Optical Fault Attacks on AES: A Threat in Violet

Jörn-Marc Schmidt, Michael Hutter, and Thomas Plos
Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria
Email:{joern-marc.schmidt, michael.hutter, thomas.plos}@iaik.tugraz.at

***Abstract*—Microprocessors are the heart of the devices we rely on every day. However, their non-volatile memory, which often contains sensitive information, can be manipulated by ultraviolet (UV) irradiation. This paper gives practical results demonstrating that the non-volatile memory can be erased with UV light by investigating the effects of UV-C light with a wavelength of $254\,nm$ on four different depackaged microcontrollers.**

We demonstrate that an adversary can use this effect to attack an AES software implementation by manipulating the 256-bit S-box table. We show that if only a single byte of the table is changed, $2\,500$ pairs of correct and faulty encrypted inputs are sufficient to recover the key with a probability of $90\,\%$, in case the key schedule is not modified by the attack. Furthermore, we emphasize this by presenting a practical attack on an AES implementation running on an 8-bit microcontroller. Our attack involves only a standard decapsulation procedure and the use of a low-cost UV lamp.

Keywords: Fault Analysis, Ultraviolet Light, UV-C, Implementation Attacks, AES

I. Introduction

Nowadays, processors are not only part of computers—they pervade everyday life. Small microprocessors, for example, are the heart of most smart cards, which are used for electronic cash or user authentication applications. They are an integral part of common mobile devices like cell phones and personal organizers.

The use of these small devices comes hand in hand with an increasing amount of private information stored on them. In order to protect the sensitive data against unauthorized access, cryptographic algorithms are typically applied. Although those algorithms are mathematically secure, new attack scenarios arise if an adversary has direct access to such a device. As a consequence, powerful attacks that exploit physical effects (e.g. the timing behavior [1] or the power consumption [2] of the device) become feasible.

Moreover, the correct functionality of a device cannot be guaranteed in an environment that is controlled by an adversary. By maliciously influencing the working conditions, an adversary can provoke faults during the computation of a device. A method that relies on such malfunctions and that tries to benefit from the erroneous result is called a fault attack.

Faults can be divided into different groups according to the duration of the fault in the device. While transient faults influence the device only once, permanent faults stay until a reset of the device. The third group are destructive faults. They are persistent after being injected. Such faults can be caused by a modification of the non-volatile memory. The work of Ross Anderson and Markus Kuhn [3] states that security fuse-bits can be erased by focused ultraviolet light. After erasing these fuse bits, the memory can be read out directly using the internal logic. This technique is also mentioned in [4]. Another proposal is due to Sergei Skorobogatov. He suggests to use a UV-light resistant marker to protect the memory cells from being modified. The ink that covers the fuse-bit cells can be removed with a toothpick [5]. However, the target of all these methods is the security fuse of the memory. In order to prevent such attacks, modern devices invert the functionality of these bits. Erasing them equals locking the internal memory. UV-light based attacks can then no longer be used to gain access to the memory of the target.

However, since the first publication of fault attacks in 1997, several attacks against secret key algorithms, including attacks against AES, have been presented. Eli Biham and Adi Shamir discussed an attack that successively erases the key bit-by-bit to reveal it [6]. This technique can be applied to arbitrary block ciphers. Other attacks assume the occurrence of transient faults, *i.e.* the computation is influenced only once. An attack described by Pierre Dusart et al. uses a precise byte error to discover the round key in 4 byte blocks [7]. Johannes Blömer and Jean-Pierre Seifert presented some attacks on different implementations [8]. They showed that if an attacker is able to set bits of the processed block precisely, it is possible to recover the entire key. Jean-Jacques Quisquater and Gilles Piret assumed a random byte error to reveal the secret key [9]. Their model was applied by Nidhal Selmane et al. to a smart card [10] and by Farouk Khelil to an FPGA implementation [11]. Another attack presented by Johannes Blömer and Volker Krummel uses bit flips and internal collisions to recover the key [12]. Chen et al. attacked the key schedule to get a full 128 bit key using precise byte errors [13]. Christophe Giraud showed how to use precise byte faults in the processed data as well as in the key schedule [14]. Combining these different faults, it is possible to determine a full 128 bit key. Using optical fault injection, Giraud was able to attack an AES implementation. More than 1 000 encryptions were needed to recover the key.

In contrast to previous works that use UV-light, we do not target the fuse bits but the memory itself. We present practical results of attacks on non-volatile memory cells of different devices using UV light. Furthermore, we show that

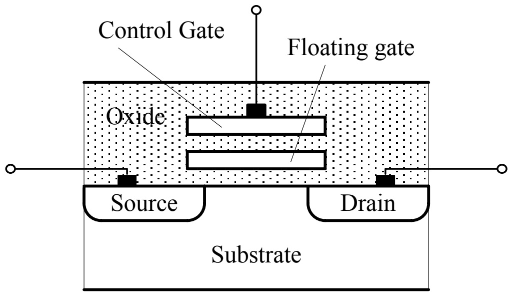

Fig. 1. Schematic of a memory cell that uses hot-carrier injection for programming.

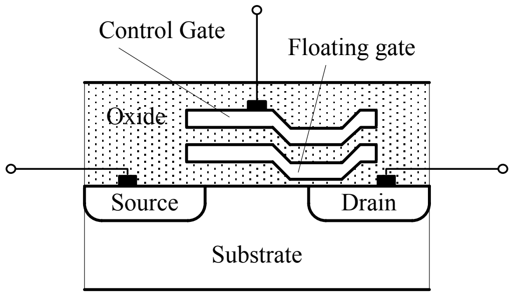

Fig. 2. Schematic of a memory cell that uses Fowler-Nordheim tunneling for programming.

these attacks are not only a hypothetical threat but emphasize its relevance by practical experiments. This is shown by performing an attack on an AES implementation that runs on an 8-bit microcontroller. The attack works independently of the fuse-bit settings and can be performed even if the memory is locked by such fuse bits.

The remaining paper is organized as follows. After discussing the theory of non-volatile memories in Section II, Section III describes practical experiments with ultraviolet light that irradiates different hardware devices. In Section IV, we present a practical attack on an AES implementation. Conclusions are drawn in Section V.

II. Theory of Non-Volatile Memories

In electronic devices, memory is needed to store information. There exist various types of memory cells for different kinds of applications. While cells of volatile memory loose the stored information when power is left, non-volatile memory cells maintain their content. Non-volatile memory, also known as read-only memory (ROM), can furthermore be classified into: non-programmable memory, one-time programmable memory, and several-times programmable memory [15].

The content of non-programmable memory is permanently integrated into hardware via so-called masks during chip manufacturing. Such mask-programmable ROMs are the cheapest form of non-volatile memory. The main drawback of this memory type is that the user cannot modify the content after the chip fabrication. One-time programmable memory overcomes this problem. There, the content is programmed after the chip manufacturing. Therefore, Erasable Programmable ROM (EPROM) devices are typically used. They are only programmable once. EPROM devices can also act as several-times programmable memory, if they are integrated into a special ceramic package which has a dedicated quartz window instead of a cheaper non-transparent plastic package. By exposing the chip die sufficiently long to UV irradiation, its memory is erased. Memory devices that can be erased without the use of UV light are Electrically Erasable and Programmable ROMs (EEPROMs) and FLASH devices [16]. Nowadays, EEPROM and FLASH devices are the most prevalent memories that can be programmed several times. In the following, the programming and erasing of EPROM, EEPROM, and FLASH devices are described in more detail.

Basically, there are two mechanisms used to electrically program non-volatile memories: hot-carrier injection and Fowler-Nordheim tunneling. Hot-carrier injection is used in EPROM devices and Fowler-Nordheim tunneling is used in EEPROM devices. Depending on their implementation, FLASH devices use either hot-carrier injection and Fowler-Nordheim tunneling or only Fowler-Nordheim tunneling.

As it is illustrated in Figure 1, the memory cells of devices that are programmed via hot-carrier injection consist of a complementary metal oxide semiconductor (CMOS) transistor with control gate, drain, source, and an additional floating gate. This gate is electrically isolated and located inside the oxide between control gate and substrate. The charge of the floating gate influences the threshold voltage (the minimum voltage that needs to be applied to the control gate of the CMOS transistor to turn it into conducting state). As long as the memory cell is unprogrammed, the floating gate does not contain any charge and the threshold voltage is minimal. By applying an external programming voltage across drain and source of the transistor, the energy of the electrons inside the substrate is increased. Electrons with sufficient energy are injected into the oxide. Applying the programming voltage also at the transistor's control gate causes the electrons inside the oxide to be deflected towards the floating gate where they get trapped [17]. Thus, the floating gate is charged and the threshold voltage raises—the memory cell is programmed. Before reprogramming, the memory cell must be erased. In EPROM devices, this is achieved by exposing the memory cell to UV light. The energy of the electrons inside the floating gate is increased by the UV irradiation. In that way, they can overcome the non-conducting oxide and run off via substrate and control gate. When using hot-carrier injection for programming FLASH devices, the memory cells are designed in a way such that their floating gate can be discharged electrically by using Fowler-Nordheim tunneling.

Memory cells that rely on Fowler-Nordheim tunneling have a similar structure as the memory cells programmed via hot-carrier injection. A major difference is the very thin oxide layer between floating gate and drain, which is depicted in Figure 2. Fowler-Nordheim tunneling describes the effect that electrons of low energy can overcome a thin oxide layer if they are subjected to a strong electric field. In an unprogrammed state, the floating gate is charged with electrons which makes the

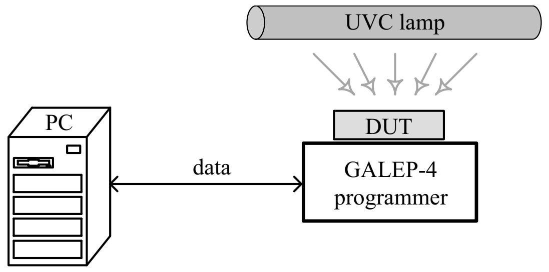

Fig. 3. Measurement setup for the UV-irradiation experiments.

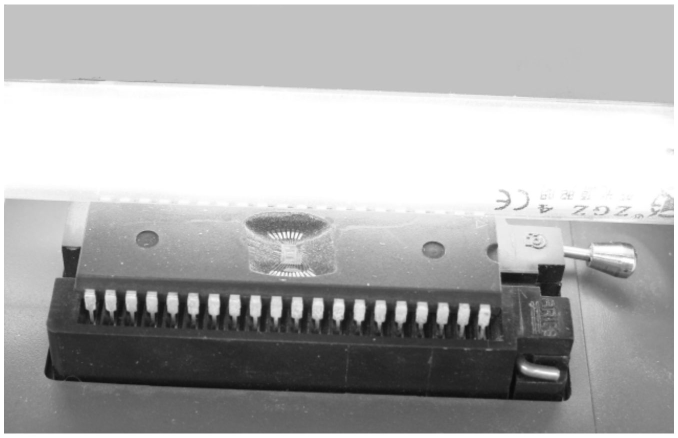

Fig. 4. UV-C fluorescent lamp irradiating the chip die of a microcontroller.

transistor non-conductive. In order to turn the transistor into conducting state, a high positive programming voltage has to be applied across drain and control gate. Electrons tunnel from the floating gate through the thin oxide to the drain discharging the floating gate—the memory cell is programmed. Erasing the memory cell can also be realized by using Fowler-Nordheim tunneling. Applying a high positive programming voltage across control gate and drain, causes the electrons to tunnel towards the floating gate where they get trapped [18].

As described above, EEPROM devices are electrically programmable and erasable via Fowler-Nordheim tunneling. In order to access the memory cells inside the EEPROM device individually, each memory cell is equipped with a so-called select transistor. In contrast, the memory cells in FLASH devices are erased at once or on a block-wise basis. Hence, select transistors are not necessary. This makes FLASH devices more attractive in terms of hardware costs.

III. Experiments

Five different hardware devices were investigated for their susceptibility to ultraviolet irradiation. We analyzed an EPROM memory chip from Cypress Semiconductor, two RISC microcontrollers from Microchip, one 8051 microcontroller, and one ATmega48 from Atmel. All devices were first decapsulated to gain direct access to their chip die, which contains the non-volatile memory. Afterwards, they were irradiated with ultraviolet light.

Consecutively, the measurement setup of our experiments is described in detail, followed by the practical results of our investigations.

A. Measurement Setup

The measurement setup is mainly composed of three components: an universal device programmer, a UV-light generator, and a measurement PC. This setup is depicted in Figure 3. In order to read data from the device under test (DUT) and write data to the DUT, we used a GALEP-4 device programmer from Conitec. The programmer supports numerous devices, which feature different non-volatile memory technologies such as EPROM, EEPROM, and FLASH.

A fluorescent lamp generated the ultraviolet light. The lamp emits a light with a wavelength of 254 nm, which is part of UV-C. Such a lamp is typically used to erase EPROM memory, to sterilize the hospital equipment in medical facilities, and to sterilize the drinking water in water-treatment applications. This lamp was placed on top of the chip surface of the target device with a distance of about one centimeter. Figure 4 shows the UV lamp placed over of a decapsulated microcontroller.

We used a PC to control the overall measurement process. The PC was connected to the GALEP-4 programmer over a parallel connection. Reading and writing of the data between PC and GALEP-4 programmer was accomplished by a Matlab script: First, the entire memory has been programmed to a fixed value of $0x00$. Afterwards, the UV lamp was turned on and the script was started. The script continuously reads the current memory content of the target device. In that way, a detailed picture of the effects of the irradiation over the test period was achieved.

B. Practical Results

Our first experiment analyzed a parallel EPROM memory device (CY27H010). It is equipped with 128 kB of memory and is shipped with a windowed package. This quartz window enables the erasure of the device by UV irradiation. Hence, it is reprogrammable. Figure 5 shows the results of these experiments. On the x-axis the UV irradiation time is drawn, the y-axis indicates the number of erased bits in the memory. At the beginning of the irradiation phase, all bits have been programmed, *i.e.* set to $0x00$. After about 4 minutes, the first bits were flipped. The whole memory was erased and contained $0xFF$ after about 8 minutes.

In the second experiment, the effects of UV light on the two 8-bit microcontrollers from Microchip (PIC16F54 and PIC16F84) were investigated. The PIC16F54 features 512 bytes of FLASH memory and no EEPROM. The PIC16F84 comprised 1 kB of FLASH memory and 64 bytes of EEPROM. Both devices were irradiated by the UV light. After about

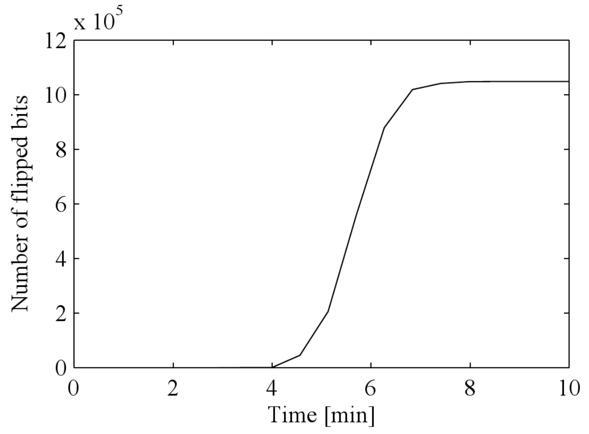

Fig. 5. Number of flipped bits of an EPROM device during 10 minutes UV exposure.

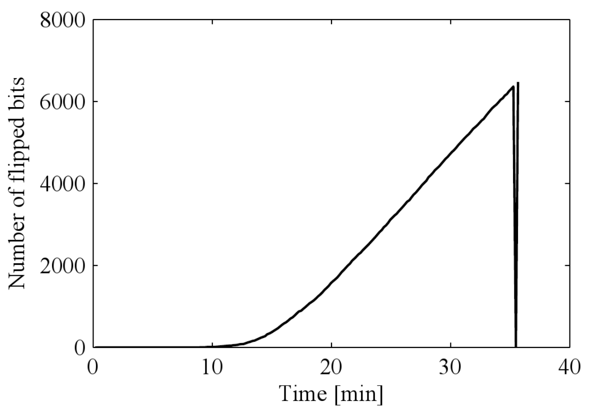

Fig. 6. Number of flipped bits of the AT89C2051 microcontroller during 40 minutes UV exposure.

10 minutes, the first bit was flipped in the memory of the PIC16F54. The entire memory was erased after 26 minutes. This was different for the PIC16F84: The first bit of the FLASH changed after 21 minutes. After about 4 hours nearly one fourth of the bits in the FLASH and EEPROM were flipped from zero to one.

Next, we evaluated the AT89C2051, an 8-bit 8051 microcontroller. It comprises 2 kB of FLASH memory and no EEPROM memory. The result of the experiment is depicted in Figure 6. The number of erased bits increased nearly linearly with the UV-exposure time. The first bits flipped after 10 minutes. However, after 35 minutes, the chip returned only zero values. This behavior results from the security fuse bits of the AT89C2051. Fuse bits are commonly used in many microcontrollers to set device-specific configurations. They can protect, for example, program code that is stored in the memory. After setting the security fuse, the program code neither can be read out nor modified via internal circuits of the device. As such fuse bits are typically realized in FLASH or EEPROM technology, they are also affected by the UV-exposure experiments. In order to prevent an adversary from deactivating a security fuse via UV light, they are logically inverted. The fuse bits get activated and prevent the reading of memory if they are erased by UV light. In our experiment, the device returned only zero values instead of the actual memory content as soon as the fuse bits were erased. The same behavior was observed and successfully evaded in the next experiment, in which an ATmega48 microcontroller was analyzed.

First, the same experiment as for the prior described devices was performed with the ATmega48. After 6 hours of UV exposure, the code-protection fuse bits of the device were modified by the UV light and were set from 0 to 1. Thus, reading of the memory content was not possible anymore. Second, the same device was set into initial state by deleting its memory content and all fuse bits. Afterwards, we took an UV resistant marker to coat the memory cells of the target device. The entire chip except a small part of the memory cells was covered by the ink. Hence, the uncovered cells were still penetrated by the UV light. After 6 hours of UV irradiation, we could successfully read out the memory content and determined that a few bits in the FLASH memory were flipped from 0 to 1.

This experiment demonstrates that it is possible to modify only selected parts of the memory and to bypass the setting of code-protection bits by UV-resistent markers.

Table I summarizes the results of the experiments. It shows the impact of the UV irradiation on the various memories of the analyzed devices. In addition to the possible erasure of EPROM, EEPROM, and FLASH-memory cells, all devices were susceptible to modifications of the fuse bits. A successful modification of the memory is denoted by *Yes* otherwise it is denoted by *No*. A sash is given if the memory is not supported by the target device. The experiments showed that it is possible to change fuse bits such as identification bytes, clock selection, watchdog configurations, and save bits that protect against unintended writing into the EEPROM memory.

The positions of flipped bits are largely uniformly distributed over the memory. However, there are devices where the EEPROM is more affected than the FLASH memory and vice versa. The time between the first bit flips and the erasure of the whole memory depends on the target device and differs with the underlying memory technology. In addition, our experiments showed that the UV exposure does not cause a fast bit-flip transaction. It is a rather slow process that takes at least some minutes and can endure up to several hours for a complete memory erasure. This slow-changing behavior offers an adversary the possibility to choose the number of bits that should be affected, by stopping the UV exposure at a specific point in time. Furthermore, the memory cells can be coated with a UV-resistant marker. This allows an adversary to target only a part of the memory, while keeping the remaining memory cells protected against the UV light by the ink. Consequently, an adversary can choose the number of bits and closely select the attacked memory region. This can be exploited to attack an AES implementation, as we show in the next section.

Type	Name	(E)EPROM	FLASH	Fuse bits
EPROM	CY27H010	Yes	-	-
MCU	PIC16F54	-	Yes	Yes
MCU	PIC16F84	Yes	Yes	Yes
8051	AT89C2051	-	Yes	Yes
MCU	ATmega48	No	Yes	Yes

TABLE I
RESULTS OF THE PRACTICAL UV-IRRADIATION EXPERIMENTS ON FIVE DIFFERENT DEVICES.

IV. DESCRIPTION OF THE ATTACK ON AN AES IMPLEMENTATION

As already described in Section III, the changing of non-volatile memory results in a destructive fault. Once it is injected, the fault persists inside the device. In contrast to transient and permanent faults, it is not necessary to inject a fault during the computation or at a specific point in time. Therefore, faults can be injected independently from the computation. The attacking process is as follows: First, we irradiated the target device with UV light and induced optical faults inside the program memory. The target device was not connected to any power supply or clock-signal source. Second, the device is connected to a standard prototyping board. This board is used to setup a serial communication between the target device and the PC. The connection allows the sending of data to the target device that encrypts the data using the Advanced Encryption Standard (AES) and sends the ciphertext back to the PC. The plaintext and received ciphertext pairs can then be used to extract the secret key that is stored inside the memory of the target device. We now briefly describe the AES algorithm [19] before we focus on a detailed description of the performed attack.

The AES defines a symmetric block cipher. It uses a block size of 128 bits and includes key sizes of 128, 192 or 256 bits. We used an AES implementation with a key size of 128 bits, denoted by AES-128. One AES-128 encryption takes ten rounds. The State of AES is written as a matrix of four times four bytes, denoted by $S_{i,j}$ with $i, j \in \{0, \ldots, 3\}$. The input of an operation is denoted by $S_{i,j}$, its corresponding output by $\bar{S}_{i,j}$. The first nine rounds consist of four operations that are processed in the following order:

- `SubBytes`: Applies a non-linear byte substitution to every byte of the block. This is denoted by $\bar{S}_{i,j} =$ S-box$(S_{i,j})$.
- `ShiftRows`: Shifts the rows 1, 2, and 3 of the State cyclically 1, 2, and 3 bytes to the left.
- `MixColumns`: Applies a linear operation on each column of the State. This can be written as a matrix multiplication for each column $i \in \{0, \ldots, 3\}$ of the State S:

$$\begin{pmatrix} \bar{S}_{0,i} \\ \bar{S}_{1,i} \\ \bar{S}_{2,i} \\ \bar{S}_{3,i} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \cdot \begin{pmatrix} S_{0,i} \\ S_{1,i} \\ S_{2,i} \\ S_{3,i} \end{pmatrix}$$

- `AddRoundKey`: XORs the block with a 128-bit round key that is derived from the initial key by a key expansion algorithm.

Three operations are performed in the last round of AES: `SubBytes`, `ShiftRows`, and `AddRoundKey`. `MixColumns` is skipped. At the beginning of an AES encryption, in contrast, an initial `AddRoundKey` operation is applied.

For our attack we assume that the AES S-box is implemented via a lookup table. This table is the target of the destructive fault. We consider two different cases: In the first case, the whole S-box table is erased. In the second case, only one or a few bytes of the table are affected by the UV light. Although the second case requires more AES encryptions and needs more effort to reveal the secret key, it can also be applied in the presence of countermeasures, as we show at the end of this section.

In the first case, *i.e.* if the whole lookup-table of the S-box is erased during the attack, the key can be derived in a straightforward way. The last round key is obtained by inverting the output and by applying an inverse `ShiftRows` operation. This is because every table lookup in the last round results in $0xFF$. Hence, it is possible to check whether the whole table is corrupted or not by verifying that the erroneous encryption of different plaintexts results in the same output. The next step is to gain the initial key from the last round key. Again, two different scenarios are possible. In case that the round keys are pre-computed and stored inside the device, a normal inversion of the key schedule is sufficient to determine the initial key. Otherwise, if the round keys are computed on the fly, the key schedule uses the corrupted S-box table. This has to be taken into account for reversing the key-schedule computation to get the initial key. Nevertheless, the key can still be computed efficiently[1].

The situation is different if only a part, e.g. a few bytes, of the table is affected by the UV light. In that case, not every ciphertext is suitable for the attack. Thus, more faulty AES encryptions are required as it is not possible to know in advance which encrypted plaintext delivers a ciphertext that can be used for the attack. Furthermore, we assume that the fault does not affect the generation of the round key. If the keys are pre-computed and stored inside the device, the faults have no influence on them. In case the keys are generated on

[1]Note that a bijective S-box is not required for an inversion of the key schedule, since the input of the S-box, *i.e.* the last word of the previous round key, can be calculated by XORing the last two words of the actual round key.

the fly, the possibility that they are affected is less than 85%. This probability is computed in Lemma 2 in Appendix A.

If only one wrong S-box byte is accessed and this happens in the last round, exactly one byte of the ciphertext differs from the correct encryption. These cases, which occur with a probability of 3.3 % for one fault in the S-box (see Lemma 1 in Appendix A), are detected by comparing correct and faulty encryptions. If only one device is available, it is necessary to compute ciphertexts in advance and to store them together with the corresponding input. After the fault injection, the stored results are compared to the erroneous output.

Thus, if correct and erroneous ciphertexts differ in only one byte, this byte is put into a *reference ciphertext* variable $(C_0, \dots, C_{15})$ at the position the byte has in the erroneous ciphertext. This is done in 6-8 of Algorithm 1.

In case that more than one byte of the lookup table is affected, a new reference-ciphertext variable is created for each fault. The faults that differ in the affected bits of the byte, can be distinguished during the attack by checking the flipped bits due to the faulty S-box access. Otherwise, a ciphertext with more than one possibility for one byte is stored.

However, for each of the 16 output-byte positions an erroneous byte is needed to create a complete reference ciphertext, *i.e.* a set of bytes $(C_0, \dots, C_{15})$ with $C_i = E \oplus K_i \, i \in \{0, \dots, 15\}$, while E denotes a constant value and K_i the last round key of position i. As shown in Lemma 1 of Appendix A, 2 500 ciphertext pairs are sufficient to obtain a complete reference ciphertext with more than 90 % probability, if only one byte of the lookup table is disturbed.

Since the reference ciphertext contains one fixed value of the S-box XORed with the corresponding key byte, there are 256 possibilities for the round key: $\{(C_0 \oplus c, \dots, C_{15} \oplus c) | c \in \{0, \dots, 255\}\}$. The right key can be found by trying all 256 possibilities for the S-box value. In case that more than one faulty byte has the same bits flipped, *i.e.* are not distinguishable by the flipped bits, for one of the faults, enough bytes have to be collected and the right combination has to be found by exhaustive search. This leads for m indistinguishable faults to $256 \cdot m^{16}$ tries, once enough bytes are collected. The whole attack is outlined in Algorithm 1.

While the generic attack strategy can be applied for any manipulation of the S-box, we further know that the faulty bits flipped from zero to one. Hence, not every key byte has to be tried during the exhaustive search, only the bits that are not influenced by the fault.

Algorithm 1 Attack for one faulty S-box byte

```
1: Initialize $C_0 \dots C_{15}$ with $-1$
2: while any $C_i == -1$, $i \in \{0, \dots, 15\}$ do
3:     set $plaintext$ to a random value
4:     encrypt $plaintext$ to $ciphertext_{correct}$ with an error-free device
5:     encrypt $plaintext$ to $ciphertext_{erroneous}$ with the faulty device
6:     if $ciphertext_{correct}$ and $ciphertext_{erroneous}$ differ in exactly one byte at some position $j \in \{0, \dots, 15\}$ then
7:         set $C_j$ to byte $j$ of $ciphertext_{correct}$
8:     end if
9: end while
10: set $plaintext$ to a random value
11: encrypt $plaintext$ to $ciphertext_{reference}$ with a error-free device
12: for $k = 0$ to 255 do
13:     encrypt $plaintext$ to $ciphertext_{compare}$ with the key $[C_0 \oplus k, \dots, C_{15} \oplus k]$
14:     if $ciphertext_{compare} == ciphertext_{reference}$ then
15:         return $[C_0 \oplus k, \dots, C_{15} \oplus k]$
16:     end if
17: end for
18: return key not found.
```

A. Performing the Attack in Practice

In order to show that the attack described above is possible, we put it into practice. Our target device was an AT89C2051 device from ATMEL. An AES implementation that uses a fixed S-box table-lookup was programmed on it. The round keys were calculated on the fly using the S-box table. The device was decapsulated and a UV resistant marker was used to put a coat over the memory cells that contained the program code. A picture of the FLASH memory is shown in Figure 7 while the cells protected with the UV-resistant ink can be seen in Figure 8. Some of the cells containing the S-box were left free. The prepared chip was exposed to UV light for about 10 minutes. After the irradiation procedure, the device was put into a standard prototyping board that provided a serial connection to the PC. A Matlab script with a reference implementation of the AES was used to control the communication with the target device and it was used to perform the attack. It turned out that one bit of the S-box was constantly flipped; another bit flipped during the attack, *i.e.* the UV exposure left it in an unstable state. As both bits were on different positions of the byte, the different faults could be distinguished by the bits flipped in the result. After 2 500 AES encryptions, a complete reference ciphertext was generated which led to the right round key. Inverting the key schedule delivered the correct initial key[2].

B. The Influence of Software Countermeasures

Some countermeasures might suppress the output if a faulty value is looked up in the table [20]. If the countermeasure leaks the information in which round the fault occurred, the attack is still applicable. Furthermore, this information even improves the previous attack. The required information can be gained by a timing attack in case the calculation is stopped whenever a fault is recognized. Otherwise, power-analysis techniques can be used to get the information.

[2]Note that other fault analysis methods can be applied as well, once a ciphertext that corresponds to the required fault model is found.

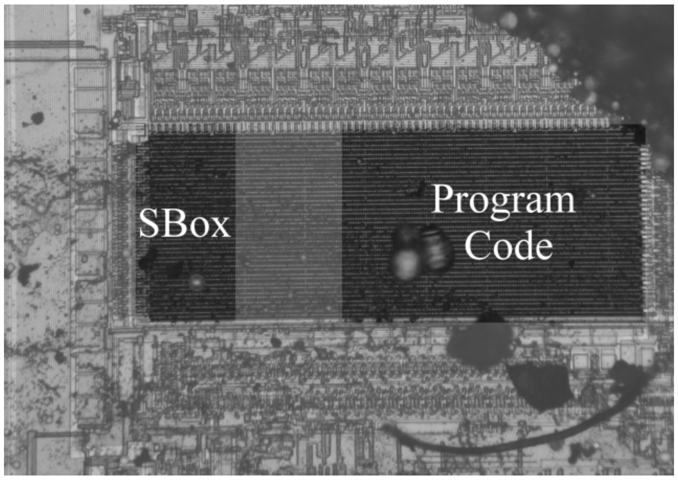

Fig. 7. FLASH memory cell layout of an AT89C2051 programmed with an AES implementation.

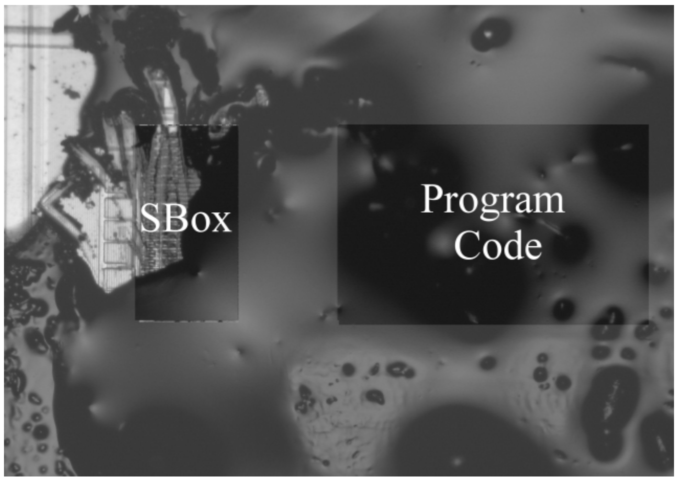

Fig. 8. FLASH memory covered with ink to protect the program code from the UV irradiation.

The attack strategy is quite similar to the method applied to implementations without countermeasures. However, here the first round the first round is targeted instead of the last one. First, an input with no error in the first round has to be found. Afterwards, all possibilities for one input byte are encrypted, while the rest of the plaintext is left constant. At least one of these inputs leads to an error in the first round, as every S-box byte is looked up at least once. With this method, it is directly possible to determine how many bytes have been affected by the fault. This procedure is repeated for every position in the plaintext, which leads to 16 bytes multiplied by the number of faults in the table that cause an error in the first round. As those bytes are XORed with the appropriate key byte of the first round, this results in $256 \cdot n^{16}$ possible keys, where n is the number of faults in the S-box: There are n faulty values for each position. As they are indistinguishable concerning the fault they were caused by, we get n^{16} possible reference ciphertexts. Assuming the worst case, for each of them, 256 bytes have to be tried until the value of the fault is found. Hence, if only one byte is faulty, $16 \cdot 256$ encryptions and 256 tries of possible keys suffice to get the right key.

C. Hardware Countermeasures and Limitations of the Attack

The attack method using UV light is destructive. Hence, modifications are not reversible[3]. If an adversary changes parts of the program code or factory settings by accident, it is likely that the device stops working. Thus, an adversary has to find the right attack position either by knowing details about the implementation and device layout or by trial and error, which can be a challenging and time-consuming task depending on the implementation and the memory size of the device.

However, most modern security smart-cards comprise countermeasures to prevent a successful manipulation. Light sensors or coating layers can inhibit a decapsulation of the package. Self-checking methods like a firmware checksum or an encryption of test-vectors and a check of the result against precomputed values can detect a malicious modification. Another way to prevent an attack is to store the information in ROM instead of FLASH or EEPROM cells.

[3]Unless the device can be reprogrammed, which is unlikely in a real attack scenario.

V. Conclusions

This paper presents practical results on how ultraviolet light can be used to erase non-volatile memory. In contrast to previous publications, our work focuses not on the fuse bits but on the memory itself. We demonstrate that an adversary has precise control concerning the number of bits that are modified, as well as the region of the memory that is affected.

Based on these observations, we present a new attack on AES and performed a practical attack on an 8-bit microcontroller. The attack can be mounted at low cost. A standard decapsulation procedure and a UV lamp are sufficient. This attack is still applicable if some countermeasures are present. We conclude that our results are not only of academical interest but also allow the modification of non-volatile memories in prevalent devices through UV irradiation.

Acknowledgements

The work described in this paper has been supported through Austrian Government funded project *ARTEUS* established under the Trust in IT Systems program FIT-IT. The information in this document reflects only the authors' views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

References

[1] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *16th Annual International Cryptology Conference – CRYPTO, Santa Barbara, CA, USA, August 18-22*, ser. LNCS, N. Koblitz, Ed., no. 1109. Springer, Heidelberg, 1996, pp. 104–113.

[2] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *19th Annual International Cryptology Conference – CRYPTO, Santa Barbara, CA, USA, August 15-19*, ser. LNCS, M. Wiener, Ed., vol. 1666. Springer, Heidelberg, 1999, pp. 388–397.

[3] R. J. Anderson and M. G. Kuhn, “Tamper Resistance - a Cautionary Note,” in *Proceedings of the 2nd USENIX Workshop on Electronic Commerce, Oakland, California, November 18-21, 1996*. USENIX Association, November 1996, pp. 1–11.

[4] H. Bar-El, “Known Attacks Against Smartcards,” Discretix Technologies Ltd., White Paper, 2004.

[5] S. P. Skorobogatov, “Semi-invasive attacks - A new approach to hardware security analysis,” Ph.D. dissertation, University of Cambridge - Computer Laboratory, 2005.

[6] E. Biham and A. Shamir, “Differential Fault Analysis of Secret Key Cryptosystems,” in *Advances in Cryptology – CRYPTO, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21*, ser. LNCS, B. S. K. Jr., Ed., vol. 1294. Springer, Heidelberg, 1997, pp. 513–525.

[7] P. Dusart, G. Letourneux, and O. Vivolo, “Differential Fault Analysis on A.E.S.” in *First International Conference on Applied Cryptography and Network Security – ACNS. Kunming, China, October 16-19*, ser. LNCS, J. Zhou, M. Yung, and Y. Han, Eds., vol. 2846. Springer, Heidelberg, October 2003, pp. 293–306.

[8] J. Blömer and J.-P. Seifert, “Fault Based Cryptanalysis of the Advanced Encryption Standard (AES),” in *Financial Cryptography – FC, 7th International Conference, Guadeloupe, French West Indies, January 27-30*, ser. LNCS, R. N. Wright, Ed., vol. 2742. Springer, Heidelberg, January 2003, pp. 162–181.

[9] J.-J. Quisquater and G. Piret, “A Differential Fault Attack Technique Against SPN Structures, with Application to the AES and KHAZAD,” in *Fifth International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, C. P. C. Walter, C. K. Koc, Ed. Springer-Verlag, 2003, pp. 77–88.

[10] N. Selmane, S. Guilley, and J. L. Danger, “Practical Setup Time Violation Attacks on AES,” in *Dependable Computing Conference, 2008. EDCC 2008. Seventh European*. IEEE ComputerSociety, May 2008, pp. 91–96.

[11] F. Khelil, M. Hamdi, S. Guilley, J. L. Danger, and N. Selmane, “Fault analysis attack on an fpga aes implementation,” in *New Technologies, Mobility and Security – NTMS, Tangier, Marrocco, November 5-7*. IEEE ComputerSociety, November 2008, pp. 1–5.

[12] J. Blömer and V. Krummel, “Fault Based Collision Attacks on AES,” in *Fault Diagnosis and Tolerance in Cryptography – FDTC, Third International Workshop, Yokohama, Japan, October 10*, ser. LNCS, L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, Eds., vol. 4236. Springer, Heidelberg, October 2006, pp. 106–120.

[13] C.-N. Chen and S.-M. Yen, “Differential Fault Analysis on AES Key Schedule and Some Coutnermeasures,” in *Information Security and Privacy – ACISP, 8th Australasian Conference, Wollongong, Australia, July 9-11*, ser. LNCS, R. Safavi-Naini and J. Seberry, Eds., vol. 2727. Springer, Heidelberg, 2003, pp. 118–129.

[14] C. Giraud, “Dfa on aes,” in *Advanced Encryption Standard – AES, 4th International Conference, Bonn, Germany, May 10-12*, ser. LNCS, H. Dobbertin, V. Rijmen, and A. Sowa, Eds., vol. 3373. Springer, Heidelberg, May 2004, pp. 27–41.

[15] J. Griffin, B. Matas, and C. de Suberbasaux, *Memory 1996: Complete Coverage of DRAM, SRAM, EPROM and Flash Memory ICs*. Integrated Circuit Engineering Corporation, 1996. [Online]. Available: http://smithsonianchips.si.edu/ice/cd/MEM96/title.pdf

[16] G. Campardo, R. Micheloni, and D. Novosel, *VLSI-Design of Non-Volatile Memories*. Springer, Heidelberg, 2005.

[17] J. J. Makwana and D. K. Schroder, “A Nonvolatile Memory Overview,” http://aplawrence.com/Makwana/nonvolmem.html, 2004.

[18] A. Bhattacharyya, “Effect of Trapping in Thin Oxides on the Write and Erase Characteristics of Floating Gate EEPROM Devices,” *Journal of Physics D: Applied Physics*, vol. 17, no. 4, pp. 799–803, April 1984. [Online]. Available: http://www.iop.org/EJ/article/0022-3727/17/4/019/jdv17i4p799.pdf

[19] National Institute of Standards and Technology (NIST), “FIPS-197: Advanced Encryption Standard,” November 2001, http://www.itl.nist.gov/fipspubs/.

[20] G. Gaubatz and B. Sunar, “Robust Finite Field Arithmetic for Fault-Tolerant Public-Key Cryptography,” in *Fault Diagnosis and Tolerance in Cryptography – FDTC, Third International Workshop, Yokohama, Japan, October 10*, ser. LNCS, L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, Eds., vol. 4236. Springer, Heidelberg, October 2006, pp. 196–210.

Appendix

Definition 1: The *Stirling Numbers of the second kind* are defined by

$$S(n,k) = 1/k! \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n, \tag{1}$$

with $\binom{k}{i}$ the binomial coefficient.

Lemma 1: Assume that all S-box lookups within an AES computation are independent and n bytes of the S-box table are disturbed. The probability that I different, uniform distributed inputs result in a faulty byte lookup for each position in the State is

$$\Pr[\text{success}] \geq \frac{16!\mathrm{S}(\lfloor pI \rfloor, 16)}{(16^{\lfloor pI \rfloor})}, \tag{2}$$

with $p \geq 0.033$ for $n = 1$ and $p \geq 0.017$ for $n = 2$ the probability that an input plaintext is mapped to a ciphertext that is affected by one specific fault out of n faults. $\mathrm{S}(n,k)$ denotes the Stirling Numbers of the second kind.

Proof. First, the probability p is considered: An AES-128 consists of 10 rounds. In each round, a `SubBytes` operation is performed, which results in 16 lookups in the S-box table, hence 160 lookups. Therefore, the probability that only one special lookup operation is incorrect is $\frac{1}{256}(1 - \frac{n}{256})^{159}(1 - \frac{n-1}{256})^{160}$. Considering one specific fault, 16 of those combinations are useful, namely all in which the error occurs in the last round. This leads to $p = 16\frac{1}{256}((1 - \frac{1}{256})^{159}) \geq 0.033$ for $n = 1$ and $p \geq 0.017$ for $n = 2$. The probability that a faulty S-box byte is addressed more than once in the last round is negligible.

Each ciphertext C_i corresponds to a Bernoulli distributed random variable Y_i, which states if C_i is suitable for the attack. All ciphertexts $C_0 \ldots C_{I-1}$ together lead to a binomial distributed random variable $X = \sum_{i=0}^{I-1} Y_i$, which gives the number of useful ciphertexts. For large n, X can be estimated by its expectation $\mathbf{E}[X] = Ip$ because its variance $\mathbf{V}[X] = \frac{(1-p)\cdot p}{I}$ aspires to zero.

Next, the number of useful inputs, which are necessary to cover each of the 16 output bytes at least once, is determined. Having m useful inputs, there are 16^m possible distributions over the 16 output bytes, $16!S(m, 16)$ cover all of them. Therefore we get a probability of $\frac{16!\cdot \mathrm{S}(m,16)}{(16^m)}$ to cover all bytes. Using $m = \lfloor Ip \rfloor$ proofs the claim.

Lemma 2: The probability that the key expansion is not affected by a single random byte failure in the S-box is ≥ 0.85. For n random faults this gives a probability of $(\frac{256-n}{256})^{40}$.

Proof. AES-128 computes ten round keys. In each of these computations, the S-box is addressed 4 times. Thus, the probability that it is not affected by a random byte failure is $(\frac{256-n}{256})^{40}$, which leads to $(\frac{255}{256})^{40} \geq 0.85$ for $n = 1$.