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

Authors' version
https://doi.org/10.1109/HST.2019.8741030
2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)

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;
†Mines Saint-Etienne, CEA Tech, Centre CMP, 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 posed against secure embedded systems. Notable among these is laser fault injection, which is often considered as the most effective fault injection technique. Indeed, laser fault injection provides 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 corruption of instructions 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 explain the observed fault model.**

***Index Terms*—Fault attack, laser injection, flash memory**

I. Introduction

Physical attacks pose a considerable threat to the security of embedded systems. Provided physical access to a device, an attacker can exploit hardware-based vulnerabilities to bypass existing security measures. Among these techniques, fault injection consists in disturbing the operating conditions of a device, while a secure computation takes place, in order to retrieve secret information or be granted unauthorised privileges. Laser fault injection features a high spatial accuracy, which enables an attacker to induce single bit-flips in static memory cells of 8-bit [1] and 32-bit microcontrollers [2]. An explanation of the fault mechanism at the architectural level was proposed, based on the physical understanding of laser injection phenomenon [3], [4]. Conversely, this technique is expensive and difficult to carry out with numerous precise parameters to tune, which might result in endless explorations of the parameters space. While increasing chip integration enables designers to integrate complex 32-bit architectures, only few work investigate laser injection on these System-on-Chip architectures [2], [5]–[7]. Besides, none of them address 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 instructions can be set. The stored value remains untouched, only the read value is altered. For example, the data, the source, or the destination register of the fetched instruction can be altered, but also the opcode itself, potentially changing the instruction itself. Such modifications give rise to severe security concerns, since an attacker can then tamper with the instructions on the fly before they are 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 detail the influence of each parameter of the laser on the injected fault. Then 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 the NOR flash memory of our target.

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

The outline of the article is as follows. In Section II, we analyse previous works on laser fault injection, pointing out the current scarcity of results and understanding of fault injection on 32-bit microcontrollers. In Section III, we detail our experimental setup. In Section IV, we describe the obtained fault model and how it is affected by the parameters of our experimental setup. In Section V, we highlight how the previously described fault model applies to implementations of a PIN verification and AES-128 algorithms by demonstrating two attacks that we performed. In Section VI, we discuss a hypothesis on the physical phenomenon accounting for the observed fault model, as well as the limitations of our setup. Finally, Section VII concludes the article.

II. Previous work

Attack schemes against physical implementations of cryptographic algorithms require to define an abstract model of a device erroneous behaviour [8]. The complexity of fault model characterisation lies in the multiple features that influence a device response to fault injection, namely its micro-architecture, technology node, and sensitivity to the fault injection technique. Therefore, a comprehensive knowledge of the fault mechanism is hard to acquire.

In order to highlight the current scarcity of results on 32-bit microcontrollers, we define four levels of abstraction in Figure 1. The algorithmic level provides a description of the fault effects on an algorithm outputs, 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 of the target device. The physical level focuses on the physical phenomenon of the fault injection.

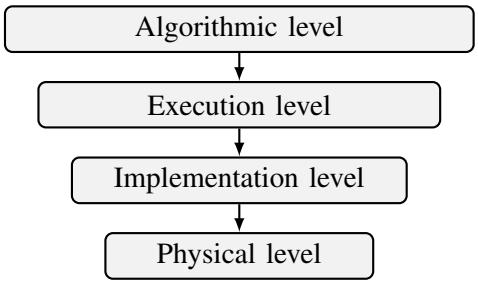


Fig. 1: Abstraction levels to describe a fault model.

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 [9]–[12]. In some cases, observations of low-level execution faults are provided, highlighting instruction or register corruption [13]–[15]. Balasch et al. demonstrated that a thorough characterisation of a device response to fault injection enables one to get a better understanding of the fault effects on the underlying hardware implementation [14]. At the same time, several authors observed that timing constraints violation could explain the observed fault models at the physical level [16], [17]. While none of these works addressed all four levels of abstraction, 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 [18] and execution level [5], [19]. However, the observed fault models lack a consistent framework. Several difficulties can be underlined while 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. They greatly increase the complexity of black-box fault effects analysis [20], [21] as already observed on 8-bit architectures [14].

Second, fault injection techniques leveraging timing constraints violation fail to catch local features of 32-bit micro-architectures. Indeed, attempts to characterise the effects of clock glitches on 32-bit architectures obtained very similar results to those with 8-bit architectures [17], [22], while different fault models were observed with optical injection depending on the injection locality [7].

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], [21]. However, chip sensitivity to the underlying physical phenomenon is not understood yet and lacks a consistent description [23].

Laser fault injection was introduced by Skorobogatov in 2002, based on related works on the simulation of ionising radiation in semiconductors devices [24]. Provided 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 reverse biased junctions of the illuminated transistors. This effect was investigated to describe the bit-flip fault model in SRAM cells [3], [4] with attacks on AES encryption [1] and secure program register [2]. 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 32-bit architecture.

Previous work on flash memory vulnerabilities pointed out the memory control logic as a sensitive area to laser fault injection [25]–[27] although the authors do not explain the underlying fault mechanism.

In this article, we characterise the effect of laser injection in the flash memory area and observe that single bit-set in data fetched from the flash memory can be performed. We then give several examples of instructions corruption. We demonstrate the validity of the fault model on the implementations of two security algorithms in Section V and propose a transistor-level explanation of the physical fault mechanism in Section VI.

III. Methods and Experimental Setup

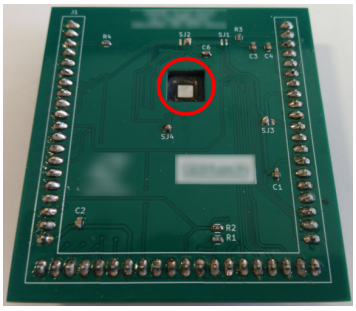
A. Target board and microcontroller

The target microcontroller that 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. It is mounted on a ChipWhisperer [28] target board, with the chip soldered below and facing up. We designed a custom target board suitable for laser injection thanks to the open-source hardware information provided for the ChipWhisperer platform[1]. The target microcontroller runs at the 7.4 MHz frequency fixed by the ChipWhisperer platform, corresponding to a 135 ns clock period. An opening was cut on the PCB board, just under the chip, to give access to its back-side.

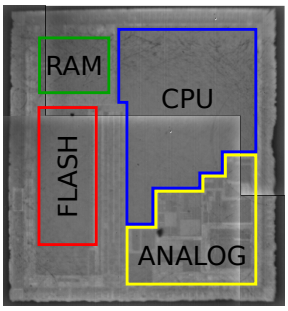
To perform laser fault injection, the back of the chip must be decapsulated to show the silicon substrate. This is performed by chemical processing before the chip is mounted on the board. The decapsulation must be carried out with great care, especially regarding the amount of chemical product used to decapsulate: too few keeps part of the die covered, thereby reducing the fault injection area, too much makes the decapsulated chip very fragile. Thinning the die was not necessary here. A picture of the board is shown in Figure 2. This target board is then mounted on the ChipWhisperer motherboard and put in place on the laser bench.

B. Laser characteristics and parameters

The laser source uses an acousto-optic technology to generate an infrared laser beam at a wavelength of 1,064 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.



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



(b) Tiled infrared picture of the target microcontroller. The flash memory is framed in red.

Fig. 2: Target microcontroller on the custom board.

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

C. Characterisation codes

Leveraging simple test codes, one can characterise the target unexpected behaviours and the fault model dependence on experimental conditions. As opposed to [18], we state that the conclusions drawn from test codes about the underlying fault mechanisms can be extended to any codes, as observed in [14]. Attack scenarios on software implementation of secure algorithms were remarkably consistent with our characterisation results. Codes given in Listings 1 and 2 were used to characterise the fault model. 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

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

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

1) 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 Characterisation of bit-set 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 and stores it in register `R0`. A fault is detected on the third bit if, for instance, the actual value stored in register `R0` is `0x00000008` after a laser injection was performed. The test word `0x00000000` was used to highlight bit-sets, as prior 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 the target instruction to prevent the effects of instruction corruption on the observed fault model.

2) Characterisation of fault sensitivity over time: The second characterisation code aims at highlighting the most fault-sensitive moments in the execution of instructions. For this, after finding out the location where a given bit can be faulted 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`.

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.

IV. Observable Fault Model

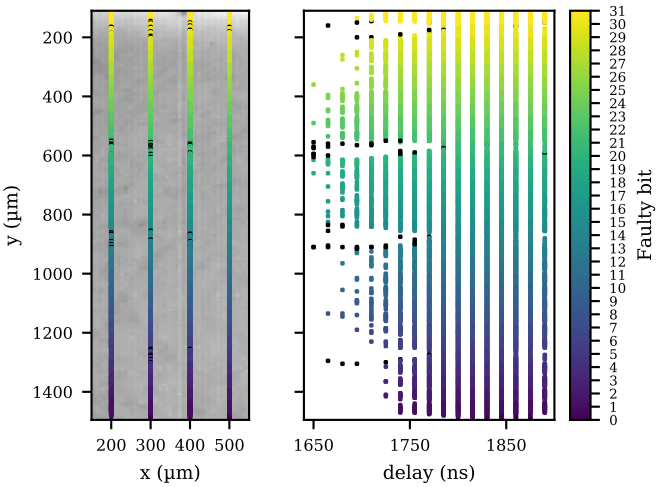
A. Parameters and types of faults

1) Characterisation of bit-set location: We observed that moving along the y-axis (longest side) on the flash

Listing 2 Characterisation of sensitivity over time.

```
1  # Initialising registers R0, R1, R4, R5, R6,
2  # R8 and R9 to 0xFFFFFFFF
3  MOVW R0, 0x0000
4  MOVW R1, 0x0000
5  MOVW R4, 0x0000
6  MOVW R5, 0x0000
7  MOVW R6, 0x0000
8  MOVW R8, 0x0000
9  MOVW R9, 0x0000
10 # Reading back the registers
```

memory area 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 the 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). Black dots show locations where the chip stopped responding. Figure 3b shows an optimal delay, around 1,850 ns here, where all the bits of the fetched word can be set depending on the y coordinate of the laser spot over the flash memory area.



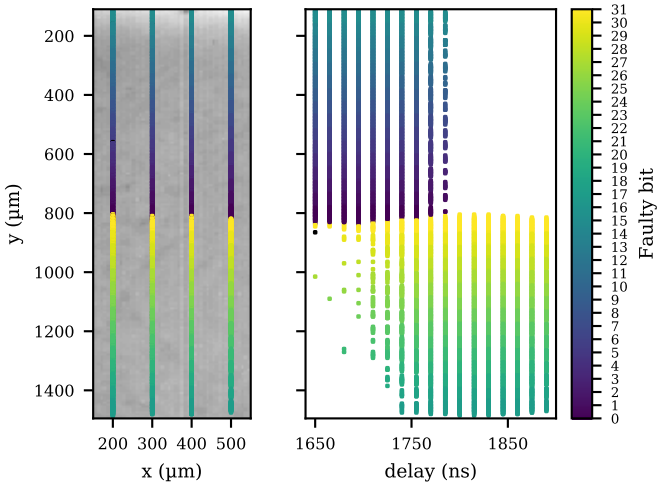
(a) Position to set the bits of the fetched data.

(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.

Figure 4 is the same as Figure 3 but for an *unaligned* word. In this situation, the upper and lower 16-bit parts 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 on Figure 4b. This observation reflects the organisation of the binary code, and supports the assumption that the sequential access to the flash memory is faulted during the fetch operation.



(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.

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 targeted bit of an instruction, the laser spot must be positioned to fault either the bit n if the instruction is aligned, or the bit $n+16$ mod 32 if the instruction is unaligned. The previous results lead to several observations. First, predictable and repeatable faults can be achieved by targeting a fixed location on the flash memory area. In the specific case where the corrupted instruction has no effect on the outputs of the test routine, the equivalent of an instruction skip is observed. Second, targeting the same instruction with the same spot location might induce an entirely different behaviour if the instruction memory alignment is different. The fault model dependency on laser injection parameters is described below.

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 reaching 100 % of fault occurrence for a range of almost 50 ns, while performing only single-bit 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 single-bit faults, careful tuning of the laser pulse power and duration is required.

2) 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. Results are shown in Figure 6, which shows how the probability of occurrence of a fault changes with the injection delay.

We observe in Figure 6 that the gap between two peaks of fault sensitivity is always a multiple of the clock period. It supports 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.

B. Modification of a MOVW instruction

As an illustrative example of the possibilities offered by the fault model, we performed fault injection on a `MOVW` instruction. This 32-bit instruction loads a 16-bit value into the lower part of a 32-bit register and resets the upper part. The opcode, the destination register (denoted as Rd) and the n-bit data (denoted as "i" or "imm$n$") of the instruction are shown in the upper part of Figure 7. An example `MOVW` instruction is 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 that we did actually perform on the target. 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 be stored was altered. Setting this bit led 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 was altered. Setting this bit led 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 was altered. This changed

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

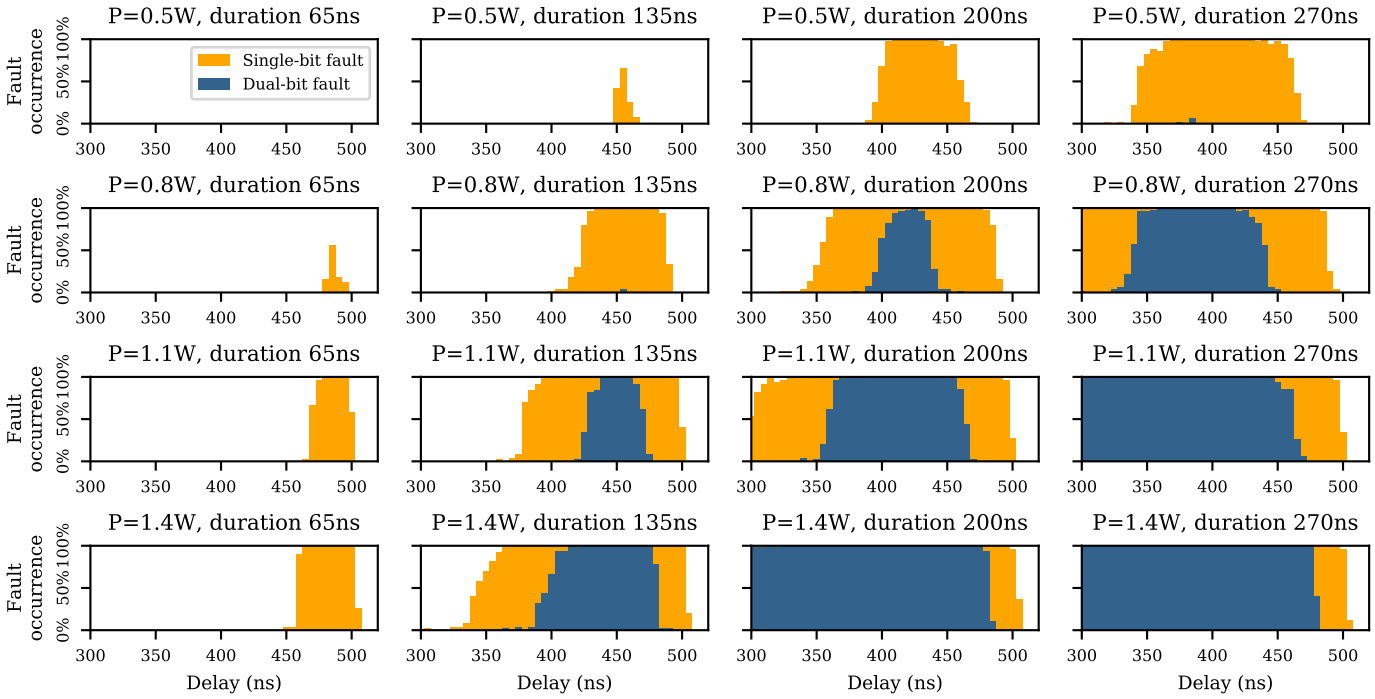


Fig. 5: Fault occurrence rate and types of faults for two laser injection parameters: power and duration.

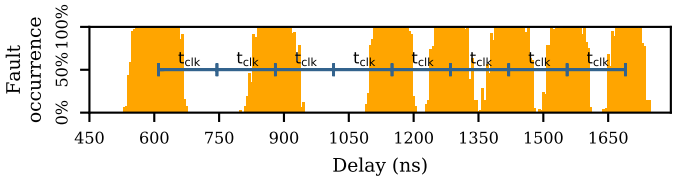


Fig. 6: Periodicity of the fault occurrence rate for a 0.8 W laser pulse of 135 ns.

the instruction from `MOVW` to `MOVT`. Setting this bit led to store `0x0000` into the upper part of `R0` instead, without altering the lower part.

These simple examples aim at illustrating the capabilities of the method. Depending on the instructions of the assembly code, advanced manipulations are possible.

V. Applications to Real-Life Codes

This section shows how to leverage the fault model described above on real-life security codes. We did perform all the experiments and obtained perfect repeatability, in accordance with the fault model described above. It is worth noting that we did not observe any degradation or wear of the Flash memory under attack.

A. 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 [29]. 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:      dummy = TRUE
7:      if trials > 0 then
8:          for i ← 0 to 3 do
9:              if user_PIN[i] != reference_PIN[i] then diff = TRUE
10:             else dummy = FALSE
11:         if diff == TRUE then trials = trials - 1
12:         else authenticated = TRUE
13:     return authenticated
```

The PIN verification algorithm is protected against simple power analysis [30] by a constant-time implementation. Therefore, an attacker cannot determine the correct digits one after the other 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 8 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

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	0	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	0	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	0	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	0	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.

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 7 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 destination register part of the `CMP` instruction to force a comparison with register `R7` instead of register `R3` (see Figure 9). The ARM convention is to store in register `R7` the address of the SRAM space, called frame pointer, allocated for the subroutine local variables. The result of the comparison is thus 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 destination register part of the `CMP` instruction to fault the comparison.

B. AES encryption

The second security use-case is the AES-128 encryption algorithm [31]. The algorithm consists of ten rounds, each round including `AddRoundKey`, `SubBytes`, `ShiftRows` and `MixColumns` transformations, except for the last round which does not include the `MixColumns` transformation. A final `AddRoundKey` is then performed, which is the transformation we targeted.

Algorithm 2 describes the `AddRoundKey` transformation. It operates on a 4x4 state matrix [31]. Going through all the sixteen possible entries, the `AddRoundKey` transformation 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.

Algorithm 2 Add_round_key transformation.

```
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} ⊕ K^{10}_{i,j}
```

As shown in Algorithm 2, the `AddRoundKey` transformation 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` transformation, 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 (see Figure 11). This causes the *for* loop to end prematurely, since the exit condition is satisfied after the first iteration. As a consequence, 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.

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).

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

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

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

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 (see Equation (3)).

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

The first byte of the tenth round key $K^{10}_{0,0}$ must then be brute-forced, which is easily 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.

VI. Discussion

A. Possible explanation for the observed fault model

The architecture of the flash memory in the microcontroller we targeted is a NOR flash (see Figure 12). Previous work has highlighted that the sensitive areas of CMOS are reverse biased PN junctions [32]. From this information, we can propose the following explanation for the observed fault model described in Section IV.

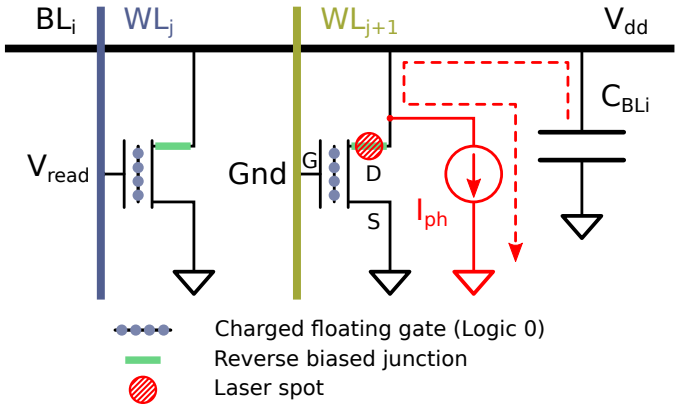


Fig. 12: Schematic of a bit-line in a NOR flash memory. The red elements indicate the effect of the laser shot.

When a bit is read from flash memory, the associated bit-line is pre-charged to V_{dd}. A floating-gate transistor, connected to this bit-line, is activated by its word-line. If the floating gate is charged, then the threshold voltage is high. If there is no charge on the floating gate, then the threshold voltage is low. A floating-gate transistor

with no voltage on its control gate is blocked, regardless of the charge stored on its floating gate. By setting an intermediate voltage V_{read} on the control gate of the floating-gate transistor, low-threshold transistors pull the bit-line to ground while high-threshold transistors do not.

In NOR flash memory, floating-gate transistors are connected in parallel between a bit-line and the ground. When a laser spot illuminates the drain of a blocked transistor, a photocurrent I_{ph} is induced between its drain and the bulk (connected to ground). This is illustrated by the red elements in Figure 12. As a consequence, its bit-line is pulled down to ground. Even though the word-line activates a floating-gate transistor that stores charges (see left-hand side of Figure 12), then the bit-line would still be pulled to ground by the laser-induced photocurrent. There is no physical mechanism to explain how a laser shot could *prevent* photocurrent from being drawn. Assuming a logic 1 stored in memory is encoded as a low voltage on the bit-line, this explains the asymmetry of the fault model which shows only bit-sets.

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 targeting specific bits, independently of the x-coordinate as shown in Figures 3a and 4a.

Finally, the power dependency of the photocurrent spatial distribution explains why two adjacent bits can be faulted with sufficient power, as shown in Figure 5. By affecting transistors connected to different adjacent bit-lines, adjacent bits can be set.

B. Limitations

1) 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.

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

3) Control flow corruption mostly: As demonstrated by two examples in Section V, faulting the control flow of a program is feasible. However, given our fault model, faulting the data is impossible most of the time 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 [33] since the AES round-key bytes are not hard-coded in the instructions.

Altering the control flow is already 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.

C. Reproducibility with a new target code and an identical microcontroller

Performing extensive characterisation and exploration of laser parameters to perform a correct fault injection is a time-consuming process and can take months. However, reproducing these results on a new target code and an identical microcontroller would be much faster. It would first require 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 coordinates at which each individual bit is set. 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 find out a valid faulty instruction. Finally, the delay injection parameter must be tuned.

VII. 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 flash memory in a very predictable manner. Based on our characterisation results, we provided practical examples of control flow and data corruption 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 the PIN verification and AES algorithms.

Acknowledgement

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] J.-M. Dutertre, A.-P. Mirbaha, D. Naccache, A.-L. Ribotta, A. Tria, and T. Vaschalde, "Fault round modification analysis of the advanced encryption standard," in *International Symposium on Hardware-Oriented Security and Trust*, 2012, pp. 140–145.

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

[3] A. Sarafianos, C. Roscian, J.-M. Dutertre, M. Lisart, and A. Tria, "Electrical modeling of the photoelectric effect induced by a pulsed laser applied to an sram cell," *Microelectronics Reliability*, vol. 53, no. 9, pp. 1300–1305, 2013.

[4] J.-M. Dutertre, V. Beroulle, P. Candelier, S. D. Castro, L.-B. Faber, M.-L. Flottes, P. Gendrier, D. Hély, R. Leveugle, P. Maistri, G. D. Natale, A. Papadimitriou, and B. Rouzeyre, "Laser fault injection at the CMOS 28 nm technology node: An analysis of the fault model," in *FDTC 2018*, 2018.

[5] E. Trichina and R. Korkikyan, "Multi fault laser attacks on protected CRT-RSA," in *Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2010, pp. 75–86.

[6] M. S. Kelly, K. Mayes, and J. 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.

[7] O. M. Guillen, M. Gruber, and F. D. 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.

[8] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of eliminating errors in cryptographic computations," *Journal of Cryptology*, vol. 14, no. 2, pp. 101–119, 2001.

[9] H. Choukri and M. Tunstall, "Round reduction using faults," in *Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2005, pp. 13–24.

[10] C. H. Kim and J.-J. 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.

[11] J. Schmidt and C. Herbst, "A practical fault attack on square and multiply," in *International Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2008, pp. 53–58.

[12] J. G. J. van Woudenberg, M. F. Witteman, and F. Menarini, "Practical optical fault injection on secure microcontrollers," in *Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2011, pp. 91–99.

[13] A. Dehbaoui, J.-M. Dutertre, B. Robisson, and A. Tria, "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.

[14] J. Balasch, B. Gierlichs, and I. 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.

[15] J. Breier, D. Jap, and C.-N. 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.

[16] N. Selmane, S. Guilley, and J.-L. Danger, "Practical setup time violation attacks on AES," in *European Dependable Computing Conference*, 2008, pp. 91–96.

[17] T. Korak and M. 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.

[18] S. K. Bukasa, R. Lashermes, J.-L. Lanet, and A. Legay, "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.

[19] N. Timmers, A. Spruyt, and M. Witteman, "Controlling PC on ARM using fault injection," in *Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2016, pp. 25–35.

[20] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "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] L. Rivière, Z. Najm, P. Rauzy, J.-L. Danger, J. Bringer, and L. Sauvage, "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.

[22] A. Barenghi, G. Bertoni, E. Parrinello, and G. Pelosi, "Low voltage fault attacks on the RSA cryptosystem," in *Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2009, pp. 23–31.

[23] S. Ordas, L. Guillaume-Sage, and P. Maurine, "Electromagnetic fault injection: The curse of flip-flops," *Journal of Cryptographic Engineering*, vol. 7, no. 3, pp. 183–197, 2017.

[24] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *International Workshop on Cryptographic Hardware and Embedded Systems*, vol. 2523, 2002, pp. 2–12.

[25] S. Skorobogatov, "Flash memory 'bumping' attacks," in *Cryptographic Hardware and Embedded Systems*, vol. 6225, 2010, pp. 158–172.

[26] ——, "Optical fault masking attacks," in *Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2010, pp. 23–29.

[27] F. Cai, G. Bai, H. Liu, and X. Hu, "Optical fault injection attacks for flash memory of smartcards," in *International Conference on Electronics Information and Emergency Communication*, IEEE, 2016, pp. 46–50.

[28] 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.

[29] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens, "FISSC: A fault injection and simulation secure collection," in *International Conference on Computer Safety, Reliability, and Security*, vol. 9922, 2016, pp. 3–11.

[30] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual International Cryptology Conference*, vol. 1666, 1999, pp. 388–397.

[31] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. 2002.

[32] C. Roscian, A. Sarafianos, J.-M. Dutertre, and A. Tria, "Fault model analysis of laser-induced faults in SRAM memory cells," in *Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2013, pp. 89–98.

[33] J. Blömer and J.-P. Seifert, "Fault based cryptanalysis of the advanced encryption standard (AES)," in *International Conference on Financial Cryptography*, vol. 2742, 2003, pp. 162–181.