Defeating a Secure Element with Multiple Laser Fault Injections

Olivier Hériveaux

 Ledger Donjon

Abstract. In 2020, we evaluated the Microchip ATECC508A Secure Memory circuit. We identified a vulnerability allowing an attacker to read a secret data slot using single Laser Fault Injection. Subsequently, the product life cycle of this chip turned to be deprecated, and the circuit has been superseded by the ATECC608A, supposedly more secure. We present a new attack allowing retrieval of the same data slot secret for this new chip, using a double Laser Fault Injection to bypass two security tests during a single command execution. A particular hardware wallet is vulnerable to this attack, as it allows stealing the secret seed protected by the Secure Element. This work was conducted in a black box approach. We explain the attack path identification process, using help from power trace analysis and up to 4 faults in a single command, during an intermediate testing campaign. We construct a firmware implementation hypothesis based on our results to explain how the security and one double-check counter-measure are bypassed.

1 Introduction

The Microchip (formerly Atmel) ATECC608A is a secure memory offering a fixed set of commands to manage secret keys, encrypt and store secret data, run cryptographic operations to perform authentication, process secure boot verification, etc. We studied a previous chip version, the ATECC508A, and reported last year a vulnerability to the manufacturer, which allows extracting a secret data slot using Laser Fault Injection [2]. Microchip is working actively to improve this circuit, and as the ATECC508A has been deprecated, we were motivated to have a look on its successor and observe it's security improvements. We don't have any access to the chip design files or source code, and all our work is done in black box approach.

Laser Fault Injection is a well known and mature technique used during the security evaluation of circuits. In brief, this powerful tool allows an attacker to precisely inject errors during the computation of a chip, to bypass security features. This technique has been presented in many previous publications [4–7], and this paper will assume the reader has basic

understanding of fault injection and power trace analysis. The reader may refer to our previous publication [2] as this new study is a continuation of our previous work.

The first part of this paper briefly presents the evaluated circuit and the targeted assets, for a particular hardware wallet application. Indeed, the presented attack only exploits one security feature of this multipurpose chip: the general purpose secret data slots. P256 key storages are not vulnerable to this attack, though we don't exclude derivative work might compromise them. We also recap the experimental setup used for fault injection.

The second part of this paper details our fault characterization work on the EEPROM memory of the chip, conducted in black box, which allowed us to identify a precise fault model, required to construct the subsequent sophisticated attack path.

The last part walks through the many progressive attack steps we led until the final successful attack. We tried different attack campaigns with one or more faults, and each result allowed us to progressively refine our firmware implementation hypothesis. In particular, we present an intermediate attack using 4 faults injection. This quadruple fault attack was unsuccessful, but its results helped us to understand the device firmware implementation and then find out that only two well chosen faults are enough to bypass the security checks. Fault injection in black box approach is known to be difficult, and therefore performing multiple faults was challenging!

2 Application

As described in our previous paper [2], the *Coldcard Mk2* hardware wallet was using the ATECC508A secure memory to store the secret seed and protect its access with a PIN code. This wallet was therefore vulnerable to our attack on the ATECC508A.

The latest revision, the *Coldcard Mk3*, uses the new ATECC608A circuit for enhanced security. The presented work shows the wallet secrets stored in the ATECC608A can still be retrieved using Laser Fault Injection, but with higher effort. One particular data slot is targeted: the PIN hash data slot, which unlocks access to the seed data slot. Moreover, this new wallet revision now encrypts the seed using a secret key stored in the main microcontroller, a STM32L496, thus attacking the ATECC608A secure memory is not enough to access the funds. STM32 devices are known to

be weak from previous publications [3], and we managed to successfully attack this microcontroller as well, with a high success probability.

3 Chip identification

Laser Fault Injection requires access to the circuit's silicon substrate. We performed backside package decapsulation and infra-red imaging to have a brief look at the chip internal structure. Surprisingly, we did not find any visible difference between this chip and its predecessor. Yet this new circuit provides more functionalities, so we considered two possible options:

- **Option 1:** The chip is exactly identical, and a programming fuse in EEPROM selects between ATECC508A or ATECC608A to enable the new features. The ATECC608A is backward compatible, and furthermore, some settings in EEPROM memory are marked as *Reserved* in the old version and used in the latest one. As we discovered through our experiments, the software implementation of the *Read Memory* command is hardened in the ATECC608A, so this hypothesis is unlikely.
- **Option 2:** Only the ROM memory, storing the firmware of the chip, has different programming. This allows the manufacturer to reuse most of the wafer masks for this new version production, which is cost-effective. Deprocessing and optical or SEM imaging can confirm or disprove this hypothesis, but we are not equipped for this.

As the circuit hardware seems to be exactly the same, we decided to try to perform the same attack which worked for the ATECC508A. In this attack, we faulted the instructions to bypass a test on the `IsSecret` data slot flag, by illuminating the ROM memory with a short laser pulse.

After several attempts, we did not manage to pass this attack. We obtained circuit errors and crashes, but not a single secret data slot could be extracted with this method. Our following work in this paper explains why the same attack did not succeed.

4 Experimental setup

Our experimental setup for the described study is similar to what we presented in our previous paper for the ATECC508A. The device under test is plugged in our *Scaffold* [1] testing motherboard. This board

sends I^2C commands to the secure memory and generates synchronization signals for fault injection.

Faults are injected using an infra-red pulsed laser source and a microscope for focusing. We used a 20X objective, hence the laser beam is about 3 μm. Nonetheless our experiments revealed that spatial resolution is not important for applying the vulnerability we found. In particular, we believe this might be reproduced with much cheaper equipment ($<$ 10k$).

Studying this circuit was done in black box approach. To understand the behavior of the chip, we used a resistor to measure the electrical current consumed by the device. The signal was amplified with an analog amplifier embedded in *Scaffold*, and we used an oscilloscope to record the traces. In the following presented power traces, the reader may observe the raw waveforms are very noisy and difficult to read. For this paper, we replayed most of the attack steps, recorded lots of traces and averaged them to clearly show the differences between faulty executions and nominal ones. That was easy to do once we found the vulnerabilities, but it does not reflect the real conditions we were working in black box.

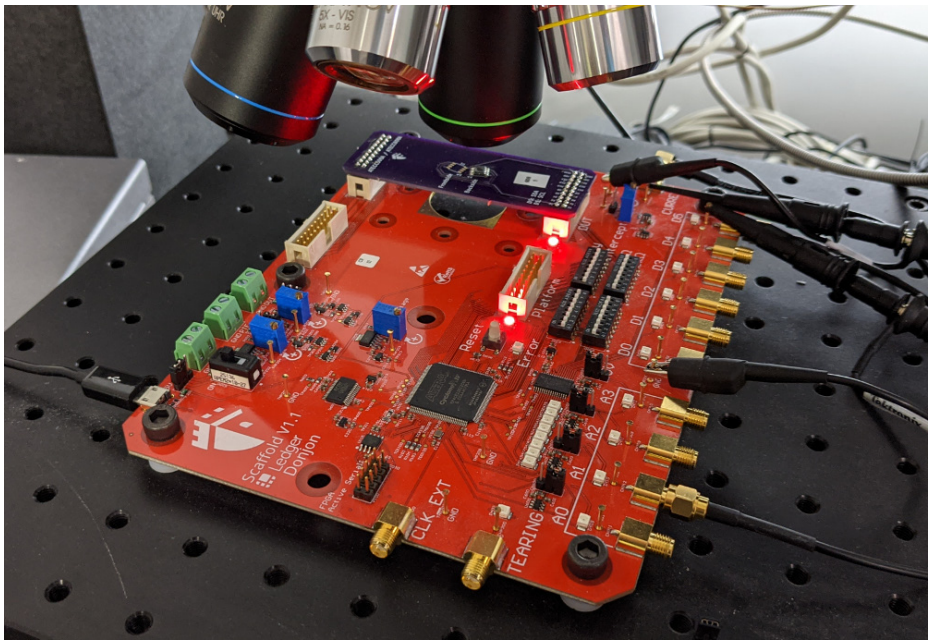

Fig. 1. *Scaffold* motherboard under the microscope

Fig. 2. Backside decapsulated ATECC608A chip, soldered on *Scaffold* daughterboard.

5 From self-test abuse to fault model identification

After 1.3 seconds of inactivity, the ATECC608A watchdog puts the chip into sleep mode automatically, for power saving. Issuing commands requires waking up the chip before, by holding low the I^2C SDA input pin for a defined duration. When executing this wake up sequence, we remarked on the power trace a 800 μs long processing operation executed by the chip. Leaving a sleep state should not be a complex operation for a chip, so we initially supposed the access conditions to the data slots might be parsed and compiled at this time and cached in RAM for further use. We decided to inject faults during the wake up sequence and then execute the *Read Memory* command without fault injection to see if wake up sequence corruption has any effect on following commands.

During this campaign, lots of faults led to an unknown error code `0x07` being returned by the chip when calling *Read Memory* after wake up. After investigation, we found out this error means the chip self-test failed. We found this information in the ATECC608A-TNGTLS documentation, which is a pre-provisioned variant of the ATECC608A and whose complete datasheet is not under NDA yet.

104814 fault injections have been performed and 1567 experiments resulted in self-test failure. Figure 3 plots the fault injection time for self-test error events. Each dot matches an experiment. Abscissa represents injection times, and ordinate corresponds to experiments number. We can observe several vertical bands, evenly spaced by 5 μs intervals, showing

that self-test errors happen at particular injection times. As faults are injected in the EEPROM memory, we concluded the circuit performs 84 EEPROM memory accesses during wake up.

Also, some bands are missing, or have only a single fault event. We supposed the fetched corresponding bytes had a special value, and we matched those "holes" to null bytes of two EEPROM CONFIG data segments, as highlighted in Table 1, and illustrated in Figure 3 (see EEPROM CONFIG segments data overlaid at the bottom).

EEPROM config address (decimal)	Data (hexadecimal)
0:	01231e310000600208ae6592ee014500
16:	c000550000008f2d8f808f438f440043
32:	00448f478f48c343c444c747c8488f4d
48:	8f430000ffffffff00000000ffffffff
64:	00000000ffffffffffffffffffffffff
80:	ffffffff00000000ffff000000000000
96:	3c005c00bc01fc01fc019c019c01fc01
112:	fc01dc03dc04dc07dc08fc01dc013c00

Table 1. The two segments of EEPROM CONFIG data read during wake up.

— Bytes 0 to 51 include device serial number and revision, various device option bits, and data slots configuration.
— Bytes 96 to 127 correspond to the keys configuration.

We understood a checksum is calculated over those configuration bytes to detect settings corruption. Indeed, the chip must have hardware support for CRC-16 with polynomial `0x8005` since it is used for commands transmission error detection. This CRC-16 engine is probably reused for the self-test.

Zooming in the last band (Figure 3 bottom zoom) reveals two bytes are fetched at very close time. We believe this is the 16-bit CRC value the configuration data checksum is compared with.

From this experiment, we can infer the fault model with confidence: we are easily able to stick bits to zero during EEPROM readout. For a fixed injection time targeting a non null byte readout, we are able to fault with 97% probability.

In Figure 4, each byte of the checked configuration data is plotted depending on its hamming weight and the number of times it was faulted. We don't have any data byte with 7 or 8 hamming weight. This figure shows the chances to fault a byte is little dependent on its hamming

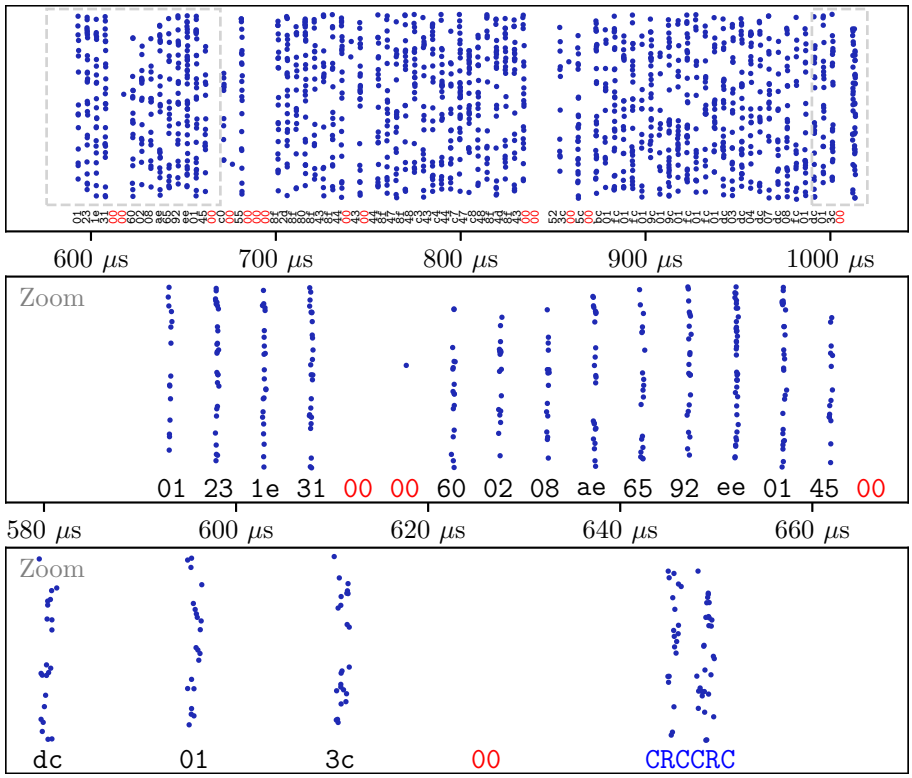

Fig. 3. Instant of fault injection in EEPROM memory, during wake up.

weight, and therefore most of the faults sets all bits of the fetched byte to zero.

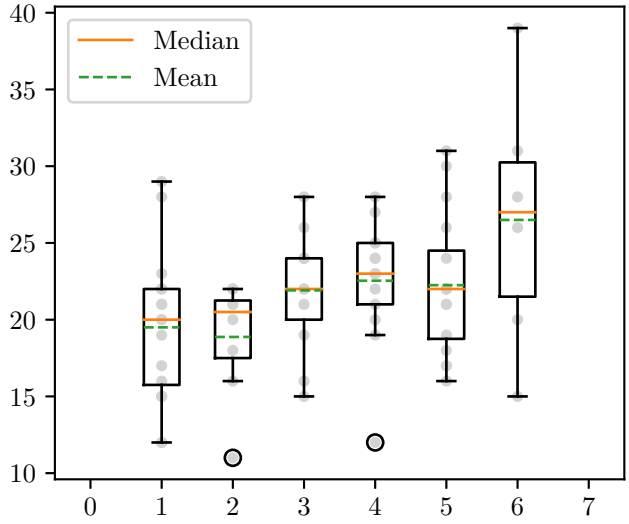

Fig. 4. Relation between byte fault count and hamming weight

We also observed few occurrences of faulted null bytes. Such faults occurred near the bottom EEPROM decoder, at the edge of our scanning area (See Figure 5). We ran another characterization campaign exclusively on the bottom decoder, and we were able to reproduce those faults. After tuning the parameters, we were able to fault null bytes with 75% probability.

6 Fault model exploitation

As already highlighted in our previous work about the ATECC508A, only one bit in the configuration of a data slot defines it as secret (See Table 2). Since we have a reliable way of overwriting a configuration byte to zero, we decided to try faulting the *Read Memory* command by smashing the configuration byte fetch in EEPROM memory to disable the `IsSecret` flag. This is a different approach as we did previously on the ATECC508A where we faulted instructions of the ROM memory, in a much less reliable way.

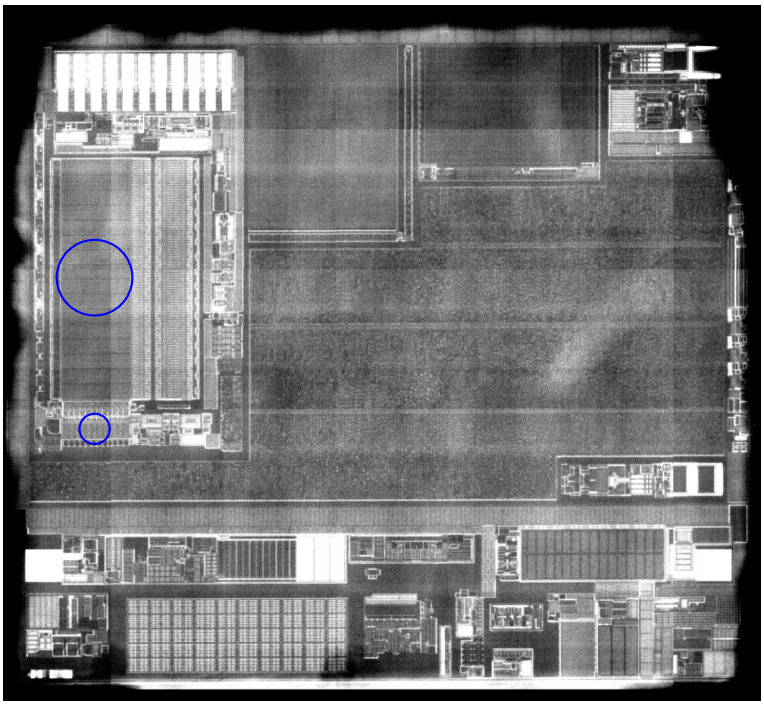

Fig. 5. Top circle area in EEPROM: sets bits to zero. Bottom circle area in EEPROM: sets bits to one.

Name	Value	Comments
Raw	`0x8f43`	Slot configuration value
Write config	encrypt	Writes are always encrypted
Write key	`0x3`	Write encryption key index
Read key	`0xf`	Read encryption key index
Is secret	**True**	This data slot can never be read
Encrypt read	False	Read are forbidden by "is secret" flag, but allowing plain text can help us if we manage to bypass "is secret" flag.
No MAC	False	MAC and HMAC commands with this data slot are allowed.

Table 2. Targeted data slot configuration (Recap from [2])

6.1 Single fault trial

The electrical current consumed by a circuit depends directly on its activity, and in particular on memory accesses and instructions executed by the processor. To find when the fault must be injected, we performed a differential power analysis in order to detect behavioral difference whether the *Read Memory* command is accepted or denied.

As the signal is very noisy, probably because of a counter-measure from the chip, we averaged 500 of them to make the execution path difference clearer. Figure 6 shows single traces in light colors, and averaged traces in dark color. The single traces have been filtered by a fifth order Butterworth 1 MHz low-pass filter to reduce noise. We can observe the execution path between accepted and denied calls differs at 363 μs.

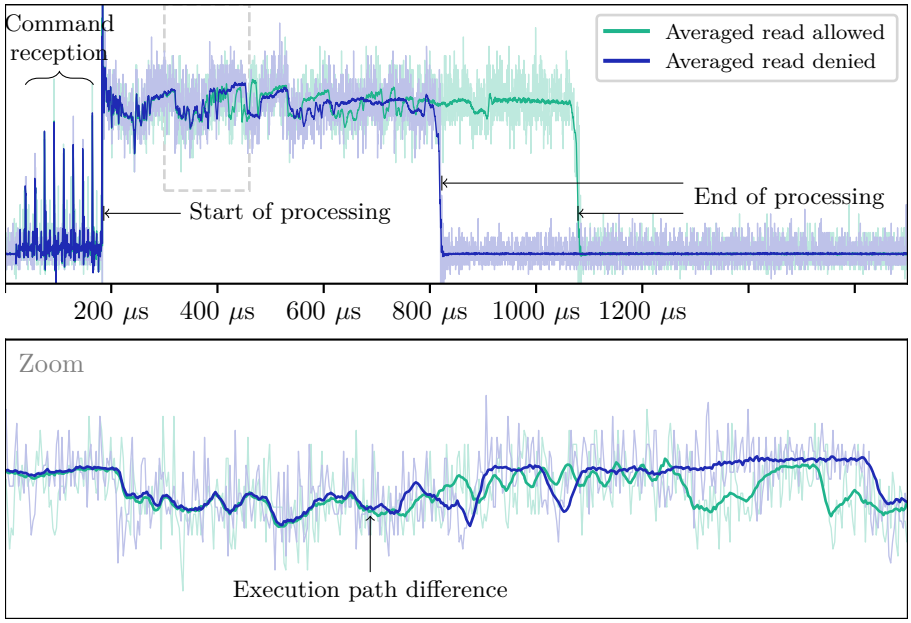

Fig. 6. Power traces comparison for *Read Memory* command. Single traces in light colors, averaged traces in dark colors.

Similar to what's been presented in our previous paper, this experiment gives the instant of execution branch depending on the `IsSecret` flag value. In the attack of the ATECC508A, we faulted the branch instruction during program execution. This time, we wanted to fault the `IsSecret` configuration byte fetch from EEPROM memory, which should happen

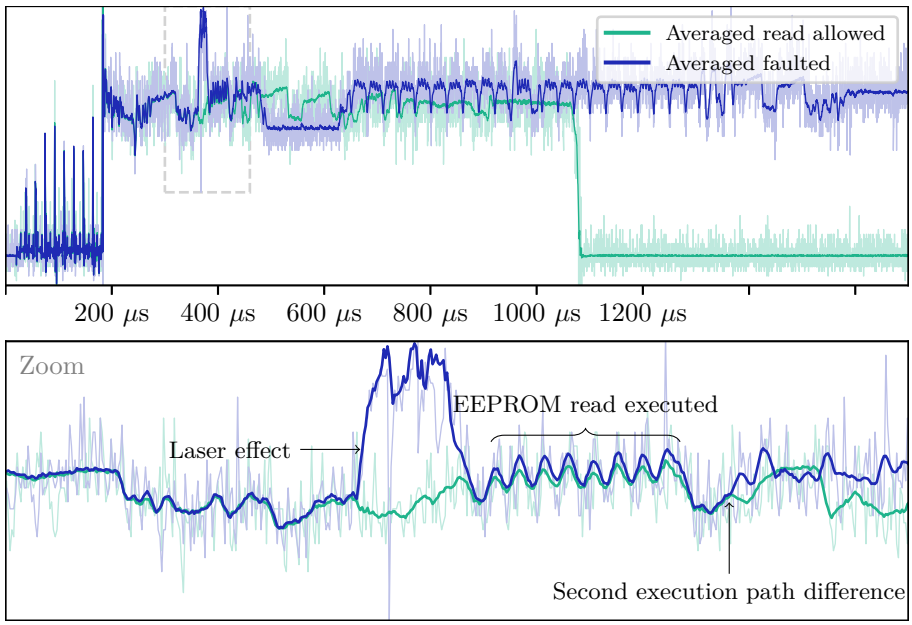

Fig. 7. Averaged first check fault bypassing.

earlier. We could not find when the memory fetch occurs from this analysis, because it is executed in both cases, but we can assume it is close to the divergence, which gave us a small time frame to try. Listing 1 is a simplified pseudo-code hypothesis of what we are trying to exploit with a single fault.

We ran an attack campaign with varying fault injection time, which quickly produced interesting faults leading to a new observable execution path (Figure 7). Unfortunately, the obtained power trace did not match the expected trace in case of success and the circuit also returned the `EXECUTION_ERROR` code.

Looking in detail the new execution trace in Figure 7, we see the 32 bytes EEPROM memory read was performed successfully, but right after the transfer loop, the traces do not match anymore which means the execution paths differ. Our interpretation was we had successfully bypassed the security check, but later the chip executes some unexpected branch. We thought the chip might perform a second security check, maybe by reading a second time the `IsSecret` flag. For this reason, we decided to try injecting a second fault to bypass this second test, still by overwriting to zero the `IsSecret` security flag stored in EEPROM memory.

```
void read_memory_command(int slot){
    uint16_t config;
    // Access condition checking
    // Fault EEPROM access here:
    eeprom_read(get_config_address(slot), &config, 2);
    if (config & IS_SECRET){
        i2c_transmit(EXECUTION_ERROR);
        return;
    } else {
        // Data fetch
        char buf[32];
        eeprom_read(get_data_address(slot), buf, 32);

        // Send response
        i2c_transmit(OK, buf);
    }
}
```

Listing 1. Pseudo-code hypothesis for single fault attack

6.2 Double fault trial

Identifying the correct time for a second fault injection was a bit harder as averaging would have required too much work. We used the differential analysis on single traces, and scanned a temporal window around the observable difference. For this paper, we finally spent time averaging the traces to make it clearer.

During this second campaign, both first and second faults successfully branched the code as planned, and the execution trace matched the expected one much longer. As shown in Figure 8, unfortunately again, a later branch in the execution was different and the chip response was still `EXECUTION_ERROR`.

6.3 Quadruple fault trial

On the right in Figure 8, we can observe a loop pattern similar to the first 32 bytes EEPROM memory read. The loop is not very clear because the jitter is more important at this time (temporal noise accumulates over time and blurs the measurements).

We understood the chip reads the content of the data slot twice, and compares the results to detect read corruptions from fault attacks. Indeed, during our characterization work, we did not manage to fault a 32 bytes memory read on a public data slot. This was possible on the older ATECC508A, but this new revision of the chip has this double-read counter-measure.

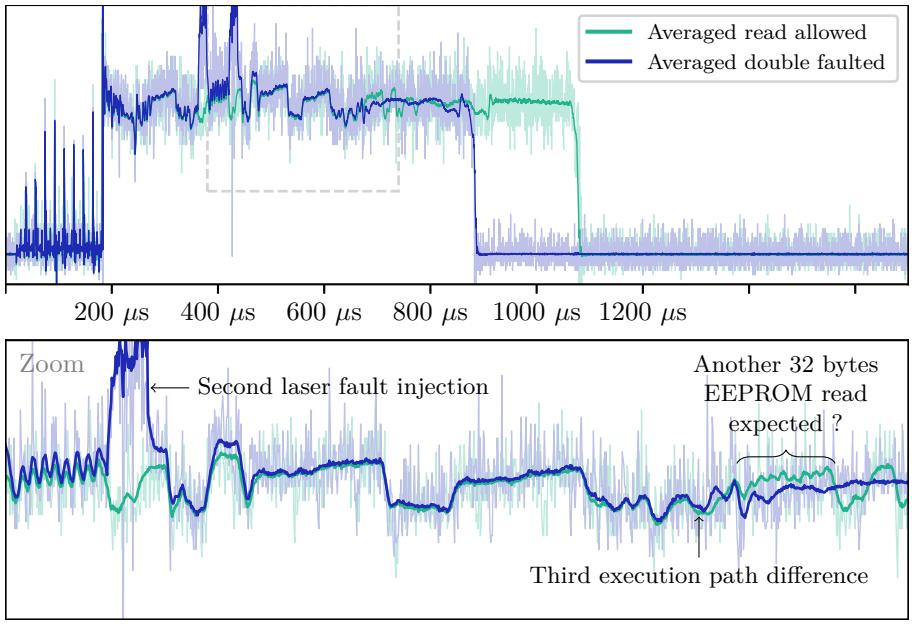

Fig. 8. Averaged first and second checks fault bypassing.

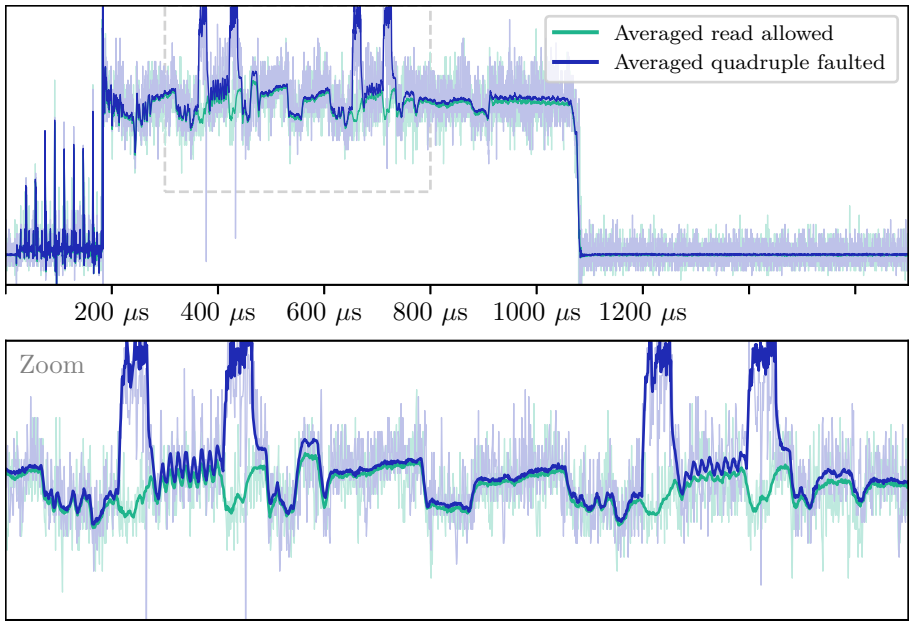

Fig. 9. Averaged trace of quadruple fault injection, which sticks to the expected trace in case of success.

Therefore, bypassing the third check was similar to bypassing the first one, and a fourth fault identical to the second one was also required.

After one day long testing campaign, one experiment with 4 faults resulted in a `OK` response from the chip, and 32 bytes of data were returned! The measured power trace was exactly matching the expected one (see Figure 9), but the returned data was different from what we initially programmed inside the device. Reusing the same fault injection settings, we managed to execute the *Read Memory* command multiple times, with the exact unfortunate issue.

7 Successful double fault attack

The ATECC608A provides the `AES` command allowing to encrypt or decrypt data using a defined key (128 bits keys according to the datasheet). We recorded and averaged power traces when executing this command, which is presented on the top trace of Figure 10. 16 bytes are encrypted, and 10 AES rounds execution is clearly visible. This remarkable pattern is also visible in the power trace of our single fault early campaign (bottom trace of Figure 10 and Figure 7). We see the AES is executed twice during the *Read Memory* execution, which matches the command read length: 32 bytes, 2 blocks.

From this, we understood that the secret data we tried to extract was probably encrypted in the EEPROM memory using the AES algorithm with an internal key. In the power trace of Figure 7, we can observe the circuit probably decrypted the data slot, and then failed at a second access checking. This decryption does not occur for public data slots, and so our first hypothesis of execution paths were incorrect. Indeed, the circuit fetches from EEPROM the `IsSecret` flag one more time to enable decryption or not.

In our quadruple fault attack, we force the circuit to follow the same execution path as for public data slots. The faults 2 and 4 do not bypass security checks, but prevent data decryption, and removing them should allow us to read and decrypt our secret data slot correctly.

After some testing of double fault attack, we managed to extract the content of the data slot, in plaintext. Since the power traces now include AES decryption, the second fault time to bypass the second security check had to be changed, as shown in Figure 11.

All those experiments helped us to infer a probable hypothetical implementation of the firmware *Read Memory* command, which is presented in Listing 2. Please note this is a very simplified model, with lots of shortcuts,

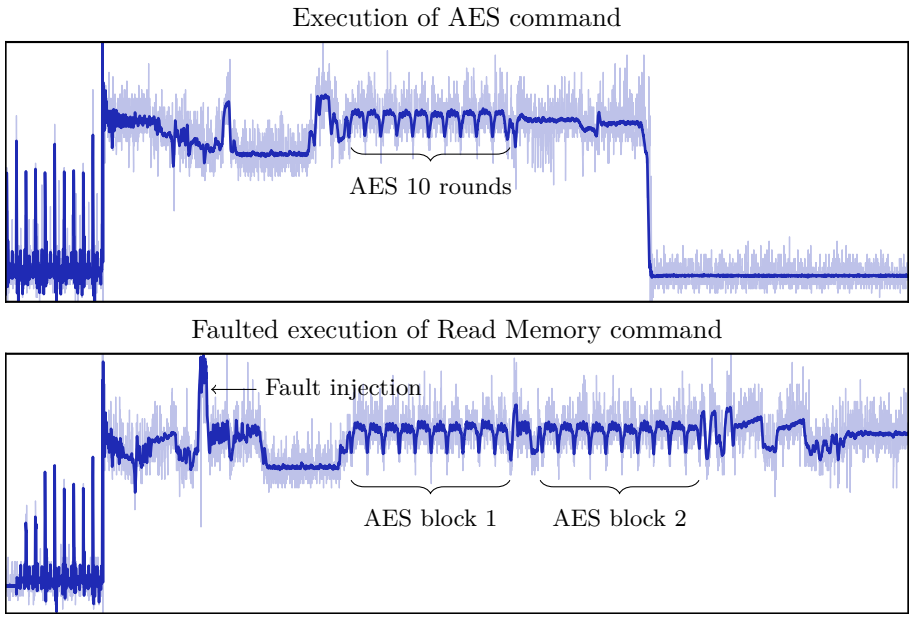

Fig. 10. AES execution in traces

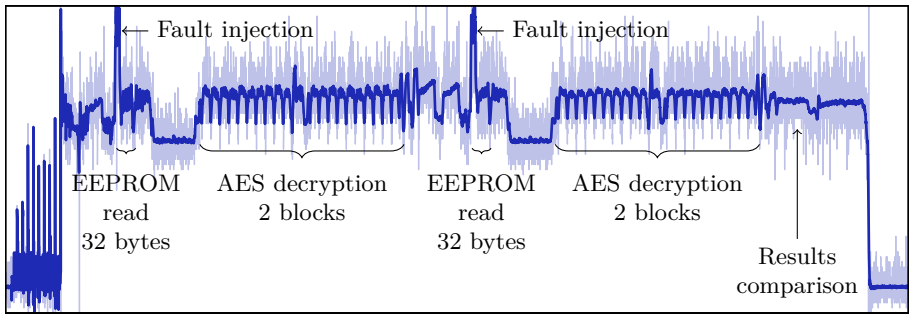

Fig. 11. Power trace for successful double-fault attack

whose goal is only to help understanding the attack path. Indeed many parameters and other functionalities of the command are ignored.

```
/**
 * Handles read memory command (called by the command dispatcher).
 * Result code and data are transmitted by I2C.
 * This is very simplified, since it ignores block and offset
 * parameters, response encryption, etc.
 * @param index Data slot number.
 */
void read_memory_command(int slot){
    // Command arguments checking
    // During attack campaigns, some faults produced PARSE_ERROR
    // responses.
    if (!slot_valid(slot)){
        i2c_transmit(PARSE_ERROR);
        return;
    }

    uint16_t config;
    // First Access condition checking
    // Fault EEPROM access here:
    eeprom_read(get_config_address(slot), &config, 2);
    if (config & IS_SECRET){
        i2c_transmit(EXECUTION_ERROR);
        return;
    }

    // First data fetch
    char buf_a[32];
    internal_get_slot_data(slot, buf_a);

    // Second access condition checking
    // Fault EEPROM access here:
    eeprom_read(get_config_address(slot), &config, 2);
    if (config & IS_SECRET){
        i2c_transmit(EXECUTION_ERROR);
        return;
    }

    // Second data fetch
    char buf_b[32];
    internal_get_slot_data(slot, buf_b);

    // Double read checking
    if (memcmp(buf_a, buf_b)){
        i2c_transmit(EXECUTION_ERROR);
    } else {
        i2c_transmit(OK, buf_a);
    }
}
```

```
/**
 * Get data slot content. Decrypt it if necessary.
 * @param index Data slot number
 * @param dest Destination buffer where the data slot content is
 *        copied.
 */
int internal_get_slot_data(int slot, char* dest){
    uint16_t config;
    // Don't fault here:
    eeprom_read(get_config_address(slot), &config, 2);
    if (config & IS_SECRET){
        char encrypted[32];
        eeprom_read(get_data_address(slot), encrypted, 32);
        aes_decrypt(encrypted, dest, SOME_INTERNAL_KEY);
    } else {
        eeprom_read(get_data_address(slot), dest, 32);
    }
    return OK;
}
```

Listing 2. Pseudo-code hypothesis corresponding to our successful double fault attack

8 Success rate improvement

Faulting 4 times a chip in a single execution is a very difficult task. Given 4 faults $F_{1..4}$ with independent success probabilities $P(F_{1..4})$, the probability to pass successfully the complete attack is:

$$P_{\text{success}} = P(F_1).P(F_2).P(F_3).P(F_4)$$

That is for instance, if every $P_{1..4}$ is 5%, which can be considered a good success rate for a single fault attack, then $P_{\text{success}} = 1/160000$, which is very low, especially when there is a chance to destroy the circuit with a fault. Therefore to make the quadruple fault attack reliable and realistic, we had to optimize the success rate of each fault, and we spent time tuning the fault injection parameters (this was done before discovering only two faults were necessary). In the end, we have reached very high success rates, as detailed in Table 3. Each experimental probability measurement is calculated over 500 experiments.

The most difficult fault to calibrate is the first one, as clock jitter makes fault injection time uncertain and the fault injection is before the execution divergence at an unknown instant (we are faulting the EEPROM configuration memory fetch, not the flag branch).

Finding the offset for the second fault is also a bit challenging for the same reasons. However, as it is very soon after the first fault, the jitter is much smaller.

Calculating the fault injection time for faults 3 and 4 is very easy. Since the power trace pattern for those two faults is the same as the first two faults, we can directly measure the time between the two EEPROM fetches on the averaged power trace for a public data slot. Then the time between fault 3 and 4 is the same as the time between 1 and 2.

Faults success rates	*Comments*
$P(F_1) = 95.8\%$	Data slot read, but `EXECUTION_ERROR` status Experimental measurement
$P(F_2) = 91.1\%$	Calculated result
$P(F_3) = 97.8\%$	Calculated result
$P(F_4) = 93.5\%$	Calculated result
$P(F_1).P(F_2) = 87.3\%$	Data slot read, but `EXECUTION_ERROR` status Experimental measurement
$P(F_1).P(F_2).P(F_3).P(F_4) = 79.8\%$	Data received, but encrypted Experimental measurement
$P(F_1).P(F_3) = \mathbf{93.7}\%$	**Successful attack** **Data received in plaintext** Experimental measurement

Table 3. Fault success rate overview

One more difficulty was to discriminate the successful faults for building up statistics. Indeed, as the chip always sends the same `EXECUTION_ERROR` response, the only way to distinguish between a successful fault and a failed one is by looking at the power trace. We found that measuring the command execution duration from the power trace was a good oracle, and therefore we could automate the selection easily.

9 Conclusion

The checksum performed in the wake up sequence of the device to verify the integrity of the configuration is probably a counter-measure to prevent EEPROM bits erasure attacks under ultraviolet light. Ironically, this was a great help for us to establish our fault model and then use this tool with enough confidence to dare performing multiple faults attack campaigns, in black box.

This work highlighted a double-check counter-measure which has been added to this new circuit revision. It is probable that other functionalities of the chip have been hardened as well. However, we demonstrated the ability to manipulate EEPROM memory fetches easily can allow an attacker to

inject many faults to disrupt a single command. Indeed our fault success rate is very high, and the attack is therefore easy to reproduce and not very risky.

We have to honestly emphasis that our previous experience on the ATECC508A device was undoubtedly helpful to find the vulnerability on its successor. For any evaluated circuit in black box approach, it may be a good idea to try breaking previous silicon revisions to understand how the chip works and what are its weaknesses, and then raise the bar by trying hardened versions.

More counter-measures can increase the security of the circuit and make this attack much harder:

— Clock jittering: by adding noise to the execution speed of the circuit, the success rate of the attack may drastically decrease as it becomes difficult to inject faults at the right time. Although hardware generated jitter is probably best, a software implementation with random delays can nonetheless be efficient.

— Light sensors: some Secure Elements embeds light sensors to detect laser illumination. However, this requires heavy hardware modifications. It can be tedious to implement and may also be patented. This is probably the best counter-measure against laser fault attacks.

— Adding error detection codes to memories highly reduces the chances of injecting an undetected fault. Once again this requires heavy silicon modifications. It also increases the surface of the circuit and consequently rises the price of the circuit.

— Killing the chip permanently on tampering detection (for instance if a double-check fails) makes very hard vulnerability research.

Today a new circuit revision is out, the ATECC608B, and the ATECC608A is not recommended for new designs anymore. This component is now available for purchase...

References

1. Ledger Donjon. Scaffold, 2019. `https://github.com/Ledger-Donjon/scaffold`.
2. Olivier Heriveaux. Black-Box Laser Fault Injection on a Secure Memory. *Symposium sur la sécurité des technologies de l'information et des communications - SSTIC 2020*, 2020. `https://www.sstic.org/2020/presentation/blackbox_laser_fault_injection_on_a_secure_memory/`.
3. Kraken. Inside Kraken Security Labs: Flaw Found in Keepkey Crypto Hardware Wallet, 2020. `https://blog.kraken.com/post/3248/flaw-found-in-keepkey-crypto-hardware-wallet-part-2/`.

4. Johannes Obermaier and Stefan Tatschner. Shedding too much Light on a Microcontroller's Firmware Protection. *11th USENIX Workshop on Offensive Technologies - WOOT 17*, 2017. `https://www.usenix.org/conference/woot17/workshop-program/presentation/obermaier`.
5. Sergei P. Skorobogatov. Optical Fault Masking Attacks. *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 23–29, 2010.
6. Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. pages 2–12. Springer, 2003.
7. Jasper. G. J. van Woudenberg, Marc F. Witteman, and Frederico Menarini. Practical optical fault injection on secure microcontrollers. *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 91–99, 2011.