Firmware extraction from real IoT devices through power analysis of AES

Francesco Benvenuto[1], Francesco Palmarini[1,2], Riccardo Focardi[1,2] and Flaminia L. Luccio[1,2]

[1]DAIS, Ca' Foscari University, via Torino 155, 30172 Venezia, Italy
[2]10Sec S.r.l., via delle Industrie, 13, 30175 Venezia, Italy

Abstract
The recent growth of Internet of Things has made embedded systems an interesting target for potential attackers. Extracting the firmware of an embedded device breaks the intellectual property of the manufacturer and makes it possible to produce functionally equivalent devices at a lower price. It is thus of ultimate importance to understand the methodologies and techniques used by attackers in order to extract the firmware, so that manufacturers become aware of the implication of their design choices for what concerns the protection of their products. In this paper, we discuss some advanced techniques and methodologies that attackers use to break the security of embedded devices. We then apply these techniques and methodologies to extract the firmware from a real device. In particular, we implement a cost-effective Correlation Power Analysis (CPA) setup that allows us to discover the confidential AES key used by the microcontroller to encrypt its code and data.

Keywords
IoT, firmware extraction, embedded system security, side channel, cryptanalysis.

1. Introduction

Embedded systems are known to be prone to different classes of attacks. With the exponential growth of the Internet of Things (IoT) we are more and more facing the so-called *patching vulnerability*, as we are surrounded by vulnerable devices that are hard or even impossible to patch [1].

In this paper, we discuss techniques and methodologies that an attacker can use to break the security of embedded devices, focusing on firmware extraction attacks. We illustrate and discuss in detail the steps that an attacker can follow to extract the firmware from a Microcontroller Unit (MCU). The process is analyzed starting from an initial phase in which information on the device is gathered so to define an *attack plan*, up to the final phase in which the firmware is extracted. Studying these steps in detail, and understanding the capabilities that an attacker can have, should hopefully help the manufacturers to improve the software and hardware protection of IoT devices.

ITASEC'21: Italian Conference on CyberSecurity, April 7–9, 2021
palmarini@unive.it (F. Palmarini); focardi@unive.it (R. Focardi); luccio@unive.it (F. L. Luccio)
https://www.unive.it/data/persone/5590470/ (R. Focardi); https://www.dsi.unive.it/~luccio/ (F. L. Luccio)
0000-0003-0101-0692 (R. Focardi); 0000-0002-5409-5039 (F. L. Luccio)

```
uint8_t check_password(char *correct_passwd, char* passwd){
    uint8_t passbad = 0;
    for(uint8_t i = 0; i < strlen(correct_passwd); i++){
        if (correct_passwd[i] != passwd[i]){
            passbad = 1;
            break;
        }
    }
    return !passbad;
}
```

Figure 1: Time dependent password check implementation.

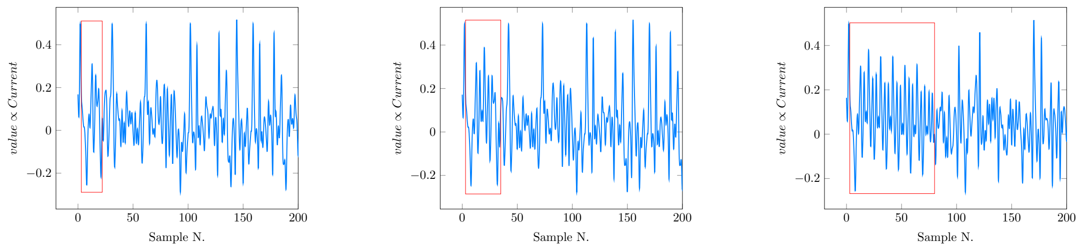


Figure 2: Power traces of an embedded device running Figure 1 code with different passwords.

We have carried out several experiments on real devices in our laboratory, and we present a representative example without providing any detail that could reveal the actual target. The case study consists in extracting the firmware from the two MCUs of an embedded device. Typically, MCUs have a specific mechanism, often called Read Out Protection (RDP), to prevent the read-out of their firmware. We will illustrate a setup based on off-the-shelf, affordable hardware that will allow us to perform advanced power analysis attacks on the AES cipher, making it possible to bypass protections, execute arbitrary code on the MCU, and extract its firmware.

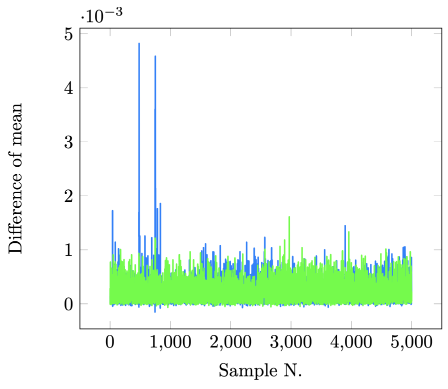
The paper is organized as follows: in Section 2 we discuss the background and the related work. In Section 3 we discuss the techniques and methodologies used by an attacker to extract firmware. In Section 4 we present our case study. We conclude in Section 5.

This article is based on the first author's Master's thesis work [2].

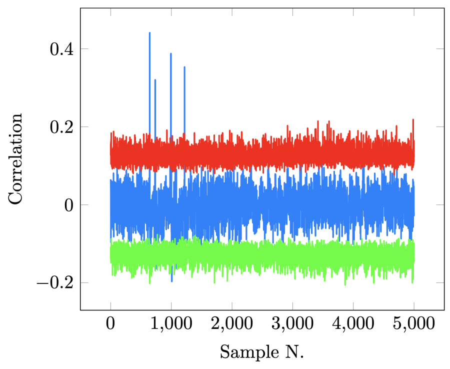
2. Background and related work

This section provides technical background on power analysis, discussing related work.

Simple Power Analysis (SPA). Simple Power Analysis is a classic power analysis technique based on the analysis of the shape of the power consumption, which is directly used to infer information about data and/or the executed code path (see, e.g., [3]). For instance, let us consider the code in Figure 1: it is a time dependent implementation of a password check which exits



(a) Differential Power Analysis (DPA).



(b) Correlation Power Analysis (CPA).

Figure 3: Comparing differential and correlation power analysis of AES.

as soon as one password character is wrong. In Figure 2 we show different power traces of an embedded device running the code listed in Figure 1 with different passwords. The rightmost power trace represents the device elaborating the correct password, the other two represent power traces for passwords that have the first wrong password character at positions 0 and 1, respectively. The red bounding box, in each trace, contains the power consumption of the for loop in the code of Figure 1. It is clearly noticeable that the red bounding box expands depending on the position of the first wrong character. The attacker can now easily brute-force, one by one, the characters of the password by observing the expansion of the box: in fact, when the box expands the attacker discovers that the character is correct and can move to brute-forcing the next one. This makes the attack polynomial with respect to the length of the password.

Differential Power Analysis (DPA). This attack is based on the observation that different processed values imply (slightly) different power consumption, observable through statistical measures [3]. For example, in the first AES round, a substitution is applied to the XOR of a round key byte with a plaintext byte. The attacker collects a big number of power traces for different plaintexts and, for each guess of the the key byte, selects a bit (e.g., the least significant one) of the output of the substitution function. If the key guess is correct, the average power consumption for the traces where the bit is 0 will be smaller than the average power consumption of the traces where the bit is 1, called *difference of means*. In Figure 3a we plot the difference of means for a wrong (green) key guess and a correct (blu) key guess. For the correct key it is possible to observe spike(s) where the value is processed. This method allows for individually guessing the bytes of the AES key.

Correlation Power Analysis (CPA). In [4, 5, 6] a new refined class of attacks, called Correlation Power Analysis, was introduced. CPA is based on defining a power consumption model for the processed data so that the correlation between modeled and actual consumption can be used to verify guesses more accurately. For example, the *Hamming weight*, i.e., the number of 1 bits in a value, can be shown to be proportional to power consumption. The attacker considers a possible guess of a key byte, computes the power consumption according to the Hamming weight model, and estimates its correlation with respect to the actual power consumption trace. This can be done using standard statistical tools such as the Pearson correlation coefficient. The most correlated guesses reveal the actual key bytes. In Figure 3b we show the correlation plot using a Hamming weight model: the correct key byte is in blue, the wrong key byte with the largest negative correlation is in green and the wrong key byte with the largest positive correlation is in red. Note that, the model offers a more accurate distinction between correct and wrong key bytes than DPA. However, we can notice that the points of highest correlation for the correct key (the blue spikes) are about at the same time as the ones of DPA. Both DPA and CPA attacks have been applied to other symmetric ciphers, like DES (see e.g., [4]) and to asymmetric ciphers (see, e.g., [7, 8, 9]).

Equipment. We briefly describe the equipment used in our power analysis setups. A *digital storage oscilloscope* is a device that acquires varying analog signal voltages as a function of time; in this work we used both Picoscope 5444D and Chipwhisperer Lite, an open source toolchain dedicated to hardware security research which is also able to perform both side-channel and fault injection attacks.[1] A *logic analyzer* (Saleae Logic Pro 8) that measures the digital voltage variation of a target signal. A *differential probe* used in combination with an oscilloscope, that allows for measuring voltage variations between any two points of the circuit.

Related work. Firmware extraction from read-protected microcontrollers is a relatively little explored field: in [10] Goodspeed defeated the password protection mechanism found in Texas Instruments' old MSP430 microcontrollers. The author used a time-based side channel attack to take advantage of an unbalanced code in the password check routine and, using voltage glitches, bypassed the security protection. The authors of [11] demonstrated how to downgrade hardware-imposed security restrictions and download the internal firmware of an STM32F0 MCU, but the attack was invasive as it required decapsulating the chip. As for the characterization of the attack parameters, in [12, 13, 14] authors have successfully applied genetic algorithms and other optimization techniques. Glitch generation using FPGAs combined with digital-to-analog converters has been studied in [15, 16] and adopted by commercial tools, such as the Riscure VC Glitcher [17]. After the seminal work by Boneh, DeMillo and Lipton [18], numerous articles (e.g., [3, 10, 16, 19, 20, 21, 22, 23]) have investigated the feasibility of applying voltage glitching to attack both microcontrollers and secure hardware, such as smartcards. Extensive surveys are provided in [24, 25]. Fault attacks against cryptographic implementations have been studied in several papers (e.g., [26, 27]). In recent years, fault injection has also been proven effective for achieving privilege escalation on a locked-down ARM processor [15] and, in the same year, authors of [28] proved that two widely used software countermeasures to fault attacks do not provide strong protection in practice.

[1]https://github.com/newaetech/chipwhisperer

3. Attacker methodology

We now present a typical firmware extraction attack methodology. This is useful to define a proper threat model and to understand the possible implications of an attack, so to consider suitable defense and mitigation techniques. Interestingly, we assume a setup based on off-the-shelf, affordable hardware, so to maximize the risk of attack: our attacker does not require a professional lab with very expensive equipment (cf. Section 2). The attacker targets the MCU firmware in order to: (i) reverse engineer it; (ii) obtain confidential information about the protocols and algorithms implemented in the firmware, breaking the intellectual property (IP) of the manufacturer; (iii) discover possible vulnerabilities, that could be exploited to compromise the device. Typically, the attacker has no prior knowledge of the attacked system: we refer to this situation as *black-box* attack scenario.

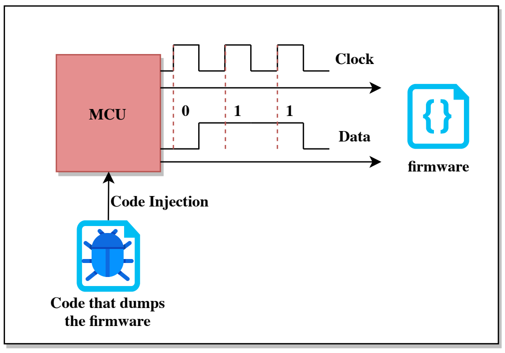
3.1. Information gathering

The first step in a black-box attack is to gather as much information about the system as possible. This allows the attacker to make informed decisions about the objective and opens the possibility of exploiting different attack techniques.

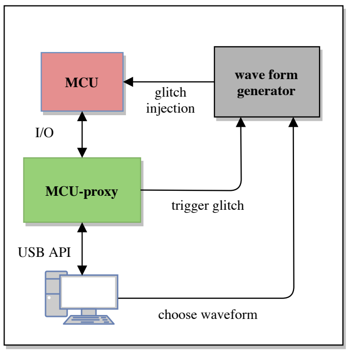
Inspecting system components. When facing an unknown hardware, the first information gathering step is to visually inspect the integrated circuit trying to understand what are the various components and how they behave. To make this task harder, a common manufacturer practice is to erase labels from the crucial components, typically from the MCU. This practice is performed to slow down the reverse engineering of the circuit and make the eventual exploitation of the device more time consuming and expensive. In fact, this could discourage occasional attackers to even start performing experiments on the device. Some companies even place a wrong label, misdirecting the attacker. A typical attack technique is to perform targeted experiments in order to discover the functionality of the MCU pins. This mapping, also called *pin-out*, allows for searching among existing MCUs the ones that match the particular pin layout. Once the MCU is known, the attacker looks for the *datasheet* describing in detail the MCU behaviour and capabilities.

Sometimes, even when the MCU name is known, it is impossible to obtain its datasheet; indeed, some companies do not publicly disclose it, so to slow down the reverse engineering process. However, it is worth noticing that it is very hard to keep an MCU datasheet secret forever. Sooner or later, these documents are leaked or released. Interestingly, erasing the MCU label and keeping the datasheet confidential are classic examples of what is referred to as *security-by-obscurity*, a typical approach in industrial security that has proved to be inefficient and that, in many cases, only reinforces a false sense of security.

Detecting encrypted content. A fairly common embedded system architecture consists of an MCU paired with an external flash memory (e.g., Espressif ESP8266 and ESP32). The flash memory might contain both code and data that can be easily leaked by connecting directly to it. A typical defensive technique is to encrypt the flash memory content, using a key that is stored securely inside the MCU. Clearly, it is important for the attacker, to immediately



(a) Code injection that dumps the firmware.



(b) Voltage glitch setup from [14].

Figure 4: Firmware extraction techniques.

detect this protection in order to decide her next move. One method that is widely used to distinguish between plain data or encrypted one is to measure its entropy: intuitively a high entropy indicates that data are encrypted, as encryption algorithms behave like pseudo-random functions producing a high-entropy output. However, as observed in [29], it is possible to obtain a high entropy also with structured file formats. For example, compressed files exhibit a high entropy. It can also happen that a small number of bytes, among the encrypted ones, are in plaintext, leading in any case to a high entropy and making it hard to notice those particular plaintext bytes.

3.2. Firmware Extraction

After the attacker has acquired enough information about the system, she devises the possible attack vectors to extract the firmware. The following list is far from being exhaustive, but is general enough to cover the most commonly used attack techniques.

Code Injection. One of the most powerful attack vectors is called *code injection*. It requires the attacker capability to inject code into the MCU, which will then be executed by the target. If the attacker knows the instruction set architecture (ISA) of the target MCU, she can craft a special payload that would dump the firmware from the inside, directly through the I/O pins of the MCU. It is enough to use two General-Purpose Input/Output (GPIO) pins, one as clock signal and the other one as data signal. Figure 4a summarizes the explained method. This attack vector works in case the MCU firmware is readable by the injected code. For example, there might exist different memory areas, isolating the injected code from the firmware and, in such a case, this attack would not work.

Fault Injection. *Fault injection* attacks are based on inducing faults in the execution so to modify the control flow and/or corrupt data [21]. There exist different types of fault injection. The cheapest and simplest one is probably voltage fault injection, i.e., injecting disturbances in the power supply rail of the target MCU so to induce faulty behavior. The injected disturbance

```
if ( condition ) {
    // dump the firmware
}
```

Figure 5: Firmware dump functionality protected by a condition.

is also known as a *voltage glitch*. In [14] it is presented a modern, cost-effective voltage fault injection setup with accurate control over the glitch shape, duration and timing. The authors show that the setup allows for highly reproducible and reliable fault injections enabling attacks that require over one million of faults on the same MCU. Figure 4b shows a simplified schema of the setup presented in the paper that uses an MCU-proxy to perform the low-level communication with the target MCU, and a waveform generator to produce the chosen glitch shape and to inject it in the supply voltage line of the target.

The ability to inject an error is a powerful capability for the attacker and can be easily leveraged to dump the MCU firmware when the MCU itself has already implemented a firmware dump functionality. Consider, as a simple example, an MCU whose bootloader contains code to dump the firmware only if a certain condition holds (cf. Figure 5). This piece of code is the ideal target for the considered attack: through fault injection, the attacker can skip/modify the jump instruction that implements the branch and dump the firmware even when the condition is false. Of course, more sophisticated attacks may be required for more complex code and/or less explicit branches (see, e.g., [14]).

As this kind of attacks can become easily destructive, it is a common practice to try them on spare MCUs before reproducing them on the real targets. Notice that, attacks made on the bootloader work with very high probability on any identical MCU. Moreover, controlling the firmware makes attacks simpler to experiment with.

IC decapsulation. The attack techniques explained so far require the device to actively execute instructions that directly or indirectly dump the firmware. *IC decapsulation* requires static, physical analysis of the MCU, i.e., decapsulating and inspecting the integrated circuit *die* with a powerful microscope. This can reveal several constructive components, like the physical bits that compose the firmware. Based on the constructive elements of the memory, it might be possible to distinguish the value of each bit. This technique, however, requires very expensive equipments, such as a scanning electron microscope (SEM), and is out of the scope of our current analysis setup that assumes off-the-shelf, affordable hardware (cf. Section 3).

4. Case study

In our laboratory we have carried out several experiments on extracting firmware from real devices. In this section, we present a representative case study without providing any detail that could reveal the actual target. Instead, we focus on the applied techniques and methodology.

We are given an embedded device with two MCUs, that we will call RED and BLUE, a flash memory and a button (cf. Figure 6). The goal is to extract the firmware of the two MCUs. We

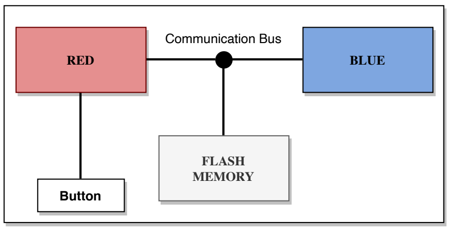


Figure 6: High level architecture of the target device.

consider a black-box scenario in which the attacker has physical access to the device: a typical situation in which the manufacturer's IP is under attack.

4.1. Information gathering

As explained in Section 3.1 the first step consists in gathering as much information as possible about the system.

Inspecting system components. The labels of the MCUs were removed but we were able to discover the names of the MCUs, and to obtain the relative datasheets using the pin-out technique described in Section 3.1. The most relevant features of RED were:

- a one-time-programmable (OTP) memory containing the firmware;
- two execution states: *development* and *deployed*. In the development state the debug interface is enabled and, through it, it is possible to read the firmware;
- an AES-128 hardware accelerator.

The most interesting features of BLUE were:

- a non-volatile memory (NVM) containing the firmware;
- various functionalities, among which one for dumping the NVM content that, unfortunately, was locked.

The flash memory was a fairly standard one, offering usual operations such as: erase, program, read, etc.

The only way for a user to interact with the device was through the button connected to RED: when pressed, RED started to read the flash content. RED communicated with BLUE only when some particular data were loaded into the flash. This information, extracted by monitoring the transferred data on the communication bus, that we probed with the logic analyzer, lead us to the idea that the flash was an essential portion of the system, and that the functionalities of the device were conditioned by the content of the flash.

The ISA of RED was publicly available, so using a disassembler software like Ghidra[2] we searched RED native code in the flash, as plaintext, with no success.

[2]https://ghidra-sre.org/

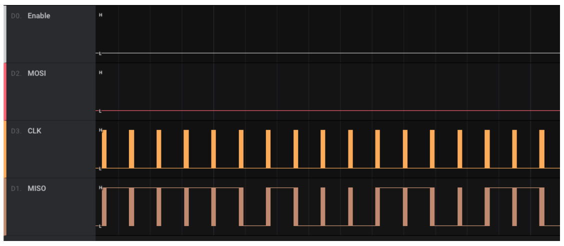


Figure 7: The RED MCU is reading the flash: each spike corresponds to a 16-byte block.

```
if (status == DEVELOPMENT) {
    run_development();
} else {
    run_deploy();
}
```

Figure 8: The hypothesis about how RED selects its running mode.

Detecting encrypted content. As explained in Section 3.1, we measured the entropy level of the flash content to understand if it was encrypted. The entropy level was really high inducing us to think that the majority of the flash content was indeed encrypted. By examining the flash content we noticed blocks of 16 bytes repeated in the memory. This led us to the hypothesis of a block cipher with a block size of 16 bytes, used in ECB mode of operation.[3] This hypothesis was strongly supported by the presence of an AES-128 hardware accelerator in the RED MCU. Furthermore, we noticed that RED read the flash data 16 bytes at the time. In Figure 7 we show the RED's reading pattern, where each of the spike corresponds to the MCU reading 16 bytes from the flash. Between each read block, there is an empty communication time lapse during which, most likely, RED was performing the decryption of the received block.

4.2. Attack Vectors

We now report the attack vectors that we devised for the two MCUs assuming a setup based on off-the-shelf, affordable hardware (cf. Sections 2, 3 and 3.2).

Attack vectors for RED. The first idea is based on the following educated guess: RED has two running states, the development mode and the deployed one. Different functionalities are available depending on the MCU state, meaning that at some point this state is checked, most likely in the bootloader. We can presume that the bootloader contains code similar to the one of Figure 8. The first attack vector aims at injecting a voltage glitch, during the bootloader state check, to enter the development mode, access the debug interface and extract the firmware. The second attack vector is based on side channel analysis. Indeed, with the assumption that AES-128 is used to decrypt the flash content, it would be possible to perform a power analysis attack to recover the AES key. This would allow us to decrypt the flash content, understand

[3] ECB mode encrypts each plaintext block independently, under the same key.

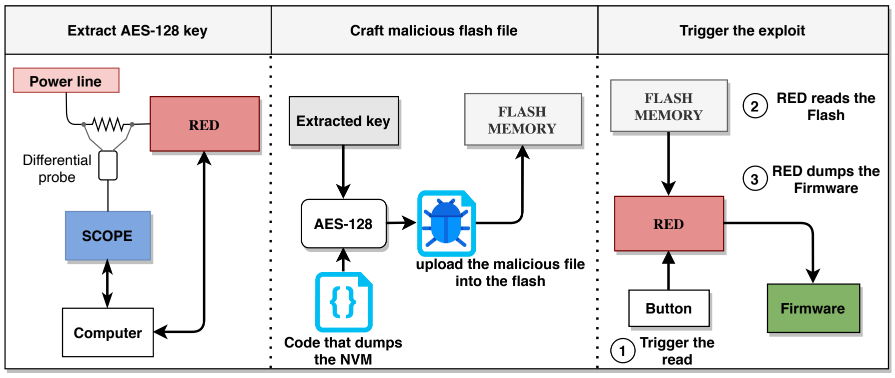


Figure 9: Steps for the attack vector based on power analysis.

its format, and then craft a malicious content to dump the MCU firmware, as summarized in Figure 9.

Attack vectors for BLUE. The only attack vector for BLUE is based on fault injection. The functionality for dumping the internal memory was disabled. Assuming this is done using a branch based on a boolean flag, the idea is to skip the branch and re-enable the functionality.

4.3. Attack results

This section describes the attack performed over the target device. The attacks are based on the attack vectors identified in Section 4.2. We bought two MCUs identical to RED and BLUE, in order to experiment attacks on a firmware under our control before reproducing them on the real targets. As explained in Section 3.2, controlling the firmware makes attacks simpler to experiment with. Moreover, attacks can become easily destructive, so it is a common practice to try them on spare MCUs. We start by describing the simplest attack on BLUE.

Voltage fault injection on BLUE. We used the setup of [14] (cf. Section 3.2). The attack consists in finding the right voltage glitch shape, its duration, and the timing for injecting the glitch. It is possible to reduce the search space of some parameters by using some reasonable constraint, e.g., if the target code is in the boot ROM after power-on reset, it is reasonable to target only the firsts milliseconds after the boot. We successfully skipped the branch and re-enabled the dump firmware functionality using a blank MCU. This took just a few days. The very same attack worked for the actual target device, and we could use the dump NVM functionality and extract the firmware.

Controlled environment for CPA. We tried the first attack vector, the one that uses fault injection, using the same approach used for BLUE, but we were unable to perform a successful attack. So we decided to try the second attack vector: AES correlation power analysis. We wrote native MCU code to encrypt and decrypt the data on the device under a parametric key so that we could control both the data and the key from an external Python program running

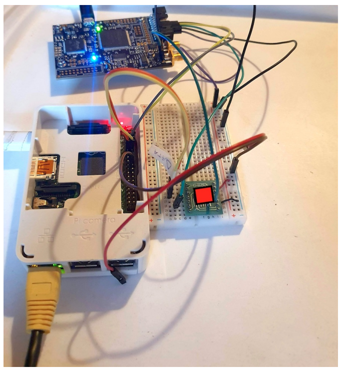


Figure 10: The controlled environment for Correlation Power Analysis

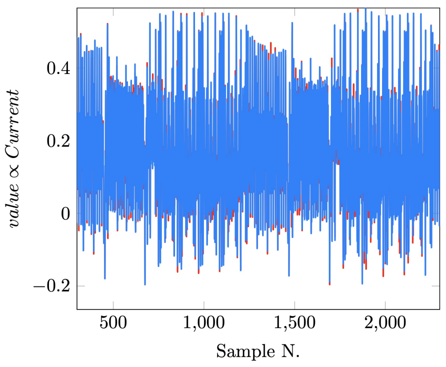


Figure 11: The initial part of two AES-128 encryption power traces on the RED MCU.

on a Raspberry Pi. To capture the power trace we used Chipwhisperer Lite. The code changed the state of a GPIO just before the start of the AES operation, in order to trigger Chipwhisperer Lite recording at the right time. To measure power consumption we placed a shunt resistor in series with the RED MCU's power supply rail. The attack setup is shown in Figure 10.

Our first experiment aimed at giving us knowledge about the "shape" of an AES-128 round. In Figure 11 we show the starting portion of two power traces that were generated by encrypting different plaintexts using the same AES key. The differences in height, between the red trace and the blue one, are due to the different plaintexts used. It is notable that in the power traces some portions repeat, due to the AES rounds. In fact, we claimed that the first round started around sample number 500 and ended around sample number 1500, where the shape of the plot starts repeating. We collected several traces to try a CPA attack and experimented with more sophisticated setups to increase accuracy of measurements, as explained below.

Attacking the RED target. To improve the quality of the analysis we milled a Printed Circuit Board (PCB). The information gathering task of Section 4.1 convinced us that the 16-byte blocks were read from the flash memory and decrypted on the fly. So, we changed the first block of the flash with our chosen data, in order to control the first input block of AES decryption. Then,

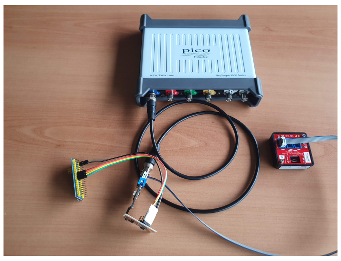


Figure 12: Correlation Power Analysis of the real target.

we triggered the button of the RED MCU to start the decryption process. We chose to use the Picoscope 5444D and the differential probe, with the idea that these changes would drastically improve the attack results. The setup (Figure 12) was so good that after only a hundred traces the best correlations values were around 0.98, and all the other guesses had correlation values around 0.01. This result convinced us that we had extracted the correct key. To confirm the correctness of the key, we decrypted some of the flash content and analyzed them with Ghidra. We found the code responsible for some of the measurable actions confirming that the key was indeed correct.

At this point, we were able to decrypt all the flash content and encrypt arbitrary data. We wanted to verify if some sort of check was performed before executing the firmware loaded from the flash. After a few attempts we discovered the presence of a checksum on the plaintext bytes. This allowed us to arbitrarily modify the firmware and make the MCU execute it. At this point, we could in principle execute arbitrary code. What we did not know was where RED started to execute the flash content. We overcame this problem using a standard *nop sled*, i.e., a block of NOP instructions, doing nothing, right in front of the code. This increases the probability for the code to be executed as any address of the NOP instructions is a valid entry point. Using this technique we were able to extract the firmware stored in the internal OTP memory, using the two GPIOs technique described in Section 3.2.

5. Conclusion

In this paper we have systematized the knowledge about the attacker techniques and methodologies for embedded devices and shown how to apply them to a real case study. In particular, we have shown two different attacks performed on the two MCUs of the analyzed device, one based on fault injection, the other one, more complex, requiring: (i) the extraction of the AES-128 key used to encrypt the code executed by the MCU, and (ii) the injection of malicious code, encrypted under the leaked key, used to dump the firmware of the MCU. Our work shows that advanced attacks on IoT devices can be performed with off-the-shelf, affordable hardware. These attacks fully compromise the device intellectual property by leaking the firmware and allowing for the development of identical clones of the target device. Manufacturers should seriously consider the adoption of more secure (but also more expensive) MCUs with hardware protections against power analysis and side channel attacks. We believe that developing more secure, affordable hardware will be an important challenge for the coming years.

Acknowledgments

This work has been partially supported by POR FESR projects DOMHO: “Sistema domotico IoT integrato ad elevata sicurezza informatica per smart building” and SAFE PLACE: “Sistemi IoT per ambienti di vita salubri e sicuri”. We would also like to thank the anonymous reviewers for their constructive and helpful suggestions.

References

[1] B. Schneier, The Internet of Things Is Wildly Insecure — And Often Unpatchable, 2014. Published on-line on Wired: https://www.wired.com/2014/01/theres-no-good-way-to-patch-the-internet-of-things-and-thats-a-huge-problem/.

[2] F. Benvenuto, Firmware extraction through power analysis of cryptographic algorithms, Master’s thesis, Ca’ Foscari University, Venice, Italy, 2021.

[3] P. Kocher, J. Jaffe, B. Jun, Differential power analysis, in: M. Wiener (Ed.), Advances in Cryptology — CRYPTO 1999, Springer Berlin Heidelberg, 1999, pp. 388–397.

[4] E. Brier, C. Clavier, F. Olivier, Correlation power analysis with a leakage model, in: Cryptographic Hardware and Embedded Systems — CHES 2004, Springer, 2004, pp. 16–29.

[5] J.-S. Coron, P. Kocher, D. Naccache, Statistics and secret leakage, in: International Conference on Financial Cryptography, Springer, 2000, pp. 157–173.

[6] R. Mayer-Sommer, Smartly analyzing the simplicity and the power of simple power analysis on smartcards, in: Ç. K. Koç, C. Paar (Eds.), Cryptographic Hardware and Embedded Systems — CHES 2000, Springer Berlin Heidelberg, 2000, pp. 78–92.

[7] F. Amiel, B. Feix, K. Villegas, Power analysis for secret recovering and reverse engineering of public key algorithms, in: International Workshop on Selected Areas in Cryptography, Springer, 2007, pp. 110–125.

[8] J.-S. Coron, Resistance against differential power analysis for elliptic curve cryptosystems, in: Ç. K. Koç, C. Paar (Eds.), Cryptographic Hardware and Embedded Systems — CHES 1999, Springer Berlin Heidelberg, 1999, pp. 292–302.

[9] J. Moon, I. Y. Jung, J. H. Park, IoT application protection against power analysis attack, Computers & Electrical Engineering 67 (2018) 566–578.

[10] T. Goodspeed, A side-channel timing attack of the msp430 bsl, Black Hat USA (2008).

[11] J. Obermaier, S. Tatschner, Shedding too much light on a microcontroller’s firmware protection, in: 11th USENIX Workshop on Offensive Technologies, WOOT, 2017.

[12] R. B. Carpi, S. Picek, L. Batina, F. Menarini, D. Jakobovic, M. Golub, Glitch it if you can: Parameter search strategies for successful fault injection, in: Smart Card Research and Advanced Applications - 12th International Conference, CARDIS, 2013, pp. 236–252.

[13] S. Picek, L. Batina, D. Jakobovic, R. B. Carpi, Evolving genetic algorithms for fault injection attacks, in: 37th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO, 2014, pp. 1106–1111.

[14] C. Bozzato, R. Focardi, F. Palmarini, Shaping the glitch: Optimizing voltage fault injection attacks, IACR Transactions on Cryptographic Hardware and Embedded Systems 2 (2019) 199–224.

[15] N. Timmers, C. Mune, Escalating privileges in Linux using voltage fault injection, in: 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017, 2017, pp. 1–8.

[16] T. Kasper, D. Oswald, C. Paar, A versatile framework for implementation attacks on cryptographic rfids and embedded devices, Trans. Computational Science 10 (2010) 100–130.

[17] Riscure, VC Glitcher datasheet, 2017. https://www.riscure.com/uploads/2017/07/datasheet_vcglitcher.pdf.

[18] D. Boneh, R. A. DeMillo, R. J. Lipton, On the importance of checking cryptographic protocols for faults (extended abstract), in: Advances in Cryptology - EUROCRYPT, 1997, pp. 37–51.

[19] O. Kömmerling, M. G. Kuhn, Design principles for tamper-resistant smartcard processors, in: Proceedings of the 1st Workshop on Smartcard Technology, 1999.

[20] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, J.-P. Seifert, Fault attacks on RSA with CRT: Concrete results and practical countermeasures, in: B. S. Kaliski, ç. K. Koç, C. Paar (Eds.), Cryptographic Hardware and Embedded Systems — CHES 2002, Springer Berlin Heidelberg, 2003, pp. 260–275.

[21] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, C. Whelan, The sorcerer's apprentice guide to fault attacks, Proceedings of the IEEE 94 (2006) 370–382.

[22] M. Hutter, J.-M. Schmidt, T. Plos, Contact-based fault injections and power analysis on rfid tags, in: European Conference on Circuit Theory and Design ECCTD, IEEE, 2009, pp. 409–412.

[23] T. Korak, M. Hoefler, On the effects of clock and power supply tampering on two microcontroller platforms, in: 2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC, 2014, pp. 8–17.

[24] M. Joye, M. Tunstall (Eds.), Fault Analysis in Cryptography, Information Security and Cryptography, Springer, 2012.

[25] A. Barenghi, L. Breveglieri, I. Koren, D. Naccache, Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures, Proceedings of the IEEE 100 (2012) 3056–3076.

[26] N. Selmane, S. Guilley, J. Danger, Practical setup time violation attacks on AES, in: Seventh European Dependable Computing Conference, EDCC-7, 2008, pp. 91–96.

[27] A. Barenghi, G. Bertoni, L. Breveglieri, M. Pellicioli, G. Pelosi, Low voltage fault attacks to AES, in: IEEE International Symposium on Hardware-Oriented Security and Trust, HOST, 2010, pp. 7–12.

[28] L. Cojocar, K. Papagiannopoulos, N. Timmers, Instruction duplication: Leaky and not too fault-tolerant!, in: Smart Card Research and Advanced Applications - 16th International Conference, CARDIS, 2017, pp. 160–179.

[29] R. Lyda, J. Hamrock, Using entropy analysis to find encrypted and packed malware, IEEE Security & Privacy 5 (2007) 40–45.