering it wouldn't need any modifications of the board and it will be open to the public soon.

4.3. Target Characterization

Performing Fault injection on a new chip on which as far as we know no one tried it before, is a very risky thesis subject. It's not possible to know in advance if a chip is vulnerable for fault injection. Therefore, the first step in glitching a device is to perform a target characterization. This step will determine if the device is vulnerable for fault injection attacks. A new chip will have to be chosen, if it turns out that the chip is not vulnerable. Multiple chips which had easier characteristics were already selected as potential backups. This risky chip was chosen due to the much greater reward that would be obtained if it was possible to glitch it. The project was not only focused on glitching the Android lockscreen anymore. The first objective was to see if it was even possible to glitch a PoP chip. Nothing could be found in the literature about glitching chips with such a configuration, without removing the upper chip.

4.3.1. Setup

To perform the characterization an EMFI setup had to be build. Riscure [24] provided everything that was needed for the project. Figure 4.1 shows an abstract representation of the characterization setup. All parts of the setup will be elaborated on below. The used setup can be seen in Figure 4.2. Through out this thesis board,target and chip will be used interchangeable to point to the same thing.

FIPY

FIPY is a SDK used in-house to control a fault injection setup using python. During the project FIPY was still under development. Starting to work with FIPY was a bit of a hard task considering that there was no documentation. The only available information was example code from someone who used FIPY to do voltage glitching. FIPY was also used to control the target instead of using the spider. This way the setup could be tailored more to our needs. The tasks of the spider will be explained in subsection Spider.
FIPY had the capability to reset the target in two ways: soft reset and hard reset. The soft reset was done by pulling one of the pins on the target low. The hard reset would command the lab power supply to cut the power to the target. This was needed because the soft reset could sometimes get stuck in a state that it wasn't able to reset the target anymore.
The code running on the target was made in such a way that it would output its results over an uart. This uart was read out, parsed and saved by FIPY. The output would get different colors depending on its contents. Green meant that the target was working normally. Yellow is a mute which means that the target didn't send anything over. Red means a successful glitch. Multiple different colors can be added. FIPY knows what color to give to each output by parsing the output and comparing them with custom rules written in python. The different outputs will be discussed in Section 4.3.4.

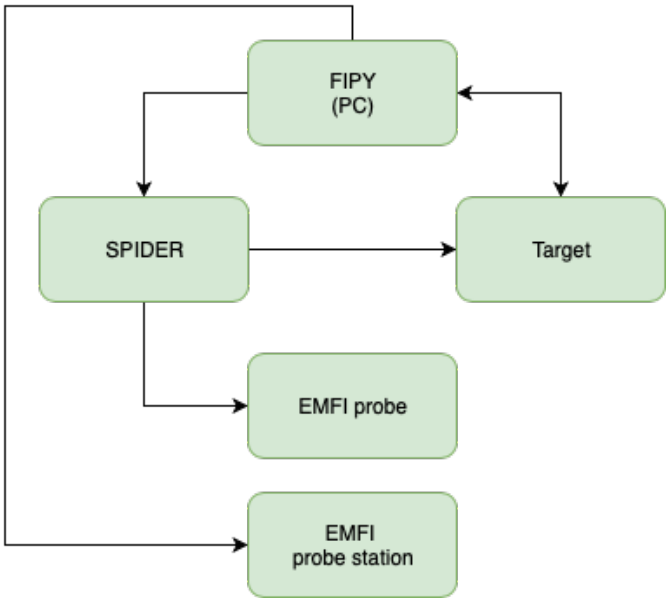

Figure 4.1: Abstract Characterization Setup

FIPY saves all the experiment data in a SQL database for later analysis. This data is not only limited to the uart output, every wanted parameter can be added to the database. Pewpew plot which is another in-house tool was used for the data analysis. This program has the capability to make figures based on the captured data and helps understand what has happened, by using its post processing capabilities.

Spider

The spider is at the core of the fault injection setup (Figure 4.3). It's able to control the FI device and target. The spider has a lot of inputs and outputs but not all will be used or mentioned in this section. The spider controls the fault injection method by generating a timed digital glitch. This digital glitch signals the FI device (EMFI probe) to attack the target device. Two FPGA cores inside the spider are trusted with the task to react on a trigger after max 4ns. This trigger is a signal from the embedded device that tells the spider, that it should send out the digital glitch. The timing of this digital glitch can be delayed with a user chosen amount of time. Programming these FPGA's is done by using an interface inside of FIPY.

EMFI Probe/Station

The EMFI probe is simply said, a bank of capacitors that can be charged to a specific value and will discharge on command. In Figure 4.4 the EMFI probe can be seen, which has three inputs: voltage adapter(24 DC), digital glitch and pulse amplitude. There are also two outputs: coil current and probe tip. The EMFI probe will fire (discharge) when it receives a signal on the digital glitch input. The strength of the produced EM field is based on the voltage in the capacitors. This voltage is controlled by the pulse amplitude, which comes from the spider's voltage out and has a range between 0V and 3,3V. Internally this value will be mapped into a range between 24V and 400V.
The coil current output is used to make the glitch visible on the oscilloscope. Figure 4.5 shows the glitch measured on the oscilloscope for 60% power. The height of the peak depends on the pulse amplitude value. A higher voltage will give a higher peak in the negative direction. The pulse width has a fixed value of approximately 20ns.
The EMFI probe is attached to the EMFI probe station, which will be called just station after this, as can be seen in Figure 4.6. The station has the capability to move the EMFI probe in the "xyz" directions. Moving the EMFI probe over the chip and glitching is called scanning the chip. FIPY controls the movement of the station via a USB connection.

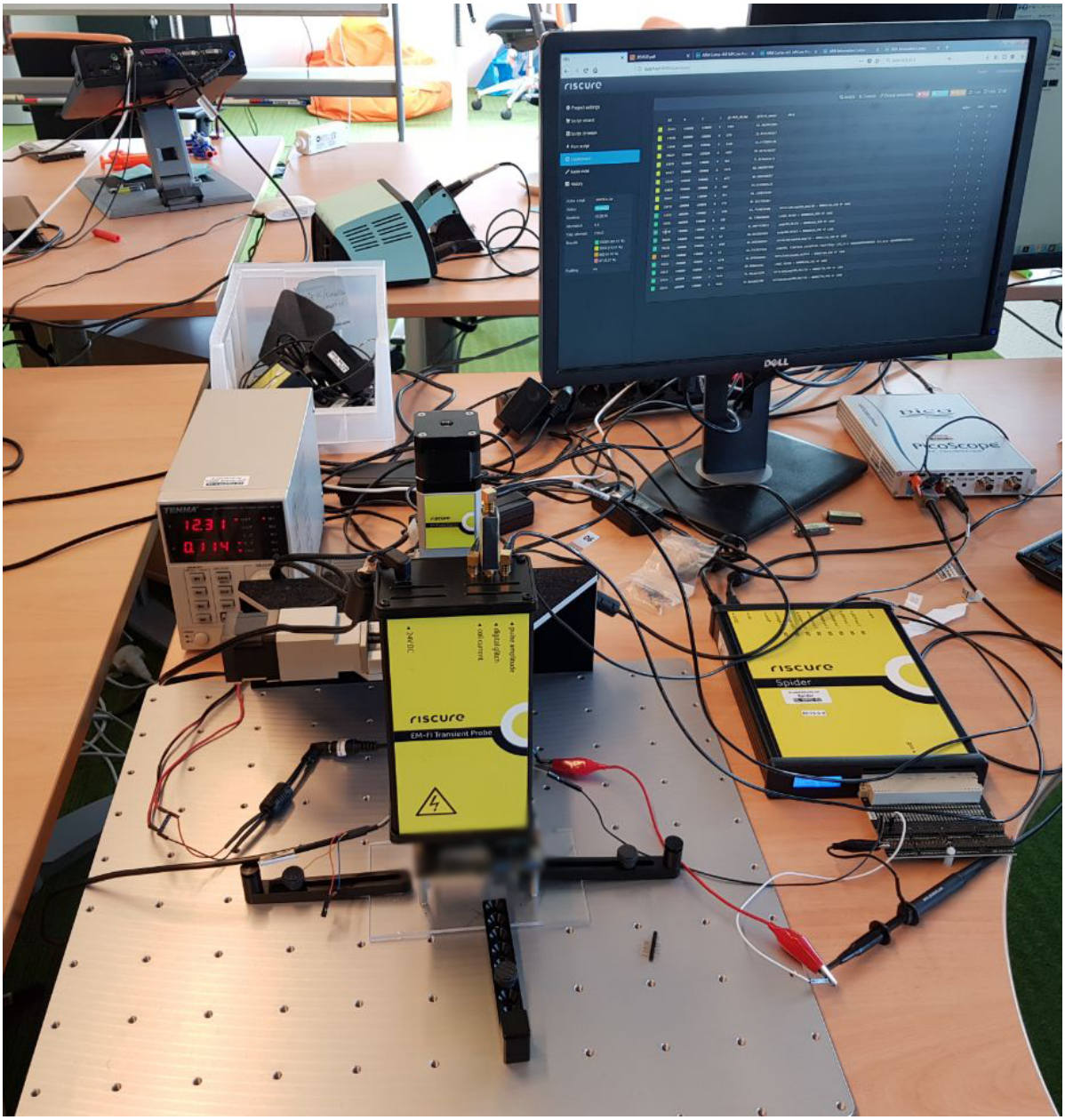

Figure 4.2: Characterization Setup

Probe Tips

There are two groups of probe tips (coils) with different sizes. These two groups differ in the direction of the magnetic field that they produce. Each group contains a 1,5mm and a 4mm tip. The smaller the tip the more precise a magnetic field can be induced. However, a smaller tip also means that the time to scan a chip increases. In the case of these 2 tips the time increase would roughly be 2,5. This makes the choice of choosing the 1,5mm tip not possible for this project considering that it would take to much time to scan the chip.

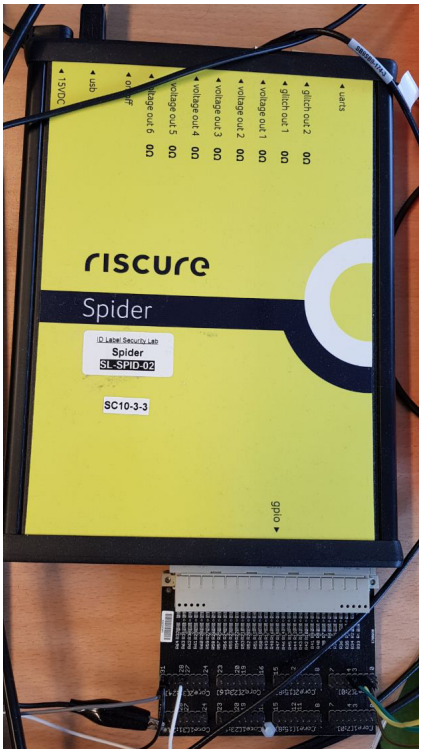

Figure 4.3: Spider

4.3.2. Setup Problems

Multiple problems were encountered while building the setup. Two of these problems will be explained here.

EMFI probe not firing

The first scans of the chip only resulted in green outputs for every power level. It was a possible result however, it was still strange that there was no reaction even for 100% power. Eventually it was discovered that the first EMFI probe broke. Apparently there was an adapter needed on the digital glitch input, which was not documented. Without this adapter the EMFI probe would receive a signal from the spider that was so strong that it would break the device. Luckily these devices are made in-house therefore it could still be repaired.

Random Voltage

During trial runs, the chip would be scanned with random voltages for different attempts. The idea was that the characterization would take less time without the need of having to do a full scan for every power level. Strangely, these scans resulted in only green or yellow outputs. Inspecting the coil current on the oscilloscope showed strange behavior. The voltage of the glitch would stay at the same level every time the EMFI probe fired, even though it was changing in software. More tests showed that using a fixed voltage for multiple attempts did change the voltage of the glitch. This was reported to the tools department. They asked us to investigate why this was happening.
A modified EMFI probe was prepared such that it was possible to measure the capacitor voltage. The test setup was copied therefore nothing had to be modified. The tests showed that the pulse amplitude and capacitor voltage were out of sync. This means that when the pulse amplitude changes, the capacitor voltage will not follow. This is possible due to a stage between the pulse amplitude signal and the capacitor voltage that translates the pulse amplitude to the range of 24V to 400V. This stage doesn't react fast enough on the voltage value change in FIPY. A quick fix would be to add a delay in the software, however this is not desired. This delay will accumulate fast over time considering that it is needed after every power change. The only real fix will be a hardware revision.

Figure 4.4: EMFI Probe

4.3.3. Running Code

To perform the characterization it's essential to run our own code on the board. This code will be specifically tailored to our needs. Preferably the code would run as soon as the board powers on as waiting for an OS to boot can take a long time. Unfortunately, there is no documented way to run code on this board without running Linux or Android.

Boot Stages

The board will be rebooted a lot during the FI attack. Waiting for Linux or Android to boot would take too much time at this stage in the project. The development board runs a first stage bootloader after power on. The ARM trusted firmware (ATF) will be executed second. The ATF contains multiple bootloader stages (BL1, BL2, BL3-1, BL3-2 and BL3-3). These stages mostly prepare the system environment such that the next stage can be executed. BL3-3 loads UEFI which boots Linux or Android. A way to run the code at an earlier boot stage was needed.

EFI Application

The first idea was to make an EFI application that could be executed by the UEFI bootloader. The problem was that there was not much information about how to make an EFI application. There is official documentation for UEFI however it was to extensive to go through and none of the examples really did what was needed. Every online search for an EFI application tutorial or example would end up pointing to the same website, which was "Programming for EFI by Roderick W.smith". It was a very short hello world example, nonetheless it gave enough insight into how to get some code running using UEFI. The EFI application that was made was an unrolled loop, which was loaded from an SD card placed into the board. Programming an EFI application is different from just normal C code, which made it hard to use the peripherals of the board.

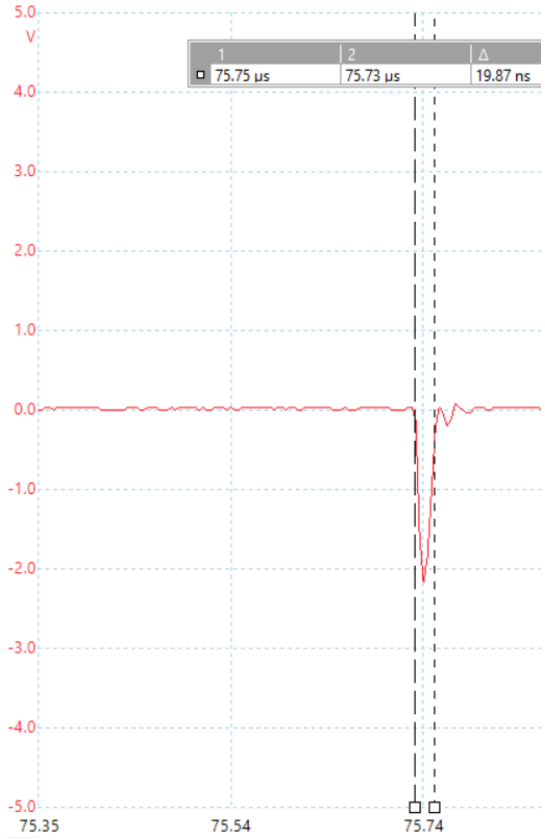

Figure 4.5: Glitch

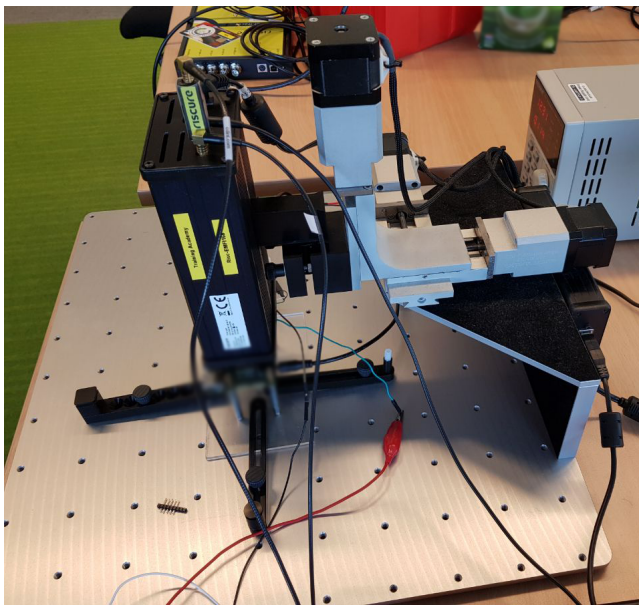

Figure 4.6: EMFI Probe Station

Modified BL1

The second idea came forth out of the first one. The UEFI bootloader had to be compiled before it could be ran on the board. It was discovered that the ATF code was actually also compiled during this process, which meant that we had access to the C code. Modifying BL1 which was part of the ATF was easier than making an

Figure 4.7: Probe tips: 4mm (red) 1.5mm (black)[6]

EFI application. BL1 was just normal C code which made it easier to use the peripherals. The second benefit was that it would take less time for the board to actually start executing our code, due to the code running in the first bootloader. BL1 is stored on the eMMC which also removed the need for an SD card.

4.3.4. Test Code

Multiple different pieces of code were needed to perform the characterization. Every piece of code is specifically tailored to give insight into what is happening during a glitch. They all have a common part which sets up the environment and a specific core part. Both parts will be elaborated on below. Most of these parts are written in ARMv8 assembly (ASM) given that, it gives more control and the compiler will not try to optimize the code. Those optimization's could break the purpose of the code. Appendix A shows a sample of the test codes.

Common Part: Peripherals

A so called trigger signal is needed as read in Section 4.3.1, which is generated by the test code. First a GPIO pin on the board is initialized as an output pin. Just before executing the core part of the test code this pin is pulled HIGH. This signals the spider that the core code is now running. At the end of the core part the pin is pulled LOW, which tells the spider that the core part has ended, therefore it should not glitch anymore.
The trigger is used to narrow down the search space and increase the success rate of the attack. Without the trigger we would have to try to guess where the core code starts and ends. This means that the search space is from the power on of the board until some sort of signal is received that the core code has ended. In a real-life situation, the trigger most often has to be improvised. Take for example, a device in which the user would have to input a pin code. The device could give a sound signal to tell the user that he can now give an input. After the pin code check it gives a different sound depending on if the password was right or wrong. This sound signal could be used as the trigger.

The UART is used to get the output results from the board. The problem with something like the more complex USB protocol is that during a glitch all parts of the chip can be influenced. Therefore a faulty output could also just be a glitch in for example the USB part of the chip. At the outside it would look like a correct glitch even though the execution of the CPU was never influenced. Using the UART doesn't guarantee that this won't happen however it reduces the chance, considering that it's a much simpler protocol that uses less hardware.

Common Part: Registers

Before manipulating the registers, first a back-up has to be made somewhere in memory. After the core code has ran this backup will be used to give all the registers their original values back. Without this backup the board will crash when it tries to follow its normal execution flow after the core code.

The hardware attack survey explained that the instruction operands can change due to fault injection. During the characterization we would like to know if an operand in the instruction changed. This is made easier by giving all the registers a known initial value, except the registers that are used by the core code. This initial value must be high enough such that it's very unlikely that it appeared just by chance. Second, this value has to be easily recognizable. Our choice went to 0xCAFEBABE (hexadecimal value). 0xCAFEBABE or something similar in the output shows that the operand has changed into a different register.
This gives us the ability to know that the operand changed into one of the registers. However it could also be possible that some registers are more often swapped for the operands than others. The 0xCAFEBABE value has to be adjusted to get more insight into the behavior of particular registers. This value must still adhere

to the previously given requirements however it must also be unique for every register. There are 31 registers of which register 0 will be used for calculations. Register 1 up to 28 will get the value 0xCXXEBABE. "XX" will be replaced with "A1" for registers 1 , "A2" for register 2 up to "BC" for register 28. Now its in most of the cases possible to trace back the output to a particular register. Registers 29 and 30 have a special purpose, the board tends to crash sometimes when these registers were given an initial value, therefore these registers are left out.

Core Part: Unrolled loop in asm

The first test will be done with an unrolled loop in assembly. An unrolled loop is used given that it gives us a big window to glitch anywhere in the code. This piece of code doesn't write its value back to memory, which is done on purpose. This is to limit the number of variables, that makes it easier to make an educated guess of what has happened during the glitch.
The core code can be seen below in Listing 4.1. Register 0 is first set to the value zero. Al the other 28 registers are already initialized to their unique 0xCXXEBABE value. The second step is incrementing the X0 register with one. This instruction will be repeated 10.000 times without jumping back, which is why it's called an unrolled loop. After the core code has ran the value inside the X0 register will be printed over the UART. The output should contain the value 0x2710. Any deviation from this value could be a successful glitch.
A whole string is printed with more information as can be seen in Figure 4.8. This is to minimize the chance that this result is due to a glitch in the UART. The chance that only the value 0x2710 changed in the whole string due to a glitch in the UART is highly unlikely.

```
1 MOV X0 , #0
2 ADD X0 , X0, #1
3     : 10.000 ADD instructions
4 ADD X0 , X0, #1
```

Listing 4.1: Unrolled Loop ASM

```
UnrolledLoopASM,0x2710 = 00002710,END OF CODE..
```

Figure 4.8: Unrolled loop in asm normal output

Core Part: Loop in asm

The goal of the second test is to glitch more complex code, which requires a more accurate timed glitch. In this test there are many different ways to end up with a different output value, based on how precise a single instruction can be glitched. Figure 4.9 shows the normal output.
The source code of the loop in assembly can be seen below in Listing 4.2. Also the difference between a normal loop and an unrolled one can be seen here. The first two lines are the same as the unrolled loop asm. The only difference is that the second line has the label "loop" in front of it. Labels are used to jump around in the code as will be seen later. On the third line X0 is compared (CMP) to the value one, left shifted with 12 zeros which is 4096. 4096 is chosen here because its the biggest value that could be made without having to use more instructions. A big value gives us a bigger window to glitch. It's preferable to keep the number of instructions as small as possible. The less instructions there are the easier it will be to back trace what has happened. The CMP instruction will set a flag if the values are equal. On the fourth line the branch if not equal (BNE) instruction will check this flag. If the value inside the X0 register was not equal to 4096 then the code will jump back to the label "loop" on line two.
It's hard to say exactly how this code can be glitched to output a different value than 4096, due to multiple different possible combinations of events. However some examples can be given of how this code could be glitched. The most obvious way is to skip line four (BNE), without this instruction the code will never jump back to the label "loop". Another way could be by glitching the comparison value in line three to set the flag when X0 is bigger or smaller than 4096.

```
1         MOV x0, #0
2 loop:   ADD x0, x0, #1
3         CMP x0, #1, lsl #12 (compares X0 to 4096)
4         BNE loop
```

Listing 4.2: Loop ASM

```
LoopASM,0x1000 = 00001000,END OF CODE..
```

Figure 4.9: Loop in asm normal output

Core Part: Loop in C

In the third test a for loop written in c code will be glitched. This for loop will increment a variable "res" as can be seen in Listing 4.3. Figure 4.10 shows the normal output. The compiler compiles this c code to the assembly code seen in Listing 4.4. On line one "res" is read from memory into the lower half of register X1. The iterator variable "i" is stored in register X0. Line two shows the instruction that will decrement the value in register X0 until it reaches the value zero. Therefore, "i" is actually not incremented from 0 to 1000 but it's decremented from 1000 to 0. On line three “res” is incremented with one and on line four "res" is written back to memory. Line five checks if the SUBS instruction of line two did set the zero flag. The BNE instruction will jump back to line one if this is not the case.
The assembly code shows that the variable "res", is every iteration read from and written back to the memory. The goal of this test is to determine if the glitches will have any effect on reading or writing the memory. In the previous tests there were no interactions with the memory, only the registers were used. Especially register 0 was often used, using a different register could give us different results. However due to timing constraints we will not be able to test multiple combinations of registers.

```
1 For(i = 0; I < 1000; i++){
2     res++
3 };
```

Listing 4.3: Loop in C

```
LoopC,0x3E8 = 000003e8,END OF CODE..
```

Figure 4.10: Loop in C normal output

```
1 Begin   LDR  w1, [x29 , 0x3c]
2         SUBS x0, x0, #1
3         ADD  x1, x1, #1
4         STR  x1, [x29 , 0x3c]
5         BNE  Begin
```

Listing 4.4: Loop in C compiled

4.3.5. Results

After many trail scans and fine tuning the setup, the characterization of the chip could finally be performed. The front side of the chip was approximately divided in a 30 by 30 grid. The EMFI probe would traverse this grid with a step size of one, using the 4mm red tip. Five glitch attempts would be performed on every spot. An attempt is one run of the test code and a glitch (firing of the EMFI probe). The core test code was chosen randomly by FIPY. All tests codes were adjusted to roughly have the same run time of 6000ns. The glitch delay was set to a random value between 2000ns and 4000ns. Glitching would start 2000ns after the trigger to let the core code first run normally.
It's not possible to use random power values as mentioned in Section 4.3.1. Instead the power is incremented with 2,5% after every scan of the whole chip. The trail scans already showed that using less than 20% power will not affect the chip. Considering this the characterization started with 20% power to reduce the amount of time needed to perform the characterization. Table 4.2 summarizes the characterization parameters.

Parameter	Value
Tip size	4mm (red)
Grid size	30 x 30
Attempts per spot	5
Glitch delay	2000ns - 4000ns
Glitch power	20%
Glitch power increase per scan	2.5%

Table 4.2: Characterization Parameters

Scan Plot Colors

The characterization results are plotted in Figure 4.11 based on the coordinates of the EMFI probe on top of the chip. It can be seen that different parts of the chip react differently to the EM waves. Green is the normal behavior of the chip. The outputs shown in Section 4.3.4 test codes will all get a green color. Yellow is a mute, which means that the board didn't respond within the given time. Most often this happens as a result of the board crashing.
Red is a successful glitch, which has to meet some requirements. First the output has to have the same length as the normal output. This way outputs that are most likely not the result of glitches in the core code can be filtered out. A longer or shorter string can be the result of a glitch in the UART or corruption of the string data. Second the printed hex value like 0x00002710 must be the only value that has changed. Third the output string has to end with "END OF CODE..". If the string doesn't end with this then the board probably crashed after our attack. The third requirement is that the board must keep functioning normally. However this doesn't mean that it should give the green output. The board needs to keep executing the code with the glitched data and or instructions.
Orange is a special case in which the board threw a platform exception. There were multiple different platform exceptions which all pointed to different faults raised by the memory management unit (MMU). The ARM manual explains that the error group 8600000xx means: "Instruction Abort taken without a change in Exception level. Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions" [7]. The second error group is 0x960000XX: "Data Abort taken without a change in Exception level. Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors" [7]. The colors and their meaning are summarized in Table 4.3.

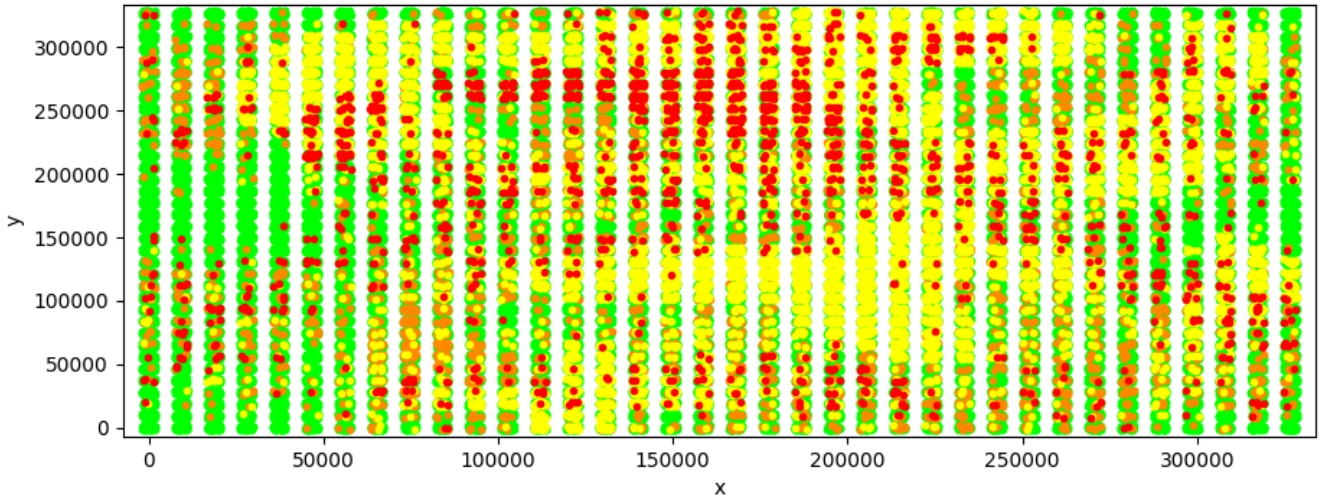

Figure 4.11: Characterization chip front side plot

Color	Meaning
Green	Normal output
Yellow	Mute
Red	Succesful glitch
Orange	Platform exception

Table 4.3: Output Colors

Power

After roughly six days the characterization was stopped by the pc due to an USB error. The characterization stopped at 85% power, which isn't a problem considering that at high powers the number of mutes is far to high to be useful, as can be seen in Figure 4.12. Our goal is to get as many successful glitches as possible, with the requirement that the chip keeps working as normal as possible. This means that the amount of mutes and platform exceptions must be reduced to a minimum. The glitch power up to 50% is plotted against the output results, except from the green output for convenience. Only the glitch power up to 50% is shown, since it can already be seen that the amount of mutes is becoming far to high. The number of successful glitches (red) and mutes (yellow) is almost the same until 27,5% power. At 32,5% power the number of mutes has already increased to double the number of successful glitches. After 32,5% the number of mutes is growing exponentially.

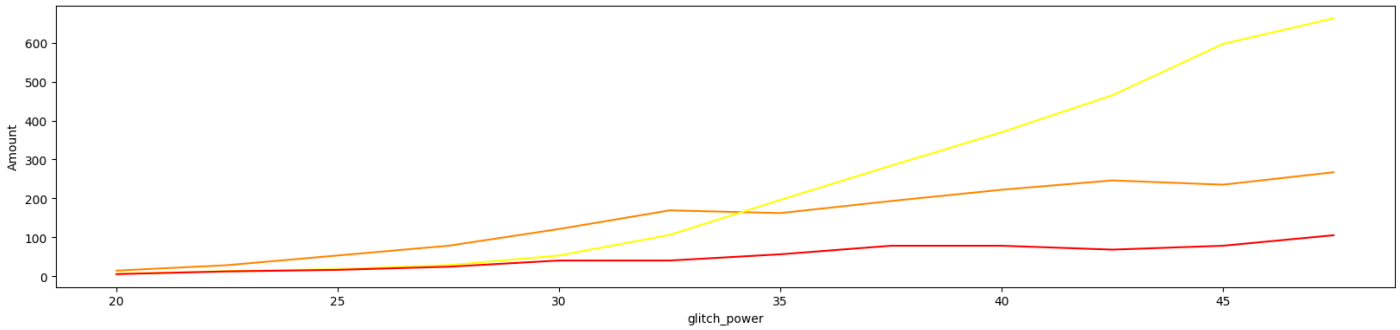

Figure 4.12: Up to 50% power plotted against the characterization results

In Figure 4.11 multiple regions with a dominant color can be distinguished. Especially the yellow and red regions are popping out. There are also regions which are mostly covered in a mix of orange and red. Figure 4.12 shows that the platform exceptions (orange) are an even worse problem than the mutes (yellow) for low powers. Fortunately, the glitch success rate can be increased by pointing the EMFI probe to mainly red regions. Figure 4.13 shows that by focusing on a small region (110K <X <130K , 240K <Y <260K), the number of successful glitches can be kept above the platform exceptions up to 67,5% power. Next to that the successful glitches even manage to get above the number of mutes up to 35% power and from 35% to 37,5% they are more or less the same. Eventually to perform the attack a fixed point will have to be chosen, which will increase the glitch success rate.

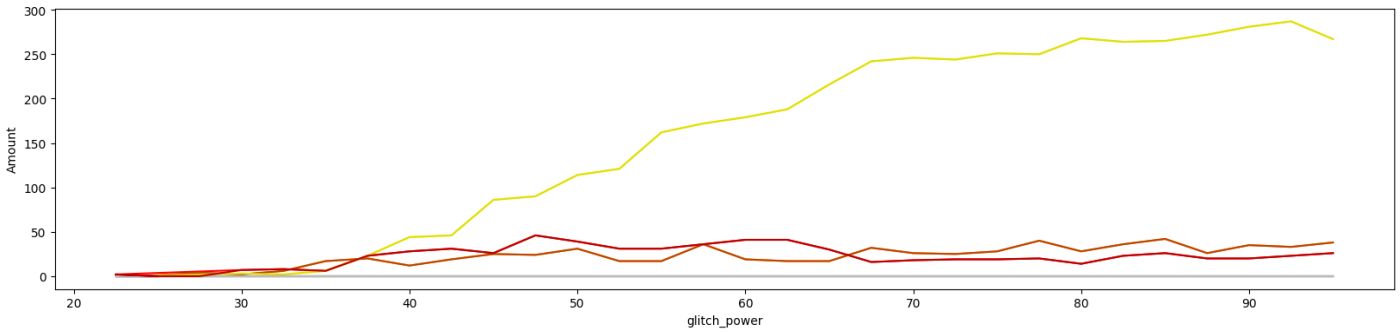

Figure 4.13: Glitch power vs output results on narrowed down region

Output Results

All three tests showed interesting glitch results with expected outputs that contained bigger or smaller values than the normal outputs. Only the loop asm and loop C have some strange outputs that contain ASCI characters. This doesn't mean that the glitches are useless depending on the goal of the attacker.
A smaller output can be the result of instruction 'skipping' or instruction corruption. We were not able (maybe even impossible) to find out which instruction was executed instead of the original one. JTAG might give some insight, however there was no equipment to use JTAG. Also it's possible that JTAG would also be glitched and therefore becomes useless. A smaller value can be the result of the ADD immediate instruction being corrupted into a SUB instruction. Knowing that the SUB instruction only differs one bit from the ADD immediate instruction. It's also possible that the destination register changed into a different register. The result of this would be that the original register is incremented one time less. All registers would need to be printed to detect this behaviour. Printing out all registers would make the code more complicated, which was not desired at this stage.
A higher value cannot be the result of instruction skipping, it's more likely that the instruction was corrupted. For example, the immediate in the instruction could have changed to a bigger value or the source operand could have changed. CXXEBABE in the output shows that the source operand changed into a different register. Table 4.4 gives a short summary of the output results, which will be elaborated on below.

Test	Output
Unrolled loop asm	Higher values Smaller values CXXEBABE Memory address
Loop asm	Higher values ASCI characters Memory address
Loop c	Smaller values ASCI characters

Table 4.4: Output Results Summary

The unrolled loop asm showed exactly the wanted results as can be seen in Figure 4.14. The output value 0x00002710 is changed into bigger and smaller values. These results were all scattered around the chip and would appear for different powers. However one region (the red square) would standout which had a 99% glitch success rate as will be seen in Section 4.4.4. Furthermore also 0xCXXEBABE ended up in the results, due to the source operand (register 0) changing into different registers. Based on the 0xCXXEBABE values these register changes can be traced back to the registers: 1, 2, 4, 8 and 24. It was already expected that especially the registers which are a power of two would appear in the output. Considering that they only need one bit flipped in the source register field of the instruction. Register 24 is an exception that needs 2 bit flips, it only appeared once in the results. More on this later in Section 4.4.1.
There were two output values that appeared most of the time. These values were 0x00000B57 and 0x00000B77. The reason for the appearance of these values is unknown. The location of the EMFI probe didn't really matter for these results, they would pop up everywhere on the chip. Also the used power didn't change the result. The only thing is that for higher powers like 50% a lot of mutes would appear instead of glitches.
The last interesting result was the appearance of memory addresses in the output for all tests. These are probably coming from the LR register (register 30) or the frame pointer (register 29), which changed into the source operand.

The loop asm was a lot harder to attack than the unrolled loop asm, the amount of successful glitches was far lower. A couple of outputs can be seen in Figure 4.15. Most often 0x00001000 was changed into 0x00009000 and a couple of times into 0x00001888. 0x00009000 was scattered around the chip and appeared 24 times. All those times it would appear when a power between 30% and 42,5% or 50% and 80% was used.
There were also smaller looking values, however the problem was that the least significant numbers were ASCI characters. They could have appeared due to the execution jumping out of the loop to soon. As a result the glitch happened in the function that prints the data, which is outside of the core code.

Amount	Output results
159	UnrolledLoopASM,0x2710 = 00000b57,END OF CODE..
148	UnrolledLoopASM,0x2710 = 00000b77,END OF CODE..
3	UnrolledLoopASM,0x2710 = 0000260a,END OF CODE..
24	UnrolledLoopASM,0x2710 = 00002708,END OF CODE..
5	UnrolledLoopASM,0x2710 = 0000270a,END OF CODE..
3	UnrolledLoopASM,0x2710 = 000028a8,END OF CODE..
7	UnrolledLoopASM,0x2710 = 00002910,END OF CODE..
3	UnrolledLoopASM,0x2710 = 1ac02ccc,END OF CODE..
13	UnrolledLoopASM,0x2710 = ca1ee023,END OF CODE..
3	UnrolledLoopASM,0x2710 = ca2ee124,END OF CODE..
7	UnrolledLoopASM,0x2710 = ca4ed0b6,END OF CODE..
2	UnrolledLoopASM,0x2710 = ca8ee150,END OF CODE..
1	UnrolledLoopASM,0x2710 = cab6fd4a,END OF CODE..
1	UnrolledLoopASM,0x2710 = ffffdc79,END OF CODE..

Figure 4.14: Unrolled loop in asm characterization outputs

Amount	Output result
8	LoopASM,0x1000 = 0000000*,END OF CODE..
16	LoopASM,0x1000 = 0000000:,END OF CODE..
34	LoopASM,0x1000 = 0000000j,END OF CODE..
8	LoopASM,0x1000 = 0000000k,END OF CODE..
3	LoopASM,0x1000 = 00001888,END OF CODE..
23	LoopASM,0x1000 = 00009000,END OF CODE..
3	LoopASM,0x1000 = 1ac107c8,END OF CODE..
1	LoopASM,0x1000 = ffffdc79,END OF CODE..
1	LoopASM,0x1000 = ffffdf75,END OF CODE..

Figure 4.15: Loop in asm characterization outputs

The loop in c only had smaller output values as can be seen in Figure 4.16. The value 0x00000007 appeared most often in the output results and was only located on the red square that was mentioned earlier. 0x00000007 would appear for different powers however the highest chance of it appearing was at 55% power. The loop in c also had ASCI values in its output just like the loop asm.

Amount	Output results
604	LoopC,0x3E8 = 000003.8,END OF CODE..
2	LoopC,0x3E8 = 0000175.,END OF CODE..
1	LoopC,0x3E< = 000003e8,END OF CODE..
5	LoopC,0x3E. = 000003e8,END OF CODE..
24	LoopC,0x3E8 = 00000007,END OF CODE..
53	LoopC,0x3E8 = 0000000?,END OF CODE..
14	LoopC,0x3E8 = 000003o8,END OF CODE..

Figure 4.16: Loop in C characterization outputs

The goal of the characterization was to find out if this chip is vulnerable for EMFI. Using this particular chip for this project was a big risk as mentioned in Section 4.2. Two main difficulties had to be overcome, which were the PoP configuration of the chip and its clock speed. The characterization showed that the best results appear for low powers up to 35%. Multiple different locations on the chip were identified which result in successful glitches. Therefore it was confirmed that this chip is definitely vulnerable for EMFI attacks.

4.4. FI Model

The characterization results showed that this chip is vulnerable for EMFI. In this section a fault injection model will be defined based on the characterization results, which will be the base for the actual attack on the Android lock screen. Here after the fault injection model will be referred to with just model.

4.4.1. What happens during a glitch?

As explained in Section 4.3.5 it's very hard to understand what happened to an instruction during a glitch. From a high level it's known that an instruction can be "skipped" or modified and data can be changed. However, there was no easy way to find out which bits in the instruction where flipped by the attack. Therefore the first question is can we find a way to get more insight into which bits are flipped. As a result of this the second question would be, is it easier to flip a bit from 0(low) to 1(high) or from 1 to 0.
To answer the first question, the MOVZ test was developed. The idea was to look for an instruction which had many zero's behind each other and an easy way to see if one of them flipped. The best candidate was the MOVZ instruction which had 21 controllable bits.
Figure 4.17 shows the MOVZ instruction encoding. This instruction will move a 16-bit immediate optionally shifted, to the provided register. The immediate and shift value are set to 0 and the register operand to register 0. All registers will get an initial value and will be printed. The output can easily be traced back to the flipped bits, by checking if the initial value of a register was changed. By analyzing the new value inside the register, it's also possible to trace back which bits were flipped inside the immediate. Unfortunately, the test will not give us insight into the bit flips in the bits 23 until 31. These bits control the type of instruction.

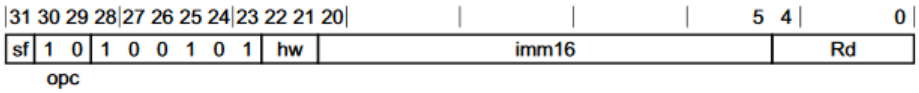

Figure 4.17: MOVZ Instruction Encoding [7]

The MOVZ test code looked like the unrolled loop in asm. Only the ADD instructions were replaced by MOVZ instructions. At the end of the test code the board would print the values inside all the registers. These outputs were stored and later processed using python.
Figure 4.18 shows the results for the MOVZ test with the immediate and shift value set to 0 and using register 0. The figure shows the chance of a register replacing the 0 register in the instruction. Register 4 and register 22 have the highest chance of replacing register 0 in the instruction.
The data about replacing register 0 was used in combination with the modified value inside the registers, to find out which bit positions in the MOVZ instruction were flipped. This resulted in Figure 4.19, from which it can be seen that the lower bits are most often glitched. The immediate value was unfortunately not so often glitched. Due to this there were no results for the higher bit positions, which can be the result of the chosen EMFI position on the chip. Bit positions 6 and 7 also had a couple of results but not enough to be seen in the figure. Different positions on the chip could result in different effects. Sadly there was not enough time to run more tests. Also if the tests were ran longer they would have resulted in more different results.
Question 2 can be answered by changing the immediate and the register operand to a value with ones. The output of this test was compared to the output of question 1, which showed that it's easier to flip a bit from a 0 to a 1 than from a 1 to a 0. An attacker could use this to roughly estimate if his attack will work. The attacker would analyze the target instruction to see which bit(s) need to be flipped. If the attacker has to flip a bit from a 1 to a 0 but he already knows that, flipping a bit from a 1 to a 0 is not possible. Then he can focus his attention on a different target without wasting a lot of time.

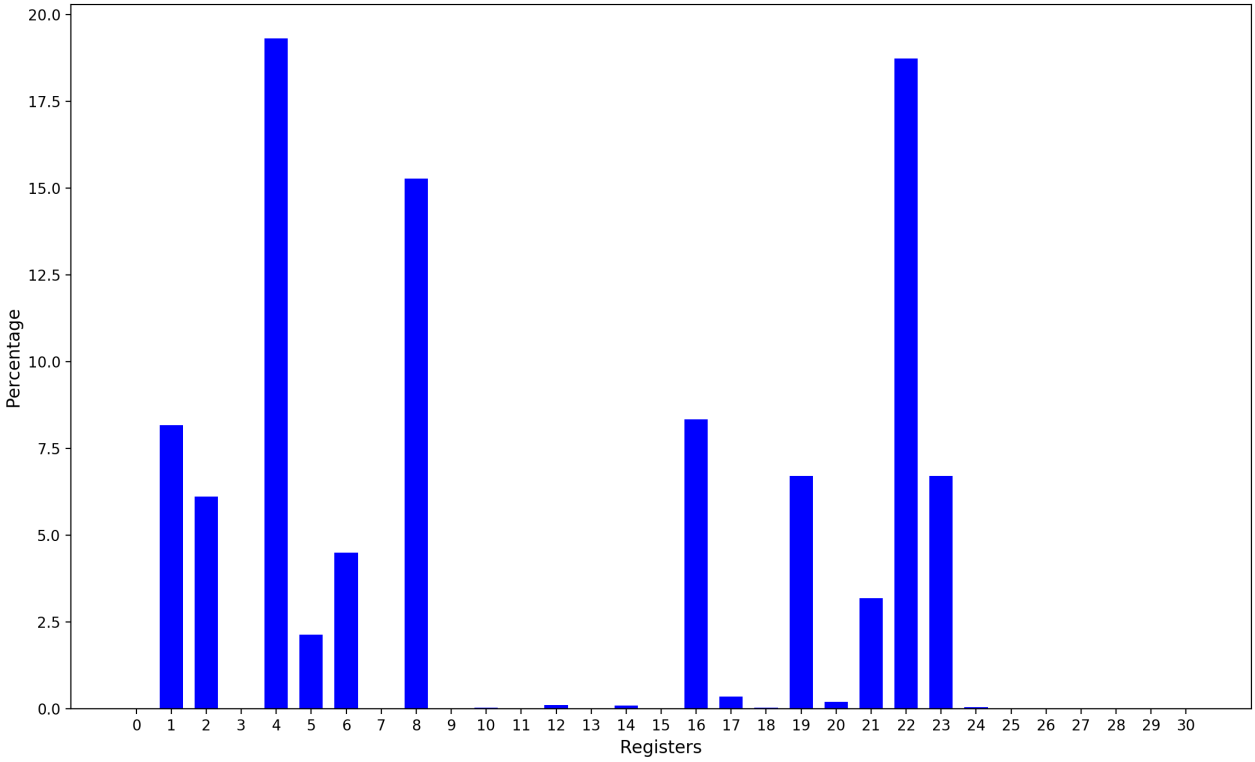

Figure 4.18: Chance of register replacing register 0 in the MOVZ instruction

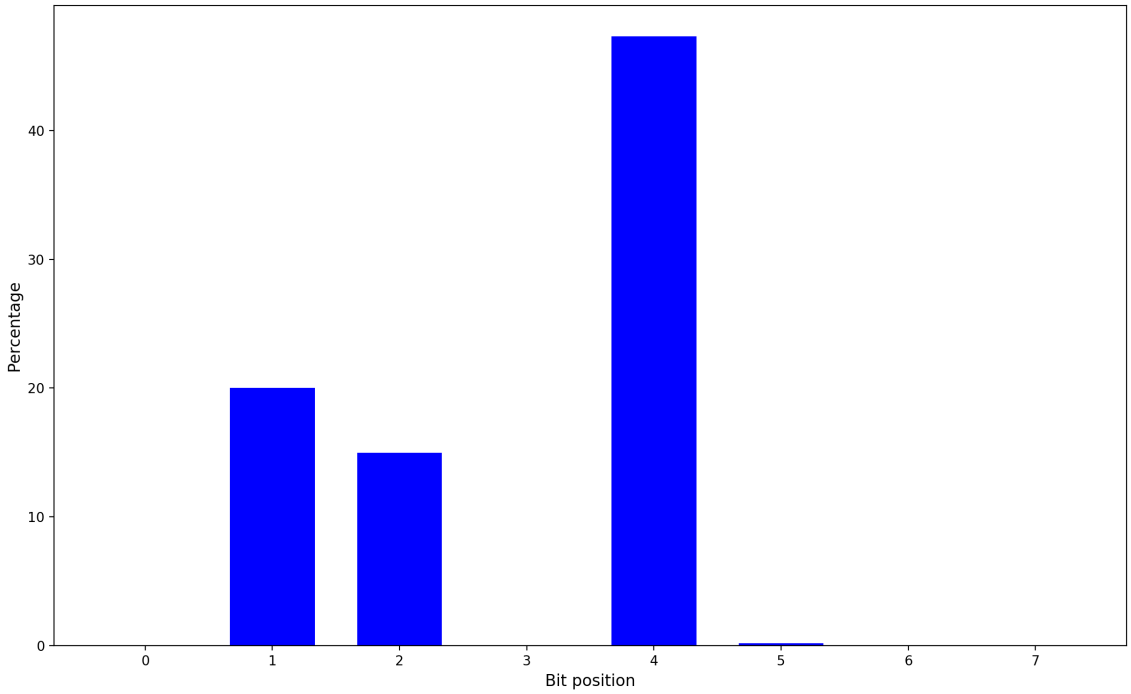

Figure 4.19: Chance of the bit flipping

4.4.2. Where does the glitch happen?

It's important to know where the glitches are induced to complete the fault injection model. The characterization already showed us which places on the chip are vulnerable for glitches. Now the probe will be fixed on a particular spot, to understand more in depth where in the chip these glitches are happening. A CPU operates in three stages: fetch, decode, execute. A glitch can happen in any of these stages.
Based on the results from the characterization, it could be seen that the glitches were probably happening in the DRAM or in one of the CPU caches (after here only named cache). This hypothesis is based on two facts. The first observation was that if we stop glitching after a successful glitch has happened, then running the same glitched test code before a reset and with another test code in between, results in the same glitch result. Figure 4.20 shows one of these results, which suggests that the corrupted instructions were stored somewhere in the cache or DRAM. This already points to the glitches happening during the instruction fetch stage.
The second observation is that when we start glitching at the second attempt after a reset (no glitch at the first attempt), then there is a very small chance that there will be a glitch, during the next attempts before a reset. During the first attempt after the reset there is an instruction transfer from the DRAM to the cache. After that the code will be ran from the cache until the next reset. This already points the answer to the question where are the glitches happening to the DRAM or the instruction transfer from the DRAM to the cache. Multiple test cases were ran on a spot on which we could reliably glitch. There were 5 possible locations of where the glitches could be happening: L1 cache, Transfer from L2 to L1 cache, L2 cache, Transfer from DRAM to L2 cache or in the DRAM.

```
UnrolledLoopASM,0x2710 = 00002732,END OF CODE
LoopC,0x3E8 = 000003e8,END OF CODE
UnrolledLoopASM,0x2710 = 00002732,END OF CODE
```

Figure 4.20: Consistent glitch result until next reset

L1 Cache

The first test case focuses on the L1 cache by only using the L1 cache during the execution of our code. A successful glitch would then mean that the glitch happened in the L1 cache. The unrolled loop in asm code was re-sized to fit into the L1 cache of 32Kb. This way it's certain that the code will always run out of the L1 cache at the second attempt after a reset. At the first attempt the code will be transferred to the L1 cache. This resulted in no glitches, which concludes that the glitches are not happening in the L1 cache.

L2 Cache

The idea was to make the test code so big such that it could fill the L2 cache (512Kb) completely. Unfortunately this was not possible due to the 40Kb restriction on the bootloader 1 code size. The code was made bigger than the L1 cache but smaller than the L2 cache, which would force a transfer between the L1 and L2 cache. The EMFI probe would only fire after the code was loaded into the cache. After a successful glitch, first another test code would be ran before running the same glitched test code. Also there would be no resets or EMFI firing for the next three attempts. This way the L1 cache would partly be refreshed with instructions of the other test codes. If the same glitched output would still appear, then this would mean that the glitch happened in the L2 cache and not during transfer.
Again there were no glitches observed. This doesn't fully point the glitches to the DRAM, considering that the test code is not a multiple times bigger than the L1 cache. Therefore there is not a lot of L1 cache refreshing going on. Furthermore, it could also be possible that the EMFI probe is not fired at the right time.
Second the code tends to stop running (yellow output), when using the L2 cache. This could mean that something is actually being glitched there in the end. To check this more code has to be put into the L2 cache or the L1 cache has to be flushed manually. Unfortunately it's very hard to flush the cache in ARMv8. There was example code in the ARMv8 manual, however it was not working.

DRAM

The last two test cases were to find out if the glitch happened in the DRAM and later the corrupted instructions were send to the L2 cache or that the corruption happened during transfer. The second possibility would

mean that the instructions in the DRAM were not affected. For these last tests the caches were disabled, this way the code has to run directly from the DRAM. After a successful glitch the same code would be ran again to see if the same result would appear. The same result would appear if the corruption happened inside the DRAM and not during transfer. If it happened during transfer, then you should not see the same glitched output when running the same test for the second time without glitching.
The CPU doesn't wait for all the instructions to be send to the cache. It will directly start executing while the cache keeps loading instructions. Therefore at the moment of glitching, the cache controller might still be busy with transferring instructions from the DRAM to the cache [97]. This makes it possible to glitch the instructions before they are loaded into the cache.
The results showed that successful glitches happened most often during transfer. Figure 4.21 shows one of these results. Keep in mind that the caches are turned off. The first line shows a successful glitch. After that the same test was ran without glitching, which resulted in the normal output. This shows that the glitch happened during transfer. Figure 4.22 shows the result of a glitch that modified the instruction inside the DRAM. Running the same test code after a successful glitch would again result in a glitched output without glitching, which is only possible if the instruction changed in the DRAM.

UnrolledLoopASM,0x2710 = 1ac0618c,END OF CODE	1
UnrolledLoopASM,0x2710 = 00002710,END OF CODE	0

Figure 4.21: Glitch during instruction transfer from DRAM to cache

UnrolledLoopASM,0x2710 = 00002707,END OF CODE	1
UnrolledLoopASM,0x2710 = 00002707,END OF CODE	0

Figure 4.22: Instruction glitched in the DRAM

4.4.3. How to use this result?

Based on the what and where the fault injection model can be defined. The five test cases showed that the most reliable glitches happen during the instruction transfer from the DRAM to the L2 cache. The same results keep appearing after a glitch, due to the corrupted instructions being stored in the L2 cache.
An attacker would need to glitch at the exact moment that the targeted instruction is transferred. However, he doesn't have the ability to disable the caches like was done in the DRAM test case. For disabling the caches, the attacker would need EL3 access rights which would mean, that the attacker already has elevated access rights to the system. Instead of disabling the caches, the attacker could trigger the system to refresh the caches. This way the attacker triggers an instruction transfer from the DRAM to the L2 cache. This could be done by for example launching an application before launching the target code. A second less practical but usable method would be to reset the device before every glitch attempt. Unfortunately, this second method would require far more time to obtain a successful glitch.

4.4.4. Special case the Red square

During the characterization a red square was observed in Figure 4.11, which had an unusual high successful glitch rate in comparison with the rest of the chip. To investigate this behaviour, the EMFI probe was fixed on the red square. This spot resulted in a 99% glitch success rate for the unrolled loop in asm. The loop in asm and loop in c would only result in strange ASCII(. * :) values in the output. The most interesting fact of this spot was that the glitch delay didn't have any influence. It didn't matter at what time the EMFI probe would fire, it would always result in a glitch.
This raised the question if the glitch was always happening in the same specific instruction. The unrolled loop asm was modified by removing ADD instructions until there were no successful glitches anymore. This lead to discovering that 8185 ADD instructions were needed before the instruction that would be glitched. 8186 ADD instructions would result in a successful glitch. This points to the possibility that the exact place in DRAM where this instruction is stored is glitched.
The code was modified to prove that the placement of the instruction is really the successful factor here. The

Loop in asm was placed after 8184 ADD instructions. Therefore the ADD instruction in the loop would be glitched, which resulted in only “LoopASM,” being printed. This is the expected result of glitching the ADD instruction such that it will be skipped, therefore the code never jumped out of the loop.
In the next test the CMP instruction was targeted, which resulted in sometimes jumping out of the loop. The output resulted in a one which means that the CMP instruction was probably skipped. The glitch success rate was lower for the CMP instruction than for the ADD instruction. This difference in success rate is due to the bits that are flipped in the instruction, which result in different effects. In the last test the BNE instruction was targeted. This resulted again in 99% success rate. The code would jump out of the loop directly at the first try. To confirm this the initial value of zero was changed into 10. This way the output should be 11 when the code jumps out of the loop, which was exactly the obtained result.

It was mentioned that it seems like that the exact place in DRAM where the instruction is stored is being glitched. A DRAM memory cell often consists out of one capacitor and one transistor. These memory cells are magnitudes smaller than the reach of the induced EM field, which makes it unlikely that only the memory cells of this instruction are glitched. Looking at the power needed for these glitches shows that a power between 25% and 28% is needed. Higher powers will directly result in mutes and lower powers will not affect the chip. Nonetheless it could be possible that more memory cells are effected, however the effect is not strong enough to induce more bit flips. DRAM consists out of far more than only memory cells. Therefore it’s also possible that a different component from the DRAM is glitched. Decapping the DRAM chip would give more insight into what is glitched. Unfortunately, there was no time to do this.

Having a point with a very high glitch success rate (depending on the target instruction) and which is independent of the glitch delay, would be a perfect weapon for an attacker. A FI model that is time independent is a huge benefit looking at the search space. The attacker only needs to glitch the DRAM before the code is transferred into the cache. This way the precision goes from needing to glitch nanoseconds precise to just glitching before running the target code.
The problem with this FI model is that the attacker has to be able to make sure that the target instruction will be placed at an exact place in DRAM. As far as we know this part makes using the red square harder than searching for the moment in time in which the instruction will be transferred from the DRAM to the cache. More research would be needed to determine the usability of the red square.

4.5. Android Characterization

A second characterization (Android characterization) of the chip needs to be performed while running the Android open source project (AOSP). Android and AOSP will be used interchangeably. The first characterization resulted in a FI model which will be used in this second characterization. Full scans of the chip are this time not needed, due to the knowledge already acquired with the previous characterization. The Android characterization is to see if the chip is still vulnerable will running an OS, considering that there could be unknown defenses and there will be a lot of services running in the background, which could interfere with the working of the glitch.

4.5.1. Setup

The setup will be the same as the first characterization setup. However, software wise there will be changes. The power will start at 20% and increment with steps of 1%. It’s now possible to use steps of 1% considering it’s already known that the board will crash for high powers.
With AOSP it’s possible to run our C code without having to package it into a bootloader. The C code was compiled with the linaro tool chain for ARM processors. Running the code from a SD card would not be possible due to the no execute protection from AOSP. The Android Debug Bridge (ADB) was used to send the file into a part of the file system that had no execute turned off. The C code had to be ran as user root to make use of "/dev/mem" for the trigger.
FIPY was modified to wait much longer before deciding that the board was not responding. This is due to the boot time of AOSP which is almost six times longer than the modified bootloader. As a result Glithing AOSP takes far more time than glitching the modified bootloader. The second difficulty was that we found out, that during the modified bootloader the CPU was running at 533 MHz and in AOSP the clock is set to 1,8 GHz. Also the DRAM clock speed was much lower than 1866 MHz during the boot stage. However in Android the DRAM will be running at 1866 MHz

4.5.2. Results

An unrolled loop in asm of a 1000(0x3e8) ADD instructions was used. Less ADD instructions were used than in the first characterization to reduce the time needed for the test. The unrolled loop asm was ran for half a day. This test already resulted in a glitch after approximately 1,5 hours, on a none optimal spot. Just before doing this test, the EMFI table was not responding anymore and it had to be reset. Due to this all the coordinates acquired from the first characterization were lost. There was not enough time to start all over, therefore the spot used in the rest of this project was the result of a fast scan fixed on 35% voltage. It might be possible that there is a spot on the chip which would result in far better results. Doing the Android characterization on a optimal spot will be future work due to the table crash and limited time.

Figure 4.23 shows the results of glitching the unrolled loop asm. Smaller and higher values are again appearing in the output. More of the same kind of results as for the first characterization would probably appear, if there was enough time to keep it running. One of most noticeable observations is that there are no platform exceptions. The board either works normal(green), mutes(yellow) or glitches successfully(red).

```
AOSPUnrolledLoopASM,res0x3e8=000003e7,EndOFCode..
AOSPUnrolledLoopASM,res0x3e8=00000406,EndOFCode..
AOSPUnrolledLoopASM,res0x3e8=001013e7,EndOFCode..
AOSPUnrolledLoopASM,res0x3e8=0ab71ac4,EndOFCode..
AOSPUnrolledLoopASM,res0x3e8=6ef982a3,EndOFCode..
AOSPUnrolledLoopASM,res0x3e8=6ef982b3,EndOFCode..
```

Figure 4.23: Successful Glitches

In Figure 4.24 the glitch power is plotted against the glitch delay. It can be seen that more power is needed to get a successful glitch than when running without AOSP. Successful glitches would start appearing for low powers around 27,5% for the first characterization. However, with AOSP the first successful glitch appears at 34% power.
The second observation is that the glitches can be divided into two groups based on their glitch delay. The first group is between 1000ns and 2000ns and the second group is between 3000ns and 4000ns. Plotting the same for the tests of the first characterization did not show a relation between the glitch delay and the successful glitches, there it looked far more random.
This separation into groups is probably the result of the services that run in the background. The test code is not the only piece of code that will be ran during the full execution of the test. This is due to parallelism techniques which are out of the scope of this thesis.
The FI model showed that a reset is needed after every glitch attempt, considering that after one run the code would be in the cache. The Android characterization again confirmed the FI model as can be seen in Figure 4.25. This figure shows the number of glitches directly after a hard reset (yes) and without a hard reset (No). From this figure it can be concluded that there is a much higher chance to successfully glitch directly after a reset, which is inline with the FI model. The results also showed a chance of 1/5 to glitch the second time the code is ran without a reset. This is probably the result of the code being refetched from DRAM.

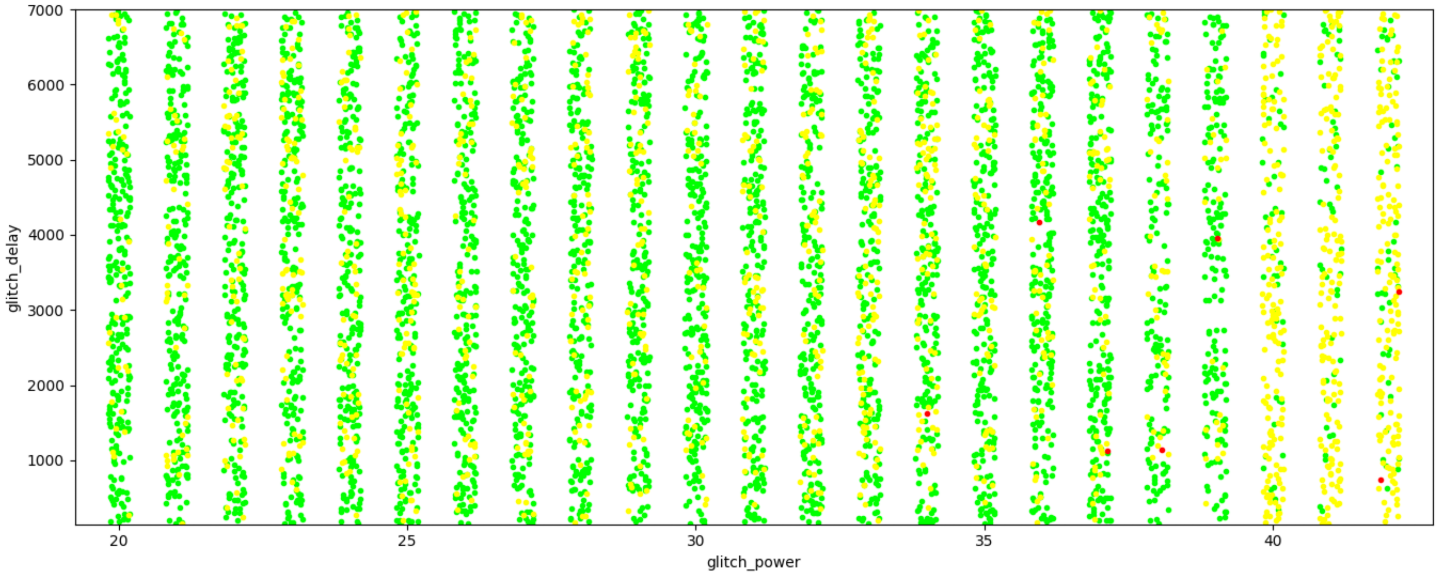

Figure 4.24: Glitch Power vs Glitch Delay

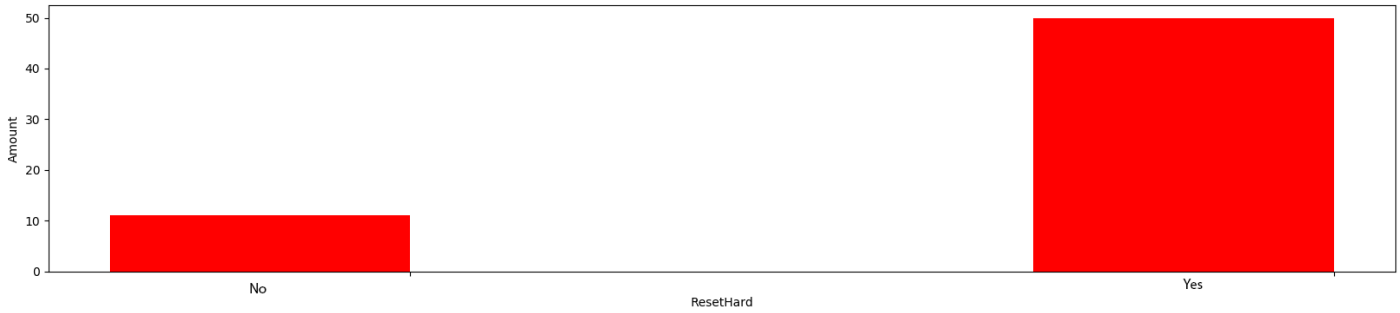

Figure 4.25: AOSP hard resets before successful glitches

The Android characterization was performed to determine, if the chip was still vulnerable while running an OS. The results showed that Android doesn't have any defenses against EMFI attacks. It's still possible to glitch instructions using the previously defined FI model. The none appearing platform exceptions are probably caught by the OS, who is able to recover from those exceptions or the chip mutes. The only difference with the first characterization is, that more power has to be used to get a successful glitch.

4.6. Simulate Lock Screen Attack

The FI attack on the lock screen is first simulated to determine the result of glitching a certain instruction, without having to run the setup for days. This massively speeds up the process of finding out if it's possible to attack the lock screen using FI. However before running the simulation it's important to know which instruction to change. Therefore first the source code of the lock screen has to be reverse engineered to understand how it's working internally.
Android is open source which means that the source code is open to the public. OEM's most of the time modify Android without opening these modifications up to the public. In that case the binaries would have to be reverse engineered to understand what is happening. Luckily, the source code for our version of Android is available online. The second step is to identify possible exploitable vulnerabilities. As last the binaries need to be reverse engineered to find and modify the target instructions.

4.6.1. Code Review

Section 3.2.5 already gave an overview of how gatekeeper is used by the lock screen. This section will go more in depth into how gatekeeper verifies a password. Reviewing the source code showed that gatekeeperd checks

for a hardware backed version of gatekeeper. Android for our board doesn't have a hardware backed implementation of gatekeeper. Instead it uses a software implementation called softgatekeeper. Softgatekeeper implements the necessary functionality needed for gatekeeper like: reading, writing and clearing the failure record.
To be able to understand the source code it's important to know the gatekeeper execution process, which can be seen in Figure 4.26. Gatekeeper starts executing when a user submits a password to unlock the lockscreen. First the failure record will be retrieved from memory and checked to see if there were no previous wrong password attempts. A timeout will directly be triggered if there are more than five wrong attempts. The user will be blocked from submitting a password until after the timeout. In case of no timeout, the failure counter inside the failure record will be incremented. A separate function is used to increment the failure counter. This function will also directly write the failure record back to memory. In the next step the failure counter will be used to compute the retry timeout, which grows exponentially based on the number of failed attempts. After this the password will be checked by the DoVerify function, which is the parameter for a branch structure that contains the true and false case for the password. In the true case the authentication token will be made and the failure record will be cleared. In case of a wrong password the computed timeout will be used to set a retry timeout after five wrong attempts. This will block the user from submitting a password to unlock the device for a certain time.

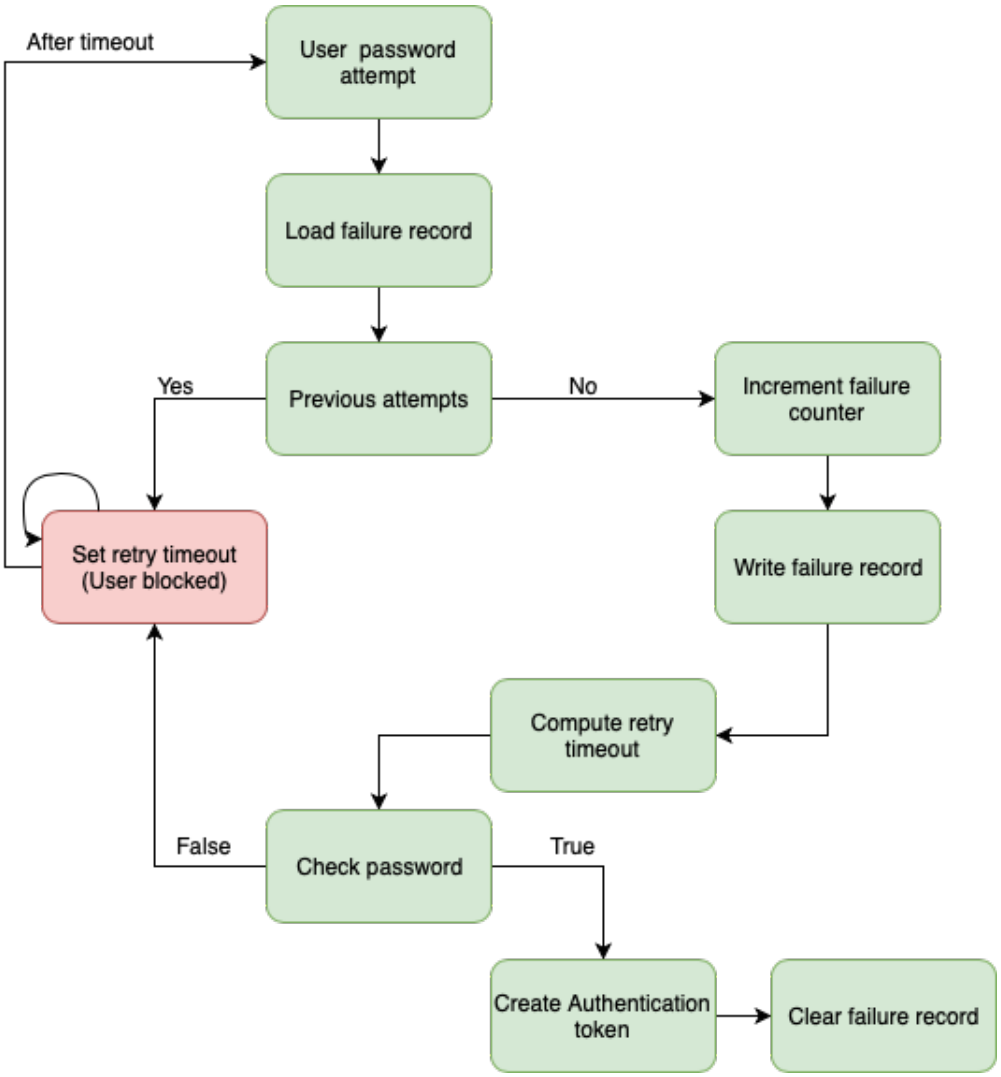

Figure 4.26: Gatekeeper Code Execution Process

4.6.2. Vulnerabilities

Five vulnerabilities were found that could be targeted with FI. These vulnerabilities all have in common that only one of them has to be glitched, to result in a lock screen that doesn't record the failed attempts anymore. These vulnerabilities will be the targets for our FI attack.

GetFailureRecord

GetFailureRecord loads the failure record from memory. A safety feature is built into this function that can be abused. Inside this function there is a branch that checks if the failure record was loaded successfully. If for some reason the failure record couldn't be loaded from memory, then a new failure record will be made. The failure counter inside this new record will be initialized to zero.
The target instruction is the branch, which will be skipped in the case of a successful glitch. As a result the code will continue like the failure record couldn't be loaded from memory, due to this the failure counter will be reset to zero on every attempt.

IncrementFailureRecord / WriteFailureRecord

The IncrementFailureRecord function simply adds one to the failure counter and writes the failure record back to memory. However the write back is done by a separate function named WriteFailureRecord. There are two possible attack targets here. The first one is the ADD instruction that will increment the failure record. If this instruction could be glitched the failure record will not be incremented.
The second attack target is in the WriteFailureRecord function. If the failure record is not written back to memory, then the old value will be loaded every time the GetFailureRecord function is executed. As a result the failure counter will not be incremented, which is achieved by glitching the store (STR) instruction.

DoVerify Branch

DoVerify is the function that compares the submitted password to a stored hash of the password and returns true if the password is correct. This function is used as a parameter for the branch structure that creates the authentication token or locks the user out. Luckily, the true case comes first, which means that the true case will be executed if the branch instruction could be skipped. As a result the authentication token is made without the correct password.

ComputeRetryTimeout

The last possible attack target is the ComputeRetryTimeout function. This function computes the timeout based on the value inside the failure counter. At the start of the function there is a branch that will check if the failure counter is not equal to zero. A failure counter of zero would mean that the retry timeout should immediately be zero. Glitching this branch instruction will result in a timeout of zero, even when the failure counter is not equal to zero.

4.6.3. Simulated Attack

Based on the code review there are five possible attack targets that have to be tested separately. The gatekeeper binary files were reverse engineered using an open source tool called radare2 [98]. Using this tool and a hex editor made it possible to manually change a specific instruction, to mimic a successful glitch. The target instructions were replaced by a useless instruction (MOV X0, X0), which simulates the glitched instruction.
After the binary modification the code would be re-flashed on the board. As last Multiple wrong pins would be submitted to check if the failure counter was not incrementing. Two of the five targets resulted in a successful attack.

The first successful target was the GetFailureRecord function. Figure 4.27 partly shows the disassembly of the GetFailureRecord function. The underlined branch if equal (BEQ) instruction was modified as can be seen in Figure 4.28. This modification makes it possible to brute force the Android lock screen, due to the failure record being recreated on every attempt.
The second successful target was the WriteFailureRecord function. Figure 4.29 partly shows the disassembly of the WriteFailureRecord function. The underlined STR instruction was modified as can be seen in Figure 4.30. This again resulted in a failure counter that was not incrementing, due to the failure record not being written back to memory.

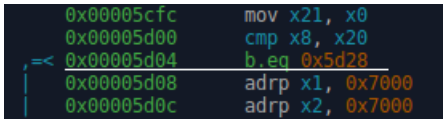

Figure 4.27: GetFailureRecord part of the binary

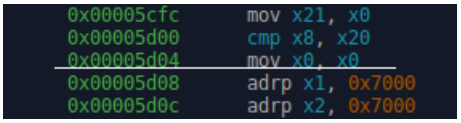

Figure 4.28: GetFailureRecord part of the binary modified

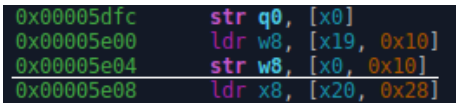

Figure 4.29: WriteFailureRecord part of the binary

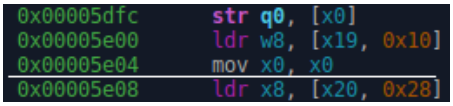

Figure 4.30: WriteFailureRecord part of the binary modified

The other three targets were a lot harder to modify, due to the compiled functions being found in multiple places. The Verify function which contains the branch of the DoVerify target, was even compiled into multiple functions with different names. This made it not possible to find the correct target instruction using radare2. The other two targets were found and modified in different places. However this resulted in the lock screen not verifying the pin anymore or it kept working normally. Removing the binary files from the board showed that the lock screen stopped working if the lib64 binary was removed. Nonetheless changing the instructions inside lib64 still didn't result in any success.
The reason for not finding the correct place to modify the code, is due to the functions being virtual functions. Virtual functions are used to tell the compiler to use late binding. Simply said this makes it possible for gatekeeperd to decide at run time, which implementation of gatekeeper it wants to use.

The goal of the simulation was to see if changing a specific instruction, would result in the lock screen not recording the failed attempts anymore. As a result the lock screen will be vulnerable to a brute force attack. Five attack targets were identified of which two resulted in a lockscreen that wouldn't record the failed attempts anymore. Therefore the simulation showed that it's definitely possible to successfully glitch the lock screen.

4.7. Lock Screen Attack

The final step in the project is to perform the attack in a real life situation, while running Android version eight on the chip. The simulation already showed that it's definitely possible to attack the lock screen by changing a specific instruction. However, finding the right moment in time to glitch will be the main problem. Especially due to the boot time, which will be even longer than in the Android characterization, considering that the whole graphical user interface has to be loaded. The FI model showed that the board will have to be reset after every attempt, therefore the boot time will be a major bottle neck for the run time of this attack.

4.7.1. Setup

The Android characterization setup will have to be modified to automate the attack. The lockscreen expects a user to input the pin code, which was automated by emulating a keyboard using an Arduino Leonardo(Figure 4.31).

This board can identify itself as a HID device and connect to the development board. Android already has the capability to accept keyboard input. The Arduino was programmed to wait for an input from the spider on it's GPIO pins. Based on this input the Arduino would: open the lockscreen, input the wrong pin code or input the correct pin code. Submitting to many wrong pin codes would trigger a retry timeout. Therefore, the right pin code was always submitted after three wrong pin code attempts to clear out the failure record. This prevents the board from triggering a retry timeout.

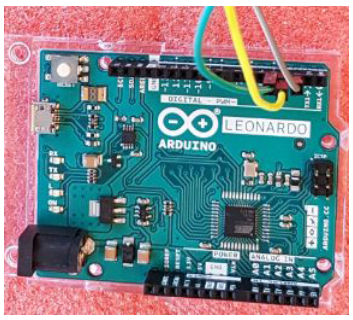

Figure 4.31: Arduino Leonardo

The gatekeeper source code was modified to incorporate a trigger. Section 4.3.4 already explained the need for a trigger. Unfortunately, gatekeeper is not able to use the same trigger method as for the Android characterization. The previously used triggers were all based on using "/dev/mem". This file gives access to the whole memory, however it's only accessible using a root user.
Gatekeeper is running as user system which has no access to "/dev/mem". Running gatekeeper as user root broke the whole lockscreen. An alternative to using "/dev/mem" was to use "/dev/gpio". This is a build-in library that gives access to the GPIO pins from user space. The disadvantage of using this method is that it deals with multiple layers of code, which could influence the reliability of the trigger. Also the uart was based on using "/dev/mem", however one of the build in uart drivers could be used as an replacement.
The gatekeeper source code was also modified to output the failure counter value over the uart. The normal output can be seen in Figure 4.32. This string was used by FIPY to again decide if a successful glitch appeared. First the failure counter is printed directly after loading it from memory. The second part of the string "FailureCounter+1=" is the failure counter after being incremented by the IncrementFailureRecord function.
After every glitch attempt the failure record will be cleared and the chip rebooted. Clearing the failure record is done by submitting the right pin code just before rebooting. After a board reboot first a wrong input was given. The output of the uart would be discarded because gatekeeper would be transferred from the DRAM to the cache at the first pin code attempt. The second pin code submit attempt is the moment that a successful glitch can be detected. A wrong input is again given and now the output should show a failure counter with value zero. If this is the case then there probably was a successful glitch. To confirm this the chip will not be reset, instead again a wrong input is given. Now the output should again show a failure counter with value zero, if the glitch was successful. In case that the glitch was not successful, then the failure counter will show a value bigger than zero.

```
VerifyCode,..FailureCounter=00000001,FailureCounter+1=00000002,EndOfCode..
```

Figure 4.32: Gatekeeper failure counter normal output.

The development board was actually supposed to have a passive cooling element on the chip. However, this couldn't be placed due to the chosen FI technique. During all the previous tests the chip wouldn't heat up that much. Therefore it was okay to use the chip without the cooling element. Running Android with the full user interface changed this quickly. The chip would become very hot and had to be cooled. The solution was to use active cooling in the form of a small desk fan, as can be seen in Figure 4.33 which shows the completed setup.

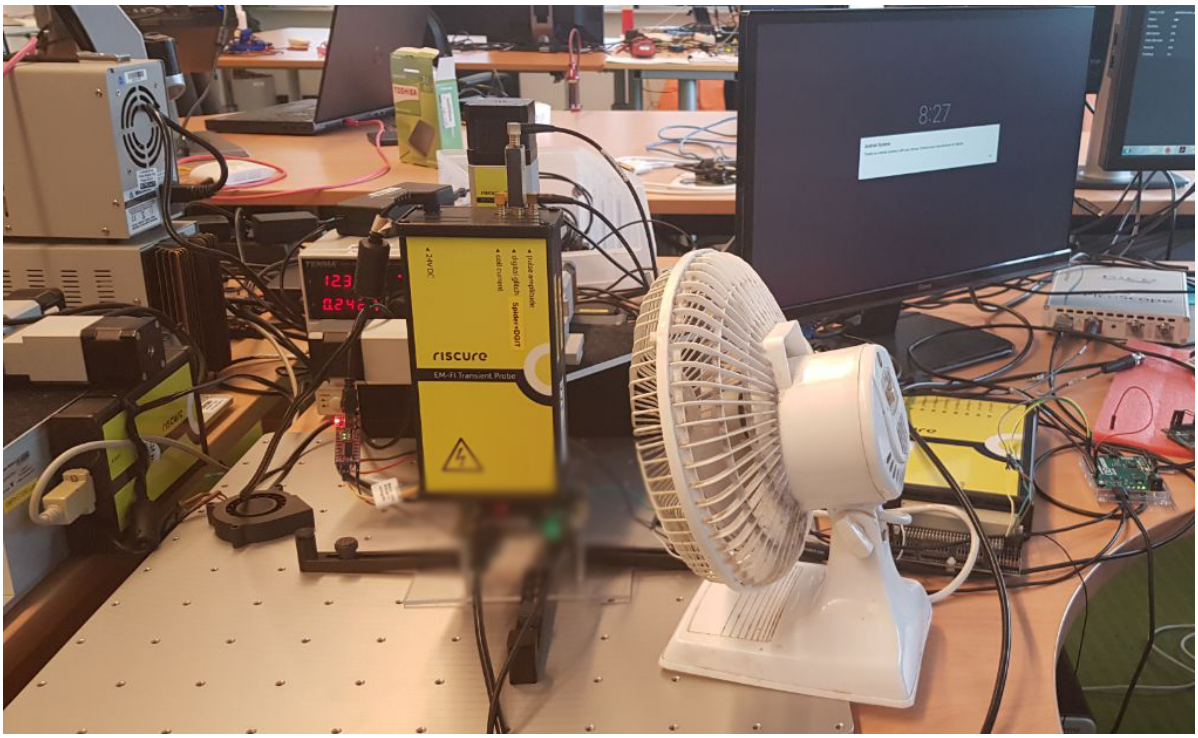

Figure 4.33: AOSP Lockscreen Attack Setup

4.7.2. Results

The setup was left running for 35 days, which resulted in four interesting outputs that contained a failure counter with value zero. These four results all printed the same string that can be seen in Figure 4.34. There are two possibilities that could have resulted in this exact output. The first possibility is that one of the attack targets was glitched successfully. The second possibility is that the first wrong pin code submit attempt didn't go through. Therefore the wrong input is only submitted once and the output results in zero.
The way to distinguish between these two possibilities is by submitting a wrong input again. FIPY will detect that the failure counter is still zero and send again a wrong input without glitching. The glitch was successful, if this would again output a failure counter with value zero. If not, then it's very likely that the first wrong input was not received.
The results after the four zero outputs were inspected. This showed that after two of the results the board crashed and after the other two the failure counter was again incremented. Therefore it can be concluded that the first wrong pin code submit attempt was not received by the chip.

```
VerifyCode,..FailureCounter=00000000,FailureCounter+1=00000001,EndOfCode..
```

Figure 4.34: Gatekeeper failure counter glitched output

In this last part the FI model was used to perform the attack in a real life situation. The goal was to make it possible to brute force the Android lockscreen by glitching the gatekeeper code. As a result Gatekeeper should not increment the failure counter anymore. Therefore, an infinite amount of failed pin code submit attempts would be possible. However the results showed that there were no successful glitches.

4.8. Discussion

The goal of this thesis was to analyze the resilience of smartphones against FI attacks, by using EMFI to glitch the Android lock screen. In the literature often old chips are used to perform FI. In this thesis we wanted to use a recently released chip, which came with two main problems as was discussed in Section 4.2. The first problem was the clock speed and the second problem was the package on package configuration. In the lit-

erature EMFI was only used on chips running at a maximum frequency of a 100 MHz and nothing was found about using EMFI on PoP chips. This made choosing such a chip for this project very risky.
It took a long time and many preparations before it could be verfied that the chip was vulnerable for EMFI. The literature pointed to the fact that the PoP configuration could act as an EM shield, which in the end could have been the case. It's hard to find out if a higher power would penetrate the DRAM and influence the CPU on the bottom, considering that the chip mutes for high powers. Nonetheless the results showed that it's definitely possible to glitch a PoP chip running on a high frequency by using EMFI.
The characterization results showed that EMFI can be used to modify or "skip" the instructions executed by the chip. Furthermore the Android characterization showed that the same glitch results can be obtained while running the Android OS. Therefore it can be concluded that the Android OS doesn't protect against FI attacks.
The simulation of the actual attack showed that the Android lock screen can be glitched by manually adjusting just one out of two specific instructions. Modifying one of these instructions resulted in the failure counter not changing anymore after a wrong pin code submit attempt, which opens the door for a brute force attack. However the real world attempt of glitching the lock screen by using EMFI was not successful. This doesn't mean that it's impossible to glitch the Android lock screen. The main obstacle in getting a successful glitch is finding the right moment in time to glitch. During the Android characterization the test code would need 7000 ns to execute. In these 7000 ns there were only two windows found of approximately a 1000 ns, that resulted in successful glitches while this test was ran for halve a day. The lock screen code (gatekeeper) runs for approximately 125000 ns which is a huge search space in comparison to the test codes. Therefore even after 35 days no successful moment in time to glitch was found.
The reason for this huge search space is not only the lockscreen code. Tests showed that building a trigger using "/dev/gpio" resulted in a very unstable trigger. The trigger window size was changing as much as double the size between attempts. Therefore it's possible that half of the time the glitches were outside of the lock screen code. The trigger was probably unstable due to the many layers of code that are used for "/dev/gpio" as mentioned in Section 4.5.
Furthermore all the tests and the attack that were performed starting from the Android characterization, were using a non-optimal spot on the chip. This was due to the EMFI table that crasht again as was explained in Section 4.5.2, which played a huge role in the chance on a successful glitch. Nonetheless it can be concluded based on the characterizations and the simulation that it's definitely possible to glitch the Android lock screen.

5 Conclusion

This chapter summarizes the achievements of this thesis and proposes possible future work. Section 5.1 gives a summary of this thesis. Section 5.2 proposes possible future work.

5.1. Summary

Chapter one motivates the need for security as security is often (partially) ignored during the design phase due to its additional costs, which makes designs vulnerable to attacks. Furthermore it describes why hardware attacks are an upcoming danger, as a vulnerability found in the hardware often stays there for the lifetime of the device. An introduction into cyber security is given to show the criteria used to decide if a system is secure. This thesis analyzes the resilience of smartphones against a certain hardware attack technique called fault injection, which is used to attack the Android lock screen.
Chapter two shows a way to classify hardware attacks by using four metrics. First, what is attacked, the so called target. Second, how is the target attacked, which attack technique is used. Third, at what phase is the target attacked. Fourth, the location in the system where the attack took place. Furthermore it classifies known hardware attacks and their proposed countermeasures, based on the four metrics.
Chapter three dives into the Android operating system and target hardware. The Android OS is built out of a stack of software components. These components and their security features are fully analyzed. The target hardware and especially the SoC is elaborated on.
Chapter four is the core of this thesis. This chapter builds up to actually attacking the Android lock screen. First, the most suitable FI technique for this attack is chosen based on the target design and characteristics of the available FI techniques. The targets vulnerability for FI is determined using that technique. Based on the results an FI model is created, which is used to test the vulnerability of the target while running the Android OS. These tests showed that the target is vulnerable for FI and that Android doesn't have any defenses against FI. After that a simulation of the actual attack is performed, which showed that it's definitely possible to attack the Android lock screen using FI. However, the real life attack was not successful, due to the large search space, unreliable trigger and possibly the non-optimal position of the EM probe.

5.2. Future work

This thesis has shown that the target chip is vulnerable for EMFI. The next step would be to perform again a characterization, however now with a smartphone that makes use of the same chip. Our expectation is that the smartphone would be vulnerable for EMFI.
From the Android characterization on a non-optimal spot had to be used. Therefore it would be logical to perform the attack again using a better spot, which might give better results. Also, a more reliable trigger would definitely improve the chance on success for this attack.
Power glitching was also one of the possible options to choose. Sadly, there was no time left to try this out. This technique would directly influence the power lines of the CPU cores without influencing the Dram. It would also be very interesting to see if it's beneficial to use EMFI and power glitching together.

Bibliography

[1] Statista. Internet of things (iot) connected devices installed base worldwide from 2015 to 2025 (in billions). [Online]. Available: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

[2] S. Mitra, H. . P. Wong, and S. Wong, "The trojan-proof chip," *IEEE Spectrum*, vol. 52, no. 2, pp. 46–51, February 2015.

[3] Apple. Application sandbox. [Online]. Available: https://developer.apple.com/library/archive/documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html

[4] Unkown. Mac vs dac. [Online]. Available: https://www.slideserve.com/lynda/security-enhanced-linux

[5] Android. Authentication. Visited on: 11-01-2017. [Online]. Available: https://source.android.com/security/authentication

[6] Riscure. Em-fi transient probe. Visited on: 05-05-2019. [Online]. Available: https://www.riscure.com/uploads/2017/07/datasheet_em-fi_transient_probe.pdf

[7] ARM, "Arm architecture reference manual," Tech. Rep., 2013. [Online]. Available: https://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf

[8] F. Consulting, "Fail to plan, plan to fail understanding the roles of lob practitioners and socs in securing iot environments," ForeScout, Tech. Rep., 2017. [Online]. Available: https://www.forescout.com/wp-content/uploads/2017/11/Forrester-Survey-Fail-To-Plan.pdf

[9] T. Micro. Samsam ransomware hits us hospital, management pays $55k ransom. [Online]. Available: https://www.trendmicro.com/vinfo/my/security/news/cyber-attacks/samsam-ransomware-hits-us-hospital-management-pays-55k-ransom

[10] C. Osborne. Malicious code hidden in advert images cost ad networks $1.13bn this year. [Online]. Available: https://www.zdnet.com/article/malicious-code-hidden-in-advert-images-cost-ad-networks-1-13bn-last-year/

[11] G. U. of Technology. Meltdown and spectre vulnerabilities in modern computers leak passwords and sensitive data. [Online]. Available: https://meltdownattack.com

[12] n. qlutoo, derrekr6. Switch security, homebrew on the horizon. [Online]. Available: https://switchbrew.github.io/34c3-slides/

[13] Synopsys. Heartbleed bug. [Online]. Available: http://heartbleed.com

[14] S. Shankland. 'heartbleed' bug undoes web encryption, reveals yahoo passwords. [Online]. Available: https://www.cnet.com/news/heartbleed-bug-undoes-web-encryption-reveals-user-passwords/

[15] Statcounter. Mobile operating system market share worldwide. [Online]. Available: http://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-201712-201812

[16] L. O'Donnell. Iot security concerns peaking – with no end in sight. [Online]. Available: https://threatpost.com/iot-security-concerns-peaking-with-no-end-in-sight/131308/

[17] Accenture and i. NowSecure, "Mobile banking applications," AccentureConsulting and NowSecure, inc., Tech. Rep., 2017. [Online]. Available: https://www.accenture.com/t20180223T145013Z__w__/us-en/_acnmedia/PDF-49/Accenture-Mobile-Banking-Apps-Security-Challenges-Banks.pdf

[18] M. D. Hill. On the meltdown spectre design flaws. [Online]. Available: http://research.cs.wisc.edu/multifacet/papers/hill_mark_wisconsin_meltdown_spectre.pdf

[19] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology — CRYPTO' 99*, M. Wiener, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397.

[20] D. Genkin, A. Shamir, and E. Tromer, "Rsa key extraction via low-bandwidth acoustic cryptanalysis," in *Advances in Cryptology – CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 444–461.

[21] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *Cryptographic Hardware and Embedded Systems — CHES 2001*, Ç. K. Koç, D. Naccache, and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 251–261.

[22] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, Feb 2006.

[23] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller," in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, Aug 2013, pp. 77–88.

[24] Riscure. [Online]. Available: https://www.riscure.com

[25] Newae. [Online]. Available: https://newae.com

[26] NewAE. Chipshouter kit. [Online]. Available: http://store.newae.com/chipshouter-kit/

[27] C. Doerr, *Network security in theory and practice*. Chrisitan Doerr, 2017.

[28] D. Pleskonjic, F. Virtuani, and O. Zoggia, "Security risk management for critical infrastructures," 10 2011.

[29] J. Joshi and C. Parekh, "Android smartphone vulnerabilities: A survey," in *2016 International Conference on Advances in Computing, Communication, Automation (ICACCA) (Spring)*, April 2016, pp. 1–5.

[30] S. Karthick and S. Binu, "Android security issues and solutions," in *2017 International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*, Feb 2017, pp. 686–689.

[31] S. Iqbal, A. Yasin, and T. Naqash, "Android (nougats) security issues and solutions," in *2018 IEEE International Conference on Applied System Invention (ICASI)*, April 2018, pp. 1152–1155.

[32] J. V. Carreira, D. Costa, and J. G. Silva, "Fault injection spot-checks computer system dependability," *IEEE Spectrum*, vol. 36, no. 8, pp. 50–55, Aug 1999.

[33] J. Karlsson, U. Gunneflo, P. Liden, and J. Torin, "Two fault injection techniques for test of fault handling mechanisms," in *1991, Proceedings. International Test Conference*, Oct 1991, pp. 140–.

[34] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults (extended abstract)," in *EUROCRYPT*, 1997.

[35] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, "Fault attacks on rsa with crt: Concrete results and practical countermeasures," in *Cryptographic Hardware and Embedded Systems - CHES 2002*, B. S. Kaliski, ç. K. Koç, and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 260–275.

[36] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *Cryptographic Hardware and Embedded Systems - CHES 2002*, B. S. Kaliski, ç. K. Koç, and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 2–12.

[37] J.-M. Schmidt and M. Hutter, "Optical and em fault-attacks on crt-based rsa : Concrete results," 2007.

[38] A. Cui and R. Housley, "BADFET: Defeating modern secure boot using second-order pulsed electromagnetic fault injection," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/woot17/workshop-program/presentation/cui

[39] N. Timmers and C. Mune, "Escalating privileges in linux using voltage fault injection," in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Sept 2017, pp. 1–8.

[40] N. Timmers, A. Spruyt, and M. Witteman, "Controlling pc on arm using fault injection," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Aug 2016, pp. 25–35.

[41] A. Vasselle, H. Thiebeauld, Q. Maouhoub, A. Morisset, and S. Ermeneux, "Laser-induced fault injection on smartphone bypassing the secure boot," in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Sept 2017, pp. 41–48.

[42] Ip piracy. [Online]. Available: https://www.techopedia.com/definition/5456/piracy

[43] S. media international. Circuit camouflage technology. Visited on: 28-11-2017. [Online]. Available: http://www2.dac.com/ebooth/files/277724!SMI_Circuit_Camo_Data_Sheet.pdf

[44] J. Zhang, "A practical logic obfuscation technique for hardware security," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 3, pp. 1193–1197, March 2016.

[45] C.-H. Chang and M. Potkonjak, *Secure System Design and Trustable Computing*, 1st ed. Springer Publishing Company, Incorporated, 2015.

[46] M. Lewandowski, R. Meana, M. Morrison, and S. Katkoori, "A novel method for watermarking sequential circuits," in *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*, June 2012, pp. 21–24.

[47] Y. M. Alkabani and F. Koushanfar, "Active hardware metering for intellectual property protection and security," in *16th USENIX Security Symposium (USENIX Security 07)*. Boston, MA: USENIX Association, 2007. [Online]. Available: https://www.usenix.org/conference/16th-usenix-security-symposium/active-hardware-metering-intellectual-property-protection

[48] E. Dubrova. Anti-tamper techniques. [Online]. Available: https://people.kth.se/~msmith/is2500_pdf/Anti-Tamper%20Techniques_elena.pdf

[49] M. Aarts, "Hardware attacks tamper resistance, tamper response and tamper evidence." [Online]. Available: http://maurice.aarts.info/papers/tamper_resistance_evidence.pdf

[50] S. Garg and J. J. Rajendran, *Split Manufacturing*. Cham: Springer International Publishing, 2017, pp. 243–262. [Online]. Available: https://doi.org/10.1007/978-3-319-49019-9_10

[51] A. Schlösser, D. Nedospasov, J. Krämer, S. Orlic, and J.-P. Seifert, "Simple photonic emission analysis of aes," in *Cryptographic Hardware and Embedded Systems – CHES 2012*, E. Prouff and P. Schaumont, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 41–57.

[52] K. Tiri, D. Hwang, A. Hodjat, B.-C. Lai, S. Yang, P. Schaumont, and I. Verbauwhede, "Prototype ic with wddl and differential routing – dpa resistance assessment," in *Cryptographic Hardware and Embedded Systems – CHES 2005*, J. R. Rao and B. Sunar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 354–365.

[53] T. S. Messerges, "Securing the aes finalists against power analysis attacks," in *Fast Software Encryption*, G. Goos, J. Hartmanis, J. van Leeuwen, and B. Schneier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 150–164.

[54] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, Apr 2011. [Online]. Available: https://doi.org/10.1007/s13389-011-0006-y

[55] M. A. Kasmi, M. Azizi, and J. Lanet, "Side channel analysis techniques towards a methodology for reverse engineering of java card byte-code," in *2015 11th International Conference on Information Assurance and Security (IAS)*, Dec 2015, pp. 104–110.

[56] Y. Lyu and P. Mishra, "A survey of side-channel attacks on caches and countermeasures," *Journal of Hardware and Systems Security*, vol. 2, no. 1, pp. 33–50, Mar 2018. [Online]. Available: https://doi.org/10.1007/s41635-017-0025-y

[57] J. Aerts and E. J. Marinissen, "Scan chain design for test time reduction in core-based ics," in *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*, Oct 1998, pp. 448–457.

[58] D. Hely, F. Bancel, M. . Flottes, and B. Rouzeyre, "Secure scan techniques: a comparison," in *12th IEEE International On-Line Testing Symposium (IOLTS'06)*, July 2006, pp. 6 pp.–.

[59] J. Dutertre, J. J. A. Fournier, A. Mirbaha, D. Naccache, J. Rigaud, B. Robisson, and A. Tria, "Review of fault injection mechanisms and consequences on countermeasures design," in *2011 6th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, April 2011, pp. 1–6.

[60] P. Maurine, K. Tobich, T. Ordas, and P.-Y. Liardet, "Yet another fault injection technique : by forward body biasing injection," 09 2012.

[61] K. Gomina, J. Rigaud, P. Gendrier, P. Candelier, and A. Tria, "Power supply glitch attacks: Design and evaluation of detection circuits," in *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, May 2014, pp. 136–141.

[62] M. Mozaffari-Kermani and A. Reyhani-Masoleh, "Concurrent structure-independent fault detection schemes for the advanced encryption standard," *IEEE Transactions on Computers*, vol. 59, no. 5, pp. 608–622, May 2010.

[63] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 361–372.

[64] O. Kommerling and M. Kuhn, "Design principles for tamper-resistant smartcard processors," 06 1998.

[65] R. Love. What are the major changes that android made to the linux kernel? [Online]. Available: https://www.quora.com/What-are-the-major-changes-that-Android-made-to-the-Linux-kernel

[66] eLinux.org. Android kernel features. Visited on: 20-11-2017. [Online]. Available: https://elinux.org/Android_Kernel_Features

[67] Oracle, *Java Platform, Standard Edition Java Virtual Machine Guide, Release 9*, Oracle, visited on: 21-11-2017. [Online]. Available: https://docs.oracle.com/javase/9/vm/toc.htm

[68] Android. Art and dalvik. Visited on: 21-11-2017. [Online]. Available: https://source.android.com/devices/tech/dalvik/

[69] ——. Application fundamentals. Visited on: 22-11-2017. [Online]. Available: https://developer.android.com/guide/components/fundamentals.html

[70] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android," Tech. Rep., 2013. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.391.2279&rep=rep1&type=pdf

[71] D. D. Downs, J. R. Rub, K. C. Kung, and C. S. Jordan, "Issues in discretionary access control," in *1985 IEEE Symposium on Security and Privacy*, April 1985, pp. 208–218.

[72] R. Ausanka-crues, "Methods for access control: Advances and limitations," Tech. Rep., 2001. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.596.2814&rep=rep1&type=pdf

[73] I. Gentoo Foundation. Selinux. [Online]. Available: https://wiki.gentoo.org/wiki/SELinux

[74] Android. Security-enhanced linux in android. Visited on: 04-12-2017. [Online]. Available: https://source.android.com/security/selinux/

[75] C. E. Landwehr, "Formal models for computer security," *Computing Surveys*, vol. 13, 1981, 0100-4892/81/0900-0247. [Online]. Available: https://dl.acm.org/citation.cfm?id=356852

[76] N. Elenkov, *Android Security Internals*. Oxford, UK: NoStarch, 2014.

[77] Android. System permissions. Visited on: 11-12-2017. [Online]. Available: https://developer.android.com/guide/topics/permissions/index.html

[78] ——. App manifest. Visited on: 11-12-2017. [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro.html

[79] nuuneoi. (2015) Everything every android developer must know about new android's runtime permission. Visited on: 11-12-2017. [Online]. Available: https://inthecheesefactory.com/blog/things-you-need-to-know-about-android-m-permission-developer-edition/en

[80] R. Chakraborty. (2016) Android working with marshmallow (m) runtime permissions. Visited on: 11-12-2017. [Online]. Available: https://www.androidhive.info/2016/11/android-working-marshmallow-m-runtime-permissions/

[81] Android. (2017) Application security. Visited on: 11-12-2017. [Online]. Available: https://source.android.com/security/overview/app-security

[82] E. Loli. Introduction to openbinder and interview with dianne hackbor. Visited on: 28-11-2017. [Online]. Available: http://www.osnews.com/story/13674/Introduction-to-OpenBinder-and-Interview-with-Dianne-Hackborn/

[83] D. Hackborne. [patch 1/6] staging: android: binder: Remove some funny usage. Visited on: 28-11-2017. [Online]. Available: https://lkml.org/lkml/2009/6/25/3

[84] T. Schreiber, *Android Binder*. Ruhr Universität Bochum, 2011. [Online]. Available: https://www.nds.rub.de/media/attachments/files/2011/10/main.pdf

[85] Android. Android interface definition language (aidl). Visited on: 28-11-2017. [Online]. Available: https://developer.android.com/guide/components/aidl.html

[86] Rishabh. (2015) Android interprocess communication (ipc) with aidl. Visited on: 28-11-2017. [Online]. Available: http://codetheory.in/android-interprocess-communication-ipc-with-aidl/

[87] Android. Bound services. Visited on: 28-11-2017. [Online]. Available: https://developer.android.com/guide/components/bound-services.html

[88] Rishabh. (2015) Android interprocess communication (ipc) with messenger (remote bound services). Visited on: 28-11-2017. [Online]. Available: http://codetheory.in/android-interprocess-communication-ipc-messenger-remote-bound-services/

[89] Android. Content providers. Visited on: 28-11-2017. [Online]. Available: https://developer.android.com/guide/topics/providers/content-providers.html

[90] ——. Intents and intent filters. Visited on: 28-11-2017. [Online]. Available: https://developer.android.com/guide/components/intents-filters.html

[91] ——. Encryption. Visited on: 19-12-2017. [Online]. Available: https://source.android.com/security/encryption/

[92] A. B. Mohamed Sabt, Mohammed Achemlal, "Trusted execution environment: What it is, and what it is not," Tech. Rep., 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01246364/file/trustcom_2015_tee_what_it_is_what_it_is_not.pdf

[93] Android. Trusty tee. Visited on: 19-12-2017. [Online]. Available: https://source.android.com/security/trusty/

[94] S. J. Murdoch. Introduction to trusted execution environments (tee) – iy5606. University of Cambridge. Visited on: 19-12-2017. [Online]. Available: http://sec.cs.ucl.ac.uk/users/smurdoch/talks/rhul14tee.pdf

[95] Android. Android keystore system. Visited on: 19-12-2017. [Online]. Available: https://developer.android.com/training/articles/keystore.html

[96] ——. Hardware-backed keystore. Visited on: 19-12-2017. [Online]. Available: https://source.android.com/security/keystore/

[97] ARM, "Programmer's guide for armv8-a," Tech. Rep., 2015. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf

[98] Pancake. Radare. Visited on: 11-12-2017. [Online]. Available: https://rada.re/r/index.html

A

Tests code sample

```
//uintptr_t watchdog =  (uintptr_t) 0x2a490000 ;

int res,SysRegB,SysRegA;
int j;
volatile int delay = 0;

uint *GpioDir = (uint *)0xFFF0F400;
uint *GpioData = (uint*)0xFFF0F004;

uint *WatchDog1CTRL = (uint*) 0xE8A07008;
uint *WatchDog0CTRL = (uint*) 0xE8A06008;
uint *WatchDog1Lock = (uint*) 0xE8A07C00;
uint *WatchDog0Lock = (uint*) 0xE8A06C00;

*GpioDir = 0x01; //Set GPIO_208 as OUTPUT pin

//Turn of the watchdog else board will reset every minute
*WatchDog0Lock = (uint) 0x1ACCE551;
*WatchDog1Lock = (uint) 0x1ACCE551;

*WatchDog0CTRL = (uint) 0x00;
*WatchDog1CTRL = (uint) 0x00;

printf("Give command in hex\n");
while(1){

  switch(GetInput()){
    case(0x61): //'a'
      printf("UnrolledLoopASM,");


  //BACKUP REGS
  asm volatile( "SUB sp, sp , #240 \n"
      "STP x1 , x2 ,[sp] \n"
      "STP x3, x4,[sp, #16]\n"
      "STP x5, x6,[sp, #32]\n"
      "STP x7, x8,[sp, #48]\n"
      "STP x9, x10,[sp, #64]\n"
      "STP x11, x12,[sp, #80]\n"
      "STP x13, x14,[sp, #96]\n"
      "STP x15, x16,[sp, #112]\n"
      "STP x17, x18,[sp, #128]\n"
      "STP x19, x20,[sp, #144]\n"
      "STP x21, x22,[sp, #160]\n"
      "STP x23, x24,[sp, #176]\n"
      "STP x25, x26,[sp, #192]\n"
      "STP x27, x28,[sp, #208]\n"
      "STP x29, x30,[sp, #224]\n");

      TrigHigh(GpioData);
```

```
51
52          asm volatile( "ldr x0, =0xCA0EBABE\n"
53              "ldr x1, =0xCA1EBABE\n"
54              "ldr x2, =0xCA2EBABE\n"
55              "ldr x3, =0xCA3EBABE\n"
56              "ldr x4, =0xCA4EBABE\n"
57              "ldr x5, =0xCA5EBABE\n"
58              "ldr x6, =0xCA6EBABE\n"
59              "ldr x7, =0xCA7EBABE\n"
60              "ldr x8, =0xCA8EBABE\n"
61              "ldr x9, =0xCA9EBABE\n"
62              "ldr x10, =0xCAAEBABE\n"
63              "ldr x11, =0xCABEBABE\n"
64              "ldr x12, =0xCACEBABE\n"
65              "ldr x13, =0xCADEBABE\n"
66              "ldr x14, =0xCAEEBABE\n"
67              "ldr x15, =0xCAFEBABE\n"
68              "ldr x16, =0xCB0EBABE\n"
69              "ldr x17, =0xCB1EBABE\n"
70              "ldr x18, =0xCB2EBABE\n"
71              "ldr x19, =0xCB3EBABE\n"
72              "ldr x20, =0xCB4EBABE\n"
73              "ldr x21, =0xCB5EBABE\n"
74              "ldr x22, =0xCB6EBABE\n"
75              "ldr x23, =0xCB7EBABE\n"
76              "ldr x24, =0xCB8EBABE\n"
77              "ldr x25, =0xCB9EBABE\n"
78              "ldr x26, =0xCBAEBABE\n"
79              "ldr x27, =0xCBBEBABE\n"
80              "ldr x28, =0xCBCEBABE\n"
81              "ldr x29, =0xCBDEBABE\n"
82              "ldr x30, =0xCBEEBABE\n"
83              );
84
85          //10000 ADD instructions
86          asm volatile(
87              "MOV x0, #0\n\t"
88              T10000);
89
90          //Restore Regs
91          asm volatile( "LDP x1 , x2, [sp] \n"
92              "LDP x3, x4,[sp, #16] \n"
93              "LDP x5, x6,[sp, #32] \n"
94              "LDP x7, x8,[sp, #48] \n"
95              "LDP x9, x10,[sp, #64] \n"
96              "LDP x11, x12,[sp, #80] \n"
97              "LDP x13, x14,[sp, #96] \n"
98              "LDP x15, x16,[sp, #112] \n"
99              "LDP x17, x18,[sp, #128] \n"
100             "LDP x19, x20,[sp, #144] \n"
101             "LDP x21, x22,[sp, #160] \n"
102             "LDP x23, x24,[sp, #176] \n"
103             "LDP x25, x26,[sp, #192] \n"
104             "LDP x27, x28,[sp, #208] \n"
105             "LDP x29, x30,[sp, #224] \n"
106             "ADD sp, sp , #240 \n");
107
108
109
110         //Move result
111         asm volatile(
112           "MOV %[result], x0 \n\t"
113           :[result] "=r" (res));
114
115         TrigLow(GpioData);
116
117         printf("0x2710 = %.8x,",res); //10.000
118         //printf("0x32c8 = %.8x,",res);
119         printf("END OF CODE\n");
120         break;
121       case(0x62): //'b'
```

```
122         printf("LoopASM,");
123 
124       //BACKUP REGS
125     asm volatile( "SUB sp, sp , #240 \n"
126         "STP x1 , x2 ,[sp] \n"
127         "STP x3, x4,[sp, #16]\n"
128         "STP x5, x6,[sp, #32]\n"
129         "STP x7, x8,[sp, #48]\n"
130         "STP x9, x10,[sp, #64]\n"
131         "STP x11, x12,[sp, #80]\n"
132         "STP x13, x14,[sp, #96]\n"
133         "STP x15, x16,[sp, #112]\n"
134         "STP x17, x18,[sp, #128]\n"
135         "STP x19, x20,[sp, #144]\n"
136         "STP x21, x22,[sp, #160]\n"
137         "STP x23, x24,[sp, #176]\n"
138         "STP x25, x26,[sp, #192]\n"
139         "STP x27, x28,[sp, #208]\n"
140         "STP x29, x30,[sp, #224]\n");
141 
142         TrigHigh(GpioData);
143 
144       asm volatile( "ldr x0, =0xCA0EBABE\n"
145           "ldr x1, =0xCA1EBABE\n"
146           "ldr x2, =0xCA2EBABE\n"
147           "ldr x3, =0xCA3EBABE\n"
148           "ldr x4, =0xCA4EBABE\n"
149           "ldr x5, =0xCA5EBABE\n"
150           "ldr x6, =0xCA6EBABE\n"
151           "ldr x7, =0xCA7EBABE\n"
152           "ldr x8, =0xCA8EBABE\n"
153           "ldr x9, =0xCA9EBABE\n"
154           "ldr x10, =0xCAAEBABE\n"
155           "ldr x11, =0xCABEBABE\n"
156           "ldr x12, =0xCACEBABE\n"
157           "ldr x13, =0xCADEBABE\n"
158           "ldr x14, =0xCAEEBABE\n"
159           "ldr x15, =0xCAFEBABE\n"
160           "ldr x16, =0xCB0EBABE\n"
161           "ldr x17, =0xCB1EBABE\n"
162           "ldr x18, =0xCB2EBABE\n"
163           "ldr x19, =0xCB3EBABE\n"
164           "ldr x20, =0xCB4EBABE\n"
165           "ldr x21, =0xCB5EBABE\n"
166           "ldr x22, =0xCB6EBABE\n"
167           "ldr x23, =0xCB7EBABE\n"
168           "ldr x24, =0xCB8EBABE\n"
169           "ldr x25, =0xCB9EBABE\n"
170           "ldr x26, =0xCBAEBABE\n"
171           "ldr x27, =0xCBBEBABE\n"
172           "ldr x28, =0xCBCEBABE\n"
173           "ldr x29, =0xCBDEBABE\n"
174           "ldr x30, =0xCBEEBABE\n"
175           );
176 
177 
178         //1000 ADD instructions
179         asm volatile(
180             "MOV x0, #0\n"
181           "loop:  ADD x0, x0, #1\n"
182             "CMP x0, #1, lsl #12\n"
183             "BNE loop\n");
184 
185 
186         //Restore Regs
187         asm volatile( "LDP x1 , x2, [sp] \n"
188             "LDP x3, x4,[sp, #16] \n"
189             "LDP x5, x6,[sp, #32] \n"
190             "LDP x7, x8,[sp, #48] \n"
191             "LDP x9, x10,[sp, #64] \n"
192             "LDP x11, x12,[sp, #80] \n"
```

```
193             "LDP x13, x14,[sp, #96] \n"
194             "LDP x15, x16,[sp, #112] \n"
195             "LDP x17, x18,[sp, #128] \n"
196             "LDP x19, x20,[sp, #144] \n"
197             "LDP x21, x22,[sp, #160] \n"
198             "LDP x23, x24,[sp, #176] \n"
199             "LDP x25, x26,[sp, #192] \n"
200             "LDP x27, x28,[sp, #208] \n"
201             "LDP x29, x30,[sp, #224] \n"
202             "ADD sp, sp , #240 \n");
203
204         //4096 ADD instructions
205         asm volatile(
206             "MOV %[result], x0\n"
207             //"MOV %[result2], x23\n"
208             :[result] "=r" (res));
209
210         TrigLow(GpioData);
211
212         printf("0x1000 = %.8x,",res);
213         //printf("0x28000 = %.8x,",res2);
214
215         printf("END OF CODE\n");
216         break;
217       case(0x63): //'c'
218         printf("LoopC,");
219
220         volatile int i;
221         volatile int res = 0;
222
223           //BACKUP REGS
224     asm volatile( "SUB sp, sp , #256 \n"
225         "STP x0 , x1 , [sp] \n"
226         "STP x2, x3,[sp, #16]\n"
227         "STP x4, x5,[sp, #32]\n"
228         "STP x6, x7,[sp, #48]\n"
229         "STP x8, x9,[sp, #64]\n"
230         "STP x10, x11,[sp, #80]\n"
231         "STP x12, x13,[sp, #96]\n"
232         "STP x14, x15,[sp, #112]\n"
233         "STP x16, x17,[sp, #128]\n"
234         "STP x18, x19,[sp, #144]\n"
235         "STP x20, x21,[sp, #160]\n"
236         "STP x22, x23,[sp, #176]\n"
237         "STP x24, x25,[sp, #192]\n"
238         "STP x26, x27,[sp, #208]\n"
239         "STP x28, x29,[sp, #224]\n"
240         "STR x30 , [sp ,#240] \n");
241
242         TrigHigh(GpioData);
243
244         //Init Regs
245         asm volatile( "ldr x0, =0xCA0EBABE\n"
246             "ldr x1, =0xCA1EBABE\n"
247             "ldr x2, =0xCA2EBABE\n"
248             "ldr x3, =0xCA3EBABE\n"
249             "ldr x4, =0xCA4EBABE\n"
250             "ldr x5, =0xCA5EBABE\n"
251             "ldr x6, =0xCA6EBABE\n"
252             "ldr x7, =0xCA7EBABE\n"
253             "ldr x8, =0xCA8EBABE\n"
254             "ldr x9, =0xCA9EBABE\n"
255             "ldr x10, =0xCAAEBABE\n"
256             "ldr x11, =0xCABEBABE\n"
257             "ldr x12, =0xCACEBABE\n"
258             "ldr x13, =0xCADEBABE\n"
259             "ldr x14, =0xCAEEBABE\n"
260             "ldr x15, =0xCAFEBABE\n"
261             "ldr x16, =0xCB0EBABE\n"
262             "ldr x17, =0xCB1EBABE\n"
263             "ldr x18, =0xCB2EBABE\n"
```

```
264             "ldr x19, =0xCB3EBABE\n"
265             "ldr x20, =0xCB4EBABE\n"
266             "ldr x21, =0xCB5EBABE\n"
267             "ldr x22, =0xCB6EBABE\n"
268             "ldr x23, =0xCB7EBABE\n"
269             "ldr x24, =0xCB8EBABE\n"
270             "ldr x25, =0xCB9EBABE\n"
271             "ldr x26, =0xCBAEBABE\n"
272             "ldr x27, =0xCBBEBABE\n"
273             "ldr x28, =0xCBCEBABE\n"
274             //"ldr x29, =0xCBDEBABE\n"
275             "ldr x30, =0xCBEEBABE\n"
276             );
277
278         //1000 ADD instructions
279         for(i = 0; i < 1000; i++){
280           res++;
281         }
282
283         //Restore Regs
284         asm volatile( "LDP x0 , x1,[sp] \n"
285             "LDP x2, x3,[sp, #16] \n"
286             "LDP x4, x5,[sp, #32] \n"
287             "LDP x6, x7,[sp, #48] \n"
288             "LDP x8, x9,[sp, #64] \n"
289             "LDP x10, x11,[sp, #80] \n"
290             "LDP x12, x13,[sp, #96] \n"
291             "LDP x14, x15,[sp, #112] \n"
292             "LDP x16, x17,[sp, #128] \n"
293             "LDP x18, x19,[sp, #144] \n"
294             "LDP x20, x21,[sp, #160] \n"
295             "LDP x22, x23,[sp, #176] \n"
296             "LDP x24, x25,[sp, #192] \n"
297             "LDP x26, x27,[sp, #208] \n"
298             "LDP x28, x29,[sp, #224] \n"
299             "LDR x30 , [sp ,#240] \n"
300             "ADD sp, sp , #256 \n");
301
302         TrigLow(GpioData);
303
304         printf("0x3E8 = %.8x,",res);
305
306         printf("END OF CODE\n");
307         break;
308       case(0x64):
309         //Togle the caches.
310         asm volatile( "MRS x0, SCTLR_EL3 \n"
311             "MOV %[SysRegB], x0 \n"
312             "EOR x0 , x0 , #(1<<12) \n" //Flip Cache bit to zero to turn off cache
313             "MSR SCTLR_EL3 , x0 \n"
314             "MRS x0, SCTLR_EL3 \n"
315             "MOV %[SysRegA], x0 \n"
316             :[SysRegB] "=r" (SysRegB),[SysRegA] "=r" (SysRegA)::"x0");
317
318         printf("Toggle cache = %x\n",((SysRegA >> 12) & 1));
319         break;
320
321       case(0x65):
322         //Trigger manually
323         delay = 0;
324
325         printf("Glitch Manually\n");
326         TrigHigh(GpioData);
327         //delay
328         for(j = 0; j < 200; j++){
329           delay++;
330         }
331         TrigLow(GpioData);
332         break;
333       default:
334         printf("Wrong choice\n");
```

```
335
336         }
337
338     }
```