Laser-induced Single-bit Faults in Flash Memory: Instructions Corruption on a 32-bit Microcontroller ⋆

Brice Colombier*, Alexandre Menu†, Jean-Max Dutertre†,
Pierre-Alain Moëllic*, Jean-Baptiste Rigaud†,Jean-Luc Danger‡

*CEA Tech, Centre CMP, Equipe Commune CEA Tech - Mines Saint-Etienne,
F-13541 Gardanne FRANCE
{brice.colombier, pierre-alain.moellic}@cea.fr;
†IMT, Mines Saint-Etienne, Centre CMP, Equipe Commune CEA Tech - Mines Saint-Etienne
F-13541 Gardanne FRANCE
{alexandre.menu, dutertre, rigaud}@emse.fr
‡LTCI, Télécom ParisTech , Institut Mines-télécom, Université Paris Saclay,
75634 Paris Cedex 13, France
jean-luc.danger@telecom-paristech.fr

Abstract. Physical attacks are a known threat to secure embedded systems. Notable among these is laser fault injection, which is probably the most powerful fault injection technique. Indeed, powerful injection techniques like laser fault injection provide a high spatial accuracy, which enables an attacker to induce bit level faults. However, experience gained from attacking 8-bit targets might not be relevant on more advanced micro-architectures and these attacks become increasingly challenging on 32-bit microcontrollers. In this article, we show that the flash memory area of a 32-bit microcontroller is sensitive to laser fault injection. These faults occur during the instruction fetch process, hence the stored value remains unaltered. After a thorough characterisation of the induced faults and the associated fault model, we provide detailed examples of bit-level corruptions of instruction and demonstrate practical applications in compromising the security of real-life codes. Based on these experimental results, we formulate a hypothesis about the underlying micro-architectural features that could explain the observed fault model.

Keywords: Fault attack, laser injection, flash memory

1 Introduction

Physical attacks pose considerable security threats to embedded systems. Provided physical access to a device, an attacker can exploit hardware-

⋆ Part of this work is funded by French ANR program (DS0901/2015) with the project PROSECCO (ANR-15-CE39-0008)

based vulnerabilities to bypass existing security measures. Among these techniques, fault injection consists in disturbing the operation of a device to retrieve secret information or be granted unauthorised privileges. Laser fault injection features a high spatial accuracy, which allows an attacker to induce single bit-flips in static memory cells of 8-bit [13] and 32-bit microcontrollers [32]. However, this technique is expensive and difficult to apply due to numerous precise parameters to tune, which might result in endless explorations of the parameters space. While increasing chip integration enables to design complex 32-bit architectures, only few works investigate laser injection on these architectures [31, 16, 32, 15]. Besides, none of them addresses the underlying fault mechanism, which makes it difficult to explain the observed fault models in a consistent framework.

In this article, we highlight the flash memory as an area of interest for laser fault injection on a 32-bit microcontroller. We observe that individual bits of the fetched instruction can be set. The stored value remains untouched, only the read value is altered. For example, the fields of the fetched instruction can be altered, but also the opcode, potentially changing the instruction. Such modifications cause severe security concerns, since an attacker can then tamper with the instructions on the fly, before they are decoded and executed.

The contributions of this article are the following. First, we highlight the sensitivity of flash memory to the single-bit bit-set fault model. We then detail the influence of each parameter of the laser on the injected fault. We apply this fault model to real-life codes and show how it undermines their security. Finally, we discuss a physical explanation for the observed faults, which is consistent with the micro-architecture of a NOR flash memory.

The outline of the article is as follows. In Section 2, we analyse previous work on laser fault injection, pointing out the current scarcity of results and understanding of fault injection on 32-bit microcontrollers. In Section 3, we detail our experimental setup. In Section 4, we describe the obtained fault model and how it is affected by the parameters of our experimental setup. In Section 5, we highlight how the previous fault model lowers the security of PIN verification and AES-128 algorithms by demonstrating two attacks that we performed on 32-bit implementations. In Section 6, we discuss a hypothesis on the physical phenomenon accounting for the observed fault model, as well as the limitations of our setup. Finally, Section 7 concludes the article.

2 Previous work

Fault attacks require to define an abstract model of the erroneous behaviour of a device [5]. The complexity of characterisation of a fault model lies in the interrelated factors which influence the response of a device to fault injection, namely its micro-architecure, technology node, and sensitivity to the fault injection technique [2]. Therefore, a comprehensive understanding of the fault model is hard to acquire. Our approach is to analyse the fault model according to the degree of knowledge it requires about the target implementation. We thus propose four levels of abstraction, in Figure 1, to depict the current understanding of fault injection on microcontrollers. The algorithmic level provides a description of the fault effects on the outputs of an algorithm, regardless of its implementation. The execution level details the faults effects on the components of the software data model. The implementation level explains how the observed behaviour is related to the hardware implementation on the target device. The physical level focuses on the physical phenomenon of the fault injection.

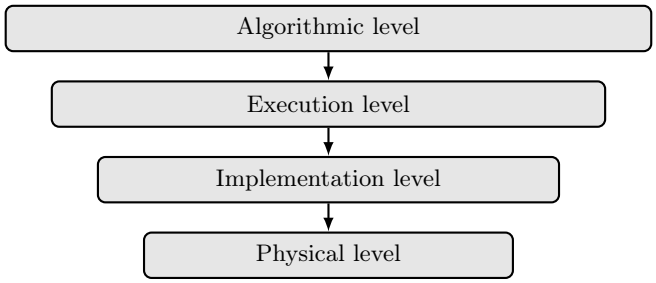


Fig. 1: Abstraction levels to describe a fault model.

Substantial work has been done to understand fault injection on 8-bit microcontrollers in the context of smart-card security. Most of the publications focus on fault description at the algorithmic level to demonstrate practical attacks on cryptographic algorithms [8, 17, 25, 33]. In some cases, observations of low-level execution faults are provided, highlighting instructions or registers corruption [11, 1, 6]. Balasch et al. demonstrated that a thorough characterisation of the response of a device to fault injection enables one to get a better understanding of the effects of the fault on the underlying hardware implementation [1]. At the same time, several authors observed that timing constraints violation could explain the observed fault models at the physical level [26, 19]. While none of

these works addressed all four levels of abstraction given in Figure 1, they reflect a global understanding of fault injection on 8-bit microcontrollers.

Current work on 32-bit architectures follows a similar timeline. Most of the publications focus so far on empirical observations at the algorithmic [7] and execution level [31, 30]. However, the observed fault models lack a consistent framework. Several difficulties can be underlined when attempting to understand the effect of fault injection on 32-bit architectures.

First, advanced technology nodes enable designers to improve the performance of a chip with architectural features like pipeline and cache mechanisms. However, they greatly increase the complexity of black-box fault effects analysis [20, 22] as already observed on 8-bit architectures [1]. Second, fault injection techniques leveraging timing constraints violation fail to catch local features of 32-bit micro-architectures. Indeed, the attempts to characterise the effects of clock glitches on 32-bit architectures got very similar results to those obtained with 8-bit architectures [3, 19], while different fault models were observed with optical injection depending on the injection locality [15]. Third, substantial work has been done to understand fault effects on 32-bit microcontroller at the execution and implementation level using local electromagnetic fault injection [20, 22]. However, chip sensitivity to the underlying physical phenomenon is not understood yet and lacks of a consistent description [21].

Laser fault injection takes advantage of related works on simulation of ionising radiation in semiconductors devices [29]. Provided an access to the die, an attacker can induce electron-hole dissociation on the path of a laser beam. As a consequence, a photoelectric current is generated in the reverse biased junctions of the illuminated transistors. This effect was investigated to describe the bit-flip fault model in static memory cells [24, 14] with attacks on AES encryption [13] and secure program register [32]. Both physical understanding and spatial accuracy of laser fault injection make this technique well suited to gain insight into the effects of fault injection on a 32-bit architecture. Previous work on flash memory vulnerabilities pointed out the memory control logic as a sensitive area to laser fault injection [27, 28] although the authors do not explain the underlying fault mechanism.

In this article, we precisely characterise the effect of laser injection in the flash memory area and observe, on the implementation level, that single bit-set in data fetched from the flash memory can be performed. We then give several examples of instructions corruption, at the execution level. We demonstrate the validity of the fault model at the algorithmic level on

implementations of two security algorithms in Section 5 and propose an explanation of the fault mechanism at the physical level in Section 6.

3 Methods and experimental setup

3.1 Target board and microcontroller

The microcontroller we used for our experiments embeds an ARM Cortex-M3 core with 128 kB of flash memory and is manufactured at the 90 nm technology node (see Figure 2a). It runs at 7.4 MHz, corresponding to a 135 ns clock period. This frequency is fixed by the ChipWhisperer platform [9].

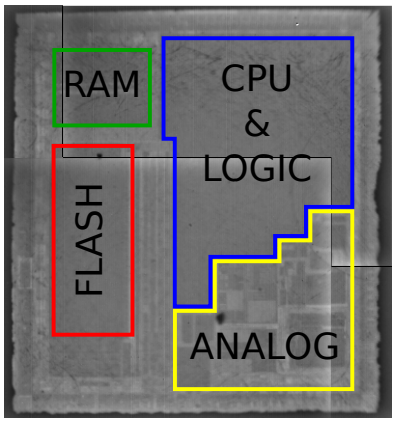
The chip is mounted on a ChipWhisperer target board and is soldered below and facing up. An opening was cut on the PCB board, just under the chip, to give access to the back-side. We designed a custom target board suitable for back-side laser injection thanks to the open-source hardware information provided for the ChipWhisperer platform[1]. To perform laser fault injection, the back of the chip must be decapsulated to show the silicon substrate. This is performed by chemical processing before the chip is mounted on the board. The decapsulation should uncover a sufficient injection area with little damage on the packaging to sustain mounting constraints. A picture of the custom board is shown in Figure 2b. This target board is mounted on the ChipWhisperer motherboard, which is then placed on the laser injection bench. The laser setup is described in the next section.

3.2 Laser characteristics and parameters

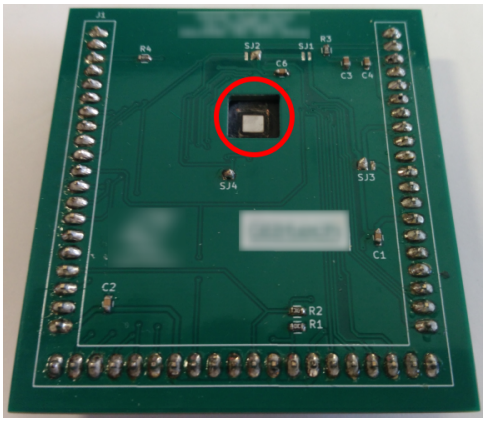
The laser source uses an acousto-optic technology to generate an infrared laser beam at a wavelength of 1064 nm. An infrared laser is a necessity to perform fault injection through the back-side since the silicon substrate is opaque to visible light. An infrared laser can go through it and impact the active regions of the transistors.

The laser source can shoot laser pulses as short as 50 ns with a peak power of 3 W. The laser pulse is directed to the focusing system by an optical fiber. The focusing system allows to a obtain a laser spot of diameter 5 µm. We manually adjust its focus with a confocal infrared camera. The laser shot is triggered by an external input, generated by the target device.

[1] `https://github.com/newaetech/chipwhisperer/tree/develop/hardware/victims/cw308_ufo_target`



(a) Infrared picture of the backside of the target microcontroller. The flash memory is framed in red.



(b) Picture of the custom target board. The decapsulated chip mounted below with the silicon substrate visible is circled in red.

Fig. 2: Target microcontroller on the custom board.

There is a constant delay between the rising edge of the trigger and the actual arrival of the laser beam on the die.

There are five injection parameters that must be tuned:

- **power**: the peak power of the laser pulse,
- **duration**: the duration of the laser pulse,
- **delay**: the delay between the arrival of the trigger on the laser source and the actual shot,
- **x-position**: the x-position on the target board,
- **y-position**: the y-position on the target board.

3.3 Characterisation codes

Leveraging simple test codes, one can characterise the target unexpected behaviours and the fault model dependency on experimental conditions. As opposed to [7], we state that the conclusions drawn from test codes about the underlying fault mechanisms can be extended to any codes, as observed in [1]. Attack scenarios on software implementation of security algorithms were remarkably consistent with our characterisation results. Codes given in Listings 1 and 2 were used to characterise the effect of fault injection in the flash memory area. Their respective usage is described below. The code was compiled into the Thumb instruction set. Therefore, instructions can either be 16 or 32-bit wide and are sometimes unaligned.

Data stored in flash memory is said to be aligned if it is stored at an address which is a multiple of 32 bits.

Bit-level characterisation of fault location The first code highlights modifications in data fetched from flash memory. The target instruction is on line 4 of Listing 1.

Listing 1 Bit-level characterisation of fault location.

```
1  test_data:
2  .word 0x00000000
3  NOP
4  LDR R0, test_data
5  NOP
6  # Reading back R0
```

This `LDR` instruction fetches the 32-bit word `0x00000000` stored at the *test_ data* label in the flash memory and stores it in register `R0`. For instance, a fault is detected on the bit of index 0 if the actual value stored in register `R0` is `0x00000001` after a laser injection was performed. The test word `0x00000000` was used to highlight bit-sets, since a previous test with the word `0xFFFFFFFF` validated that we were not able to induce bit-resets. The advantage of faulting raw data instead of an instruction is that it allows to observe bit-sets on a whole 32-bit word, whereas a 32-bit instruction always contains several `1`s, for which the bit-set is not observable. The `LDR` instruction executes in two clock cycles. In the first clock cycle, the offset of the address at which the data is stored is computed. In the second clock cycle, the data is actually read and stored in `R0`. The second clock cycle is the one we target. Dummy instructions (`NOP`) are inserted before and after to isolate and fault the target instruction only.

Characterisation of fault sensitivity over time The second characterisation code aims at highlighting the most fault-sensitive moments in the execution of a sequence of instructions. For this, after finding out the location where a given bit is faulty thanks to the code given in Listing 1, we swept over the injection delay with a 10 ns step to target consecutive instructions shown in Listing 2, from line 3 to 9. Under normal conditions, after executing the code shown in Listing 2, the output consists in several 32-bit `0x0000FFFF` values stored in registers `R0`, `R1`, `R4`, `R5`, `R6`, `R8`, and `R9`.

Listing 2 Characterisation of sensitivity over time.

```
# Initialising registers R0, R1, R4, R5, R6,
# R8 and R9 to 0xFFFFFFFF
MOVW R0, 0x0000
MOVW R1, 0x0000
MOVW R4, 0x0000
MOVW R5, 0x0000
MOVW R6, 0x0000
MOVW R8, 0x0000
MOVW R9, 0x0000
# Reading back the registers
```

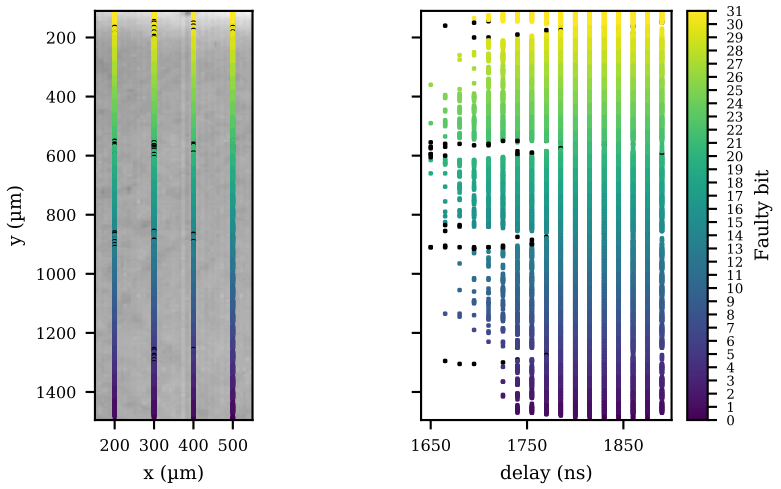
The results obtained with these codes are given in the next section, where we present the influence of the laser parameters on the fault injection process.

4 Observable fault model

4.1 Parameters and types of faults

Bit-level characterisation of fault location We observed that moving along the y-axis (longest side) in the flash memory area (see Figure 2a) allows to precisely target the bits of the fetched data one after the other. Conversely, moving along the x-axis (shortest side) does not change the affected bit. Figure 3a shows a mapping of *test_ data* faulty bits with a x-step of 100 µm and a y-step of 5 µm for an *aligned* word. The laser power is set to 1.1 W with a pulse duration of 135 ns. It clearly shows that the affected bit is directly related to the y-position (see color code on the right-hand side of Figure 3).. Figure 3b reports which bit is faulty at a given y-coordinate as a function of the delay between the trigger and the laser shot. At an optimal delay, around 1850 ns here, all the bits of the fetched word can be set depending on the y-coordinate of the laser spot over the flash memory area.

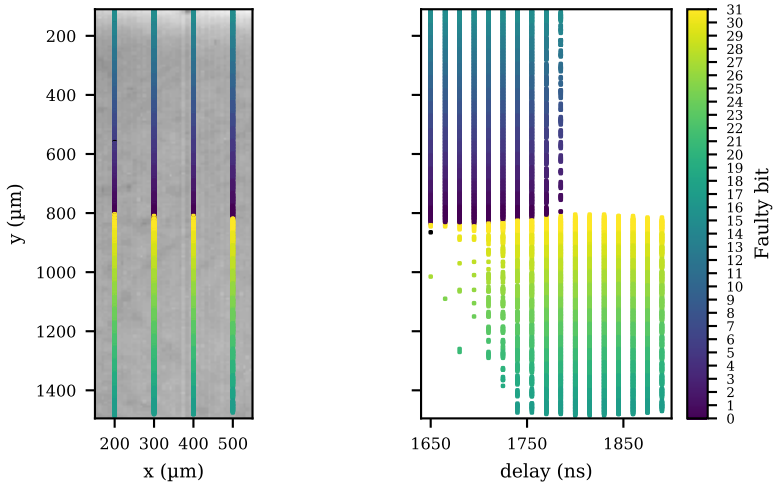
Figure 4 is the same as Figure 3 but for an *unaligned* word. In this situation, the upper and lower 16-bit halves of the accessed data are swapped (see Figure 4a). This behaviour is better understood by analysing the injection timing: the sixteen least significant bits are faulty one clock period (135 ns) before the sixteen most significant bits, as shown in Figure 4b. This observation reflects the organisation of the binary code and supports the assumption that the sequential access to the flash memory is affected during the fetch operation.



(a) Position to set the bits of the fetched data.

(b) Timing of the optimal delay to set the bits in the fetched data.

Fig. 3: Influence of the x, y and delay parameters on the fault injection on *aligned* data. Black dots show locations where the chip stopped responding.



(a) Position in to set the bits of the fetched data.

(b) Timing of the optimal delay to set the bits in the the fetched data.

Fig. 4: Influence of the x, y and delay parameters on the fault injection on *unaligned* data. Black dots show locations where the chip stopped responding.

When performing the fault injection, an attacker does not know if the target instruction is aligned or not. Therefore, according to Figures 3 and 4, if n denotes the target bit in a given instruction, the laser spot must be positioned to fault either the bit n if the instruction is aligned, or the bit $n + 16 \bmod 32$ if it is unaligned.

These results demonstrate that predictable and repeatable faults on words read from the flash memory can be achieved by targeting a fixed location on the flash memory area while a fetch operation occurs. However, targeting the same instruction with the same spot location might induce an entirely different behaviour if the instruction memory alignment is different.

After we found a position at which a bit can be set, we explored the power and duration parameters. The results are presented in Figure 5, for a laser power ranging from 0.5 to 1.4 W and a pulse duration from 65 to 270 ns. From these results, it appears that increasing the power and the duration of the laser pulse increases the success rate of the fault injection. One very interesting setting is 0.5 W of power and 200 ns of duration. Indeed, it allows to reach 100% of fault occurrence for a range of almost 50 ns, while performing only monobit faults. Performing a fault on two adjacent bits with 100% probability is then possible by increasing the power to 1.1 W. These results show that in order to obtain monobit faults, careful tuning of the laser pulse power and duration is required.

Characterisation of fault sensitivity over time After running the code shown in Listing 2 at a position where a specific bit can be set, it appeared that some moments in time are more prone to fault injection than others. Figure 6 shows how the probability of occurrence of a fault changes with the injection delay.

We observe on Figure 6 that the separation between two peaks of fault sensitivity is always a multiple of the clock period, which support the assumption that the fault injection is synchronous with the chip internal activity. Besides, we observe on the left-hand side of Figure 6 that the interval between two consecutive faults is not constant. As this feature has not been documented yet, we assume that the fetch timing depends on the pipeline activity. However, for every instruction, there is a delay parameter that allows to fault it with 100% probability.

These characterisation results show that individual instructions can be targeted. Provided the right injection parameters, single bit-set can be achieved on all the bits of an instruction or word fetched from the flash memory.

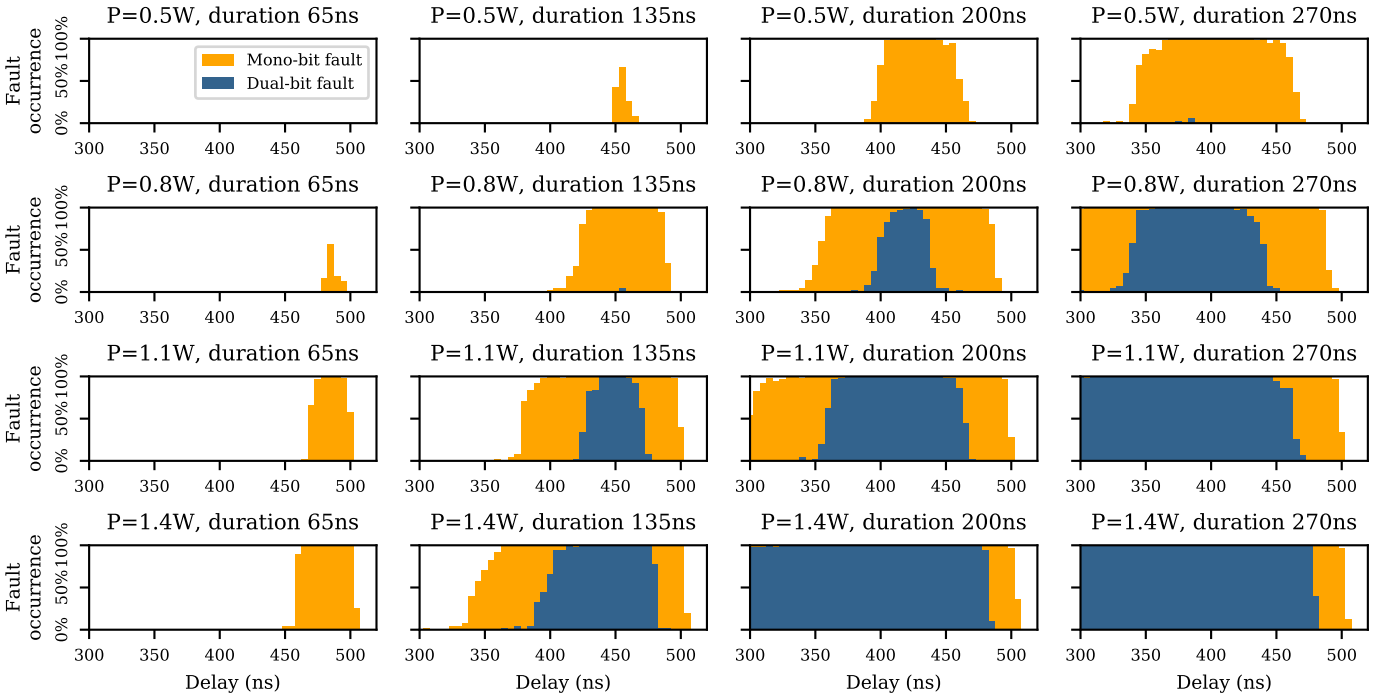


Fig. 5: Occurrence and types of faults for two laser injection parameters: power and duration.

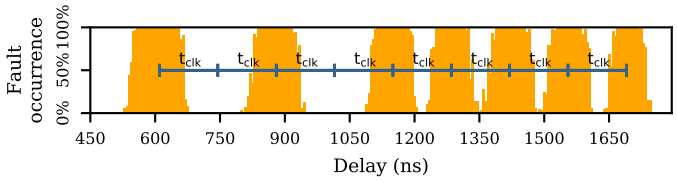


Fig. 6: Periodicity of the fault occurrences for a 0.8 W laser pulse of 135 ns.

4.2 Modification of a MOVW instruction

As an illustrative example of the possibilities offered by the fault model, we performed fault injection a `MOVW` instruction. The purpose of this 32-bit instruction is to load a 16-bit value into the lower half of a 32-bit register. The opcode part, the destination register part (denoted as Rd) and the n-bit data part (denoted as imm$n$ or i) of the instruction are given in the upper part of Figure 7. An example of `MOVW` instruction is also given where `0x0000` is stored in `R0`. This information is given in the ARM Architecture

Reference Manual[2]. We illustrate the impact of the fault model with three instruction modifications we performed on the target microcontroller.

bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reference instructions																																
Generic `MOVW`	1	1	1	1	0	i	1	0	0	1	0	0	imm4				0	imm3			Rd				imm8							
`MOVW, R0, 0`	1	1	1	1	0	i	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Data corruption																														↓		
`MOVW, R0, 4`	1	1	1	1	0	i	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0
Destination register corruption																								↓								
`MOVW, R1, 0`	1	1	1	1	0	i	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0
Opcode corruption									↓																							
`MOVT, R0, 0`	1	1	1	1	0	i	1	0	**1**	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 7: Examples of achievable corruptions on a `MOVW` instruction. The arrow indicates which bit is set by laser injection.

By performing a bit-set on the bit of index 2 of the instruction, the data to store is altered. Setting this bit leads to store `0x0004` instead of `0x0000` into `R0`.

By performing a bit-set on the bit of index 8 of the instruction, the destination register is altered. Setting this bit leads to store `0x0000` into `R1` instead of `R0`.

Finally, by performing a bit-set on the bit of index 23 of the instruction, the opcode is altered. This changes the instruction from `MOVW` to `MOVT`. Setting this bit leads to store `0x0000` into the upper part of `R0` instead and resetting the lower part.

These are simple examples, aimed at illustrating the capabilities of the method. Depending on the instructions found in the assembly code, advanced manipulations are possible. Examples of using this fault model on real-life security codes are given in the next section.

5 Applications to real-life codes

In the following experiments, based on our characterisation results, we set the laser power to 0.8 W and duration to 135 ns. We achieve perfect repeatability at the correct injection delay.

[2] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html`

5.1 PIN verification algorithm

In order to apply the fault injection technique described above to a real security test-case, we targeted a constant-time 4-digit PIN verification algorithm with hardened booleans [12]. Its description is given in Algorithm 1.

Algorithm 1 Constant-time 4-digit VerifyPIN with hardened booleans.

```
1: trials = 3
2: reference_PIN[4] = {1, 2, 3, 4}
3: procedure VerifyPIN(user_PIN[4])
4:    authenticated = FALSE
5:    diff = FALSE
6:    if trials > 0 then
7:       for i ← 0 to 3 do
8:          if user_PIN[i] != reference_PIN[i] then
9:             diff = TRUE
10:      if diff == TRUE then
11:         trials = trials - 1
12:      else
13:         authenticated = TRUE
14:   return authenticated
```

This PIN verification algorithm is protected against simple power analysis [18] by a constant-time implementation. Therefore, an attacker cannot determine the correct digits by simply observing the execution time of the algorithm. This is achieved by systematically comparing all the digits of the user and reference PINs (see *for* loop on line 7 of Algorithm 1). Thus a perturbation attack is required to break such an implementation.

A first approach to perform a successful authentication using a fault attack without providing the correct user PIN could be to change the initialisation value of the *authenticated* variable (see initialisation on line 4 of Algorithm 1). By setting it to `TRUE` instead of `FALSE`, the authentication is successful even if the user PIN is wrong. However, the target implementation that we used employs hardened booleans. This common technique consists in storing booleans in bytes and encoding `TRUE` as `0x55` and `FALSE` as `0xAA` for instance. In this case, two bit-sets and two bit-resets are needed to turn `TRUE` into `FALSE`, making the attack very challenging and impractical in our fault injection setup since we can only perform bit-sets.

The approach we explored is then to corrupt the trials counter (see line 6 of Algorithm 1). Indeed, if we can bypass this comparison, then

an exhaustive search over all the possible PINs becomes feasible. The *if* instruction is compiled into the assembly code shown in Figure 8. The `CMP` instruction compares the *trials* variable, stored in `R3`, with 0. Then the `BLE` instruction branches to *address* if the result of the comparison is "less or equal".

C code	Assembly code
`if (trials > 0)`	`CMP R3, 0` `BLE address`

Fig. 8: C and assembly code for an *if* branch.

We chose to alter the register part of the `CMP` instruction to compare register `R7` instead of register `R3`, as shown in Figure 9. The ARM convention is to store the frame pointer in register `R7`, that is the address of the memory space allocated for the subroutine local variables. Thus the result of the comparison is always positive and the branch is never taken. Even if the trials counter reaches zero, the user and reference PINs are still compared. Therefore, an attacker can iterate over all the possible 4-digit PINs until authentication succeeds.

bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reference instructions																
Generic `CMP`	0	0	1	0	1	Rd			imm8							
`CMP R3, 0`	0	0	1	0	1	0	1	1	0	0	0	0	0	0	0	0
Register corruption ↓																
`CMP R7, 0`	0	0	1	0	1	**1**	1	1	0	0	0	0	0	0	0	0

Fig. 9: Fault on the register part of the `CMP` instruction to fault the comparison. The arrow indicates which bit is set by laser injection.

5.2 AES encryption

The second security use-case is the AES-128 encryption algorithm [10]. The algorithm consists in ten rounds, each round including the `AddRoundKey`, `SubBytes`, `ShiftRows` and `MixColumns` transformations, except for the last

round which does not include the `MixColumns` transformation. Another `AddRoundKey` is finally performed, which is the operation we targeted.

Algorithm 2 describes the `AddRoundKey` operation. It operates on a 4x4 state matrix [10]. Going through all the sixteen possible entries, the `AddRoundKey` operation consists in XORing the state matrix entry $S_{i,j}$ and a tenth round-key byte $K^{10}_{i,j}$, where i denotes the column and j denotes the row of the state matrix.

Algorithm 2 Add_round_key function.

1: **procedure** ADD_ROUND_KEY
2: **for** i ← 0 to 3 **do**
3: **for** j ← 0 to 3 **do**
4: $S_{i,j} = S_{i,j} \oplus K^{10}_{i,j}$

As shown in Algorithm 2, the `AddRoundKey` operation consists in two nested *for* loops. The C and assembly codes for this construction are shown on Figure 10.

C code	Assembly code

`for (int i=0; i<4; i++)`
`{`
`  for (int j=0; j<4; j++)`
`  {`
`  ...`
`  }`
`}`
	`MOV R0, 0`
	`addr_i:`
	`MOV R1, 0`
	`addr_j:`
	`...`
	`ADD R1, 1`
	`CMP R1, 3`
	`BLE addr_j`
	`ADD R0, 1`
	`CMP R0, 3`
	`BLE addr_i`

Fig. 10: C and assembly code for two nested *for* loops.

In order to fault the final `AddRoundKey` operation, we chose to alter the control flow and prematurely exit the *for* loops. By performing a fault on the `ADD` instruction, we can modify the data part and add 5 instead of 1 to the loop variable, as shown in Figure 11. This causes the *for* loop to end prematurely, since the exit condition is satisfied after the first iteration.

Below, the faulty bytes of the ciphertext are given by the expression $C_{i,j} \oplus K^{10}_{i,j}$ where $C_{i,j}$ denote the correct byte of the ciphertext found on the i-th column of the j-th row of the state matrix after completion of a fault-free encryption.

bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reference instructions																
Generic `ADD`	0	0	1	1	0	Rd			imm8							
`ADD R0, 1`	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1
Data corruption														↓		
`ADD R0, 5`	0	0	1	1	0	0	0	0	0	0	0	0	0	**1**	0	1

Fig. 11: Fault on the data part of the `ADD` instruction to prematurely escape a *for* loop. The arrow indicates which bit is set by laser injection.

By faulting the inner *for* loop on its first execution, in the first column of the state matrix, the three last bytes of the first row are faulty. The resulting ciphertext $\tilde{C}^{\text{inner}}_{i,j}$ is given in Equation (1).

$$\tilde{C}^{\text{inner}}_{i,j} = \begin{cases} C_{i,j} \oplus K^{10}_{i,j} & \text{if } i = 0,\ j \in [1..3] \\ C_{i,j} & \text{otherwise} \end{cases} \tag{1}$$

By faulting the outer *for* loop, only the bytes of the first column of the state matrix are XORed with the associated tenth round-key bytes. The last three columns are faulty. The resulting ciphertext $\tilde{C}^{\text{outer}}_{i}$ is given in Equation (2).

$$\tilde{C}^{\text{outer}}_{i,j} = \begin{cases} C_{i,j} \oplus K^{10}_{i,j} & \text{if } i \in [1..3],\ j \in [0..3] \\ C_{i,j} & \text{otherwise} \end{cases} \tag{2}$$

Holding these two faulty ciphertexts, the attacker can recover the last fifteen bytes of the tenth round key by XORing the fifteen faulty bytes with the bytes of the correct ciphertext as shown in Equation (3).

$$K^{10}_{i,j} = \begin{cases} \tilde{C}^{\text{inner}}_{i,j} \oplus C_{i,j} & \text{if } i = 0,\ j \in [1..3] \\ \tilde{C}^{\text{outer}}_{i,j} \oplus C_{i,j} & \text{if } i \in [1..3],\ j \in [0..3] \end{cases} \tag{3}$$

The first byte of the tenth round key $K^{10}_{0,0}$ must then be brute-forced, which is done in 2^7 attempts on average. The whole AES key can then be recovered by reversing the key schedule. To conclude, altering the control-flow of AES encryption and obtaining two faulty ciphertexts allows an attacker to fully recover the AES key with an average complexity of 2^7.

6 Discussion

6.1 Possible explanation for the observed fault model

The architecture of the flash memory in the microcontroller we targeted is a NOR flash memory (see Figure 12). In NOR flash memory, floating-gate transistors are connected in parallel between a bit-line and the ground. Previous work have highlighted that the sensitive areas in CMOS technology are the reverse biased PN junctions [23]. From this information we can propose the following explanation for the observed fault model described in Section 4.

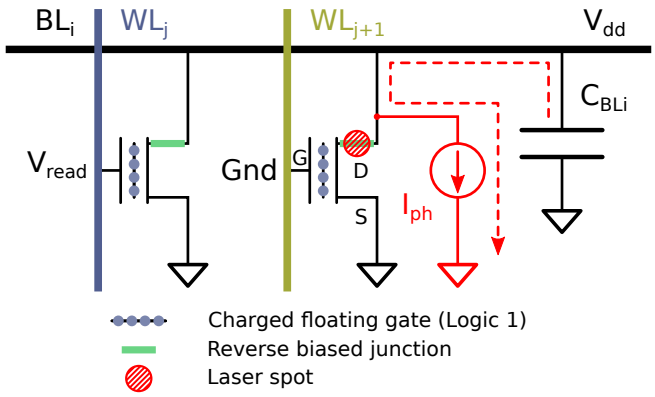


Fig. 12: Schematic of a bit-line in a NOR flash memory. The red elements indicate the effect of the laser shot.

We assume that a logic 1 is stored as a low voltage on the bit-line. When a bit is read from flash memory, the associated bit-line is pre-charged to V_{dd}. If the floating gate is charged, then the threshold voltage is high. If the floating gate is not charged, then the threshold voltage is low. By setting an intermediate voltage V_{read} on a word-line, low-threshold transistors pull the bit-line to ground while high-threshold transistors do not.

When a laser spot illuminates the drain of a blocked transistor, a photocurrent is induced between its drain and source (see red elements in Figure 12). If a low-voltage transistor is activated by the word-line, it pulls the bit-line to ground and a logic 1 is read. However if a high-voltage transistor is activated by the word-line, a photocurrent can be induced between its drain and source by the laser injection. As a consequence, its bit-line is pulled down to ground and a logic 1 is read. This explains the

asymmetry of the fault model, consisting only of bit-sets. We expect other targets embedding NOR flash memory to exhibit a similar asymetric fault model.

This physical mechanism can be applied to any floating-gate transistor of the flash memory. Assuming that bit-lines are horizontal and word-lines are vertical, it explains why we can sweep over the bits of a fetched instruction as we move the laser spot along the y-axis, affecting the bit-lines one after the other. However, moving along the x-axis affects transistors connected to the same bit-line, setting the same bit. This explains why moving along the y-axis allows to target specific bits, independently of the x-coordinate as shown in Figures 3a and 4a.

Finally, since the photocurrent spatial distribution depends on the injection power, it explains why two adjacent bits are faulty only with sufficient power, as shown in Figure 5. If the photocurrent distribution is large enought, transistors connected to different adjacent bit-lines are affected and adjacent bits are set.

6.2 Limitations

Mono-spot laser The fact that the laser we use has only one spot limits the number of bits that can be simultaneously set in the instruction. We observed either a single bit-set or two adjacent bit-sets. A multi-spot laser setup is thus useful to set multiple non-adjacent bits and extend the range of reachable modified instructions.

Bit-set only The observed fault model only consists of bit-sets. We did not observe any laser-induced bit-reset in this region of the circuit. Even though this limits the range of reachable modified instructions, Section 5 shows that this fault model still has numerous applications.

Control flow corruption mostly As demonstrated by two examples in Section 5, faulting the control flow of a program is feasible. However, given our fault model, faulting the data is often difficult while targeting the flash memory. Indeed, data is not hard-coded in the instructions but instead stored in RAM and fetched when needed. For example, it is impossible to perform safe-error attacks on AES encryption [4] since the AES round-key bytes are not hard-coded in the instructions.

Still, altering the control flow is an effective way to lower the security of algorithms though. In future works, some arithmetic operations could be modified to actually alter the data. However, this is very algorithm-specific and must be investigated for each case.

6.3 Reproducibility with a new target code and an identical microcontroller

Performing extensive characterisation and exploration of laser parameters to perform a successful fault injection is a time-consuming process and takes months. However, given the knowledge acquired and detailed above, reproducing these results on a new target code and an identical microcontroller would be much faster. First, it requires to decapsulate the chip and mount it on a suitable board for back-side laser injection. Then, the code shown in Listing 1 with a laser power of 0.5 W and duration of 200 ns can be used to find the y-coordinate at which each individual bit is set. An access to the assembly code of the target application is needed to identify the target instruction. After that, the ARM Architecture Reference Manual is used to identify a valid faulty instruction. Finally, the delay injection parameter must be tuned.

7 Conclusion

This article presented a new laser fault injection attack on the flash memory of a 32-bit microcontroller. Provided the right injection parameters, an attacker can set individual bits of the words fetched from the flash memory in a very predictable manner. Based on our characterisation results, we provided practical examples of fault injection affecting common security algorithms. Finally, we discussed how the hardware features of a NOR flash memory can explain the observed fault model. Future works on the topic will focus on examining state-of-the-art software countermeasures such as control flow integrity that may be relevant against the attacks that we demonstrated on implementations of a PIN verification and AES algorithms.

Acknowledgment

The authors would like to thank Colin O'Flynn from NewAE Technology Inc. for his help in providing the reference designs and bill of materials for the custom ChipWhisperer target board. We thank the Micro-PackS platform too for the PCB design and chip decapsulation.

References

[1] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. "An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2011, pp. 105–114.

[2] Hagai Bar-El et al. “The Sorcerer’s Apprentice Guide to Fault Attacks”. In: *Proceedings of the IEEE* 94.2 (2006), pp. 370–382.
[3] Alessandro Barenghi et al. “Low Voltage Fault Attacks on the RSA Cryptosystem”. In: *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2009, pp. 23–31.
[4] Johannes Blömer and Jean-Pierre Seifert. “Fault Based Cryptanalysis of the Advanced Encryption Standard (AES)”. In: *International Conference on Financial Cryptography*. Vol. 2742. 2003, pp. 162–181.
[5] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. “On the Importance of Eliminating Errors in Cryptographic Computations”. In: *Journal of Cryptology* 14.2 (2001), pp. 101–119.
[6] Jakub Breier, Dirmanto Jap, and Chien-Ning Chen. “Laser Profiling for the Back-Side Fault Attacks: With a Practical Laser Skip Instruction Attack on AES”. In: *Workshop on Cyber-Physical System Security*. 2015, pp. 99–103.
[7] Sebanjila Kevin Bukasa et al. “Let’s shock our IoT’s heart: ARMv7-M under (fault) attacks”. In: *International Conference on Availability, Reliability and Security*. 2018, 33:1–33:6.
[8] Hamid Choukri and Michael Tunstall. “Round Reduction Using Faults”. In: *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2005, pp. 13–24.
[9] Colin O’Flynn and Zhizhang (David) Chen. “ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research”. In: *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Vol. 8622. 2014, pp. 243–260.
[10] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. 2002.
[11] Amine Dehbaoui et al. “Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES”. In: *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2012, pp. 7–15.
[12] Louis Dureuil et al. “FISSC: A Fault Injection and Simulation Secure Collection”. In: *International Conference on Computer Safety, Reliability, and Security*. Vol. 9922. 2016, pp. 3–11.
[13] Jean-Max Dutertre et al. “Fault Round Modification Analysis of the advanced encryption standard”. In: *International Symposium on Hardware-Oriented Security and Trust*. 2012, pp. 140–145.
[14] Jean-Max Dutertre et al. “Laser fault injection at the CMOS 28 nm technology node: an analysis of the fault model,” in: *FDTC 2018*. 2018.
[15] Oscar M. Guillen, Michael Gruber, and Fabrizio De Santis. “Low-Cost Setup for Localized Semi-invasive Optical Fault Injection Attacks - How Low Can We Go?” In: *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Vol. 10348. 2017, pp. 207–222.
[16] Martin S. Kelly, Keith Mayes, and John F. Walker. “Characterising a CPU fault attack model via run-time data analysis”. In: *International Symposium on Hardware Oriented Security and Trust*. 2017, pp. 79–84.
[17] Chong Hee Kim and Jean-Jacques Quisquater. “Fault Attacks for CRT Based RSA: New Attacks, New Results, and New Countermeasures”. In: *Information Security Theory and Practices*. Vol. 4462. 2007, pp. 215–228.
[18] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis”. In: *Annual International Cryptology Conference*. Vol. 1666. 1999, pp. 388–397.

[19] Thomas Korak and Michael Hoefler. "On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2014, pp. 8–17.

[20] Nicolas Moro et al. "Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2013, pp. 77–88.

[21] Sébastien Ordas, Ludovic Guillaume-Sage, and Philippe Maurine. "Electromagnetic fault injection: the curse of flip-flops". In: *Journal of Cryptographic Engineering* 7.3 (2017), pp. 183–197.

[22] Lionel Rivière et al. "High precision fault injections on the instruction cache of ARMv7-M architectures". In: *International Symposium on Hardware Oriented Security and Trust*. 2015, pp. 62–67.

[23] Cyril Roscian et al. "Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2013, pp. 89–98.

[24] A. Sarafianos et al. "Electrical modeling of the photoelectric effect induced by a pulsed laser applied to an SRAM cell". In: *Microelectronics Reliability* 53.9 (2013), pp. 1300–1305.

[25] Jörn-Marc Schmidt and Christoph Herbst. "A Practical Fault Attack on Square and Multiply". In: *International Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2008, pp. 53–58.

[26] Nidhal Selmane, Sylvain Guilley, and Jean-Luc Danger. "Practical Setup Time Violation Attacks on AES". In: *European Dependable Computing Conference*. 2008, pp. 91–96.

[27] Sergei Skorobogatov. "Flash Memory 'Bumping' Attacks". In: *Cryptographic Hardware and Embedded Systems*. Vol. 6225. 2010, pp. 158–172.

[28] Sergei Skorobogatov. "Optical Fault Masking Attacks". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2010, pp. 23–29.

[29] Sergei P. Skorobogatov and Ross J. Anderson. "Optical Fault Induction Attacks". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Vol. 2523. 2002, pp. 2–12.

[30] Niek Timmers, Albert Spruyt, and Marc Witteman. "Controlling PC on ARM Using Fault Injection". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2016, pp. 25–35.

[31] Elena Trichina and Roman Korkikyan. "Multi Fault Laser Attacks on Protected CRT-RSA". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2010, pp. 75–86.

[32] Aurelien Vasselle et al. "Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2017, pp. 41–48.

[33] Jasper G. J. van Woudenberg, Marc F. Witteman, and Federico Menarini. "Practical Optical Fault Injection on Secure Microcontrollers". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2011, pp. 91–99.