UNIVERSITY OF AMSTERDAM

Faculty of Physics, Mathematics and Informatics
Graduate School of Informatics
System and Network Engineering MSc

Proving the wild jungle jump

Research Project 2

James Gratchoff
james.gratchoff@os3.nl

July 8, 2015

Abstract

Fault injection has been proven to be a successful attack vector against secure systems and algorithms. It induces faults that are logically not possible in software. This paper presents results that prove the feasibility of corrupting the program counter of the processor using power fault injection. The likelihood of the fault model is also expressed. The tests are performed on a widespread processor, namely the ARM Cortex A9. The tests show that by introducing a precise drop of power in the voltage line of the processor, assembly instructions can be skipped or corrupted. This behaviour can cause jumps in memory space and can result in code execution controlled by an attacker. This kind of attack can be applicable against secure devices that are for example implementing secure boot.

Contents

Chapter 1

Introduction

Fault injection attacks are known to be a successful and cheap way to attack embedded systems [12, 14]. Fault injection involves two phases: the injection, that is the attack vector, where the attacker deliberately injects an interference in the integrated circuit (IC) and the monitoring that proves whether the attack is effective or not. The goal is to change the behaviour of the IC's operation by using different injection techniques without destroying the device. The usual purpose of fault injection is to bypass or corrupt security mechanisms. The fault injection techniques differ in their domain space, however they all require having access on the IC. These common techniques are:

1. **Power fault injection**: Introducing unattended glitches in the power supply to change the behaviour of the IC operation.
2. **Clock fault injection**: Introducing shorter clock frequency cycles than the target ones to shorten the cycles and change the behaviour of the IC operation.
3. **Optical fault injection**: Introducing light (i.e. laser) on a specific area of the IC to change its behaviour.
4. **Electromagnetic fault injection** : Introducing magnetic pulses using a probe on a specific area of the IC to change its behaviour.
5. **Temperature fault injection**: Introducing temperature difference on a specific area of the IC to change its behaviour.

Fault injections or glitches can be detected but are hard to rule out completely. However, embedded software can be hardened against fault injection attacks (e.g. checksums, variable redundancy) and hardware can be designed as a countermeasure (e.g. voltage/frequency detectors, active shields)[9, 13]. Hardware manufacturers are more and more interested in testing their hardware against this kind of attack.

This research is focused on power fault injection and intends to prove if it is possible to corrupt the CPU's program counter (PC) in such a fashion that it points to an arbitrary address. This attack vector will be defined as a wild jungle jump in the rest of this paper. The purpose of such an attack is to run arbitrary attacker code on a secure device.

1.1 Research question

Overall discussion of the significance and motivation for this project resulted in the following research question:

What is the feasibility of a wild jungle jump?

This research question can be subdivided into several subquestions.

1. *How can the PC be corrupted?*
2. *What is the likelihood of a glitch corrupting the PC?*
3. *What are the repercussions of a corrupted PC?*

Chapter 2

Scope and contribution

Due to time constraints, the scope of the project had to be carefully defined. The project is targeting a Wandboard Solo [8] that integrates a Freescale i.MX6 Solo processor with an Cortex-A9 single core (ARM). This board has been chosen for its processor that can be found in many embedded devices (e.g Smart phones, smart reader). The project is restricted to the ARM architecture and other architectures (e.g. x86, AMD64) are left out of the scope. The project has a practical approach and is only tackling one type of fault injection, that is the power fault injection.

The project's main contribution is proving the possibility of corrupting the PC in a controllable way. This paper presents a fault attack that allows running arbitrary code on an embedded device by modifying the program counter. Then the author presents the likelihood of successfully performing a power FI attack.

Chapter 3

Related work and background

From the early nineties, fault injection was known to be an effective and cheap attack vector for example for smart cards and embedded systems. In the last ten years an increase in this field of research has been observed. Indeed as system's security becomes more and more advanced, attackers always tries to find new attack vectors. Fault injection is highly dependent on the targeted hardware. This section describes the previous research related to fault injection on similar infrastructure and describes in details the targeted ARM architecture.

3.1 Fault injection on ARM

Some research has been done regarding fault injection on ARM based embedded devices. Barenghi et al. [10] were the first to perform a successful power fault injection attack on a cryptosystem running on a complete operating system.

The processor targeted in this research was an ARM9E (ARMv5TE). By applying techniques known in open-literature, they describe the errors induced in the computation that lead to retrieve all AES round keys, similar results were obtained when combining a fault injection with a RSA ciphertext attack. Prior to performing their attack they tested the effectiveness of their power fault injection setup by attempting to infer a computation running on the target. They deduced that the instructions prone to power fault injection attack were the memory operations. On the other hand, arithmetical and branch operations were not affected by the power FI. They explained these results due to the low capacitance design of the CPU registers; which compensate for the slowdown introduced by the glitch. This research also highlighted that the load operation was the only memory operation to report faulty results. The store operation was not affected by the glitches due to the buffer present in the bus between the CPU and memory; this buffer helps the performing of a correct write operation in memory by reducing the capacitive load. Indeed memory instructions are transferred through the bus interface; this bus being more power expensive it tends to be more sensitive to the underfeeding in voltage than other instructions.

However this does not mean that other instructions are not prone to power fault injection. Moreover on normal execution the processor does not always transfer the operations between CPU and memory as compilers can optimize the code to limit this expensive transfer.

Riviere et al [14], successfully performed an electromagnetic fault injection (EMFI) on Cortex-M (ARMv7M) architectures. They created an attack targeting the cache instruction that, with the right FI parameters, was highly reproducible and inducing a fault in up to 96 % of cases. This attack was also shown effective against cryptographic algorithms. Finally, another EMFI was successfully performed by Thessalonikefs [16], on the same architecture targeted in this research (Cortex-A9 (ARMv7-A)). Thessalonikefs's attack resulted into instruction skipping, MMU exceptions (that lead to a reset) and wrong value on the output register.

3.2 ARM architecture

The targeted ARM infrastructure is a 32 bit Cortex-A9 MPCore microprocessor. This processor is based on a RISC infrastructure. Due to its presence in many embedded devices such as the iPhone 4S, Apple TV, or the Samsung Galaxy SIII, the Cortex-A9 is an interesting target for this research.

3.2.1 Registers

The processor is based on a RISC infrastructure and has 37 registers arranged in several banks (User, FIQ, IRQ, Supervisor, Abort, Undefined and System [4]). These banks are determined by the processor mode and this mode changes if an exception occurs. The bank that is the point of interest in this research is the user bank, where most tasks run. This bank implements 17 registers of 32 bit word size to perform operations. These registers have different name and purposes. R0 to R12 corresponds to the general purpose register used to hold information that is used by the processor to perform operations. R13 is the stack pointer, that, as its name implies, points to an address on the stack. It is 32 bit word aligned. R14 is referenced as the Link Register that points out what the last instruction processed was. The last 15 registers described are specific to the user bank. R15 is the program counter (PC), that points to the processor which memory address to execute next and thus determines what the flow of instruction for the processor to execute is. The PC is also word aligned and is used by all banks. If corrupted with another address, the PC will effectively jump and the processor will start executing from this address; this jump is the target of the research. Finally, R17 is the Program Status Register (PSR) that holds information about the processor while running to report errors in case of crash.

3.2.2 Memory

The memory can be seen as two components, the memory and the bus (AMBA AXI). This bus permits access to (store or load) data into memory. Operations related to memory are more expensive computationally and in power than, for

example, arithmetic operations. The Cortex-A9 accesses memory using two caches (L1 or L2) that help translating virtual addresses to physical addresses and speed the memory access. The instruction and data caches are separated [1].

3.2.3 Instructions

In the targeted ARM architecture, the instructions are divided into several types:

- **Memory** (e.g. load (LDR), store (STR))
- **Arithmetic** (e.g. add (ADD), substitute (SUB))
- **Logical instructions** (e.g. move (MOV), and (AND), or (OR))
- **Flow control** (i.e. branch with link (BL))

The Cortex-A9 implements a variety of instructions that are 32 bits long, the decoding and meaning of these instructions can be found in appendix A. These instructions are architecture-dependent. When an instruction is received by the processor it is first fetched, decoded, executed, then a memory access might be required and finally the write back process is performed. This sequential flow is respecting the von-Neumann model [7]. ARM architectures are able to do pipelining, meaning that instructions can be processed concurrently or pre-computed.

Chapter 4

Test environment and approach

This section describes the test environment setup and the methodology for the experiments.

4.1 Test environment

The required resources for this project have been provided by Riscure [6]. It includes the Wandboard Solo and some hardware to perform fault injection such as:

1. **Picoscope 5203** - Digital oscilloscope used in this research to monitor the power consumption of the code instructions to determine the correct glitch boundaries.
2. **Riscure Glitch Amplifier** - Amplifier targeting embedded systems that requires a more powerful glitch. It is used to amplify the glitch sent by the VC Glitcher.
3. **Riscure VC Glitcher** - Precise and repeatable glitch generator.

The list of software required to carry out this project is:

1. **Picoscope 6.0** - Oscilloscope software used to output the results of the voltage monitoring.
2. **Riscure Inspector FI 4.8.3** - Software to define parameters and program the VC Glitcher.
3. **Riscure FI GraphIt 1.0** - Live and Post FI Perturbation Visualization Software

The setup is as presented in figure 4.1. The FI target is a Wandboard that is running Linux with the Archlinux distribution [3] and the U-boot bootloader [5]. The VC Glitcher is connected to the Glitch Amplifier that is itself hooked to the target board voltage input (Vcc). The VC Glitcher is connected to the reset pin of the Wandboard to have control over the board if a reset is needed

during the tests. The VC Glitcher analog glitch line and the Wandboard trigger line are connected to the Picoscope to monitor the Wandboard's triggers and the glitch introduced in the IC. Finally, the board is itself connected to the workstation with a serial cable where Inspector, FI GraphIt and Picoscope are running.

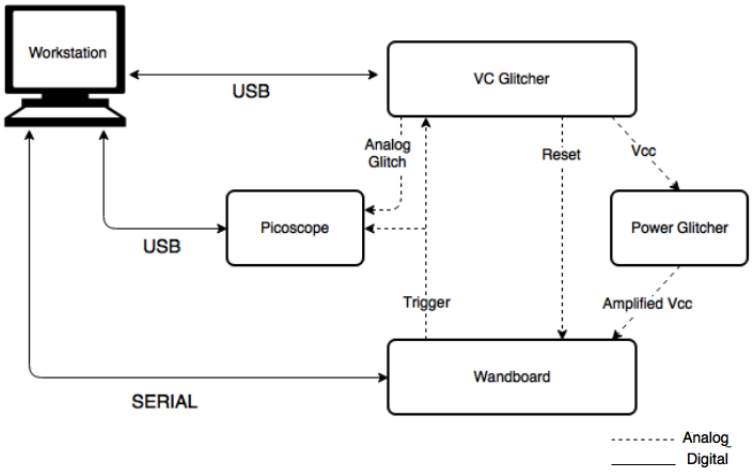

Figure 4.1: Schematic overview of the test environment

4.2 Methodology

In order to determine the feasibility of a wild jungle jump, it is essential to determine an approach for the research. The first step of the research is to get familiar with the power fault injection tools and techniques. Following this, assumptions where a wild jungle jump can be feasible are made. Then assembly code is implemented to allow the testing of these assumptions. Once the code is implemented and flashed on the Wandboard, it is to be tested using different glitch parameters. Finally, the output of these tests are analysed to find the right fault injection parameters (e.g. time, glitch voltage, glitch offsets). The outcome of the research is to determine possible situations (i.e. code constructs) where a wild jungle jump is possible. An attempt at establishing a relationship between the fault injection parameters and the code constructs is made.

4.2.1 Glitch generation

The glitch is the cause of the PC corruption. It is dependent on several parameters that need to be finely tuned to allow increasing the likelihood of such IC behaviour. These parameters needed to be tuned are detailed below and are represented in a schematic form in figure 4.2 :

1. Glitch voltage: power pulse or drop injected in the IC (Measured in Volts).
2. Glitch offset: Time to wait after a clock cycle phase before sending a glitch (in microseconds).

3. Glitch cycles: The number of cycles targeted by the glitch.
4. Wait cycles: The number of cycles to wait before sending a glitch.
5. Glitch length: Length of the power pulse injected in the IC (in microseconds).

The fault injection inserted in the IC is expected to be tuned in such a fashion that the chances of corrupting the PC are higher. In order to find the best parameters for the targeted instrumentation the glitch parameters will be defined in a certain range and swept through using a random pattern using Inspector. This method has proven to be one of the most successful [11]. These testings will allow to determine where the PC corruption is more likely to happen. Then the parameters will be narrowed to increase the success rate.

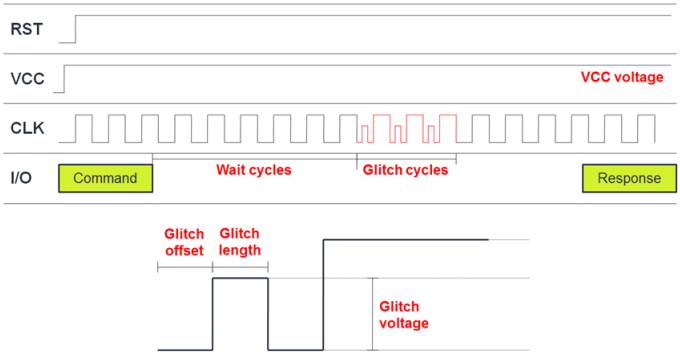

Figure 4.2: Representation of the glitch parameters needed to perform a power fault injection

4.2.2 Determining the result of the glitch

An inspector module, written in java, is the core of the FI test. It sends the command to the board (through the serial connection), performs the glitch using the defined parameters and outputs the result in a SQLite database. It also allows for control of the board in case it is not responding (i.e. boot loop) and sends a reset to the board.

To analyse the result of the tests, the written java module differentiates the expected result of the glitch from the expected result of the instrumentation, the resets and the abnormal behaviour of the processor (e.g. data abort, undefined instruction). These selections allow for an overview of the test and enable better understanding of the processor behaviour, depending on the parameters set for the test. The module outputs these results with an associated color to a SQLite database that is then crawled by the FI GraphIt tool to produce readable graphs and enables analysis of the database file in a readable manner.

4.2.3 Possible situation of corrupted PC

A program counter can be corrupted in many manners, however the likelihood for it to happen is highly dependent on the processor infrastructure, the glitch parameters and the present code. Assumptions of the most probable ways of corrupting a PC are presented in this section.

Regarding the targeted RISC architecture, the most probable manner of corrupting the PC would be by corrupting an instruction in the assembly code. By doing so, a glitch can modify an instruction in such a manner that, for example, the destination register used in a MOV instruction would be modified from a normal register (R0-R12) to the PC register, resulting in the PC being modified by another value. If this value is a pointer, the processor will thus move to this address and execute the following instructions present at that address. An assembly example of the wanted behaviour is presented in figure 4.3.

```
LDR R0, =memory address | LDR R0, =memory address
MOV R1, R0              | MOV PC, R0
```

Figure 4.3: Representation of the PC corruption by modifying an instruction. (Left: instructions before the glitch, Right: instructions affected by the glitch)

Another way of corrupting the PC would be to skip a set of instructions that would lead to a part of the code executed on the target behaving differently. This case is highly dependent on the way the compiler produces the assembly code and how the linker is assembling the functions together. An attempt at finding examples where this sequence corruption can happen will be made, in assembly output from the compiler. An assembly example of the wanted behaviour is presented in figure 4.4.

```
push{r0}                | push{r0}
MOV r0, =memory address | MOV r0, =memory address
pop {r0}                | pop {r1}
pop {pc}                | pop {pc}
```

Figure 4.4: Representation of the PC corruption by skipping an instruction. (Left: instructions before the glitch, Right: instructions affected by the glitch)

4.2.4 Instrumentation and expected behaviour of the glitch

The code executed on the target should be carefully designed to remove possible false positives and increase the likelihood of a wild jungle jump. This section describes the different codes implemented in this research in order to prove possible ways of a wild jungle jump. The code has been written in ARM assembly to control its execution flow in the processor and be sure the compiler does not introduce code optimization. Finally, synchronization is needed between the embedded system and the glitching platform, which is explained in the initialization section.

Initialization

In order for the platform to detect the critical section targeted by the glitch synchronisation is needed. To allow this the section is delimited by power triggers using gpio output. The triggers can go from a high power line to a lower power line (or vice verca), one is set before the critical section and is then restored to its initial value after it using a second one. These triggers are used as a time synchronization point between the target and the attacker. As the trigger is not immediate and takes some micro seconds to reach its lower power line, another initialization is needed to compensate for this time in the critical section (See delta in figure 4.5). To compensate, some no operations (NOP) are computed before entering the critical section, permitting the glitching hardware to be synchronized and ready to glitch.

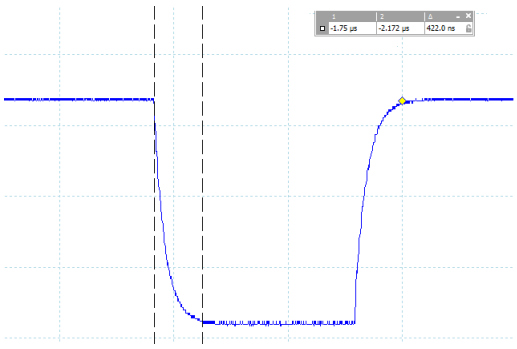

Figure 4.5: Trigger delay measurement

Instruction skip characterization

The first instrumentation intends to prove the possibility of skipping instructions with power fault injection. The goal is to characterize if instructions can be skipped by introducing a glitch without disrupting the execution of the code. The critical section in this implementation is an unrolled loop that increments a counter. The end of the critical section returns the counter and a recognizable string is printed when finished. The length of the skip can be then deduced by subtracting the value of the counter from the length of the loop.

Gluing functions using instructions skip

The second test intends to prove the possibility of gluing functions together by skipping a set of instructions. The goal of this instrumentation is to skip the end and the start of two functions that are closely located in memory. This model intends to prove that old values present in the registers in the first function can be reused in the second one. To prove such fault a precise number of instructions need to be skipped. The target of the glitch is to skip the register flush and set and thus to execute an address that has been loaded into a register in the first function.

If this test results in proving the model to be successful, an attempt will be made to find a real sequence that could be corrupted in compiled open source code. If an instruction is found to be able to be corrupted or skipped, the project will have a real life implementation. The purpose of this is to find a possible weakness proving the research target on a code that is already compiled and used. It is important to mention that the reason for which the author focuses on already compiled code is that the assembly code produced depends on the compiler (version or chain) used. Using an already compiled code removes the problem of the code being compiler dependent.

Instruction corruption characterization

The second set of instrumentations was implemented in such a way as to prove the likeliness of a instruction being corrupted, by replacing in the destination field a general purpose register (R0-R12) by the PC register (R15). To confirm a successful glitch, a payload function returning an identifiable string when called is inserted in a farther part of memory. The main function targeted by the glitch is not calling this function but is performing a variety of MOV operations between the registers that hold the location of this function. To call the payload function, the location of this function should be loaded into the PC and that is the expected fault model for the glitch. An example representing the desired behaviour is presented in figure 4.6. In this example the expected behaviour after a successful glitch would be to change the value of R1 into the PC so it jumps to another location in memory. Another more realistic test using this function was performed with a single MOV operation and will be the final target of this fault model.

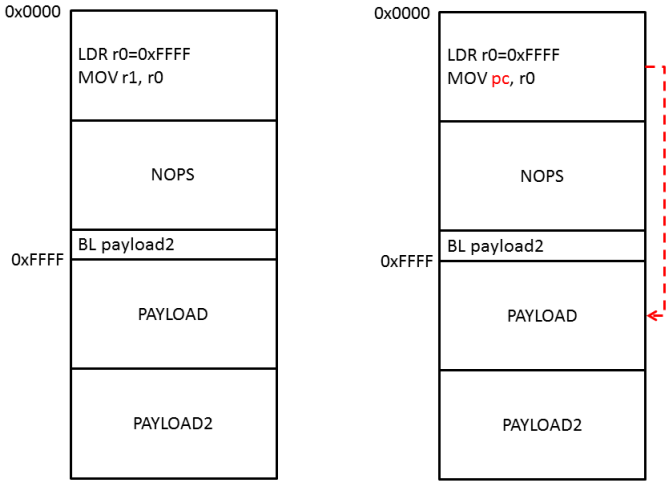

Figure 4.6: Schematic representation of the code behaviour when successfully glitched. (Left: code in normal usage, Right: code affected by the glitch)

It is important to mention that when linked, functions are more likely to end up close to each other. As glitches are also likely to skip a certain number of lines, it is necessary to move the payload function to a further location in memory so the tests will not result in false positives. To do so a consequent number of NOP (no operations) were added between the payload and the executed function. The code was also adapted to distinguish if the result of a successful glitch is due to a skip or an instruction corruption. The memory was laid out in such a manner that if a skip happened the processor would result in going to the second payload function, located under the first one. To do so, a branch with link (BL) pointing towards the second payload was placed after the end of the critical section and before the first payload function. If the processor jumped to the second payload function it would then mean that a sequence had been skipped or the value of the register holding the address had been modified by the glitch to an address located between the critical section and the BL to the second payload function. A representation of the memory layout is presented in figure 4.7.

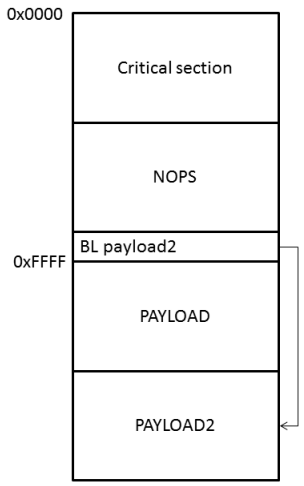

Figure 4.7: Schematic representation of the memory layout

Corrupting the PC during a memory copy

As explained by Barenghi et al [10], fault injections are more likely to be successful on memory operations and more precisely on load instructions due to the fact that these instructions are more power expensive and thus more prone to fault injections. The expected behaviour of this fault model is the same as in the instrumentation discussed above. The implemented code should be relevant to normal use cases and, as most assembly code is produced using a compiler, the results of this research would be more meaningful if the tested instructions were likely to be produced by this compiler. The second instrumentation is then implementing a memory copy. This memory copy will be referred to as memcpy in the rest of the paper. This type of function was chosen due to its operations.

A memcpy meets the requirements for the fault models targeted in the instruction corruption as indeed it uses a source and a destination memory address as an input and performs a load (LDR) and store (STR) instruction to copy from flash/memory to memory. In order to increase the likelihood of a successful glitch, the address being copied is an array of pointers to the attackers code. This means that every word copied, containing the attackers code address, will be a single load instruction. A schematic representation of the instrumentation and its expected behaviour after a successful glitch is represented in figure 4.8.

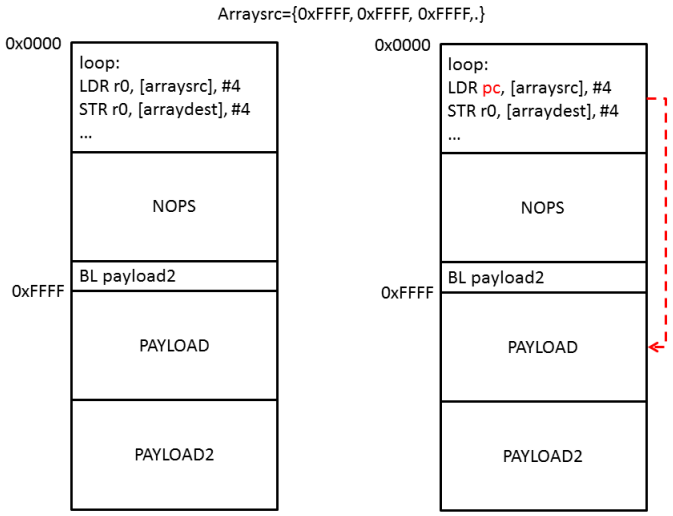

Figure 4.8: Schematic representation of the LDR corruption behaviour when successfully glitched. (Left: code in normal usage, Right: code affected by the glitch)

Chapter 5

Results and Analysis

This section analyses the results of the test performed on different code implementations and tries to explain their causes and consequences. The results are summarized into a possible attack model. The likelihood of every fault model being accurate is also expressed. It is essential to mention that in fault injection any fault model that has a likelihood that is not null means that it is successful. This likelihood can be expressed in percentage of success. The likelihood of finding exploitable code in open source implementations is also expressed.

5.1 Glitch influence

The following section describes the influence of the glitch on the different instrumentations produced and tries to derive conclusions on the cause of these glitches.

5.1.1 Instruction skip

Characterization

In order to prove the feasibility of instruction skips on the targeted environment, the critical section targeted by the power FI is an unrolled loop that increments a counter at every instruction. The results of this test have proven the feasibility of such a fault model as indeed after a successful glitch the counter, incremented at every instruction, reported a value lower than the length of the loop. The number of skipped instructions can be of different amount, ranging from one to the full length of the loop tested (ten thousand instructions). This test does not prove that it is the largest instruction amount that a glitch can skip without stopping the execution flow of the processor (e.g. reset, data abort, misalignment). This behaviour is explained by the fact that the glitch of power in the processor is enough to skip one or a set of instructions and lower than the threshold that leads to an exception handle or reset of the board. The likelihood of such a fault is high and its percentage of success is of 45%.

Gluing functions and fault model likeliness in compiled code

The first goal of this instrumentation was to prove that instruction skip can be used to glue two functions together and that registers set in the first functions could be reused in another function. This fault model has been proven successful when a precise amount of instructions are skipped. In the instrumentation created the fault injection resulted in a wild jungle jump. The likelihood of such a fault is lower as the glitch is targeting a skip on a certain amount of instructions only. The percentage of success for this test is 0.01%. Due to time constraints, a second FI test with narrowed parameters has not been performed; thus this success rate could not be raised.

However this implementation is really specific; that is why attempts were made to find examples of such operations where an attacker could have control over the address in the first function in already compiled code. The goal was to find weaknesses in code without the chain dependency of the compiler. Despite an in depth code walkthrough in two related ARM code implementation (U-boot and busybox) no possible wild jungle jump has been found for this situation. This does not mean that this fault model could be reused for other purposes, such as to change the output value of a function by changing its register set.

5.1.2 Instruction corruption

Characterization: Moving a variety of register

The third test, that had for goal to corrupt an instruction, has proven the feasibility of corrupting a normal purpose destination register into the PC register during a MOV operation. During this test it was feasible to change the destination register to the PC, resulting in the value contained in the source register being loaded in the program counter. This glitched instruction resulted in the processor jumping to the target address held in the source register. The tests have been performed over a variety of registers sorted by destination register number. The successful glitches tend not be correlated with time, so it is thus safe to say that the glitch does not have a time dependency regarding the function. Which induces that the glitches are not more successful on different registers. The likelihood of creating such a fault is low however compilers often use this kind of instructions as they are less expensive than memory instructions. The success rate of this test was raised at most at 0.16%.

Moving one register

After moving towards a more realistic approach, with a single MOV instruction, the same results were observed as for the variety of registers. However an unusual behaviour was noticed as some registers are more prone to be changed into the PC register. The goal of the glitch is to flip/keep all bits of the destination registers at one. It resulted that as an example no successful glitches were noticed when the destination register was set to '0111' (R7), however successful glitches happened when the register was set to '0110' (R6). This behaviour could not be explained and could not be further researched due to time constraints. The success rate when using the R6 destination register was raised to 5%. This implementation has been proven more likely to happen as the time to

perform the function is reduced; it is more likely to introduce the correct glitch, thus flipping the correct bits at the right time.

A real life example of the instruction corruption model is present in the U-boot source code (Figure 5.1). This example has not been tested however, it would be likely to be exploitable and it would be relevant to many implementations. The function is a cyclic redundancy check (CRC). It does respond to the requirements as it loads into a register a value that the attacker has control of. This value can be for example data stored in flash. The other requirement is also respected as the function is moving this value into another register. However it does not implement a normal MOV but a MVN that corresponds to a normal move operation with a bitwise logical NOT operation on the value held in source. So if the malicious code is located at 0xFF00 (1111 1111 0000 0000) the computed CRC should be equal to 0x00FF (0000 0000 1111 1111). In order to execute the arbitrary code three steps would be required:

1. Locate the malicious code address.
2. Craft the data in such a way that the CRC computation results in the inverse of the address pointing to the malicious code.
3. Perform power FI on the CRC function to modify the R0 register to the PC register.

The success of the FI is not likely as the probability for the glitch to flip four bits from zero (R0) to one (PC) is low. However it is possible and this behaviour has been seen during tests.

```
; uint32_t __fastcall crc32(uint32_t crc, const Bytef *p, uInt len)
EXPORT crc32
crc32
crc = R0                ; uint32_t
p = R1                  ; const Bytef *
len = R2                ; uInt
STMFD           SP!, {R3,LR}
MVN             crc, crc ; crc
BL              crc32_no_comp
MVN             R0, R0
LDMFD           SP!, {R3,PC}
; End of function crc32
```

Figure 5.1: Code source of the CRC32 function

Targeting the memcopy

The goal of the memcopy implementation is to prove the feasibility of an instruction corruption on a load operation. It results that a load operation can also be the target of a instruction corruption where the destination address is set to the PC. The result of this instruction corruption is the same behaviour as the instruction on a MOV operation where a wild jungle jump is performed. The success rate for this test is of 3.4%. The test showed that the likelihood for this corruption to happen is low. Similar code can be found in most Unix-like

system. The memcopy gives the opportunity to the attacker to have control over the source and destination. In the attack performed the source is an array containing multiple time the start address value of the attacker's code. By copying this value from source to destination it is possible to introduce a glitch that modifies an instruction in such a way that the address value is loaded into the PC register; resulting in the execution of the attacker's code. This example proves that it can be possible to run arbitrary code on a secure device using FI. Moreover in the first step of secure boot implementation, memory copy are often used to copy from memory to flash.

5.2 Conclusion

The possibility of corrupting the program counter by skipping instructions has been proven to be a feasible attack vector. Combined with the right fault injection parameters this FI can result in gluing two functions together, resulting in the execution of a function with the registers value set in the previous function. To reproduce such an attack on different targets dependencies arise; as indeed this attack model is highly dependent on what the compiler does as any different version of compiler or chain will give a different assembly output. To prove the likelihood of such attack models in real life without these dependencies, some already compiled code, available in the open source community was studied. Situations where an attacker had control over a source that would result in a register and this register being reused in a following function could not be found.

The move (MOV) and load (LDR) operations have been proven successful attack vectors for a wild jungle jump as indeed when successfully glitched it is possible to jump to the targeted code by using fault injection. This situation is dependent on the code structure. The requirement for the code structure is that it implements the operation (MOV or LDR) with as a source a pointer address that the attacker has control of. An example of such a code is the CRC32 function or the memory copy case. These addresses or pointers are moved from memory to registers and it is from these instructions that the attack was designed.

Chapter 6

Conclusions

This research goal was to estimate the likelihood of a program counter corruption with the use of power fault injection on a low power computer that implemented a widespread processor (ARM Cortex-A9).

The results have shown that it is feasible to corrupt the PC in different ways, resulting in the execution of code located at a control address (by an attacker). This research also resumes the most probable way of corrupting the program counter using fault injection and explains the likelihood of the fault model presence in already compiled code.

The first fault model proven feasible is the instruction skip. Indeed a power glitch, introduced with the right parameters, can result in the skip of a single or a large amount of instructions. The success rate of the test that proved this model successful is of 45%. This fault model can be used to glue functions together resulting in old register value being executed by the processor. The success rate for the test proving this model successful is of 0.1%.

The second fault model proved that it is feasible to modify a single instruction in such a way that a normal purpose register, used to perform internal processor operation, can be modified to the PC. The success rates of the tests that proved this model successful is of 5% when corrupting a MOV instruction and 3.4% for the LDR instruction. This model has proven that if, in a function, an attacker has control over the source or destination address in memory it is possible to perform a wild jungle jump, resulting in the code located at the targeted address being executed. This situation could happen in the CRC32 U-Boot function or in the memory copy example that has been proven feasible in this research. The memcopy function can be coded in several ways to enhance performance and most of these different functions, presented in the ARM support knowledge articles [2], are also vulnerable to this fault model.

Finally code has been analyzed to find possible real life scenario where a set of instructions can be skipped in such a fashion that the previous value held in a register (preferably a controlled pointer) is moved into the program counter

instead of the skipped value. However, this situation could not be found in the examples analysed.

To conclude a wild jungle jump has been proven feasible using power FI in the test cases evoked above; however there is a low likelihood of reproducing such a fault on another device as it is highly dependent on the compiler (version or chain) used and requires having an understanding of the assembly code and the layout of the memory. Moreover finding the right parameters for the FI can be tedious as it requires long and precise testing.

Chapter 7

Future work

As future work the author would recommend proving that a wild jungle jump is also feasible on other architectures such as x86, MIPS or AMD. Another interesting topic that has not been investigated in this research is the countermeasures against the program counter corruption. Last but not least, the feasibility of a wild jungle jump could be proven doable using other FI techniques such as EMFI or optical FI.

Finally, fault injection is becoming a topical trend and is a feasible attack vector against embedded systems as proven in this research. That is why it is important to raise awareness about FI attacks and their applicability to secure devices.

Bibliography

[1] Cortex-A9 technical reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388f/Caccifbd.html, 2010.

[2] What is the fastest way to copy memory on a cortex-a8?, ARM technical support knowledge articles. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13544.html, 2011.

[3] Archlinux – a simple lightweight distribution. https://www.archlinux.org/, 2015.

[4] The arm instruction set. http://simplemachines.it/doc/arm_inst.pdf, 2015.

[5] Das U-Boot – the universal boot loader. http://www.denx.de/wiki/U-Boot, 2015.

[6] Riscure. https://www.riscure.com/, 2015.

[7] von-Neumann architecture. https://en.wikipedia.org/wiki/Von_Neumann_architecture, 2015.

[8] Wandboard. http://www.wandboard.org/, 2015.

[9] Naccache Tunstall & Whelan Bar-El, Choukri. The sorcerer's apprentice guide to fault attacks. 2004.

[10] Koren & Naccache Barenghi, Breveglieri. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. 2012.

[11] Batina Menarini Jakobovic Boix Carpi, Picek and Golub. Glitch it if you can: parameter search strategies for successful fault injection. http://cardis.sec.t-labs.tu-berlin.de/proceedings/CARDIS2013_16.pdf, 2013.

[12] Jovanovic & Polian. Kumar. Precise fault-injections using voltage and temperature manipulation for differential cryptanalysis. 2014.

[13] M. Oostdijk M. Witteman. Secure application programming in the presence of side channel attacks. 2008.

[14] Rauzy Danger Bringer & Sauvage. Riviere, Najim. High precision fault injections on the instruction cache of ARMv7-m architectures. 2015.

[15] Ryan. Decoding the arm instruction set. http://emucode.blogspot.nl/2010/09/decoding-arm-instruction-set.html, 2010.

[16] Thessalonikefs. Electromagnetic fault injection characterization. 2014.

Appendices

Appendix A

ARM instruction encoding

This table[15] describes how the instruction are encoded in ARM. The instruction are 32 bit long and have all a different purpose.

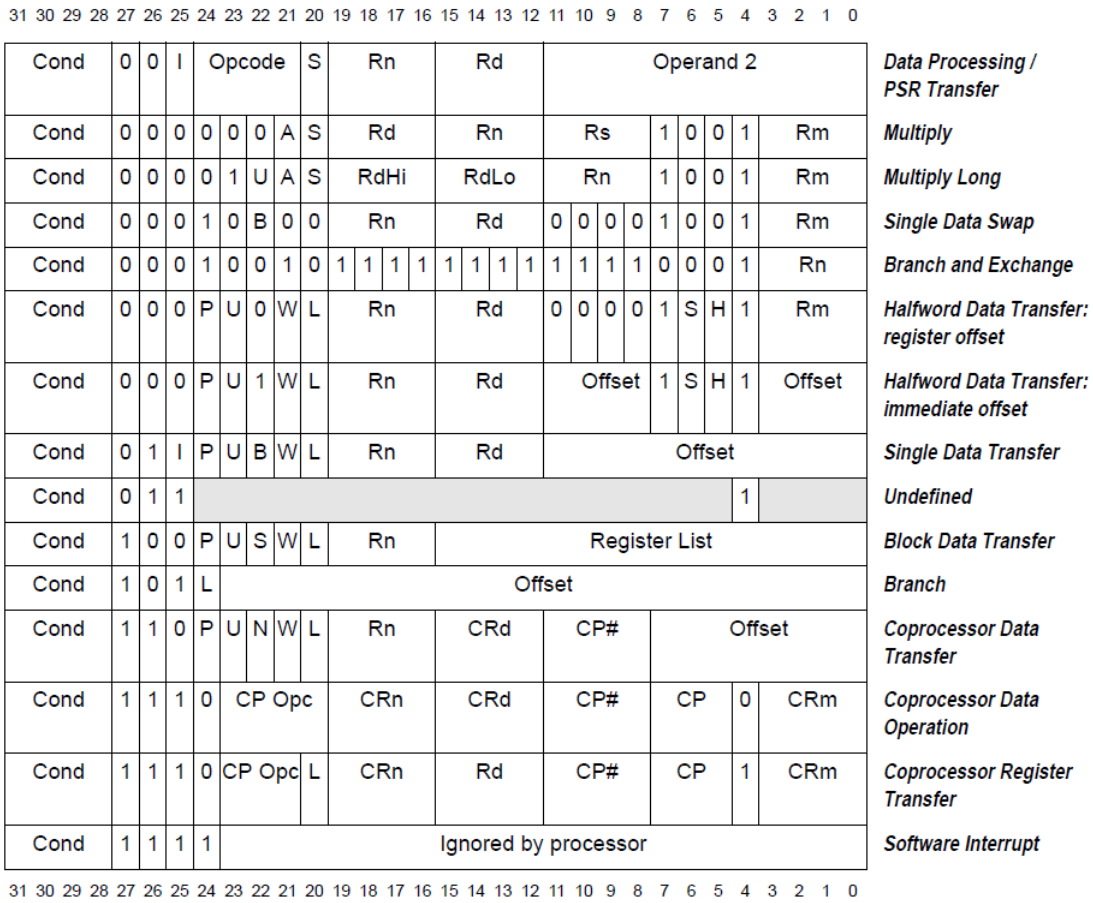

Figure A.1: ARM instruction set encoding