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 a considerable threat to the security of embedded systems. Provided an access to a physical instance of the target system,

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

hardware-based vulnerabilities can be exploited. Among physical attacks, fault attacks consists in disturbing the behaviour of the device, thereby allowing an attacker to gain knowledge of secret data or control over the device by exploiting the modified operation. Laser fault injection is probably the most powerful injection method since it provides the best spatial resolution. Conversely, it is also expensive and complex to carry out with numerous precise parameters to tune that can rapidly result in endless parameters space exploration. When applied to simple 8-bit targets, this fault injection method proved effective [26, 13, 5]. The fault injection mechanism is well understood after substantial work on laser injection [22, 23]. However, only little work mentions its application to complex 32-bit modern microprocessors [29, 16, 30], which makes it difficult to explain the observed fault models in a consistent framework. Thus knowledge obtained about the feasibility of attacks on 8-bit targets is not directly transferable to complex 32-bit ones.

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, in a fetched instruction, the data, the source or destination register can be tampered with, but also the opcode itself, potentially changing an instruction completely. Regarding security, such a modification is very powerful, 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 investigate the influence of each parameter of the laser setup on the injected fault. We then apply this fault model to real-life codes and show how it undermines their security. Finally, we provide a physical explanation for the observed fault model, which is consistent with the micro-architecture of the 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 previously described fault model applies to implementations of the VerifyPIN and AES-128 algorithms. 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 injection techniques aim at exploiting vulnerabilities in the implementation of secure algorithms by disrupting the operation of an electronic device. Attack schemes against cryptographic algorithms leverage so called fault model, as introduced by Boneh et al. to factor the RSA modulus [4]. The complexity of fault model characterisation lies in the multiple factors that influence a device response to fault injection, namely its micro-architecture, technology node and susceptibility to the physical phenomenon accounting for the fault. Since a comprehensive knowledge of the fault mechanism is hard to acquire, several levels of abstraction can be used to simplify the analysis of a device behaviour.

For this purpose, we introduce four levels of abstraction on 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 accounting for the fault injection. Fault description covering the four levels of abstraction provides a deep understanding of a fault injection attack.

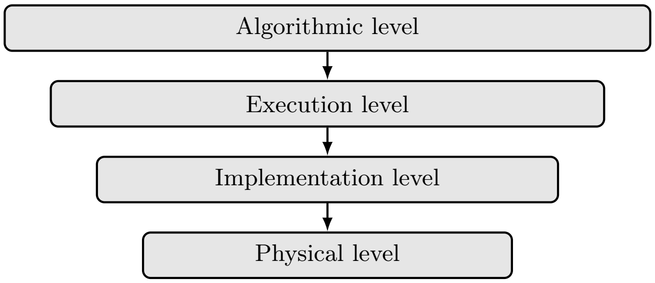


Fig. 1: Abstraction levels for the description of a fault model

Laser fault injection was introduced by Skorobogatov in 2002, based on related works on the simulation of ionizing radiation in semiconductors devices [27]. Provided an access to the chip surface, 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 illuminated transistors. This effect was investigated to describe the bit-flip fault model in static memory cells [23, 14] with practical attacks on AES encryption [13] and secure program register [30]. 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 the target micro-architecture.

As for the target devices, substantial work has been done to understand fault injection on 8-bit microcontrollers. As stated before, a disctinction can be made between fault models observed at the algorithmic level, such as round reduction [8], procedure skip [24], corruption of one byte of the AES state [25, 11] and fault models described at the execution level, such as instruction skip [11, 5] and instruction corruption [1]. Although the latter descriptions can account for the former in most of the cases, an in-depth characterisation of the effect of a fault enables one to gain insight into the vulnerabilities of the underlying hardware implementation. For instance, Balasch et al. gathered evidence that a fault on instruction fetch and memory transfer operations accounts for instruction and data corruption [1]. At the same time, several authors observed that timing constraints violation could account for the observed fault models [25, 18]. While none of these works addressed all four levels of fault 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 [6] and execution level [29, 28]. Difficulties arise as one try to apply insight gained on 8-bit architectures on 32-bit architectures.

First, as transistor size shrinks and chip size increases, the number of injection configurations grows exponentially, making exhaustive parameters exploration impractical. Pipeline mechanism, interruptions and fetch prediction, which improve performance of 32-bit architectures, increase the complexity of fault effects analysis.

Second, fault injection techniques leveraging timing constraints violation fail to catch local features of 32-bit micro-architectures. Indeed, several attempts to characterise the effects of clock glitches on 32-bit architectures obtained very similar results to those with 8-bit architectures [2, 18], while different fault models were observed with optical injection depending on the injection locality [15].

Third, previous work on electromagnetic fault injection lack a physical description [20]. This difficulty was underlined in a first attempt to describe fault models on 32-bit micrcontrollers at the execution and implementation level based on experimental results [19]. The authors of [21] faced the same difficulties incriminating the cache read mechanism of a 32-bit architecture.

All these considerations make laser fault injection well suited to characterise the effects of fault injection on 32-bit architectures at the physical level. Previous works on laser fault injection in flash memory emphasize on the effects of laser injection at the algorithmic and execution level [26, 7]. Conversely, in this article, we address all four levels of abstraction by characterising single bit-set in data fetched from flash memory. We provide a transistor-level explaination of the physical fault mechanism and demonstrate the validity of the fault model on the implementations of two secure algorithms.

3 Methods and experimental setup

3.1 Target board and microcontroller

The target microcontroller that we used for our experiments embeds an ARM Cortex-M3 core and 128 kB of flash memory and is manufactured at the 90 nm technology node. It is mounted on a ChipWhisperer [9] target board, with the chip soldered below and facing up. We designed a custom target board suitable for laser injection, designed 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 the 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. Indeed, too few product keeps part of the die covered, thereby reducing the fault injection area. Conversely, too much product exposes the bonding wires, making the decapsulated chip very fragile. A picture of the board is shown in Figure 2. This target board is mounted on the ChipWhisperer motherboard, which is then put in place on the laser 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

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

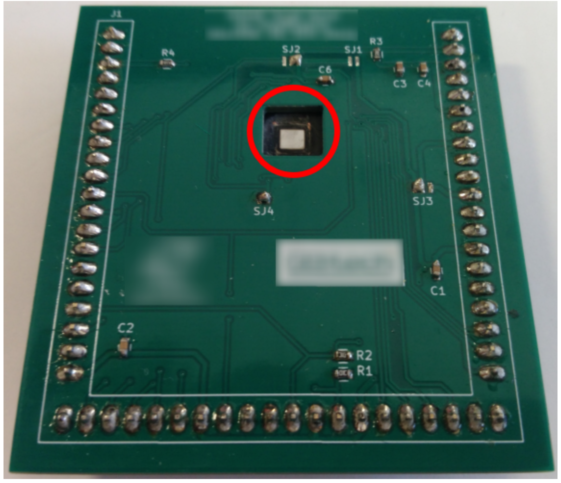


Fig. 2: Picture of the custom target board. The decapsulated chip mounted below with the silicon substrate visible is circled in red.

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 maximum power of 3 W. The laser pulse is directed to the focusing system by an optical fiber. The focusing system allows to focus the laser to a spot of diameter 5 µm. We assume here that the laser beam is correctly focused to the smallest spot size. The laser shot is triggered by an external input, generated by the target device as it is usually the case for fault injection. 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 is able to characterise the target unexpected behaviours and the fault model dependence on experimental conditions. As opposed to [6], we state that the underlying fault mechanism accounting for the observed fault models does not depend on the code being executed, as observed in [1]. Attack scenarios on software implementation of secure algorithms were remarkably consistent with out characterisation results. The following two codes were used to characterise the type of fault we obtained. 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 is said to be aligned if it is stored at an address which is a multiple of 32 bits.

Characterisation of bit-set location The first code highlights bit-sets in the 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 data stored at the *test_ data* label and stores it in `R0`. It executes in two clock cycles. In the first clock cycle, the offset to 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. The advantage of faulting raw data instead of an instruction is that it allows to observe the bit-set on all the 32 bits, whereas a 32-bit instruction always contains several `1`s, for which the bit-set is not observable. Dummy instructions (`NOP`) are inserted before and after the target instruction to make sure that only this instruction is affected by the laser shot. Under normal conditions, the observable output is the 32-bit `0x00000000` value, stored in `R0`.

Characterisation of fault sensitivity over time The second characterisation code we used aims at highlighting the most fault-sensitive moments in the execution of instructions. For this, after finding out where a given bit can be faulted thanks to the code given in Listing 1, we modify the delay parameter to sweep over the target instructions, shown in Listing 2, from line 3 to 9. Under normal conditions, the output after executing the code shown in Listing 2 consists of several 32-bit `0x0000FFFF` values stored in all the registers.

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
```

4 Observable fault model

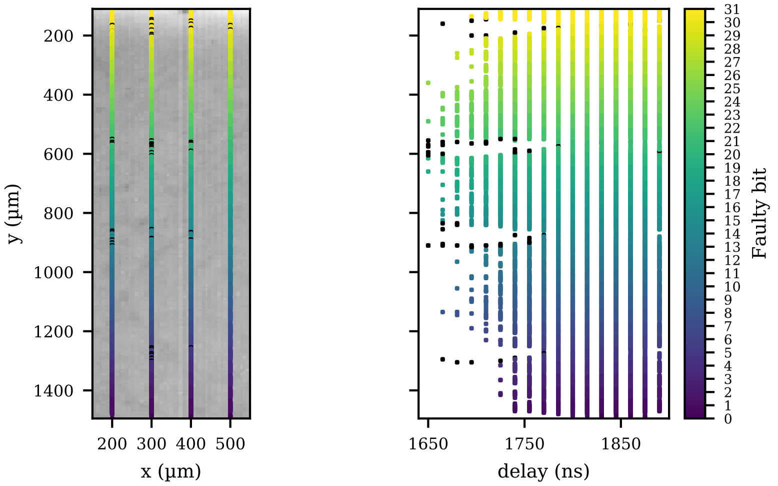
4.1 Parameters and types of faults

We present now the influence of the laser parameters on the fault injection process.

Characterisation of bit-set location We observed that moving along the y-axis on the flash memory area allows to precisely target the bits of the fetched data one after the other. Conversely, moving along the x-axis does not change the affected bit. Figure 3a shows a cartography of the faulty bits with a x-step of 100µm and a y-step of 5µm for *aligned* data. It is clearly visible that the affected bit is directly related to the y position. Figure 3b shows that there is an optimal delay, around 1850 ns in this case, where all bits can be faulted.

Figure 4 is the same as Figure 3 but for *unaligned* data. In this situation, the upper and lower parts of the data are swapped (see Figure 4a). Moreover, they are not fetched at the same time, since the upper part is faulted one clock period after the lower part, as shown in Figure 4b.

When performing the fault, an attacker does not know if the target instruction is aligned or not. Therefore, according to Figures 3 and 4, if the n-th bit of the instruction is targeted the laser spot must be positioned



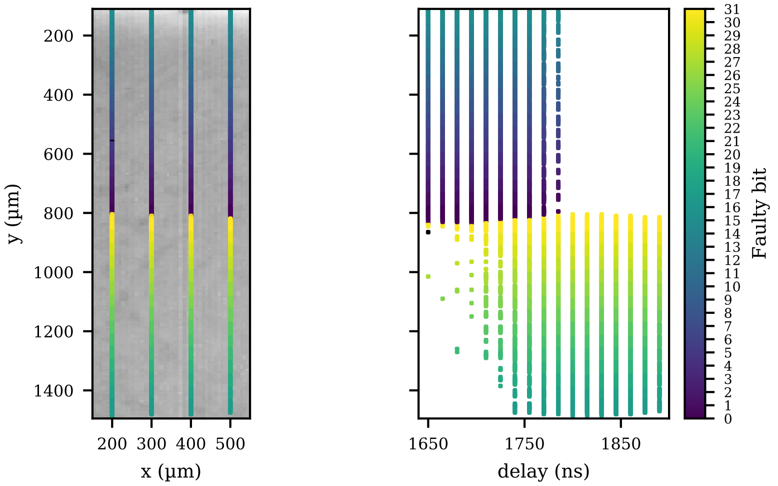
(a) Position in flash memory to fault the bits of the fetched data

(b) Timing of the optimal delay to fault 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.

where the bit n is faulted if the instruction is aligned, and at the position where the bit $n + 16 \bmod 32$ is faulted if the instruction is unaligned. After we found a position at which a bit can be faulted, 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 occurence for a range of almost 50 ns, while performing only mono-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 mono-bit 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 faulted, 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 over time.



(a) Position in flash memory to fault the bits of the fetched data

(b) Timing of the optimal delay to fault 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.

Two interesting things can be observed in Figure 6. First of all, the separation between two peaks of fault sensitivity is always a multiple of the clock period. It implies that, over a clock period, some instants are sensitive to fault injection while others are not. Moreover, as observed on the left-hand side of Figure 6, at some periods, no instruction is faulted. Thus in order to move the fault from one instruction to the other, it is sometimes necessary to wait more that one clock cycle. This could be explained by the fact that the microcontroller uses a 3-stage pipeline. Therefore, we can assume that the fetched timing depends on how full the pipeline is already.

To conclude, the delay parameter has an influence on the fault occurrence. Nevertheless, we observed that for every instruction there is a correct delay parameter that allows to fault this instruction with 100% probability.

4.2 Modification of a MOVW instruction

As an illustrative example of the possibilities offered by the fault model, we chose to alter a `MOVW` instruction. The purpose of this 32-bit instruction

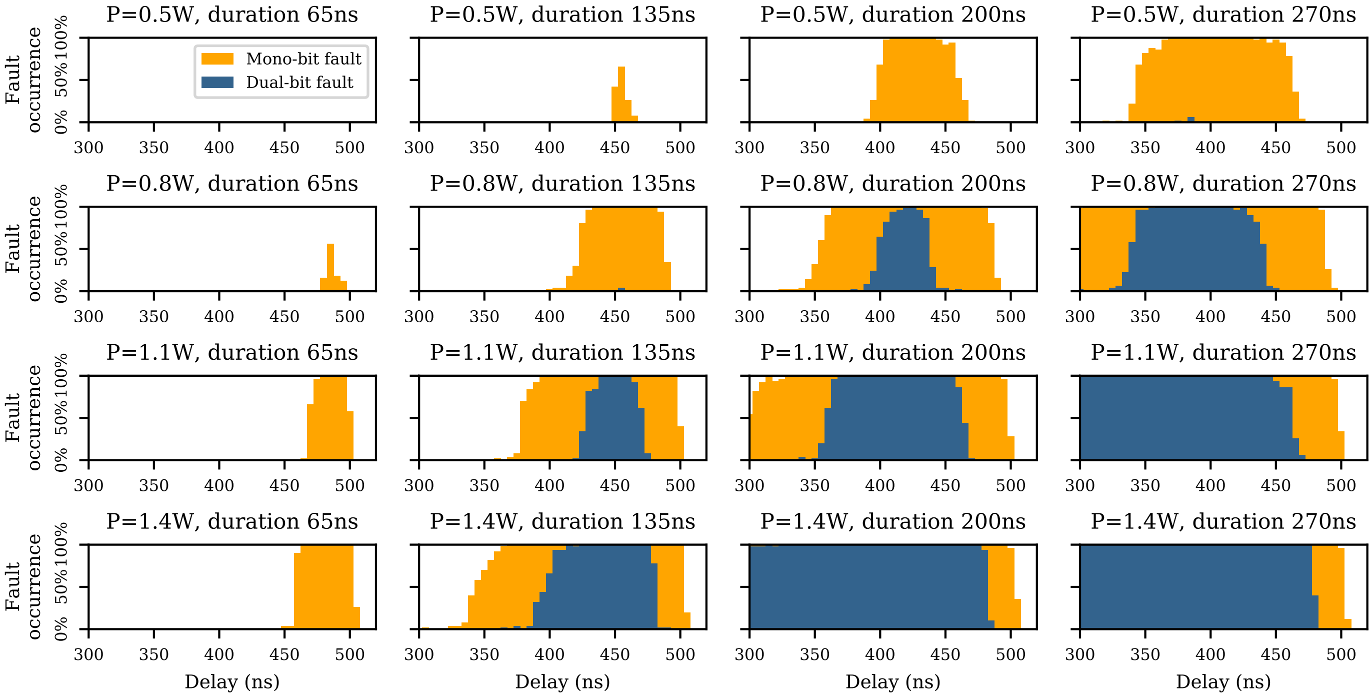


Fig. 5: Occurrence and types of faults for two laser injection parameters: power and duration.

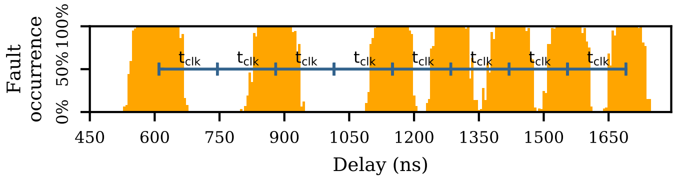


Fig. 6: Periodicity of the fault occurrences for a 0.8 W laser pulse of 135 ns

is to load a 16-bit value into the lower part 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*) 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.

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

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 2nd bit of the instruction, the data to be stored is altered. Setting this bit leads to store `0x0004` instead of `0x0000` into `R0`.

By performing a bit-set on the 8th bit 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 23rd bit of the instruction, the opcode is altered. This completely 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 codes are given in the next section.

5 Applications to real-life codes

5.1 VerifyPIN bypass

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

This VerifyPIN algorithm is protected against simple power analysis [17] by a constant-time implementation. Therefore, an attacker cannot determine the correct digits one after the other by simply observing

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
```

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 preferred approach that 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”, that is if the `Z` or `N` flags are set.

We chose to alter the register part of the `CMP` instruction (see Figure 9). Indeed, in the assembly code, we observed that before the *trials* variable is compared, the frame pointer is stored in `R7`. The value of the frame pointer is always greater than zero by construction. Therefore, by changing the

C code	Assembly code

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

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

register part from `R3` to `R7`, the comparison does not set either the `Z` or the `N` flag.

Even if the trials counter reaches zero, this will have no effect and the user and reference PINs will still be compared. Therefore, an attacker can simply iterate over all the possible 4-digit PINs until the correct one is found and authentication is successful.

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
Data 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 that we targeted is the AES-128 encryption algorithm [10]. The algorithm consists in 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. This final `AddRoundKey` 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 XORing the state matrix entry and a tenth round-key byte. We assume here that the round-keys are pre-computed and stored in an 11x4x4 array.

Algorithm 2 Add_round_key function.

```
procedure ADD_ROUND_KEY
  for i ← 0 to 3 do
    for j ← 0 to 3 do
      state[i][j] = state[i][j] ^ round_key[round][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 in Figure 10.

```
C code                                          Assembly code


for (int i=0; i<4; i++)                         MOV R0, 0
{                                               addr_i:
  for (int j=0; j<4; j++)                       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 (see Figure 11). This causes the *for* loop to end prematurely, since the exit condition is satisfied after the first iteration. The following notation is used: C_i is the i-th byte of the ciphertext and K_i^{10} is the i-th byte of the tenth round-key.

By faulting the inner *for* loop on its first execution, in the first column of the state matrix, only the first byte is XORed with the associated tenth round-key byte. The resulting ciphertext $\tilde{C}_i^{\text{inner}}$ is given in Equation (1).

$$\tilde{C}_i^{\text{inner}} = \begin{cases} C_i \oplus K_i^{10} & \text{if } i \in [1..3] \\ C_i & \text{otherwise} \end{cases} \tag{1}$$

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 outer *for* loop, only the first row bytes of the state matrix are XORed with the associated tenth round-key bytes. The resulting ciphertext $\tilde{C}_i^{\text{outer}}$ is given in Equation (2).

$$\tilde{C}_i^{\text{outer}} = \begin{cases} C_i \oplus K_i^{10} & \text{if } i \in [4..15] \\ C_i & \text{otherwise} \end{cases} \qquad (2)$$

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)).

$$K_i^{10} = \begin{cases} \tilde{C}_i^{\text{inner}} \oplus C_i & \text{if } i \in [1..3] \\ \tilde{C}_i^{\text{outer}} \oplus C_i & \text{if } i \in [4..15] \end{cases} \qquad (3)$$

The first byte of the tenth round key K_0^{10} 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.

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 (see Figure 12). Previous work have highlighted that the sensitive areas of CMOS are reverse biased PN junctions [22]. From this information we can propose the following explanation for the observed fault model described in Section 4.

When a bit is read from flash memory, the associated bit-line is precharged. A floating-gate transistor, connected to this bit-line, is activated

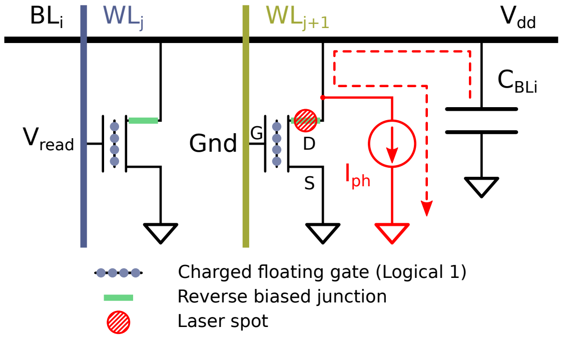


Fig. 12: Schematic of a bit-line found in a NOR flash memory. The red elements indicate the effect of the laser shot.

by the word-line. If charges are present on the floating gate, then the threshold voltage is high. If there are no charges on the floating gate, then the threshold voltage is low. By setting an intermediate voltage on the control gate of the floating-gate transistor, low-threshold transistors pull the bit-line to ground while high-threshold transistors do not.

When a laser spot is shot on the drain of a blocked transistor, a photocurrent is induced between the drain and the source. On a floating-gate transistor in flash memory, is has the effect of pulling the bit-line to ground. If we assume that a logical 1 stored in memory is encoded as a low voltage, then this explains why we induce only bit-sets. Therefore, even though the word-line activates a floating-gate transistor that stores charges, then the bit-line would still be pulled to ground because of the photocurrent. There is not physical mechanism to explain how a laser shot could *prevent* photocurrent from being drawn. This explains the unipolar character of the observed model, consisting only of bit-sets.

The physical mechanism detailed above can be applied to any floating-gate transistor connected to the bit-line. If we assume that bit-lines are horizontal, while word-lines are vertical, then this explains why we can sweep over all the bits of the instruction while moving the laser spot along the y-axis, affecting the bit-lines one after the other. However, moving along the x-axis only changes the affected floating-gate transistor. Since these affected transistors are all connected to the same bit-line, then the same bit is set. This explains why moving along the y-axis allows to target specific bits, while moving along the x-axis does not change anything.

Finally, faulting two adjacent bits with sufficient power can be explained by the fact that the spatial distribution of the photocurrent induced by the laser spot is Gaussian. By shooting with high power, the induced photocurrent can affect multiple floating-gate transistors. If these transistors happen to be connected to different bit-lines, then these two bit-lines are both pulled to ground. This explains why only adjacent bits can be faulted simultaneously, and why it only happens with sufficient power.

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 can only set either a single bit or two adjacent bits. Indeed, it is not possible with this setup to set two bits that are not adjacent to one another, let alone more than two bits. A multi-spot laser setup could be beneficial in this regard, allowing to set multiple non-adjacent bits. It would greatly extend the range of reachable modified instructions.

Bit-set only The observed fault model only consists of bit-sets. Therefore, it is impossible to perform a bit-reset so far. Even though this limits the range of reachable modified instructions, previous sections showed that this fault model can still have serious consequences.

Control flow corruption mostly As demonstrated by the two examples given in Section 5, faulting the control flow of a program is feasible. However, given our fault model, faulting the data is impossible most of the time. Indeed, data is not hard-coded in the instructions but instead stored in RAM and fetched when needed. Therefore, it is impossible for example to perform safe-error attacks on AES encryption [3] since the AES key bytes are not encoded in the instructions.

Altering the control flow is already an effective way to lower the security of algorithms though. Moreover, some arithmetic operations could be modified to actually alter the data. However, this is very algorithm-specific and must be investigated specifically for each of them.

6.3 Reproducibility with a new target code on 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 run on 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 pulse power of 0.5 W and duration of 200 ns can be used to find the x and y coordinates at which each individual bit is set. Having access to the assembly code of the target application is then required, to identify the target instruction. After that, the ARM Architecture Reference Manual must be used to find out a valid instruction into which the target instruction can be transformed. Finally, the delay injection parameter must be tuned. The authors estimate that this overall procedure requires approximately one or two weeks to carry out.

7 Conclusion

This article presented a new laser fault injection attack in flash memory of a 32-bit microcontroller. This attack has the potential to selectively set individual bits of the instruction while it is fetched. It implies that an attacker has the capability to alter the instructions executed by the microcontroller. We detailed the fault model before giving examples of applying this attack to greatly affect the security of common algorithms. Finally, a physical explanation for the observed fault model was given. Future works on the topic could focus on examining the efficient of state-of-the-art software countermeasures against fault attacks., based on redundancy, desynchronisation or control-flow integrity, on this new attack.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Feifei Cai et al. “Optical fault injection attacks for flash memory of smartcards”. In: *International Conference on Electronics Information and Emergency Communication*. IEEE. 2016, pp. 46–50.

[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 attacks on integrated circuits: From CMOS to FD-SOI”. In: *International Conference on Design & Technology of Integrated Systems in Nanoscale Era*. 2014, pp. 1–6.

[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] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis”. In: *Annual International Cryptology Conference*. Vol. 1666. 1999, pp. 388–397.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] Sergei Skorobogatov. “Flash Memory ’Bumping’ Attacks”. In: *Cryptographic Hardware and Embedded Systems.* Vol. 6225. 2010, pp. 158–172.

[27] 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.

[28] 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.

[29] 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.

[30] 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.