Fault injection attacks are a powerful technique to influence the intended behavior of embedded systems. They can be used to exploit or bypass robust security features found in secure embedded systems. Examples of such attacks include differential fault analysis (DFA) and bypassing authentication mechanisms. An embedded system’s authenticated boot chain (i.e. secure boot) is an interesting target for fault injection. The initial boot stages are of limited size which means logically exploitable vulnerabilities are not guaranteed to be present. In this paper, we introduce an ARM specific fault injection attack strategy for exploiting embedded systems where externally controlled data is loaded in the program counter (PC) register of the processor. This allows an attacker to control the target’s execution flow which eventually will lead to arbitrary code execution on the target. We first simulate the attack using a common fault model, after which we demonstrate the practicality of the attack using a development platform designed around an ARM based, fast and feature rich system on chip (SOC). We conclude with an overview of effective and non-effective countermeasures against this fault injection attack technique.

Keywords—fault injection; ARM; exploitation; system on chip; embedded systems.

I. INTRODUCTION

Embedded systems are often designed around a single system on chip (SOC), which often includes one or more central processing units (CPU). These can be implemented using different architectures, such as ARM or MIPS. The fault injection attack technique described in this paper is ARM 32-bit (AArch32 [1]) specific. This architecture is from now on referred to as ‘ARM’. The practicality of the fault injection attack technique is demonstrated using simulation and experimentation on a development platform designed around a fast and feature rich SOC which implements the following functionality: up-to 1 GHz ARM core, 32-bit DDR3 at 400 MHz, 2D/3D graphics, Gigabit Ethernet, HDMIv1.4 controller, among other functionality.

Software running on embedded systems is often found to be logically exploitable as shown in [2] (integer overflow), [3] (null pointer dereference) and [4] (incomplete signature parsing). The research presented in [5] shows that open source software has roughly 0.434 defects per thousand lines of code, out of which an unknown, but definitely smaller number, is logically exploitable. Therefore, it is likely that software exists without logically exploitable flaws, especially on smaller code bases. In such situations, other attacks may be used to exploit the embedded system. An example of such an attack is fault injection, which works by injecting physical faults into the embedded system to change its intended behavior. Attackers require physical access to the embedded system to perform fault injection; remote exploitability is typically not feasible.

II. FAULT INJECTION BACKGROUND

Faults can be injected by manipulating the chip’s environmental conditions. Typically, the following fault injection are considered:

- **Clock Fault Injection** Faults are injected in the target’s clock signal by switching between a faster and slower clock. This variant of fault injection is only effective for chips using an external clock which can be controlled by an attacker. Most embedded systems do not run directly from a clock and therefore this particular technique is often not very effective.

- **Voltage Fault Injection** Faults are injected in the target’s power domain by switching between different voltage levels. This technique is often effective on larger feature rich chips as they typically have multiple separated power domains, as they are derived externally (i.e. on the PCB). Therefore, it is possible to inject faults in one power domain without affecting the others.
• **Electromagnetic Fault Injection** Faults are injected in the target using electromagnetic emissions, generated by driving a high current through a coil. An important characteristic of this technique is its localized nature. Using electromagnetic fault injection, it may be possible to affect only parts of the chip.

• **Optical Fault Injection** Faults are injected in the target using a laser beam which injects a fault locally inside the chip. The chip’s package blocks the light from reaching the die directly. Therefore, the chip must be decapsulated to expose the chip’s internals to the laser beam.

These techniques have been proposed in [6], [7], [8], [9], [10], [11], [12], [13] and [14]. The attack proposed in this paper does not rely on a specific fault injection technique. However, for demonstration purposes we considered only voltage fault injection.

The injected fault can, in principle, have an impact on any stage of the fetch-decode-execute cycle performed for each instruction, which is shown in [13] and [14]. Additionally, any optimizations implemented by the CPU, such as pipelining [15], add to the complexity of executing a single instruction. Therefore, it is typically unknown what exactly goes wrong within the CPU when its behavior is changed due to fault injection, whereas the modified behavior itself is easier to measure. We consider a generic fault model, likely applicable to a wide range of targets, where a variable amount of bits in the instruction are flipped as a result of fault injection. Two types of behavior are possible using this fault model:

• **Instruction corruption** The original instruction is modified into an instruction that has an impact on the behavior of the device. In practice, it may modify the instruction to any other instruction supported by the architecture.

• **Instruction skipping** Effectively a subset of instruction corruption. The original instruction is corrupted into an instruction that does not have an impact on the behavior of the device. The resulting instruction does not change the execution flow or any state that is used later on.

Invocation of specific behavior is not a trivial task, as the low level control required to do this is often limited. However, it is possible to identify the more probable results while assuming that bit flips affecting single or all bits are more likely than complex patterns of bit flips.

### III. Attacking ARM Load Instructions

On ARM based chips, LDR and STR instructions are often used when data needs to be copied from memory address A to memory address B. These instructions use the internal registers to temporarily store the loaded data before it is copied to the destination memory address. Variants of the LDR and STR instruction are shown in [16] where different instructions are provided also supported in the ARM architecture. The LDR instruction loads four bytes from the address stored in register R0 into register R3, after which the STR operation copies the contents from register R3 to the memory address stored in register R1. The assembly, hex and binary representation of both instructions are shown in Table I.

<table>
<thead>
<tr>
<th>ARM Assembly</th>
<th>HEX</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR R3, [R0]</td>
<td>E5903000</td>
<td>00110000 11100000 00000000</td>
</tr>
<tr>
<td>STR R3, [R1]</td>
<td>E5813000</td>
<td>00110000 11100000 00000000</td>
</tr>
</tbody>
</table>

An efficient copy operation in ARM utilizes the LDMIA and STMIA instructions [16], which can copy multiple words with a single instruction. The LDMIA instruction copies multiple words from a memory address into multiple general purpose registers from which they are copied into another memory address. The LDMIA instruction loads four 32-bit words from the address stored in register R1 into the registers R4 to R7, after which the STMIA instruction copies to contents from the registers R4 to R7 to the address stored in register R2. The exclamation mark (!) indicates the addresses stored in register R1 and register R2 are automatically incremented with the amount of data being copied (i.e. four times 32-bits equaling 16 bytes). The assembly, hex and binary representation of both instructions are shown in Table II.

<table>
<thead>
<tr>
<th>ARM Assembly</th>
<th>HEX</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDMIA R1!,R4-R7</td>
<td>E8B100F0</td>
<td>00000000 11100000 11110000</td>
</tr>
<tr>
<td>STMIA R2!,R4-R7</td>
<td>E8A200F0</td>
<td>00000000 11100000 11110000</td>
</tr>
</tbody>
</table>

From an attacker’s point of view, these load and store operations are an interesting target for a fault injection attack as the instructions:

- operate on attacker controlled data.
- are not part of a fault injection protected section in the code as they are not considered security sensitive. All security, such as authentication and decryption, is implemented after the load and store operations.
- are executed multiple times consecutively with attacker controlled data which decreases the attack’s timing.
dependency. The attack does not require to target one specific instruction.

A. Controlling PC

An ARM processor includes 16 registers. Simply said, register R0 to R12 are general purpose registers, register R13 (SP), register R14 (LR) and register R15 (PC) are special. All registers can be modified directly using a wide variety of ARM instruction whereas the PC register can only be directly modified using specific instructions. For example, the following instructions can directly modify the PC register: LDR, MOV, ADD, SUBS, etc. In all other major architectures, such as MIPS, X86, X64, and even ARM64, the PC cannot be accessed by most instructions, especially not directly. These architectures modify the PC indirectly using jump or stack-related instructions.

Depending on the instruction under attack, a single or double bit corruption in the instruction is required to load an attacker controlled value in PC. The characteristic of ARM, where the PC can be accessed directly by most instructions, effectively leads to the applicability of this attack. The required bit flips to load the value into the PC register are underlined in Table III.

<table>
<thead>
<tr>
<th>Table III</th>
<th>GLITCHED LDR AND LDMIA INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM assembly</td>
<td>HEX</td>
</tr>
<tr>
<td>LDR PC,[R0]</td>
<td>E590F000</td>
</tr>
<tr>
<td>LDMIA R1!,R4-R7,PC</td>
<td>E8B180F0</td>
</tr>
</tbody>
</table>

While we do not perform an exhaustive search for all possible instruction encodings leading to control of the PC register, we wish to highlight the impact of the destination register encoding for both the LDR and LDMIA instruction. The LDR instruction requires a specific bit pattern, whereas the LDMIA instruction allows a multitude of different bit patterns as long as the bit for the PC register is set. This is indicated by the ‘x’ character in III. We hypothesize that the success rate when attacking the LDMIA instruction is higher than when attacking the LDR instruction.

Setting the PC register as the destination for the LDR or LDMIA instruction effectively allows an attacker to load an arbitrary value from which the next instruction will be fetched by the CPU. This is a first step towards arbitrary code execution. Typically, an attacker needs to perform additional tasks in order to exploit an embedded system. For example, an additional task is to fill the memory with executable code from which the processor will execute.

B. Practical attack scenarios

In this section, we describe two practical attack scenarios: a boot attack and a runtime attack. Both scenarios are generic and could be applied to almost any embedded system. However, attacks described in this section are always target dependent and their success rate may vary. Also, when data is copied using a different method than the described ARM instruction it may affect the applicability of the attack. For example, utilizing dedicated hardware to perform direct memory access (DMA) based copy operations will not be affected by the presented attack. However, such copy operations may be vulnerable to other attacks where the parameters passed to the DMA engine are corrupted using fault injection. Such attacks are out of scope for this paper.

This paper only explores the scenarios where the attacker injects code into the system in the form of an executable binary payload which is executed by modifying a load instruction.

C. Secure boot attack

The first attack scenario described in this paper is a boot attack, performed during the initialization of the embedded system. Although a SOC includes several types of internal memory, such as ROM, RAM or Flash, it typically also uses external memory components. The SOC cannot trust code from the external memory, since these components are physically accessible to other parties. Therefore, the usual procedure is to first copy code from external memory to internal memory, and then perform a cryptographic verification of the loaded code. Secure boot is the process of loading and verifying data from external memory during the startup of the system. A generic secure boot sequence is shown in Figure 1.
An attacker who wants to execute arbitrary code in a high privilege context, may choose to attack the secure boot mechanism. The boot of an embedded system is often implemented using multiple stages which are executed in a different privileged context. The first stage, often referred to as the ROM code, is an immutable stage which contains the first code executed after power-on reset. The ROM code is responsible for loading, authenticating and decrypting the next stage, which is often stored in external flash.

This multi-stage approach allows the embedded system to gradually drop privileges until the final stage is reached. The different stages are copied into faster volatile memory before they are decrypted and authenticated. It is likely this copy operation is implemented as described in Section IV, especially considering these are based on instructions from ARM. The boot flow of BL1 after power-on reset is as follows:

1) The BL1 image, including its signature, is copied from external flash memory into volatile memory (e.g. internal SRAM or DDR) using LOAD and STORE instructions.
2) The BL1 image is decrypted inside volatile memory using a symmetric algorithm (e.g. AES). The decryption can be performed by software or hardware (e.g. cryptographic accelerators).
3) The BL1 image is authenticated using an asymmetric cryptographic algorithm in combination with a hashing algorithm. An immutable public key should be used by the ROM code to guarantee a chain of trust.

An important observation that must be made here is that the attack is not affected by any form of decryption or authentication of the external firmware, because the attack is performed prior to such mechanisms. We target the copy phase of BL1, effectively bypassing any security provided by the secure boot chain. The entire attack can be broken down into the following steps:

- First, the destination address inside the volatile memory of the copy operation must be recovered. For example, this can accomplished by reverse engineering or analyzing the documentation. This address is used as the pointer value in the next step.
- Second, the original contents in external flash must be overwritten with a malicious payload which has the following format: shellcode + pointers, which is shown in Figure 2. The shellcode is malicious executable code that gives the attacker privileged access.
- Finally, the fault is injected after the target has copied the shellcode into its internal memory and while it is copying the pointers into its internal memory. A successful fault will corrupt the a load or store instruction so that the pointer value is copied into the PC register which will effectively execute the shellcode.

Run time control gained in this way can bypass the authentication and encryption protection of secure boot. Secure boot implementations can use Direct Memory Access (DMA) engines [17] to copy data to volatile memory. However, if any part of the unauthenticated data, for instance the signature, is copied, the attack still applies.

D. Trusted execution environment (TEE) attack

The second attack scenario is a runtime attack, performed after the embedded system is booted. More and more security sensitive embedded systems are equipped with a so-called trusted execution environment (TEE) which separates itself from the rich execution environment (REE). A general description of such a system is described in [18]. The REE typically communicates with the TEE using a dedicated API, where the TEE will copy data from the REE to its own memory. A logical representation of a system that implements a REE and a TEE is shown in 3.
We again assume that the copy operation is implemented using the instructions described in Section III. The logical flow of a call from the REE to the TEE is as follows:

1) The REE sets up the required data for an API command exposed by a Trusted App in the TEE, using the shared buffer.
2) The REE uses an exception to notify the TEE a command is ready inside the shared buffer.
3) The TEE copies the API command data into its own context using a copy operation.

Again, because the fault will be injected during the copy operation, the attacker will not be restricted by any verification performed by the TEE. The following steps can be performed to mount an attack:

- First, an attacker needs to identify an application programming interface (API) command into the TEE which fulfills the following requirements: data must be copied from the non-secure context to the secure context. Such functionality can be derived from the API documentation, passed parameters or reverse engineering.
- Second, the data originating from the non-secure context is under control of the attacker, assuming full control of the REE is established. Therefore, the attack has full control over what data passed into the API command. Depending on the implementation of the TEE, the provided shellcode could be executed from non-secure memory. Otherwise, the destination address must be recovered by other means.
- Third, the data going into the API command is constructed similarly to the secure boot attack described earlier: shellcode + pointers (see Figure 2).
- Finally, the fault is injected when the payload is in place, while the TEE is copying the pointer values. A successful fault will corrupt the a load or store instruction so that the pointer value is copied into the PC register which will effectively execute the shellcode in the secure context.

IV. SIMULATION

We used a simulation program to determine the likelihood a load instruction is modified into an instruction that loads externally controlled values into PC. The simulation program is executed natively on an development boards that contains an ARM processor that implements the ARMv7-A architecture which supports the ARM execution state. We executed the simulation program as a user-land Linux application on top off Ubuntu 12.04 LTS [19].

A generic fault model is used where we corrupt the instruction with a hamming distance difference up-to 4, to limit the simulation runtime. Using this technique we test 41448 different instructions derived from the original instruction. Pseudo code for the bit flipping code is shown in Figure 4.

```
flips = [ 1, 2, 4, ... 2147483648 ]
s = Set()
for a in flips:
  for b in flips:
    for c in flips:
      for d in flips:
        s.add(a | b | c | d)
```

Figure 4. Pseudo bit flipper code

The bit flipper code is implemented in a Python wrapper that passes the flipper value as a parameter to a C application. This application uses the flipper value to flip bits in the target instruction, which is part of piece of shellcode stored in a local buffer. The C application jumps to the shellcode after the target instruction’s bits are flipped. Another function, from now on named the identifier function, prints a string on the serial interface, which is used to identify if the PC register is modified to the controlled value. This approach is very fast and allows simulating the proposed attack vector natively.

A. Simulation: LDR instruction

The simulation program loads data, using an LDR instruction, from a memory address stored in register R0 into register R3. The source data consists of a pointer pointing to the identifier function. We identified multiple ways to corrupt the LDR instruction into an instruction that loads the controlled value into the PC register. These are shown in Table IV. The difference between the original instruction and the corrupted instruction is also listed, expressed as the hamming distance (HD).

<table>
<thead>
<tr>
<th>Modified LDR instructions</th>
<th>Differences (HD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR R3, [R0]</td>
<td>0</td>
</tr>
<tr>
<td>LDR PC, [R0]</td>
<td>2</td>
</tr>
<tr>
<td>LDRGE PC, [R0]</td>
<td>3</td>
</tr>
<tr>
<td>LDRLE PC, [R0]</td>
<td>4</td>
</tr>
</tbody>
</table>

The LDR instruction can be implemented using a different register (e.g. register R6 instead of register R3) which influences the required fault model. Hence, when R7 is used, only a single bit corruption will modify the PC register into the controlled value.
B. Simulation: LDMIA instruction

The simulation is repeated using the LDMIA instruction, where the target instruction loads data from a memory address stored in register R0 into register R4, R5, R6 and R7. The source data consists of pointers pointing to the identifier function.

We identified multiple ways to corrupt the LDMIA instruction into an instruction that loads the controlled value into the PC register. These are shown in Table V. The difference compared to the original instruction, expressed as the hamming distance, is also listed.

<table>
<thead>
<tr>
<th>Modified instruction</th>
<th>Differences (HD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDMIA R0!, R4-R7</td>
<td>0</td>
</tr>
<tr>
<td>LDMIA R0!, R4-R7,PC</td>
<td>1</td>
</tr>
<tr>
<td>LDMDA R0!, R4-R7,PC</td>
<td>2</td>
</tr>
<tr>
<td>LMDGT R0!, R4-R7,PC</td>
<td>2</td>
</tr>
<tr>
<td>LMDGE R0!, R4-R7,PC</td>
<td>2</td>
</tr>
</tbody>
</table>

A successful glitch may change the original operands into anything, as long as the bit is set to copy the data from R1 into PC.

The simulation showed that the PC register might be modified by corrupting LDR and LDMIA instructions using fault injection, as long as the chosen fault model is applicable. We believe it provides enough confidence to demonstrate the practicality during the experiments.

V. EXPERIMENTAL RESULTS

We use the same development platform used during the simulation phase as the target. We created a test bench to demonstrate the practicality of the described attack scenarios. We use commercially available FI hardware and software [20] to inject a glitch in the core power domain of the target, which powers ARM processor.

This paper only considers voltage fault injection, other fault injection methods are not applied. However, the proposed attack may be applicable to all fault injection techniques introduced in Section I.

A. Target modification

Multiple modifications are made to the platform, which allow us to perform effective fault injection. These can be summarized as follows:

- A power cut on the PCB is made to be able to control the power to the ARM processor using our fault injection equipment.
- The capacitance in the core power domain which powers the ARM processor, is reduced to a minimum by removing all the external capacitors from the PCB. These capacitors are in place to stabilize the power signal of the target when low frequency and high frequency anomalies are present in the power domain, such as our injected faults. In theory, removing the capacitors should destabilize the target under normal conditions but we do not observe such behavior.
- A trigger is implemented using a general purpose IO pin of the target. The test program running in the target can set this trigger shortly before the injection moment, which simplifies the timing of the attack.
- The reset signal of the target is fed into our fault injection setup which allows us to reset the target when required. Fault injection may cause the target to enter an unrecoverable state which can only be overcome using a full system reset.

B. Fault injection setup

Dedicated high speed hardware is used to generate a pulse, and an amplifier is used to glitch the power line of the target. An logical representation of the setup is shown in Figure 5. The setup includes a PC that controls the experiment and configures the glitch parameters: Normal VCC, Glitch VCC, Glitch Length and Glitch Delay. The Normal VCC is the power provided to the target power domain. The Glitch VCC is the voltage level during the glitch, which is subtracted from the Normal VCC. The Glitch Length is the duration of the glitch. The Glitch Delay is the duration between the moment the trigger is detected, and the glitch is injected.

![Figure 5. Fault injection setup used for the experiments](image-url)

The test program is copied from external flash and executed from external DDR. The ARM processor runs at 800 MHz on a single core. We power the target during all experiments at 1.1 volt, even though the target’s original power supply provides 1.2 volt. The target is stable between 1.1 volt and 1.3 volt. We assume the injected faults are more effective when the target operates on its minimal voltage. Hence, for this reason we also remove the capacitors from the
target’s PCB. However, we did not investigate further if this hypothesis is correct.

C. Characterization phase

The effectiveness of the fault injection setup is verified using a characterization phase where a simple test program is attacked using fault injection. The results of the characterization phase serve as a baseline to support our hypothesis that the instruction encoding influences the fault injection success probability.

The resulting counter value is sent back to the software running on the workstation. The trigger signal is set before the first ADD instruction and set low after the last ADD instruction. A snippet of the code is shown in Figure 6.

```
MOV R1, #0
MOV R0, [%loopcnt]
LOOP:  ADD R1, R1, #1
      ADD R1, R1, #1
      SUBS R0, R0, #1
      BNE LOOP
      ...
```

Figure 6. Test program: ADD instructions

Using a randomized parameter search, where we randomize the glitch length and the glitch voltage, we determine optimal parameters where the counter value, stored in register R1, is corrupted, which is used as an indication for a successful glitch.

We perform 18,000 experiments using the glitch parameters in Table VI.

<table>
<thead>
<tr>
<th>Glitch parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal VCC</td>
<td>1.1 volt</td>
</tr>
<tr>
<td>Glitch VCC</td>
<td>Random between -1.4 volt and -1.0 volt</td>
</tr>
<tr>
<td>Glitch length</td>
<td>Random between 0 ns and 1000 ns</td>
</tr>
<tr>
<td>Glitch delay</td>
<td>Random between 30 µs and 35 µs</td>
</tr>
</tbody>
</table>

Table VI: PROFILING GLITCH PARAMETERS

Figure 7 shows a plot of the experiments where the chip’s behavior is indicated by plotting the glitch length against the glitch voltage. No effect is plotted as 'Expected', a successful glitch where the counter value changed is plotted as 'Success' and all resets and mutes are plotted as 'Mute'.

The characterization showed the fault injection is effective in changing the behavior of the target using voltage fault injection.

D. Experiments

The test applications used for the experiments are implemented as an additional command in U-Boot [21], which is the de facto standard boot loader embedded systems and is publicly available as open source software. The primary purpose of U-Boot is typically to load a subsequent stage, such as the Linux Kernel. However, U-Boot provides a functionality rich environment in the form of a shell which can be utilized to perform various operations. We do not use any of the functionality provided by U-Boot other than our test applications.

We reuse the parameter ranges identified in the characterization phase to optimize the initial parameter search.

1) Experiment 1: LDR

The 1st experiment consists of 25 subsequent LDR instructions which copy from a local buffer into register R7. The instructions are placed inside a loop which is executed 100 times, resulting in the execution of 2500 LDR instructions which take approximately 300 microseconds. A snippet of the test application is shown in Figure 8.

```
... MOV R0, [%loopcnt] MOVW R1, #0x0F1C MOVT R1, #0x2FFA LOOP:
      LDR R4, [R1]
      LDR R5, [R1]
      LDR R6, [R1]
      LDR R7, [R1]
      SUBS R0, R0, #1
      BNE LOOP
      ...
```

Figure 8. Test program: LDR instructions

The local buffer is filled with pointers pointing to a function that prints a string on the serial interface and then enters an
endless loop. The address of the local buffer (0x2FFA0F1C) is stored in register R1 using a MOVW and MOVT instruction. The fault is injected when the LDR instructions are executed. A successful attack loads one of the pointers from the local buffer into the PC register, which results in the execution of shellcode that prints a string on the serial interface.

We start off with a parameter sweep to identify efficient glitch parameters using the parameters outlined in Table VII. In total we perform 10,000 experiments, which takes roughly 4 hours.

Table VII
EXPERIMENT I: INITIAL GLITCH PARAMETERS

<table>
<thead>
<tr>
<th>Glitch parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal VCC</td>
<td>1.1 volt</td>
</tr>
<tr>
<td>Glitch VCC</td>
<td>Random between -1.4 volt and -1.0 volt</td>
</tr>
<tr>
<td>Glitch length</td>
<td>Random between 700 ns and 1000 ns</td>
</tr>
<tr>
<td>Glitch delay</td>
<td>Random between 30 µs and 35 µs</td>
</tr>
</tbody>
</table>

Figure 9 shows a plot of the experiments where the chip’s behavior is indicated by plotting the glitch voltage against the glitch length. We grouped the resets and mutes to generate a clear picture.

Out of the 10,000 experiments, we identify 1 (0.01%) successful glitch, 3436 (34.36%) mutes/resets and 6563 (65.63%) glitches did not yield influence on the target’s operation. The resets are caused by different exceptions, such as data abort and illegal instruction.

We perform 1000 experiments with the parameters of the successful glitch, which are shown in VIII. We perform this experiment to approximate the reproducibility of a successful glitch.

Using these parameters, we identify 7 successful glitches (0.7%) where the expected string is printed on the serial interface.

Table VIII
EXPERIMENT I: FINAL GLITCH PARAMETERS

<table>
<thead>
<tr>
<th>Glitch parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal VCC</td>
<td>1.1 volt</td>
</tr>
<tr>
<td>Glitch VCC</td>
<td>-1.293 volt</td>
</tr>
<tr>
<td>Glitch length</td>
<td>862 ns</td>
</tr>
<tr>
<td>Glitch delay</td>
<td>31 µs</td>
</tr>
</tbody>
</table>

2) Experiment 2: LDMIA

The 2nd experiment consists of 25 subsequent LDMIA instructions, which copy from a local buffer into register R4, R5, R6 and R7. The instructions are placed inside a loop which is executed 100 times, resulting in the execution of 2500 LDMIA instructions which takes approximately 330 microseconds. A snippet of the test application is shown in Figure 10.

```
... MOV R2, %[loopcnt]  
MOVW R1, #0x0F24  
MOVT R1, #0x2FFA  
LOOP:  
LDMIA R1, (R4-R7)  
...  
LDMIA R1, (R4-R7)  
SUBS R2, R2, #1  
BNE LOOP  
...  
```

Figure 10. Test program: LDMIA instructions

The local buffer is filled with pointers pointing to a function that prints a string on the serial interface and then enters an endless loop. The address of the local buffer (0x2FFA0F24) is stored in register R1 using a MOVW and MOVT instruction. The fault is injected when the LMDIA instructions are executed. A successful attack loads one of the pointers from the local buffer into the PC, which results in the expected string being printed on the serial interface.

We start off with a parameter sweep to identify efficient glitch parameters using the parameters outlined in Table IX. In total we perform 10,000 experiments, which takes roughly 4 hours.

Table IX
EXPERIMENT 2: INITIAL GLITCH PARAMETERS

<table>
<thead>
<tr>
<th>Glitch parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal VCC</td>
<td>1.1 volt</td>
</tr>
<tr>
<td>Glitch VCC</td>
<td>Random between -1.4 volt and -1.0 volt</td>
</tr>
<tr>
<td>Glitch length</td>
<td>Random between 700 ns and 1000 ns</td>
</tr>
<tr>
<td>Glitch delay</td>
<td>Random between 30 µs and 35 µs</td>
</tr>
</tbody>
</table>

Figure 11 shows a plot of the experiments where the chip’s behavior is indicated by plotting the glitch voltage against the glitch length. We grouped the resets and mutes to generate a clear picture.
Out of the 10,000 experiments, we identify 27 (0.27%) successful glitches, 2,653 (26.53%) mutes/resets and 7,320 (73.20%) glitches did not yield influence on the target’s operation. The resets are caused by different exceptions, such as data abort and illegal instruction.

We perform 1,000 experiments with the parameters of a successful glitch, which are shown in Table X. We perform this experiment to approximate the reproducibility of a successful glitch.

### Table X

<table>
<thead>
<tr>
<th>Glitch parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal VCC</td>
<td>1.1 volt</td>
</tr>
<tr>
<td>Glitch VCC</td>
<td>-1.321 volt</td>
</tr>
<tr>
<td>Glitch length</td>
<td>946 ns</td>
</tr>
<tr>
<td>Glitch delay</td>
<td>31 31 µs</td>
</tr>
</tbody>
</table>

Using these parameters, we identify 27 successful glitches (2.7%) where the expected string is printed on the serial interface.

3) **Conclusions: LDR vs LDMIA**

The success rate and reproducibility of a successful attack is significantly different between the LDR and LDMIA instruction. The impact of the destination register encoding difference for each instruction is likely the reason, which matches our earlier hypothesis: the success rate when attacking the LDMIA instruction is higher than when attacking the LDR instruction due to the difference in instruction encoding.

### VI. COUNTERMEASURES

Mitigation of fault injection is possible at the hardware and software level. Three principle approaches include:

- **Deflect** Manipulation of a computer chip typically requires a fault to be injected at exactly the right time to affect a specific instruction. If the code execution progress is fully deterministic an attacker can use a precise time base to determine the best moment in time for the fault to occur. It is possible to deflect the fault and lower the predictability by running the processor at varying speed, or by introducing random delays. This could be implemented at a hardware level, using an unstable clock, or a software by having waiting loops of variable length.

- **Detect** If faults are timely detected it is possible to prevent insecure behavior. In hardware this could be done with sensors that measure the environmental conditions and generate alarms when extraordinary conditions are detected. In software this could be done by double checking conditions before branching, or to verify sensitive data values. Also it is possible to check that the expected program flow was executed.

- **React** Even with reduced hit probability and increased detection probability it is still possible that an attack would succeed, given enough time. Fault response uses detected faults to impose a penalty. This could be a temporary penalty, like a waiting time before the behavior targeted by the attack can be repeated, or a more permanent one like terminating the device.

An extensive list of countermeasures using these approaches is discussed in [22]. However, many of these countermeasures are not very effective against the presented attack vector. To summarize:

- Firstly, random delays are used to decrease the success rate for a fault injection attack. However, copy operations are often implemented as a loop and are of significant length, resulting in the consecutive execution of vulnerable instructions, effectively rendering the randomization ineffective as a successful glitch can occur at different moments in time.

- Secondly, detection of fault injection works requires that the checks are executed. When the fault triggers a jump to a different code section this countermeasure is bypassed. An alternative could be to apply traps, small code segments that detect anomalies due to fault injection, interleaved with the copy instructions. However, this does not rule out the described attack.

- Lastly, reaction is possible. In case a trap would detect a fault, this could trigger blowing a fuse (or OTP cell) that would block, or terminate, the target. This countermeasure should be designed with care to prevent unintentional defects.

Traditional logical exploitation protection mechanisms such as NX/DEP [23] where code can only be executed from designated pages in memory, and ASLR [24] where memory
addressing is randomized, increase the complexity of the attack. However, these hurdles can be overcome, using techniques utilized by logical exploitation, and can therefore not mitigate the fault injection threat.

In our experiments we confirm that power glitching is more difficult when complex bit flips are required for the attack. A redesign of the instruction set that would maximize hamming distance from valid to potentially harmful instructions, could be an interesting approach to reduce the sensitivity for fault attacks. However, the risk imposed by fault injection would not be entirely mitigated.

In any case, we believe hardware countermeasures are required to increase the robustness of embedded chips. For example, such countermeasures are described in [11] and [9]. However, typically these countermeasures aim to prevent a single fault injection technique. Therefore, a combination of different countermeasures must be considered when the risk imposed by all fault injection techniques must be mitigated.

VII. Conclusion

In this paper we describe a fault injection attack strategy which poses a significant risk towards secure embedded systems that implement the ARM 32-bit (AArch32) architecture. We show it is possible to load attacker controlled values in the program counter (PC) register which may compromise the overall security of an embedded system.

We simulated the fault injection attack strategy to form a hypothesis on its practicality. The hypothesis stressed the fact that the destination register encoding difference for the LDR and LDMIA instruction impacts the attack’s success rate. The experiments performed on a development platform design around a popular, ARM-based, fast and feature rich system on chip (SOC), indicate that the hypothesis is correct.

The described fault injection strategy is architecture specific, but its principles may be applicable to other architectures. More complex code constructs may allow an attacker to control the program counter (PC) register using fault injection.

ACKNOWLEDGMENT

This research is supported by Riscure. We thank our colleagues at Riscure, including Lukasz, for all their wonderful insights and pearls of wisdom. We are immensely grateful for their help reviewing preliminary versions of this paper. Their input immensely improved the readability of this report.

REFERENCES


