BRNO UNIVERSITY OF TECHNOLOGY

BRNO UNIVERSITY OF TECHNOLOGY

FACULTY OF INFORMATION TECHNOLOGY

FACULTY OF INFORMATION TECHNOLOGY

DEPARTMENT OF INTELLIGENT SYSTEMS

DEPARTMENT OF INTELLIGENT SYSTEMS

SOFTWARE AND HARDWARE BASED FAULT INJECTION ATTACKS AGAINST THE CPU AND MCU

SOFTWARE AND HARDWARE BASED FAULT INJECTION ATTACKS AGAINST THE CPU AND MCU

MASTER'S THESIS

MASTER'S THESIS

AUTHOR **Bc. MAREK LÖRINC**

AUTHOR

SUPERVISOR **Ing. MARTIN PEREŠÍNI**

SUPERVISOR

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (UITS) Academic year 2021/2022

Diploma thesis assignment

Student: **Lörinc Marek, Bc.**
Program Information Technology
Field of study: Intelligent Devices
Title: **Software and Hardware Based Fault Injection Attacks against the CPU and MCU**
Software and Hardware Based Fault Injection Attacks against the CPU and MCU
Category: Security
Assignment:

1. Study the issue of software and hardware fault injection in CPU and MCU calculations. Focus in particular on faults caused by voltage changes.
2. Read and familiarize yourself with at least two specific studies (from the literature) that deal with the issue of voltage changes.
3. Based on the studies, perform a security analysis and create a protocol that describes in detail the replication of attacks in practice.
4. Verify the selected attacks on real hardware. Perform attacks on at least two different Intel x86 processors or ARM-based MCUs.
5. Discuss the results achieved by replicating attacks and the characteristics of defenses against such attacks. Propose extensions and improvements to defenses against voltage modification attacks.

Literature:

- https://plundervolt.com/
- http://voltjockey.com/
- https://arxiv.org/abs/1912.04870
- https://zt-chen.github.io/voltpillager/
- https://platypusattack.com
- https://media.ccc.de/v/36c3-10859-trustzone-m_eh_breaking_armv8-m_s_security

When defending the semester part of the project, the following is required:

- 1-3

For detailed binding instructions for preparing the thesis, see
https://www.fit.vut.cz/study/theses/ Thesis supervisor: **Perešíni Martin, Ing.**
Head of department: Hanáček Petr, doc. Dr.
Ing. Date of assignment: November 1,
2021
Submission date: May 18, 2022
Date of approval: November 3, 2021

Abstract

The thesis deals with attacks that cause errors in CPU and MCU calculations. A short change in CPU or MCU voltage is used to cause the error. The theoretical part of the thesis describes how to cause and exploit these errors. This part also describes the most well-known protection against hardware attacks, which is the trusted execution environment. Triggering an error in this environment is the primary goal of attacks that cause errors. The practical part deals with the replication of PlunderVolt and VoltPillager attacks on Intel processors with the SGX trusted execution environment enabled. Several experiments were performed to trigger errors in RSA and AES encryption within the SGX enclave. Known analytical methods were used on these errors, which successfully obtained the encryption key. The practical part also deals with the replication of an attack on ARM microcontrollers with the TrustZone-M trusted execution environment enabled.

Abstract

The thesis deals with attacks that cause faults in CPU and MCU calculations. A short voltage change in CPU or MCU is used to trigger the error. The theoretical part of the thesis deals with the description of how to cause and exploit these errors. This section also describes the most well-known protection against hardware attacks, which is a trusted execution environment (TEE). Injecting a fault into TEE is the primary target of fault attacks. The practical part deals with the replication of PlunderVolt and VoltPillager attacks on Intel processors with an activated TEE SGX. Several experiments were performed to trigger faults in RSA and AES encryption within the SGX enclave. To obtain the encryption key from these errors, known analysis methods were used. The practical part also deals with the replication of the attack on ARM microcontrollers with an active TEE TrustZone-M.

Keywords

attacks causing errors, hardware injection errors, software injection errors, PlunderVolt, VoltPillager, voltage change errors, TrustZone-M(eh), hardware, Intel SGX, ARM Trustzone

Keywords

fault attacks, software fault injection, hardware fault injection, PlunderVolt, VoltPillager, voltage fault injection, TrustZone-M(eh), hardware, Intel SGX, ARM Trustzone

Citation

LÖRINC, Marek. *Software and hardware fault injection in CPU and MCU calculations*. Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Martin Perešíni

Software and hardware fault injection in CPU and MCU calculations

Declaration

I declare that I have prepared this diploma thesis independently under the supervision of Ing. Martin Perešíni. I have cited all literary sources, publications, and other sources from which I have drawn.

. . . . . .

Marek Lörinc
May 13, 2022

Acknowledgements

I would like to thank my thesis supervisor, Ing. Martin Perešín, for his expert assistance in the completion of this thesis.

Contents

Chapter 1

Introduction

Injecting errors into MCU and CPU calculations is one of the biggest threats that an attacker can use to obtain sensitive data. Even if we focus on voltage injection errors, this method is simple and affordable compared to others. In the case of software error injection, there are no costs for special equipment.

To protect against any attacks, primarily hardware attacks, a trusted execution environment (such as Intel SGX, ARM TrustZone) is used, which should ensure the confidentiality of application data under any conditions through encryption. However, this has proven to be insufficient security, as an attacker can exploit these bugs to obtain a private key in encryption, for example, using differential error analysis. In fact, some microcontrollers have successfully bypassed the entire trusted execution environment. This made it possible, for example, to obtain the PIN from the TREZOR One hardware wallet.

In the past, the feasibility of these attacks was often criticized, and even if an attack was possible, the equipment required would be very expensive. However, this is no longer the case, as demonstrated by this work, because several attacks were successfully replicated at a low cost for the attack hardware.

1.1 Structure of the work

Chapter 2 explains what error-inducing attacks are, their types, possible implementation, and the most common types of errors they induce. Chapter 3 introduces the reader to technologies and designs aimed at increasing the security of computing technology through hardware enhancements and related software modifications. These techniques are based on encryption. Therefore, Chapter 4 discusses the most common encryption algorithms used. The goal of injecting an error is most often to cause an error in this encryption. Chapter 5 explains to the reader how these induced errors can be exploited using various known analyses. Chapter 6 describes one software attack and two hardware attacks that use voltage changes to inject errors. These attacks have also been carried out and their implementation is described in Chapter 8. The implementation follows the protocol created in Chapter 7, which describes the general procedure for replicating an attack that causes an error by changing the voltage. Chapter 9 describes the results achieved by replicating the attacks and suggestions for improving defense against the attack.

Chapter 2

Attacks causing errors

An attack that causes errors is an attack on physical electronic devices such as chip cards, hardware security modules[1] or CPUs. The attack consists of **loading** the device **with** external means (e.g., voltage, light) in order **to generate errors**. These errors lead to **a failure of** the system's **security** (key renewal, acceptance of a false signature, PIN code renewal) [12].

A successful attack consists of two steps (Figure 2.1):

- injecting an error
- exploiting the error

The first step is to introduce an error at an appropriate time during the process. Error injection can be **hardware** or **software based**. The second step is to exploit the erroneous result or unexpected behavior. Exploiting errors depends on the design and implementation of the software. In the case of an algorithm, it will also depend on its specification, as exploit will usually be combined with **cryptanalysis.** Depending on the type of analysis performed, the error injection will have to be performed at **a precise moment** or approximately within a given time period [12].

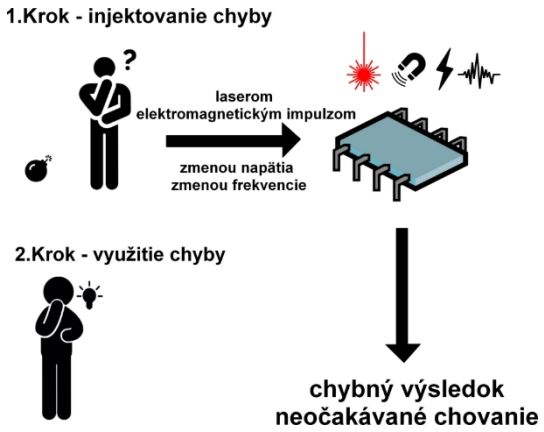

Fig. 2.1: Process of an attack causing errors

[1] physical computing device that protects and manages digital keys, performs digital signature encryption and decryption functions, strong authentication, and other cryptographic functions [5]

2.1 Hardware fault injection

Hardware-controlled fault injection techniques use **separate hardware** to inject external faults. The goal is to induce physical stress on the hardware and thereby cause an error in the victim's software [56]. There are several methods for hardware fault injection that use different tools. Fault injection tools can be compared based on their characteristics, which are summarized in Tables 2.1 and 2.2.

The following features are essential for each tool [10]:

- **Effect of the tool**. In this case, the effect can be global or local. A local bug indicates that the attacker could precisely limit the effect of the bug to a specific location, compared to a global bug, where the attacker cannot actually control where the bug is inserted.
- **Requirements for the target device**. Unpacking means that the outer layer of the device must be opened in order to access the chip inside. So it will be necessary to open the device either by chemical or mechanical means.
- **Damage** refers to the condition of the device after the flaw has been injected. With some flaw injection tools, the target device may be destroyed. Therefore, the choice of tools can be adjusted depending on the availability and importance of the target device.
- **Device design detail**. Some of the tools require knowledge of the device in order to inject the fault correctly, while in other cases partial or no knowledge is required.
- **Accuracy** of **the** fault injection **tool**. The time parameter refers to the accuracy of the timing when the fault is injected. The position indicates how accurately the location of the fault in the target device needs to be determined. The cost of the injection device in this case is closely related to accuracy. To achieve higher accuracy, the fault injection device will be more expensive.
- **Technical and professional skills** required to operate the device. In this case, some specialized tools, such as lasers or ion beams, can be more dangerous and cause potential damage. Therefore, a higher level of skill is required to operate the device safely.

Method	Effect	Unpacking	Design detail	Damage
Change in tension	Global	No	Partially necessary	No
Frequency change	Global	No	Required	No
Warm-up	Global	No	Required	Possible
EM pulse	Local	No	Partially necessary	Possible
Light impulse	Local	Yes	Necessary	Possible
Laser beam	Local	Yes	Required	Possible
Light radiation	Local	Yes	Not required	Yes
Ion beam	Local	Yes	Needed	Yes

Table 2.1: Most commonly used methods for injecting errors and their characteristics 1 [10]

Method	Accuracy - Time	Accuracy - Position	Cost	Skills
Voltage change	Moderate	Low	Low	Medium
Frequency change	High	Low	Low	Medium
Warming	None	Low	Low	Low
EM pulse	Moderate	Moderate	Low	Medium
Light impulse	Moderate	Moderate	Medium	Medium
Laser beam	High	High	High	High
Light radiation	Moderate	Low	Low	Medium
Ion beam	High	High	High	High

Table 2.2: Most commonly used methods for injecting errors and their characteristics 2 [10]

Voltage change attack

A voltage attack involves manipulating the power supply of the target devices. A simple example would be **to reduce the supply voltage**, which introduces timing errors due to propagation delays. This insufficient power supply has been shown to cause errors in devices during cryptographic operations. However, this does not provide good timing accuracy, making it difficult to target a specific instruction [11].

The ability to target a specific instruction can be achieved by **a short change in the** supply **voltage** at a specific location within a specific time window. Positive and negative voltage changes can be inserted into the external power supply
, both positive and negative voltage changes can be inserted, where the change is very short (typically in the range ofns toμs) and attempts to cause errors on specific instructions [57].

A successful voltage change attack depends on many parameters that the attacker can control [29]:

- nominal power supply
- voltage change width
- voltage change pulse width
- delay (in relation to the target operation)
- CPU/MCU temperature

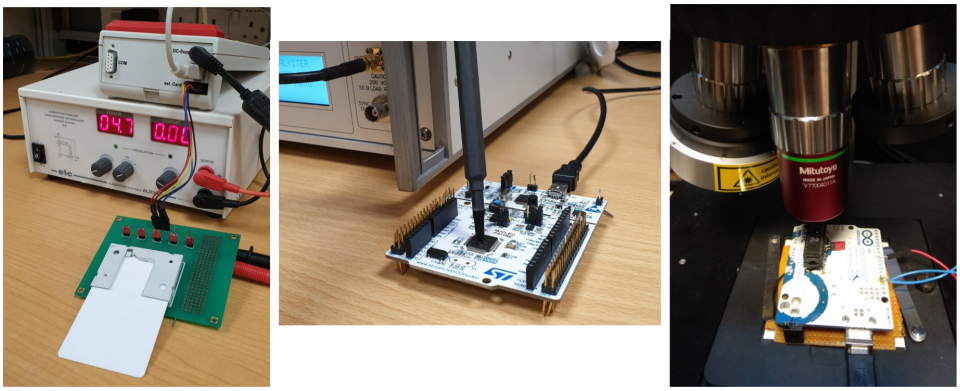

Fig. 2.2: Attack by voltage change on a smart card (left), electromagnetic pulse (center) on a microcontroller, and laser on a microcontroller (right) [10]

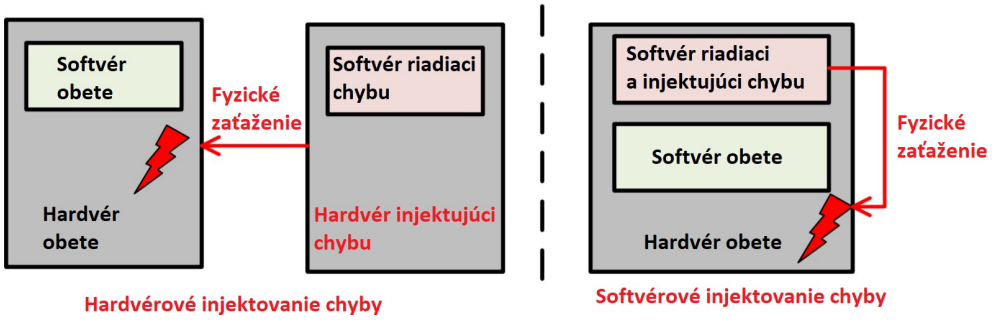

Fig. 2.3: Comparison of hardware and software fault injection [56]

2.2 Software error injection

Software-based fault injection techniques are controlled by **malicious software** (i.e., fault injection software and control software) running on the **same** hardware platform as the victim software to be attacked. This malicious software changes the physical operating conditions of the target hardware. While hardware-controlled techniques require physical access to the target system, software bug injection also allows for remote attacks [56].

Initially, these attacks were interesting in scenarios where the attacker is unprivileged or even has their own space allocated. However, with trusted computing technologies such as Intel SGX, ARM TrustZone, and AMD SEV, privileged attackers must also be considered part of the corresponding threat models. However, there are still attacks that have been able to deal with this (see Chapter 6).

The most commonly used techniques for software injection errors are:

- DVFS interface manipulation
- Inducing memory corruption errors

2.2.1 Manipulation of the DVFS interface

In modern systems, Dynamic Voltage Frequency Scaling (DVFS) is the primary power management technique used, which regulates the operating voltage and frequency of the microprocessor based on the workload. In a typical DVFS scheme, core-level controllers manage the frequency and voltage of the processor through **voltage regulators** that are on a separate chip.

One of the most famous software attacks [52] demonstrated that an attacker can exploit the interface between software drivers and hardware controllers to cause malfunctions in multi-core processors. Here, the attacker uses a malicious kernel-level driver running on the processor core to set the operating voltage and frequency of another core. This method allows the attacker to violate the time constraints of the victim's core settings by overclocking and underpowering for a certain period of time. The attacker controls the temporary location with overclocking endpoints or undervolting periods. The severity of the bug is determined by **the overclocking frequency and undervolting value.**

This method does not require any additional hardware for injecting errors or physical access to the target device.

2.2.2 Inducing memory corruption

In this method of injecting errors, the attacker inserts faults into memory cells by exploiting reliability issues in modern memories such as DRAM and Flash memory chips. Gradual miniaturization in manufacturing technology has enabled memory manufacturers to significantly reduce the price per bit by placing smaller memory cells closer together. However, this also increases electrical interference between memory cells. Accessing a memory cell electrically disturbs the surrounding memory cells. A disturbed memory cell loses its value and memory corruption occurs when the amount of electrical interference exceeds the noise threshold of that cell [56].

An attacker can cause memory corruption through an unprivileged program to inject corruption. This program repeatedly accesses a set of memory cells (i.e., the attack memory cells) to cause corruption in a set of compromised memory cells storing security-sensitive data. This allows the attacker to compromise the memory space of a security-sensitive program from the attacker's controlled memory space. Memory corruption errors are mostly exploited in DRAM and NAND Flash memory chips [17].

2.3 Characteristics of error models

Attacks exploiting bugs are based on manipulating the device in such a way that it functions abnormally. From the device's response, which may be an erroneous result, an error message, or some form of security reset (including destruction of the device), the attacker wants to learn something about the secrets hidden in the device. Since cryptographic algorithms must be public in order for users to trust them, there is no such thing as secrecy. Therefore, an attacker can determine which variables are used and which values depend on the secret key. This makes it possible to determine what kind of error will cause a certain response that the attacker can observe. For example, if one bit in the secret key flips during an attack and the device does not detect this error, the output is an incorrect result with **a specific pattern.** By comparing this incorrect result with the correct one, the attacker may be able to deduce one bit of the secret key. The attacker can also target the flow of operations so that certain operations are repeated or omitted. To achieve and exploit the desired effect, they must have knowledge of how a particular physical attack will affect the logical flow of the attacked algorithm. Only then will it be possible to determine the probability of success and determine the secret data from the erroneous output [42].

The description of the error-inducing attack must specify the error model. The model clarifies the attacker's capabilities and must include parameters such as error type, timing, location, error injection accuracy, and number of failures. First-order models assume that the attacker is only capable of inducing a single error during the execution of the algorithm, while second-order models assume that it is possible to inject more than one error [29].

2.3.1 Error model - precise bit flipping

Proponents of error-inducing attacks try to demonstrate the effectiveness of their attacks by making as few assumptions as possible (for example, the location of the error is considered unknown, which is later recovered through thorough analysis). From the perspective of a differential error attack

(see Chapter 5), the strongest model flips **one specific bit** (the attacker chooses both the round of the cipher and the location of the bit). Let us consider the impact of changing one input bit of the AND operation:
$y = x_{(0)} \wedge x_{(1)}$. If it is possible to flip exactly $x_{(0)}$, then by assessing whether the output has changed or not, it is possible to obtain $x_{(1)}$ (no change in output means $x_{(1)} = 0$, otherwise $x_{(1)} = 1$). In this way, it is possible to attack any AND gate. It is also interesting that this model
breaks countermeasures such as detection or infection. Even if the cipher is 'protected' by detection or infection (which effectively prevents obtaining an erroneous output), the attacker is able to check whether flipping one bit of the AND operation changes the output or not [10].

2.3.2 Single/multiple fault model

Most published works on error-inducing attacks assume a single-error model. According to this model, an attacker can inject errors at most once during a single execution of the cipher (which may affect multiple locations). So if two sets of errors are injected, say in the first and last rounds of encryption, this would be considered a violation of the model [10].

Generic modeling of attacks with multiple errors becomes much more difficult because, theoretically, any set of parameter values for triggering the first error can be combined with any set of parameter values for triggering the second error. Of course, restrictions can be introduced, for example, the same type of error can be triggered twice. Even with these restrictions on attackers, their capabilities become stronger. For example, an attacker can now trigger errors in both a variable and a procedure testing that variable. This defeats many countermeasures designed to resist single-error attacks [29].

2.3.3 Random/Deterministic Error Model

The random error model is the most commonly used in the literature [46]. Here, the attacker can control in which round errors can be injected, but cannot control the value changed by the error injection. Essentially, the injected error results in the reversal of one or more bits of the operand value. In general, the attacker can control the intensity, location, and duration of external interference, but cannot control the accuracy of the error injection (it is assumed that the impact of the error injection is unknown). In certain cases, it can be assumed that the exact location of the error is also unknown (the complexity of the attack is multiplied by the number of error locations [26]). Depending on the specific attack, the target for error injection may be a word (byte/nibble) or a series of bits. In addition to bit flipping, various models can be used where specific bits are set to 1 or reset to 0 [14]. For byte-oriented ciphers such as AES, a random byte error is most often assumed.

2.3.4 Error model - setting/resetting multiple bits

The prerequisites for using this model are as follows [30]:

1. Setting/resetting multiple bits can be introduced by injecting an error into a given device
2. Unknown plaintext can be encrypted multiple times
3. Various incorrect or correct outputs can be compared in pairs without revealing their values

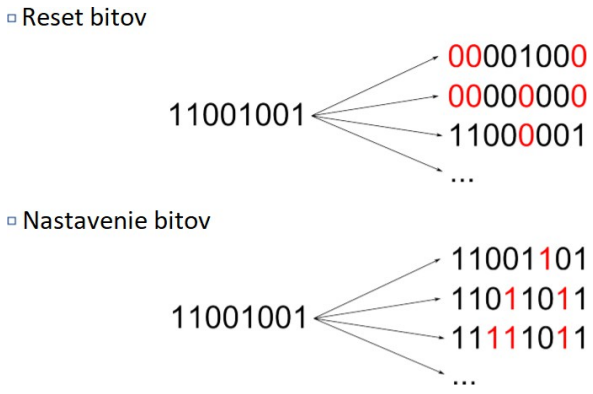

Fig. 2.4: Example of setting and resetting multiple bits

The model with setting/resetting multiple bits was observed during error injection using an electromagnetic pulse [38] or a laser [50], where it is stated that during laser error injection into SRAM, the bit flip error model is not applicable. Only errors using the multiple bit setting/reset model are possible. An example of multiple bit reset and setting is shown in Figure 2.4.

2.3.5 Error model - instruction skip

Instruction skipping is an error that results in skipping, which means **that** one program instruction **is not executed** during runtime (as if the program flow skipped the erroneous instruction). Several studies have examined instruction skipping using EM pulses. The studies deal with skipping a single instruction [39], but also with skipping several instructions in a row [49]. Several works also deal with skipping instructions using a laser, where a single instruction is skipped with high accuracy and high success rate. This is then used to perform a successful differential error attack on AES [25].

The following code[2] may be a suitable candidate for instruction skipping:

```
bool firmware_is_valid = validate_firmware();
if(!firmware_is_valid)
    abort();
boot();
```

Listing 2.1: C code suitable for demonstrating instruction skipping

If the conditional instruction is successfully skipped, potentially dangerous firmware may be booted as safe.

[2] Taken from https://media.ccc.de/v/36c3-10859-trustzone-m_eh_breaking_armv8-m_s_security#t= 907

Chapter 3

Trusted Computing

Computer security is extremely important in today's information technology world. Most computer users have already encountered viruses, spam, phishing, or other types of malware, or even threats to their privacy and theft of confidential information. That is why, back in 1980, the military defined criteria for evaluating the reliability of PC systems (TCSEC). However, these criteria primarily address the security **of operating systems**, but as it later turned out, **hardware** also plays an important role**.** Even if the operating system considers the hardware to be trustworthy because there is no alternative way to test and verify the correctness of the hardware, this does not mean that the hardware cannot be compromised.

For this reason, the **Trusted Computing Group (TCG)** was formed in 2003 with the goal of developing, defining, and freely promoting specifications for Trusted Computing. Increased security should be ensured through transitive trust properties. The TCG has expanded its scope beyond computers to other devices and systems such as storage, mobile devices, servers, and peripherals.

In computer security, the concept **of Trusted Computing (TC**) can be considered as a computer system whose entities have a certain level of assurance that a given part or the entire computer system behaves in the expected manner. An entity can be considered to be a person using a computer system or a program running on a remote machine, and the degree of security assurance can cover all aspects of the system or only part of it [8].

TC must ensure [8]:

- **Storage protection**. The TCG has introduced **the Trusted Computing Platform (TCP)**, which provides a trusted disk function. This feature encrypts all data directly on the disk, and the encryption speed matches the throughput of the disk interface, so the process is essentially invisible to the user during normal operation. This means that if a trusted disk is stolen, rebuilt, or taken out of service, it remains protected.

- **Secure Online Transactions**. To protect customer and employee data from Internet attacks, Personal Information Manager (PIM) software, secured by a hardware chip in TCP, isolates contact information, passwords, bank access codes, and credit card numbers. With encryption keys stored locally in TCP, copies are automatically transferred to the Key Transfer Manager Server, providing both protection and recoverability of information.

- **Protecting data and networks from viruses and malware**. In this case, relying on antivirus and personal firewalls on portable PCs is not acceptable for securing the corporate network. An authorized user can access the network from an external site to simply check their email. If the user's computer has a virus, for example, it can spread to the network. By taking advantage of TCP, these deceptive endpoints can be detected. The TCP specification creates a level of trust in the status of the endpoint and also ensures the presence, status, and version of mandatory application software.

- **Digital rights management**. Trusted Computing would enable companies to create a digital rights management system that would be very difficult to circumvent. An example is downloading a music file. Remote verification could be used to prevent the music file from being played except with a specific music player that enforces the recording company's rules. Sealed storage would prevent the user from opening the file with another player. The music would be played in a shielded memory, preventing the user from making an unrestricted copy of the file while it is playing, and secure inputs/outputs (I/O) would prevent the capture of what is sent to the sound system.

3.1 Trusted Platform Module

The first device defined by the TCG is **the Trusted Platform Module (TPM)**, which is encapsulated within the TCP by connecting a single chip to the motherboard or by embedding the functionality within other components. The TPM is typically implemented as a microcontroller on the motherboard that stores **passwords, digital keys, and certificates** providing unique identification.

The TPM essentially creates `a chain of trust` (Figure 3.1), in which the TPM acts as `the root` of integrity verification for multiple components of the computing environment that are necessary to create a `trusted boot path`.
environment necessary to create a trusted boot path. Each layer (component) is responsible for verifying the next layer after it, so that the machine as a whole is considered trustworthy only if the entire chain is verified. Just one non-functional link will cause the chain of trust to fail [35].

3.2 Trusted Execution Environment

However, TPM modules are limited compared to modern trust technologies: they do not allow arbitrary application execution, nor are secure input/output (I/O) operations feasible without additional processes such as TPM-enabled virtual machines. One solution is **a Trusted Execution Environment (TEE)**, which shares hardware with an untrusted **Rich Execution Environment (REE(**[1)]), such as Android, but runs independently with hardware-enforced **isolation.** It allows critical applications to run with strong confidentiality and integrity guarantees alongside a potentially compromised REE, while providing TPM-like features such as **secure storage and verified boot.** In fact, the TEE has its own **CPU, memory, and peripherals.** The only way to communicate with the external environment is through the TEE application interface [51]. The described TEE architecture is shown in Figure 3 .2.

[1] n environment that provides and manages a broader operating system

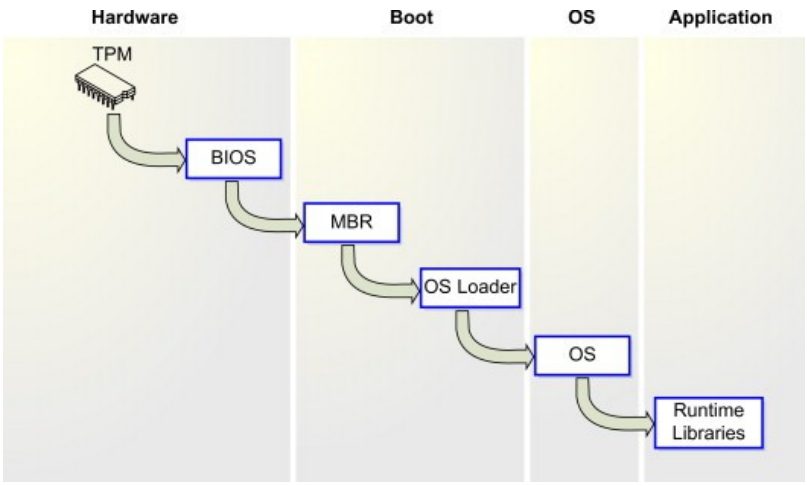

Fig. 3.1: Illustration of the chain of trust. The figure shows that components can be both hardware and software [35].

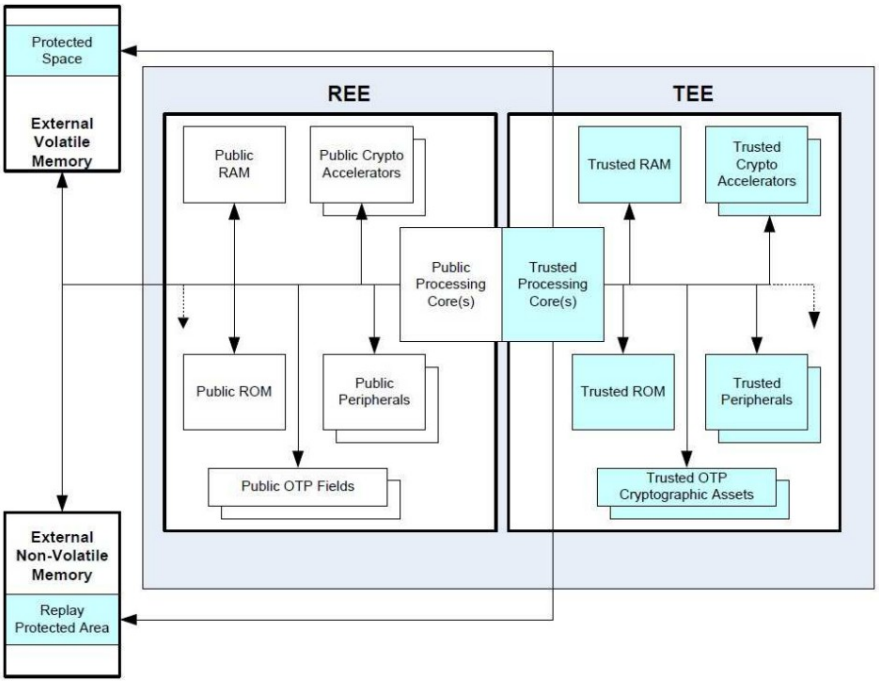

Fig. 3.2: TEE hardware architecture [6]

TEEs are used to run sensitive applications, such as fingerprint matching, with hardware-enforced isolation via a trusted hardware element, such as a CPU. Trusted applications (**TAs)** reside in their own memory areas, and such isolation is used to prevent unauthorized access from the REE space. TAs can allocate shared memory space with trusted applications or expose developer-defined functions that are mediated by highly privileged secure monitoring.

The best-known implementations of TEE are **ARM TRUSTZONE** and **Intel SGX**. Intel SGX is used in servers and larger portable devices, such as PCs or laptops with Intel processors. ARM TRUSTZONE is primarily used on devices with ARM architecture. These include various microcontrollers, single-board PCs, including Raspberry PI, and systems on chip (SoC), where smartphones are predominant [51].

3.2.1 Intel SGX

Intel SGX extends the Intel processor architecture with a set of new instructions that allow application developers to create a secure, hardware-isolated container called **an enclave**. The enclave protects **the confidentiality** and **integrity** of application content from all other software on the platform, including potentially untrusted OS. Since the platform's main memory may be under the control of an attacker, SGX ensures that enclave data is **encrypted and** integrity is protected before the data leaves the CPU. Enclaves also provide secure storage by allowing data to be sealed so that only a specific enclave can decrypt it.

Unlike TPM, all SGX calculations are performed by **the CPU**, resulting in a significant increase in performance. A typical use case for SGX is **the isolation** and **protection** of security-sensitive application **components**. Minimizing the size of these trusted components reduces the risk of security vulnerabilities and reduces the load on the verifier during remote verification [31].

Attestation

To increase confidence that the software is running securely within an enclave on an updated Intel SGX platform and security level, Intel SGX provides an attestation mechanism. Intel provides two types of enclave attestation in Intel SGX, **local** and **remote [4].**

During local attestation, the enclave asks the hardware to generate a credential, known as `a report`, and send `the report` to another enclave on **the same platform** that can verify it. The enclave `report` contains the following data [54]:

- measurements of code and data in the enclave
- public key hash at an independent certificate provider (ISV)
- user data
- other security-related status information
- signature block above the above information

For remote attestation, the application can send `the` enclave `report` to a special enclave (**Quoting Enclave - QE**) to create a type of authorization that reflects the status of the enclave and

platform. This authorization is called QUOTE and is signed with the EPID private key [4]. QE is an enclave provided by Intel that can process the enclave report and convert the report to QUOTE. Only QE has access to the Intel EPID key. QUOTE is a data structure used for remote attestation, and its main content is the same as the content of the report. The QUOTE structure contains the following data [54]:

- code and data measurement in the enclave
- public key hash at an independent certificate provider (ISV)
- product ID
- enclave security version number
- enclave attributes
- user data
- enclave attributes
- signature block above the above data

Implementation

To implement this concept, SGX introduces a new set of instructions for the x86 architecture. This allows the BIOS to allocate a memory area restricted for use by the processor (Processor Reserved Memory - PRM). Enclave data and code are located in the Enclave Page Cache (EPC), a subset of PRM. To ensure its confidentiality, EPC is encrypted using the Memory Encryption Engine (MEE), a CPU component that ensures that enclave data runs only within the limits of the CPU. The encryption key is **randomly generated** by the CPU and is changed every cycle and never crosses its boundaries. Unauthorized requests for enclave memory are blocked by the CPU and treated as non-existent memory addresses. Thus, only the enclave has access to its own information.

The system code that controls the physical memory of the computer also uses SGX instructions to manage EPC. However, this method is considered unreliable in the SGX threat model, so security checks are performed to ensure the legitimacy of all operations involving EPC. For this check, the CPU uses the Enclave Page Cache Map (EPCM) structure, which is a matrix with an entry for each EPC page. It contains three basic pieces of information about it [19]:

- Whether the page is in use
- Which enclave the page belongs to
- The type of page

This allows several enclaves to be loaded at once without interference between them.

To ensure that only authenticated enclaves can run, Intel provides **the** described **enclave signing** attestation **mechanism**. Once the enclave is instantiated, the CPU performs its measurement and stores it in the MRENCLAVE register. Its value is then compared with the signature structure. This means that if they are identical, the enclave will be successfully loaded. A register for storing the author key hash is also reserved: MRSIGNER [31].

Access to the enclave is performed through a limited and well-defined interface consisting of two types of functions: **ECALL** is a function that an untrusted application can

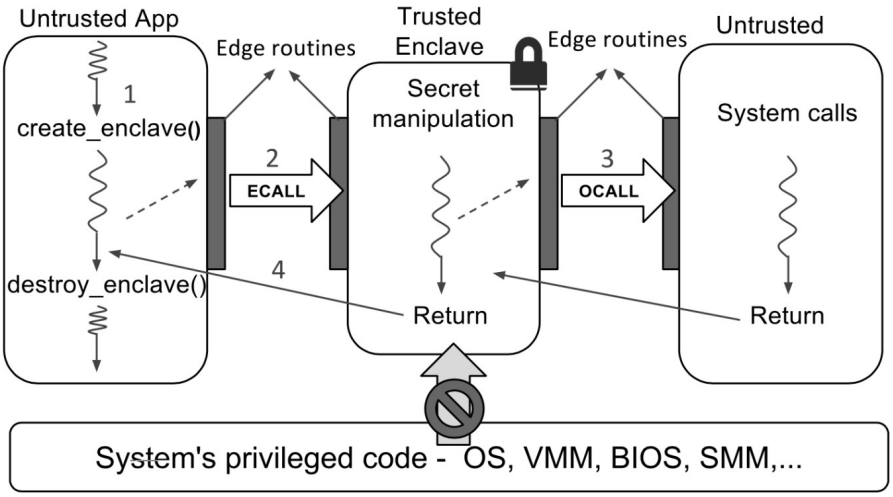

Fig. 3.3: Intel SGX application launch procedure [19]

used for calling execution within enclaves, and **OCALL** is a function that an enclave can use to access untrusted elements. These functions must be defined in a special file using the `Enclave Definition Language (EDL)`. This file contains a function declaration similar to the C language, but with special attributes to specify the direction, size, and type of data that will cross the enclave boundary. From this file, a special tool (Edger8er) generates routines (`edge routines`) that safely manipulate the relevant parameters [31].

Procedure for launching Intel SGX applications

Figure 3.3 shows the process of launching an application using Intel SGX. First, an untrusted application **creates an enclave (1)** and, when it needs to manipulate sensitive data, uses an **ECALL** call to move the execution **to the enclave (2)**. However, enclaves do not have direct access to system resources. They also cannot execute functions from dynamic libraries. If any of these are needed, the enclave uses **OCALL (3).** When the enclave is no longer needed, it is destroyed and its **data is deleted (4).**

Data security is guaranteed as long as the data is located within the enclave and is lost when the enclave is destroyed. A sealing mechanism can also be used if permanent storage is required. When protecting data using cryptography, special attention must be paid to the storage of secret keys. However, the SGX architecture allows the CPU to generate **a** unique **128-bit AES-GCM key,** which is a sealing key, for a specific enclave running on a specific platform for data encryption. The sealing key does not need to be stored because it is generated by the CPU at runtime and never leaves the processor. The developer
can choose to attach the sealing key to the `MRENCLAVE` or `MRSIGNER` register so that
only the same enclave or any enclave from the same author can unseal the data. The sealing key is also attached to a unique persistent CPU key. This means that only the CPU that sealed the data will be able to unseal it [19].

3.2.2 ARM TrustZone

TrustZone is a hardware security mechanism introduced in 2004 in Arm application processors (ARMv6 architecture, Cortex-A). In 2016, TrustZone was modified

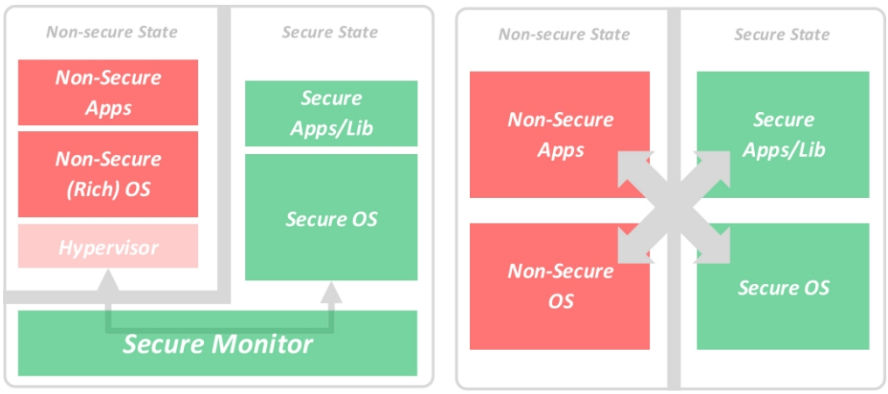

Fig. 3.4: TrustZone technology for ARM Cortex-A (left) and ARM Cortex-M (right) [44]

to cover the new generation of Arm microcontrollers (Cortex-M). TrustZone includes security extensions for systems on chip (SoC) covering **the processor**, **memory**, and **peripherals** [24]. TrustZone security is based on the idea of dividing the entire hardware and software of a system on chip (SoC) into two worlds: **a secure world and a normal world.** The secure world is everything that runs when the processor state is marked as secure. The normal world is everything that runs when the processor is in an unsecured state. Hardware barriers are created to prevent components of the normal world from accessing secure resources. The secure world is not restricted. Specifically, the system prevents the normal world from accessing [41]:

- Areas of physical memory marked as secure
- System control that applies to the secure world
- Switching states outside of approved mechanisms

TrustZone for application processors

TrustZone for application processors was developed to increase security for Cortex-A series processors. As mentioned in section 3.2.2, the most important architectural change at the processor level is the introduction of two worlds: the secure world and the normal world. Figure 3.4 illustrates this concept. At any given moment, the processor operates exclusively in one of these worlds. The world in which the processor is currently operating is determined by the value of the processor's new 33rd bit, also known as the `Non-Secure` **bit**. The value of this bit is read from the `Secure Configuration Register (SCR)` and propagated throughout the system to memory and peripheral buses.

TrustZone is an extra processor mode that is responsible for preserving the state of the processor whenever a world change occurs. This processor mode is called **monitor mode** and ensures that the current state of the world from which the processor is leaving is saved and the state of the world it is entering is restored. The processor can be put into this mode using the new privileged `Secure Monitor Call (SMC)` instruction, or by configuring a hardware exception or interrupt (IRQ, FIQ).

To strengthen the hardware isolation between worlds, the processor has expanded the versions of special registers as well as some system registers (accessible via coprocessor 15 on Armv7-A and

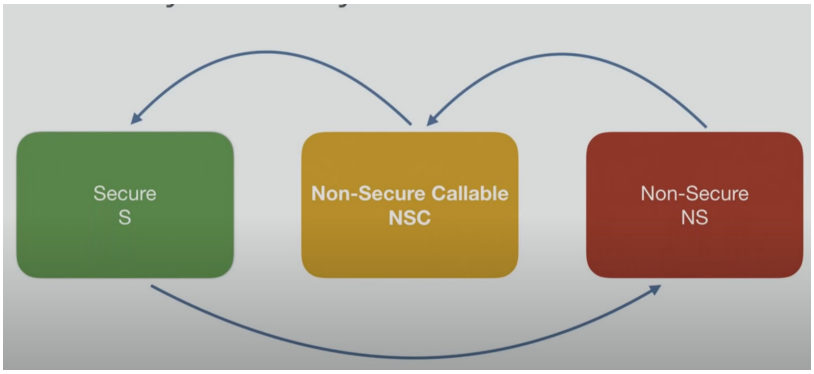

Fig. 3.5: TrustZone-M memory security states[2]

using MSR and MRS instructions on Armv8-A). In the normal world, security-critical system registers and processor core bits are either completely hidden or subject to a set of access permissions controlled by the secure world software [44].

TrustZone-M

From a high-level perspective, TrustZone-M for ARM Cortex-M microcontrollers is similar to the variant in Cortex-A processors. In both implementations, the processor can operate in a secure or unsecured state. However, there are important differences, as Cortex-M has been optimized for faster context switching and deterministic execution. As a result, the basic mechanisms of TrustZone-M differ significantly from the original TrustZone specification. In TrustZone-M, the execution state is based on **memory mapping, and** switching between worlds occurs **automatically** when code exceptions are handled [44].

TrustZone-M **does not include a monitoring mode**, which reduces the latency of switching between worlds, resulting in more efficient transitions. TrustZone-M supports multiple secure entry points to bridge software between the two worlds. For this purpose, the Instruction Set Architecture (ISA) has been extended with three new instructions, including Secure Gateway (SG). With the exception of stack pointers stack pointers and a few special registers, most registers in the Cortex-M architecture are **shared** between secure and non-secure states.

In terms of memory, the physical address space is divided into **secure** and **non-secure** sections. In addition, in TrustZone-M, the secure memory space is divided into two types: **secure and non-secure (**Non-Secure Callable - NSC). NSC is a special secure memory location used to store SG instructions. It is the entry point for every explicit transition between non-secure and secure states. This means that when transitioning from an unsecured state, **it is not** possible to transition directly to a secure state; instead, an intermediate NSC state is used. Conversely, it is possible to transition directly from a secure state to an unsecured state, as shown in Figure 3.5 [45].

The memory state can be configured using the Secure Attribution Unit (SAU) and/or the Implementation Defined Attribution Unit (IDAU). These units determine the state of the address. The stricter state wins. This means that if, for example, the SAU has a secure state and the IDAU has an insecure state, the address will have **a secure state**.

[2] Taken from https://media.ccc.de/v/36c3-10859-trustzone-m_eh_breaking_armv8-m_s_security

Some microcontrollers have only **one** of these units. For example, the SAM L11 MCU has only an IDAU, which directly determines the address status.

The TrustZone-aware Memory Protection Unit (MPU) allows each world to have a local set of memory access permissions by providing a different MPU interface for each world. The Nested Vectored Interrupt Controller (NVIC) allows interrupts to be configured as secure or unsecured. If the state of the incoming interrupt is equal to the currently executing state, the exception sequence is similar to that of previous M-series processors. The main difference occurs when an unsecured interrupt occurs during the execution of secure code. In this case, the processor automatically places all secure information into a secure stack and clears the contents of the registers [45].

Chapter 4

Cryptography

This work primarily deals with attacks that cause errors during code execution in the **Intel SGX** and **ARM TRUSTZONE** trusted execution environments. To prevent communication from being exposed, these environments most often encrypt it using **AES (Advanced Encryption Standard) and RSA (Rivest–Shamir–Adleman)** encryption algorithms**.** The purpose of this chapter is to explain how the RSA and AES encryption algorithms work. Only then can the next chapter explain what errors are triggered in these encryption algorithms and how to exploit these errors, as this is directly related to their functionality.

4.1 RSA encryption

RSA (Rivest–Shamir–Adleman) is a **public-key** cryptosystem (asymmetric encryption, Figure 4.1) that is often used for secure data transmission. In public-key encryption, the encryption key is public and **different** from the decryption key, which is secret (private). The RSA user creates and publishes a public key based on two large **prime numbers** together with an auxiliary value. The prime numbers are kept secret. Anyone can encrypt messages using the public key, but only someone who knows the prime numbers can decrypt them [48].

The security of RSA is based on the assumption that breaking down a large number into prime factors (factorization) is a very difficult task. From the number $n = pq$, it is therefore practically impossible to find the factors p and q in a reasonable amount of time, because there is no known factorization algorithm that

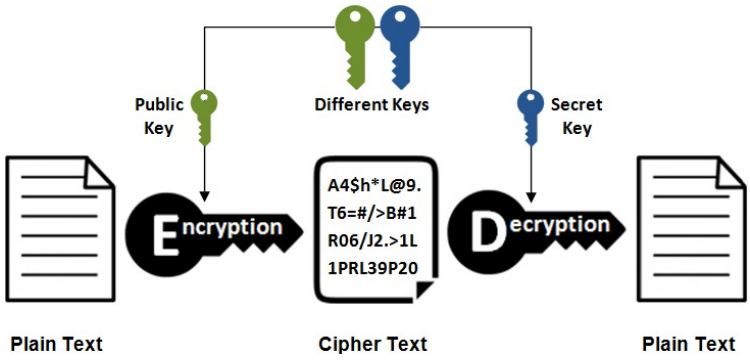

Fig. 4.1: Asymmetric encryption [21]

would work in polynomial time relative to the size of the binary representation of the number n . In contrast, multiplying two large numbers is an elementary task [3].

The RSA algorithm involves four steps: key generation, key distribution, encryption, and decryption.

4.1.1 Key generation

The key for RSA is generated as follows [36]:

1. Two different large random prime numbers are selected p , and q .
2. Their product is calculated $n = pq$.
3. The value **of the Euler function** $\phi(n)$ is calculated, which determines the number of all natural numbers such that $1 \leq k \leq n$ and their greatest common divisor $NSD(k,n) = 1$. From this, it is clear that for the prime number p , $\phi(p) = p - 1$. To calculate the value of the Euler function for the general argument n , we use multiplicativity, which can be used for two indivisible numbers. Since the prime numbers p and q are indivisible, the Euler function is calculated as:

$$\phi(n) = \phi(pq) = \phi(p) * \phi(q) = (p-1) * (q-1)$$

4. An integer e (public exponent) is chosen that is smaller than $\phi(n)$ and is not divisible by $\phi(n)$.
5. d (private exponent) is chosen so that

$$e * d \mod \phi(n) = 1$$

The pair of numbers (n,d) becomes the private key, and the pair (n,e) becomes the public key.

4.1.2 Key distribution

Let's assume that Bob wants to send information to Alice. If they decide to use RSA, Bob must know Alice's public key to encrypt the message, and Alice must use her private key to decrypt the message.

In order for Bob to send encrypted messages, Alice sends her public key (n,e) to Bob via a reliable but not necessarily secret channel. Alice's private key d is never distributed.

4.1.3 Encryption and decryption

After calculating all the variables needed to generate the key, it is possible to encrypt and decrypt the message using the algorithm. This is, of course, due to the fact that a public key has been created, consisting of n and e . The formula for encrypting the message m is as follows:

$$c = m^e \mod n$$

The formula [7] is used to decrypt the encrypted text c :

$$m = c^d \mod n$$

4.1.4 Example of use

There are two actors in this scenario: Alice and Bob. Alice wants to send a message to Bob and wants to encrypt it using RSA encryption. Bob chooses two prime numbers, p = 101 and q = 113. Using these two numbers, the variable n is calculated as $n = p * q$ = 11413. The next step is to calculate the value of $\phi(n)$ as follows:

$$\phi(n) = (p-1) * (q-1) = 100 * 112 = 11200$$

Bob chooses an exponent, say e = 3. Normally, it would be wise to choose a larger exponent for security reasons. In this case, however, we will use a small number for simplicity. Using this exponent, Bob is able to generate a private key, so that:

$$d = e^{-1} \bmod 11200 = 6597$$

Bob publishes the public key variables, n and e, which can be used to encrypt messages in plain text. The key generation process is complete and the message transmission process continues.

The next step is encryption. Let's assume that Alice wants to send a message with a single character "U". To convert the character to an integer, we use the ASCII table, where the character "U" has a value of 85. Alice wants to send encrypted text, so she calculates the value of c as:

$$c = 85^{(3)} \bmod 11413 = 9236$$

She sends this value to Bob. For a larger number of characters, the value c is calculated for each character.

Bob obviously knows the value of the secret key (6597), which he uses to decrypt the message received from Alice as follows [7]:

$$9236^{6597} \bmod 11413 = 85 = U$$

4.1.5 Use of the Chinese Remainder Theorem

To increase efficiency, many libraries use the following optimization based on the Chinese remainder theorem for decryption.

Since the recipient knows the secret prime numbers p and q, they can calculate the following modular components [55]:

1. $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$
2. $C_p = C \bmod p$ and $C_q = C \bmod q$
3. $M_p = C_p^{d_p} \bmod p$ and $M_q = C_q^{d_q} \bmod q$

This reduces the calculation time since d_p, d_q < d and C_p, C_q < C. In fact, their size is approximately half, and therefore, ideally, we achieve approximately a 4-fold acceleration. The final report is then calculated as [55]:

$$M = [M_p (q^{(-)(1)} \bmod p) q + M_q (p^{-1} \bmod q) p] \bmod n$$

4.2 AES encryption

The AES algorithm is a specific implementation of the general Rijndael algorithm, named after its creators Joan Daemen and Vincent Rijmen [20].

The AES cipher is fast in both software and hardware (e.g., using AES-NI) and, unlike its predecessor DES, does not use a Feistel network. Instead, it uses **a** so-called **SP-network,** i.e., a substitution-permutation network. The SP-network determines the number of mathematical operations performed in block cipher algorithms [37].

The input and **output** of the AES algorithm consists of **a data block** with a length of 128 bits (16 bytes) representing plaintext. These 16 bytes are represented in **a 4x4 matrix**, which AES treats as a byte matrix. The length **of the encryption key** in the AES algorithm is selected from three possible values: 128, 192, and 256 bits. Other lengths of input, output, and encryption key are not permitted by the standard. The key size determines **the number of rounds** of the AES encryption algorithm. This means that a 128-bit key undergoes 10 rounds of encryption, a 192-bit key undergoes 12 rounds, and a 256-bit key undergoes 14 rounds [1].

4.2.1 Encryption

The AES encryption process is shown on the left side of Figure 4.2 and can be divided into three basic phases: **the initial phase**, **the rounding** phase, and the **final round** phase. Before the encryption process itself, the encryption key is expanded from its original size to the size required for the entire encryption process (all rounds). The initial phase of encryption consists of only one operation, "key addition," where the original encryption key is added to `the state` field (in this case, plaintext). In each round, the operations are performed sequentially, and of the four basic operations performed during the AES algorithm, only the "key addition" operations are parameterized by the input encryption key. All other operations are reversible during encryption and decryption and do not guarantee security, only confusion and diffusion (confusion and diffusion). In the final round, the "matrix multiplication" operation is omitted due to the easy inversion of the decryption process. The encryption process can be summarized in the following points [34]:

- Key expansion
- Initial phase
  - Adding an encryption key
- Rounds
  - Nonlinear byte substitution
  - Row rotation
  - Matrix multiplication
  - Adding the key for a given round
- Final round
  - Nonlinear byte substitution
  - Row rotation
  - Adding the key for the given round

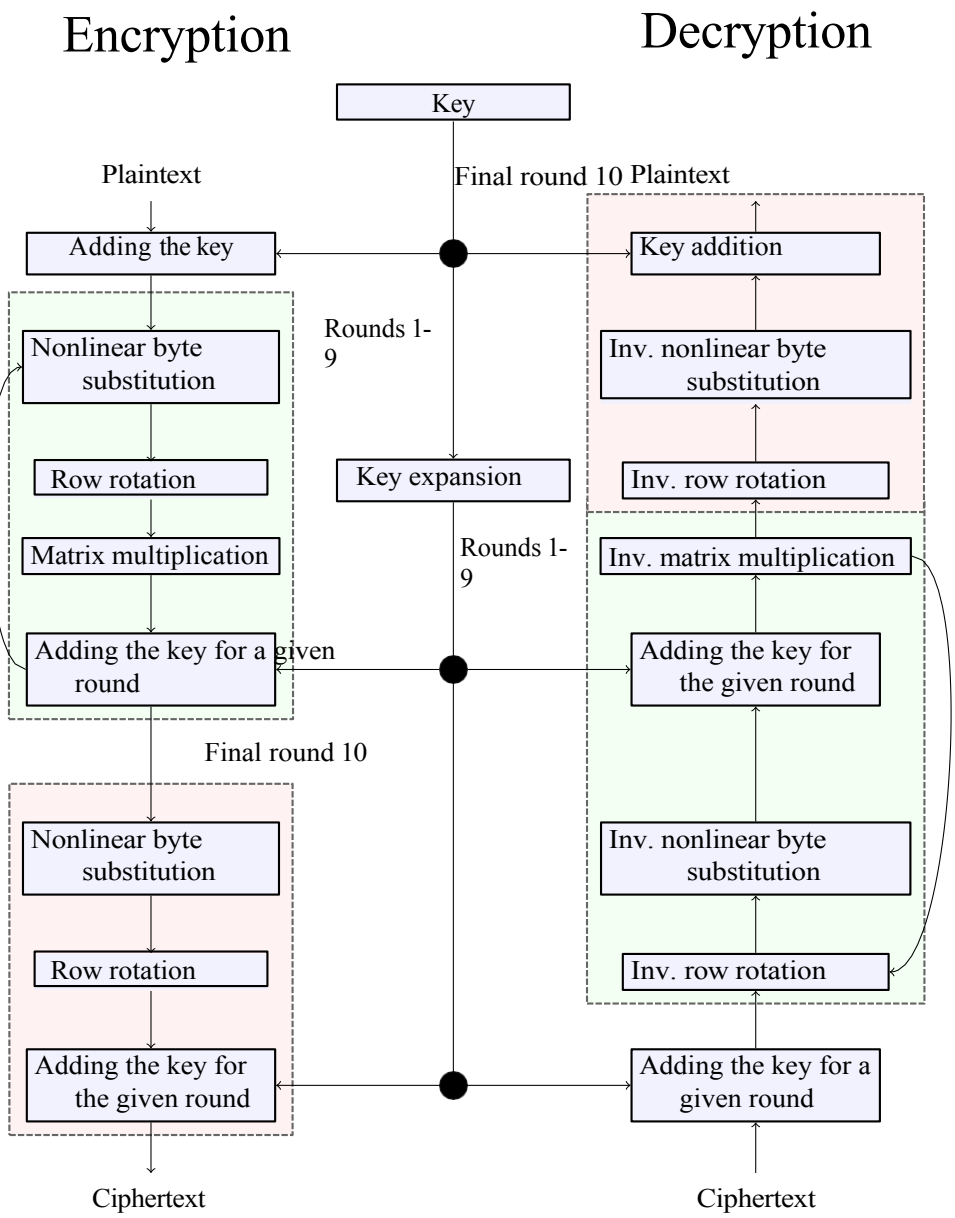

Fig. 4.2: Structure of encryption and decryption using the AES algorithm (128-bit key, 10 rounds)

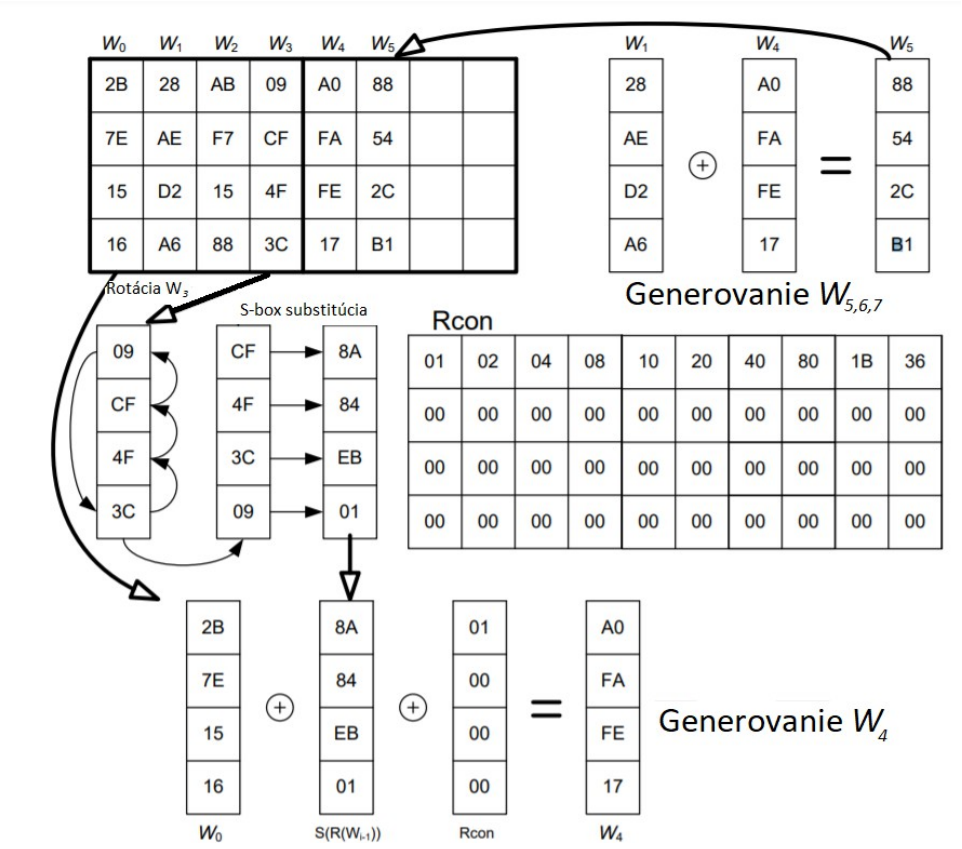

Fig. 4.3: Key expansion [34]

Key expansion

The AES algorithm is based on the expansion (extension) of the AES key for encrypting and decrypting data. Each round has a new key. The key expansion routine creates **subkeys** for each round word by word, where a word is an array of four bytes. The routine creates $4 * (N_r + 1)$ words. Where N_r is the total number of rounds [43]. The procedure for the AES-128 standard is shown in Figure 4.3 and is as follows. The encryption key (initial key) is used to create the first four words. The key size is 16 bytes. The first four bytes are represented as $w_{(0)}$, the next four bytes in the second column represent $w_{(1)}$, the following four bytes in $w_{(2)}$ and in the fourth column $w_{(3)}$ the last four bytes.

The figure shows that the keys w_5, w_6, and w_7, written in the form of four-byte words, are generated simply using the XOR operation. The key w_4 is created in a much more complex way using several functions. These functions are applied to the word $w_{(3)}$ of the encryption key. The functions performed are as follows [34]:

- Cyclic shift of the bytes of the word w_3 one position to the right.
- Substitution of the shifted bytes according to the corresponding table for S-box.
- The result is added to the word $w_{(0)}$ using the XOR operation.
- The result of the previous three steps is added using the logical XOR operation with the constant **Rcon**. This constant is defined for each round. Each value of the constant

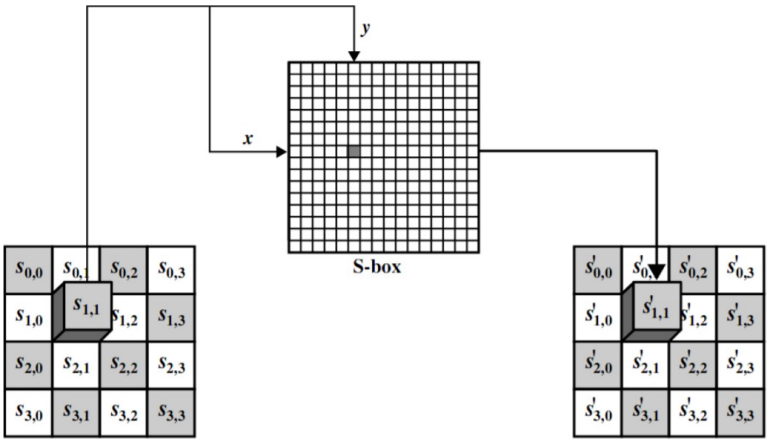

Fig. 4.4: Nonlinear byte substitution [1]

EA	04	65	85
83	45	5D	96
5C	33	98	B0
F0	2D	AD	C5

87	F2	4D	97
EC	6E	4C	90
4A	C3	46	E7
8C	D8	95	A6

Fig. 4.5: Real example of nonlinear byte substitution in the AES algorithm

Rcon is represented by four bytes. The three lowest bytes are always zero. The figure 4.3 shows the values of the **Rcon** constant for 10 rounds.

Figure 4.3 shows the expansion for the first round key. The other keys are expanded identically. This means that 4 words are always generated, where the first is generated in a more complex way involving a shift, substitution, the Rcon constant, and an XOR operation. Only the XOR operation is used in the other words.

Nonlinear byte substitution

The first phase of each round begins with a nonlinear byte substitution. This phase depends on **the** substitution **S-box table** (Table B.1), according to which each byte `of the state` is gradually replaced. For example, in AES, if we have hex 53 in `the state,` it must be replaced with hex ED. ED is created from the intersection of 5 and 3 [1].

Row rotation

The main idea of this step is to cyclically shift the bytes of the state to the left in each row except for the first row. The bytes in the first row remain unchanged. The second row is shifted circularly to the left by one byte. The third row is cyclically shifted by two bytes to the left. The last row is cyclically shifted three bytes to the left. The size of the new state does not change and remains the same as the original size of 16 bytes, but the position of the bytes has shifted, as shown in Figure 4.6.

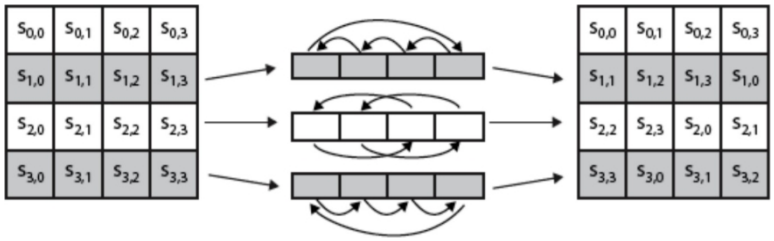

Fig. 4.6: Row rotation [1]

87	F2	4D	97
EC	6E	4C	90
4A	C3	46	E7
8C	D8	95	A6

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

Fig. 4.7: Real example of row rotation in the AES algorithm

Matrix multiplication

Another essential step is to mix the columns using matrix multiplication. Multiplication is performed in each state. Each byte of one row of the transformation matrix is multiplied by each value (byte) in the column of the `state` matrix. In other words, normal **multiplication** of **two** 4x4 **matrices** is used, with the difference that instead of adding the multiplied values, we use the **XOR** operation **[** 1]. The matrix representation of the calculation is as follows:

$$\begin{bmatrix} S'_{0,0} & S'_{0,1} & S'_{0,2} & S'_{0,3} \\ S'_{1,0} & S'_{1,1} & S'_{1,2} & S'_{1,3} \\ S'_{2,0} & S'_{2,1} & S'_{2,2} & S'_{2,3} \\ S'_{3,0} & S'_{3,1} & S'_{3,2} & S'_{3,3} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix}$$

It follows that, for example, the value *of* $S'_{0,0}$ is calculated as:

$$S'_{0,0} = (\{02\} \cdot S_{0,0}) \oplus (\{03\} \cdot S_{1,0}) \oplus S_{2,0} \oplus S_{3,0}$$

Adding the key for a given round

Adding the key for a given round is the **most important** phase in the AES algorithm. Both the key and the input data (also referred to as `the state`) are arranged in a 4x4 byte matrix. This step provides much greater security when encrypting data, as this operation is based on creating a relationship between the key and the ciphertext. The ciphertext comes from the previous phase. In this phase, a subkey is also used and combined with `the state.` The master key is used to derive the subkey in each round using the Rijndael key schedule. The size of the subkey and `the state` is the same. The entire key addition phase of a given round consists of combining each byte of the subkey with each byte `of the state` using the XOR bitwise operation [32].

For example the value of $S'_{0,0}$ in `the` newly created `state` S' will be calculated as:

$$S'_{0,0} = S_{0,0} \oplus K_{0,0}$$

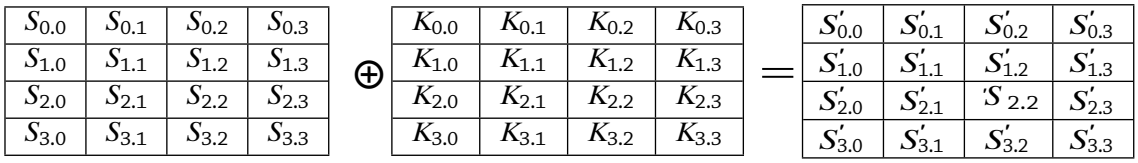

Fig. 4.8: Adding the key for a given round

4.2.2 Decryption

The decryption process is derived from encryption and is shown on the right half of Figure 4.2. The basic difference is the inversion of some operations and the change in the order of individual operations. The decryption key used is identical to the encryption key, but is read in reverse order. The inverse decryption scheme is as follows [34]:

- Key expansion
- Initial phase
  - Adding the encryption key
- Rounds
  - Inverse row rotation
  - Inverse nonlinear byte substitution
  - Adding the key for a given round
  - Inverse matrix multiplication
- Final round
  - Inverse row rotation
  - Inverse nonlinear byte substitution
  - Adding the key for the given round

The description of individual functions is essentially identical to the description of functions during encryption, with the difference that the operations work inversely.

Chapter 5

Differential error attacks

As described in Chapter 2, external noise such as electromagnetic radiation, power supply voltage fluctuations, or frequency fluctuations can cause errors in electronic devices. Attackers exploit these characteristics of electronic devices to deliberately introduce errors into devices running cryptographic algorithms. The attacker can then use analysis of the erroneous output to reveal **the secret key.** This type of attack is known as a differential error attack, which was originally introduced in the **Bellcore** attack [15]. Their attack is based on the algebraic properties of modular arithmetic and is therefore only applicable to public-key cryptosystems such as RSA.

E. Biham and A. Shamir [13] extended this attack to various secret key cryptosystems such as DES and called it **differential fault analysis** (DFA), which is based on a combination of differential cryptanalysis and fault analysis. In addition, they applied differential cryptanalysis to Data Encryption Standard (DES) encryption.

Differential error analysis has become the most commonly used method of error analysis in attacks on symmetric block ciphers. This method is the first choice for testing error resistance in new cryptographic algorithms because of its simplicity and ability to recover the secret key with a low number of erroneous ciphers [16].

A DFA attack consists of injecting an error into **the transition state** of the cipher, usually during one of the last rounds. The error then **spreads** further, resulting in the ciphertext being incorrect. The difference between the original and erroneous ciphertext is then analyzed, providing the attacker with information about the secret key. DFA exploits the properties of nonlinear operations commonly used in cryptosystems [16].

It has been shown that most block cryptosystems are vulnerable to DFA and there are currently no ciphers that can prevent this analysis. DFA has been used to successfully attack ciphers such as AES [2], DES [13], and PRESENT [9].

5.1 The Bellcore attack

The attack can be used for both classic RSA encryption, which involves a single modular exponentiation, and **RSA-CRT**, which uses the Chinese remainder theorem to increase efficiency (see section 4.1.5 for more details). In the first case, the attack requires several invalid signatures, while in the second case, only **one** invalid signature is sufficient. In this case, the RSA-CRT variant is described.

This attack is based on several assumptions [29]:

- The RSA implementation uses the Chinese remainder theorem

- The attacker can introduce an error either in s_p or in s_q
- The attacker can record an invalid signature
- The attacker knows the correct signature or the initial message m.

Let us assume that the final signature s is calculated using the Gaussian recombination method with private keys p, q:

$$s = CRT(s_p, s_q) = (s_p q\,(q^{(-)(1)} \mod p)) + (s_q p\,(p^{-1} \mod q)) \mod n$$

Let us assume that an error occurred in the calculations s_p and the incorrect signature s' is:

$$s' = CRT(s'_p, s_q) = (s'_q\,(q^{-1} \mod p)) + (s_q p\,(p^{-1} \mod q)) \mod n$$

The difference between the signatures can be calculated as:

$$\begin{aligned}\Delta &= s - s' \\ &= (s_p q\,(q^{-1} \mod p)) + (s_q p\,(p^{-1} \mod q)) - (s'_q\,(q^{-1} \mod p)) - (s_q p\,(p^{-1} \mod q)) \\ &= (s_p q\,(q^{-1} \mod p)) - (s'_q\,(q^{-1} \mod p)) \\ &= (s_p - s')_q\,(q^{-1} \mod p) \mod n\end{aligned}$$

Therefore, the greatest common divisor (GCD) between Δ and n is equal to q:

$$NSD(m, \Delta) = NSD(pq, (s_p - s')_q\,(q^{-1} \mod p)) = q$$

Arjen Lenstra noticed that an attack against CRT-RSA can be performed with an initial message m [33]. If an error occurs during the calculation of s_p, then for a valid signature s and an invalid signature s', the following relationship applies:

$$\begin{aligned}s^e &= (s')^e \mod q \\ s^e &\neq (s')^e \mod p\end{aligned}$$

where e is the public exponent

The difference $(s')^e - m$ can be obtained as:

$$\begin{aligned}(s')^e - m &= (s')^e q\,(q^{-1} \mod p) + (s')^e p\,(p^{-1} \mod q) - m \\ &= (s'_p)^e q\,(q^{-1} \mod p) + (s'_q)^e p\,(p^{-(1)} \mod q) - \\ &\quad - (s_p)^e q\,(q^{-1} \mod p) - (s_q)^e p\,(p^{-1} \mod q) \\ &= (s'_p)^e q\,(q^{-1} \mod p) - (s_p)^e q\,(q^{-1} \mod p) \\ &= ((s'_p)^e - (s_p)^e) q\,(q^{-1} \mod p)\end{aligned}$$

Then NSD for modulus n and the difference $(s')^e - m$ is equal to q:

$$NSD(n, (s')^e - m) = NSD(pq, ((s'_p)^e - (s_p)^e) q\,(q^{-1} \mod p)) = q$$

5.2 Differential error analysis used for AES encryption

This section defines the strategy for performing error analysis. It is assumed that the attacker will cause an error in the byte of input to **the eighth round**. It is also assumed that the error corresponds to the random error model described in section 2.3.3, where this byte becomes a random and unknown value. The described technique of differential error analysis on AES is taken from Tunstall et al. [53].

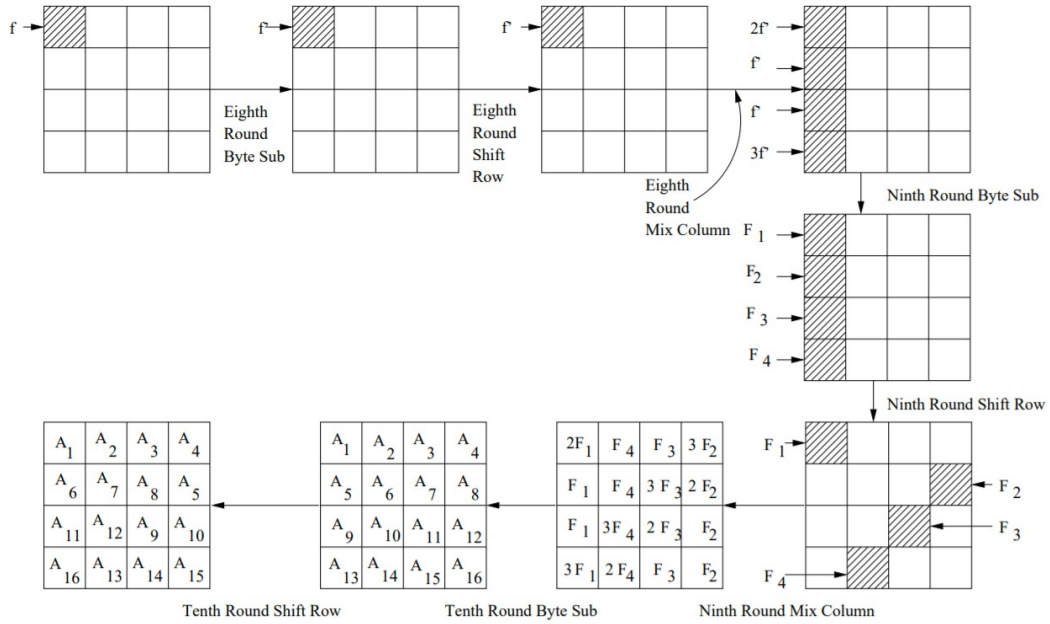

Fig. 5.1: Propagation of an error induced in the eighth round of the AES cipher

5.2.1 First step of the error-inducing attack

If an error is induced in a byte of the state matrix, which is then the input to the eighth round, the **matrix multiplication** operation (MixColumn) at the end of the round propagates this error to the entire state column. The **row rotation** operation (ShiftRow) at the beginning of the next round then shifts these bytes to occupy different columns. Another **matrix multiplication** operation then carries the error into the remaining twelve bytes.

This process is illustrated in Figure 5.1, which shows the propagation of the error in the byte read at the start of the eighth round. This is followed by the XOR difference between the state matrices of the two results, one error-free and the other erroneous. This is used as the basis for differential error analysis.

If the error is triggered at the start of the eighth round and the status of the differences after the **rotation of** the ninth round, then it is possible to obtain the following set of equations, which include values from the key bytes k_1 ,k_8 ,k_{11} and k_{14} ,thereby obtaining an expression for 32 bits $\mathbf{K_{10}}$.

$$2\delta_1 = S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1)$$
$$\delta_1 = S^{-1}(x_{14} \oplus k_{14}) \oplus S^{-1}(x'_{14} \oplus k_{14})$$
$$\delta_1 = S^{-1}(x_{11} \oplus k_{11}) \oplus S^{-1}(x'_{11} \oplus k_{11})$$
$$3\delta_1 = S^{-1}(x_8 \oplus k_8 \oplus S^{-1}(x'_8 \oplus k_8$$

Where $\delta_1, k_1, k_8, k_{11}$, and k_{14} are unknown values $\in \{0, \ldots, 255\}$

The above system of equations can be used to reduce the possibilities for those 32 bits of the key. The attacker selects a value for δ_1 and determines which values k_1 ,k_8 ,k_{11} and k_{14} satisfy the equations using four independent exhaustive searches. Each equation returns 0 , 2 ,

or 4 hypotheses. If any of the four equations cannot be satisfied, then any hypotheses for that value of δ_1 can be rejected.

The same technique can be used to recover information on the remaining bytes.
last subkey. This means that the information on the remaining key bytes can be derived using the following set of equations. To obtain information about k_2, k_5, k_{12} and k_{15}, an attacker can use:

$$3\delta_2 = S^{-1}(x_5 \oplus k_5) \oplus S^{-1}(x'_5 \oplus k_5)$$
$$_2\delta_2 = S^{-1}(x_2 \oplus k_2) \oplus S^{-1}(x'_2 \oplus k_2)$$
$$\delta_2 = S^{-1}(x_{15} \oplus k_{15}) \oplus S^{-1}(x'_{15} \oplus k_{15})$$
$$\delta_2 = S^{-1}(x_{12} \oplus k_{12}) \oplus S^{-1}(x'_{12} \oplus k_{12})$$

To obtain information about k_3, k_6, k_9 and $k_{(16)}$, an attacker can use the following equations:

$$\delta_3 = S^{-1}(x_9 \oplus k_9) \oplus S^{-1}_9(x' \oplus k_9)\ 3\delta_3 = S^{(-)(1)}(x_6 \oplus k_6) \oplus S^{(-)(1)}_6(x' \oplus k_6)$$
$$_2\delta_3 = S^{-1}(x_3 \oplus k_3) \oplus S^{-1}(x'_3 \oplus k_3)$$
$$\delta_3 = S^{-1}(x_{16} \oplus k_{16}) \oplus S^{-1}(x'_{16} \oplus k_{16})$$

Finally, to obtain information about k_4, k_7, k_{10} and $k_{(13)}$, an attacker can use the following equations:

$$\delta_4 = S^{-1}(x_{13} \oplus k_{13}) \oplus S^{-1}(x'_{13} \oplus k_{13})$$
$$\delta_4 = S^{-1}(x_{10} \oplus k_{10}) \oplus S^{-1}(x'_{10} \oplus k_{10})$$
$$_3\delta_4 = S^{-1}(x_7 \oplus k_7) \oplus S^{-1}(x'_7 \oplus k_7)$$
$$_2\delta_4 = S^{-1}(x_4 \oplus k_4) \oplus S^{-1}(x'_4 \oplus k_4)$$

It is worth noting that the equations have the same structure and therefore have similar solutions. Evaluating each set of equations is expected to return 2^8 unique hypotheses for the corresponding key bytes. Therefore, the attacker expects to have $2^{(32)}$ key hypotheses of the secret key used.

5.2.2 Analysis of the first step of the attack causing errors

The first step of the attack uses four sets of equations to reduce the AES key space. This section determines the expected number of key hypotheses that the attacker will have in each phase of the attack.

In order to analyze the number of valid hypotheses in the first phase of the attack, the first set of equations given in Section 5.2.1 is analyzed, where $\delta_1 \in \{1, \ldots, 255\}$. If $\delta_{(1)}$ is equal to zero, then it could be said that the expected error was not injected. So if $\delta_{(1)}$ is zero, it means that $x_1 = x'_1$ and all 256 key hypotheses are possible. First, consider that the first equation is in this set:

$$2\delta_1 = S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1)$$

The values x_1 and x'_1 are known from the correct and incorrect ciphertexts. For a given value $2\delta_{(1)}$, there will be 0, 2, or 4 valid key hypotheses. The probability of a hypothesis for all $\delta_1 \in \{1, ..., 255\}$ is approximately one, and thus 256 key hypotheses arise when all possible values of $\delta_1 \in \{1, ..., 255\}$ are considered.

The same can be said for each of the four equations in the set above. However, for a given value of δ_1, each of the four equations is expected to provide approximately one hypothesis for the key byte. These values will provide one hypothesis for the quadruple of key bytes $\{k_1, k_8, k_{11}, k_{14}\}$. Given that the attacker will have to take into account all values in the set {1, ..., 255}, there are 256 possible values for the quadruplets $\{k_1, k_{(8)}, k_{(11)}, k_{(14)}\}$. After analyzing the four equations defined in section 5.2.1, the attacker expects $2^{(32)}$ key hypotheses.

5.2.3 The second step of the attack causing errors

To further reduce the key hypotheses, the relationship between the key from the ninth round and the key from the tenth round is used.

We consider the AES key scheduling algorithm, the ninth round key, **K_9**, and the generation of the tenth round key, **K_{10}**. The key schedule is invertible, and **K_9** can be expressed in terms of the elements of **K_{10}**. The value **of K_9** can be expressed as:

$$\begin{bmatrix} k_1 \oplus S(k_{14} \oplus k_{10}) \oplus h_{10} & k_5 \oplus k_1 & k_9 \oplus k_5 & k_{13} \oplus k_9 \\ k_2 \oplus S(k_{15} \oplus k_{(11)}) & k_6 \oplus k_2 & k_{10} \oplus k_6 & k_{14} \oplus k_{10} \\ k_3 \oplus S(k_{16} \oplus k_{12}) & k_7 \oplus k_3 & k_{11} \oplus k_7 & k_{15} \oplus k_{11} \\ k_4 \oplus S(k_{13} \oplus k_9) & k_8 \oplus k_4 & k_{12} \oplus k_8 & k_{16} \oplus k_{12} \end{bmatrix}$$

We can see that the erroneous values in the first column of the state matrix at the output of the eighth round of **matrix multiplication** are $(2f', f', f', 3f')$, where f' is any non-zero value in $F(2)_8$. Using the **inverse matrix multiplication** operation (InverseMixColumn) and the mutual relationships between the erroneous values, the following equation can be defined:

$$\begin{aligned} 2f' = & S^{(-)(1)}(14(S^{(-)(1)}(x_1 \oplus k_{1)} \oplus k') \oplus 11(S^{(-)(1)}(x_{14} \oplus k_{14)} \oplus k') \oplus \\ & 13(S^{-1}(x_{11} \oplus k_{11}) \oplus k') \oplus 9(S^{-1}(x_{8} \oplus k_{8}) \oplus k')) \oplus \\ & S^{-1}(14(S^{-1}(x'_{1} \oplus k_1) \oplus k') \oplus 11(S^{-1}(x'_{14} \oplus k_{14}) \oplus k') \oplus \\ & 13(S^{-1}(x'_{11} \oplus k_{11}) \oplus k') \oplus 9(S^{-1}(x'_{8} \oplus k_8) \oplus k')) \\ = & S^{-1}(14(S^{-1}(x_1 \oplus k_1) \oplus ((k_1 \oplus S(k_{14} \oplus k_{10}) \oplus h_{10}))) \oplus \\ & 11(S^{-1}(x_{14} \oplus k_{14}) \oplus (k_2 \oplus S(k_{15} \oplus k_{11})))) \oplus \\ & 13(S^{-1}(x_{11} \oplus k_{11}) \oplus (k_3 \oplus S(k_{16} \oplus k_{12}))) \oplus \\ & 9(S^{-1}(x_{8} \oplus k_{8}) \oplus (k_4 \oplus S(k_{13} \oplus k_9)))) \oplus \\ & S^{-1}(14(S^{-1}(x'_{1} \oplus k_1) \oplus ((k_1 \oplus S(k_{14} \oplus k_{10}) \oplus h_{(10)}))) \oplus \\ & 11(S^{-1}(x'_{14} \oplus k_{14}) \oplus (k_2 \oplus S(k_{15} \oplus k_{11}))) \oplus \\ & 13(S^{-1}(x'_{11} \oplus k_{11}) \oplus (k_3 \oplus S(k_{16} \oplus k_{12}))) \oplus \\ & 9(S^{-1}(x'_{8} \oplus k_{8}) \oplus (k_4 \oplus S(k_{13} \oplus k_9)))) \end{aligned}$$

Similarly, the following equations can be defined:

$$f(\cdot) = S^{(-)(1)}(9(S^{(-)(1)}(x_{13} \oplus k_{13}) \oplus (k_{(4)} \oplus k_{(9)}) \oplus 14(S^{(1)}(x_{10} \oplus k_{10}) \oplus (k_{10} \oplus k_{14})) \oplus$$
$$11(S^{-1}(x_7 \oplus k_7) \oplus (k_{15} \oplus k_{11}) \oplus 13(S^{(-)(1)}(x_4 \oplus k_4) \oplus (k_{16} \oplus k_{12})) \oplus$$
$$S^{-1}(9(S^{-1}(x'_{13} \oplus k_{13}) \oplus (k_4 \oplus k_9) \oplus 14(S^{(-)(1)}(x'_{10} \oplus k_{10}) \oplus (k_{10} \oplus k_{14})) \oplus$$
$$11(S^{-1}(x'_7 \oplus k_7) \oplus (k_{15} \oplus k_{11}) \oplus 13(S^{-1}(x'_4 \oplus k_4) \oplus (k_{16} \oplus k_{12}))$$

$$f(\cdot) = S^{(-)(1)}(13(S^{(-)(1)}(x_9 \oplus k_9) \oplus (k_9 \oplus k_5) \oplus 9(S^{(-)(1)}(x_6 \oplus k_6) \oplus (k_{10} \oplus k_6)) \oplus$$
$$14(S^{-1}(x_3 \oplus k_3) \oplus (k_{11} \oplus k_7) \oplus 11(S^{(-)(1)}(x_{16} \oplus k_{16}) \oplus (k_{12} \oplus k_8))) \oplus$$
$$S^{-1}(13(S^{-1}(x'_9 \oplus k_9) \oplus (k_{(9)} \oplus k_{(5)})) \oplus 9(S^{(-)(1)}(x'_6 \oplus k_{(6)}) \oplus (k_{10} \oplus k_{(6)}))) \oplus$$
$$14(S^{-1}(x'_3 \oplus k_3) \oplus (k_{11} \oplus k_7) \oplus 11(S^{-1}(x'_{16} \oplus k_{16}) \oplus (k_{12} \oplus k_8)))$$

$$f'' = S^{-1}(11(S^{(-)(1)}(x_5 \oplus k_5) \oplus (k_5 \oplus k_1) \oplus 13(S^{(1)}(x_2 \oplus k_2) \oplus (k_6 \oplus k_2)) \oplus$$
$$9(S^{-1}(x_{15} \oplus k_{15}) \oplus (k_7 \oplus k_3)) \oplus 8(S^{(-)(1)}(x_{12} \oplus k_{12}) \oplus (k_8 \oplus k_4)) \oplus$$
$$S^{-1}(11(S^{-1}(x'_5 \oplus k_5) \oplus (k_5 \oplus k_1) \oplus 13(S^{(-)(1)}(x'_2 \oplus k_{(2)}) \oplus (k_{(6)} \oplus k_{(2)})) \oplus$$
$$9(S^{-1}(x'_{15} \oplus k_{15}) \oplus (k_7 \oplus k_{(3)})) \oplus 8(S^{-1}(x'_{12} \oplus k_{12}) \oplus (k_8 \oplus k_4)))$$

The second phase of the attack is linked to the first phase and is used to further reduce the number of key hypotheses.

5.2.4 Analysis of the second step of the attack causing errors

The expected number of hypotheses generated in the second step of the attack is based on a similar reasoning as the analysis of the first step described in section 5.2.2. Considering the second equation defined in section 5.2.3, it can be rewritten as:

$$f(\cdot) = A \oplus B$$

where the variables A and B are defined as:

$$A = S^{(-)(1)}(9(S^{(-)(1)}(x_{13} \oplus k_{13}) \oplus (k_{(4)} \oplus k_{(9)}) \oplus 14(S^{(-)(1)}(x_{10} \oplus k_{10}) \oplus (k_{10} \oplus k_{14})) \oplus$$
$$11(S^{-1}(x_7 \oplus k_7) \oplus (k_{15} \oplus k_{11}) \oplus 13(S^{(-)(1)}(x_4 \oplus k_4) \oplus (k_{16} \oplus k_{12}))$$
$$B = S^{(-)(1)}(9(S^{(-)(1)}(x'_{13} \oplus k_{13}) \oplus (k_{(4)} \oplus k_9) \oplus 14(S^{(-)(1)}(x'_{10} \oplus k_{10}) \oplus (k_{10} \oplus k_{14})) \oplus$$
$$11(S^{-1}(x'_7 \oplus k_7) \oplus (k_{15} \oplus k_{11}) \oplus 13(S^{-1}(x'_4 \oplus k_4) \oplus (k_{16} \oplus k_{12}))$$

A and B can be considered random values in F_{2^8}. For given values of f'' the difference between A and B will be equal to f'' with probability $\frac{1}{2^8}$. Using the same reasoning the probability that all four equations were valid is $(\frac{1}{2^8})^4 = \frac{1}{2^{32}}$.

It is necessary to consider all possible values of f'', i.e. {0, ..., 255}. The given key hypothesis will therefore be valid for any value of f'' with a probability of $2^8 * \frac{(1)}{2^{32}} = \frac{(1)}{2^{24}}$. In the first step of the attack, it is expected that the attack will return $2^{(32)}$ hypotheses, each of which still being considered at the end of the second step with probability $\frac{1}{2^{24}}$. It could therefore be expected that the second step of the attack will produce 2^8 possible key hypotheses.

5.2.5 Attack on other bytes

The previous sections describe an attack where differential error analysis is based on the knowledge that an error was triggered in the first byte of the state matrix. However, it should be noted that the analysis returns a very small number of hypotheses. Therefore, it is possible to perform 16 independent analyses, assuming that the error is triggered in all 16 bytes of the state at the beginning of the eighth round. The attacker can expect this to generate $2^{(4)} * 2^{(8)} = 2^{(12)}$ valid key hypotheses, which is still a trivial search.

Chapter 6

Theory of attacks exploiting voltage variation

This chapter describes the principle of the most well-known attacks that exploit voltage variation. In addition to dividing attacks into hardware and software (see Chapter 2), these attacks are divided according to **the** processor **architecture** they were used on, either Intel(x86) or ARM. The attacks are not portable for the following reasons [27]:

- The ARM architecture allows virtually **unlimited** core voltage and frequency settings. This means that an attacker can freely choose any combination of frequency and voltage, allowing them to use extremely dangerous voltage/frequency settings exclusively from software to carry out an attack. In contrast, the x86 architecture offers only a fixed, predefined list of P states that are tested to be safe for the manufacturer's normal operating conditions before being released to the market.
- The ARM architecture allows each core to function effectively in its **own** P state. On x86, all physical cores operate within the same P state, which means that the same voltage setting will apply to both the attacker and the victim core, and therefore errors cannot be easily limited to a given core as on the ARM architecture.
- Platforms based on x86 architecture use extensive **security measures** and implement **defenses** to detect, prevent, and recover from hardware errors during runtime. Such rescue mechanisms are virtually non-existent on ARM, which significantly increases the reliability of error introduction and possible exploitation scenarios.
- Since power management is one of the key factors on mobile devices (ARM), the relevant hardware mechanisms are extensively **documented** and tools are **readily available**. For the x86 architecture, **there is** virtually no official documentation on low-level power management, so even simple tests usually involve costly **reverse engineering** of microarchitectural elements that may differ on other generations of processors.

6.1 PlunderVolt

All information about the PlunderVolt attack is obtained from a study of this attack [40].

Modern CPUs are very well optimized to maximize performance and efficiency while maintaining proper functionality for specified operating conditions. In fact

Attack name	Attack type	Architecture
PlunderVolt	Software	Intel
VoltPillager	Hardware	Intel
TrustZone-M(eh)	Hardware	ARM

Table 6.1: Known attacks exploiting voltage changes and their basic characteristics

Most modern processors cannot run continuously at their maximum frequency because this would consume a large amount of energy, producing too much heat. Therefore, processors keep the clock frequency and supply voltage as low as possible, increasing them **dynamically** only when necessary. Higher frequencies require higher voltages for the processor to function properly, so they should not be changed independently. In addition, there are other types of power consumption that affect the appropriate choice of frequency-voltage pairs for specific situations.

Reducing the supply voltage was also important in the development of the latest DRAMs. The supply voltage is gradually reduced, resulting in less charge in the capacitors storing individual bits – this led to the well-known **Rowhammer** effect [28]. A large amount of research on this effect has resulted in a number of practical attacks, such as privilege escalation, introducing errors into cryptographic primitives, or reading inaccessible memory locations. Therefore, the scientific community and industry have made considerable efforts to eliminate the Rowhammer effect. This has reached the point where Intel ultimately considers main memory to be **untrustworthy storage and** fully **encrypts and** verifies all memory within the Intel SGX enclave [23]. In a Plundervolt attack, an attacker exploits privileged software and uses Intel's undocumented voltage scaling interface to compromise the integrity of computations within the Intel SGX enclave. Plundervolt carefully **monitors the** processor's **supply voltage** during enclave computation, causing predictable processor errors. As a result, even Intel SGX memory encryption/verification technology cannot protect against a Plundervolt attack.

/verification technology cannot protect against a Plundervolt attack.

6.1.1 Test settings

A standard Intel SGX attacker model is assumed, where the attacker has full control over all software running outside the enclave (including privileged system software such as the operating system and BIOS). The attacker's ability to read and write to **the MSR register,** for example through a malicious `ring 0` kernel module or an attack framework such as `SGXStep`, is crucial.

Voltage scaling on Intel processors

Reverse engineering has confirmed the existence of an undocumented MSR register used to adjust the operating voltage on Intel processors.

Figure 6.2 shows how a 64-bit value in **MSR 0x150** can be divided into an index and **a voltage offset**. By specifying a valid index, the system software can choose which CPU components should have the voltage change applied. The CPU core and cache **share** the same voltage, and the higher voltage will be applied to both the core and the cache. The desired voltage offset is encoded as an 11-bit signed integer relative to the core's base operating voltage. This value is expressed in units of $1/1024V$ (approximately $1mV$), allowing for a maximum voltage change of $1V$. After the software has successfully sent the voltage scaling request, it takes some time for

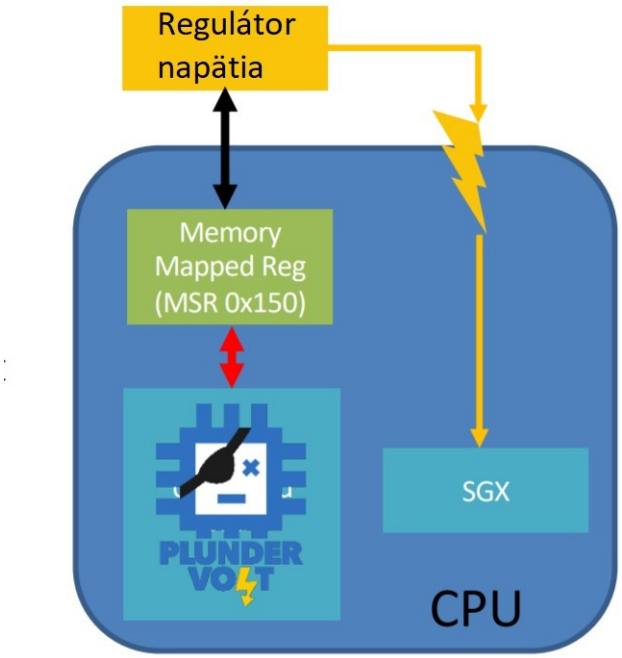

Fig. 6.1: Injecting a bug into an Intel processor with SGX support using a Plunder-Volt attack
Volt

until a physical voltage change occurs. The current operating voltage can be requested from the documented MSR 0x198 register.

Setting voltage and frequency

In order to reliably find **the frequency/voltage pair** causing the error, it is necessary to configure the CPU to run at **a fixed frequency**. The frequency can be set, for example, using the *cpupower* command. Undervolting is applied by writing to the MSR 0x150 register (for example, using the Linux kernel's msr module) just before the victim enters the enclave via **ECALL** through an untrusted host program. Upon returning from the enclave, the host program immediately resets the stable operating voltage. In addition to the msr kernel module, an attacker can also rely on more accurate undervolting methods, for example, if configuration latency needs to be minimized. Therefore, the SGX-Step enclave execution control framework has been extended with x86 interrupt and gateway call functionalities. This makes it possible to execute privileged instructions to read and write to the MSR register directly before the victim enters the enclave.

One of the challenges for a successful Plundervolt attack is setting the undervolting parameter so that the processor generates **incorrect results** for certain instructions while still allowing other instructions **to function correctly**. This means that excessive undervolting leads to system failure and **freezing**, while too little undervolting **does not produce** any errors. Therefore, it is necessary to find the correct undervolting value by carefully reducing the core voltage (for example, by 1 mV per step) until a failure occurs, but without crashing the system. In practice, PlunderVolt attack researchers have found that undervolting to

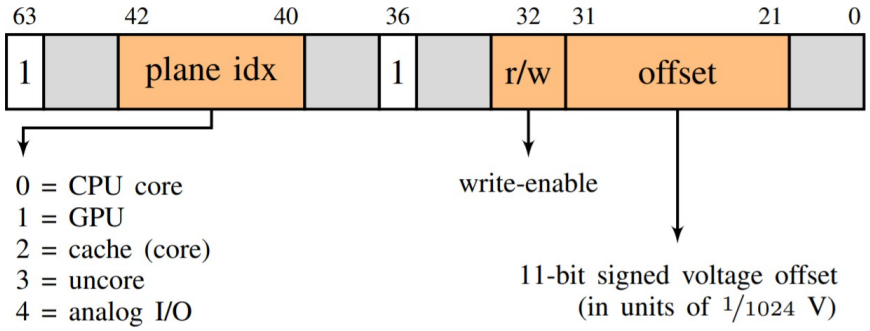

Fig. 6.2: Structure of the undocumented MSR register with address 0x150 [40]

short time intervals of $-100\,mV$ to $-260mV$, depending on the specific CPU, frequency, and temperature.

6.1.2 Causing a multiplication error in the enclave

The first step in practically injecting errors into SGX enclaves was to analyze x86 instructions in **isolation**. Although we were unable to trigger an error with simple arithmetic instructions (such as addition and subtraction) or logical instructions (such as shifts and OR/XOR/AND), we were able to trigger an error using **multiplication**. This may be because multipliers typically have a longer critical path compared to adders or other simple operations. Or it may be because multiplication is likely to be the most aggressively optimized due to its frequent use in source code.

Consider the following implementation, which performs simple multiplication (the code is compiled into symbolic instruction language with the *imul* instruction) in a loop inside the ECALL handler:

```
uint64_t multiplier = 0x1122334455667788;
uint64_t var = 0xdeadbeef * multiplier;
while(var == 0xdeadbeef * multiplier){
    var = 0xdeadbeef; var
    *= multiplier;
}
var ^= 0xdeadbeef * multiplier;
```

Listing 6.1: C code demonstrating the multiplication bug

It is obvious from the code that the program should **never terminate** because the loop condition is always true. However, experiments have shown that undervolting the CPU just before entering the enclave causes bit flipping (see 2.3.1) in *the* variable *var*, typically in byte 3 (counting from the least significant byte as byte 0). This allows the enclave program to terminate. The erroneous output is then XORed with the desired value to highlight only the erroneous bit(s). In this specific configuration, the output is always 0*x* 04000000.

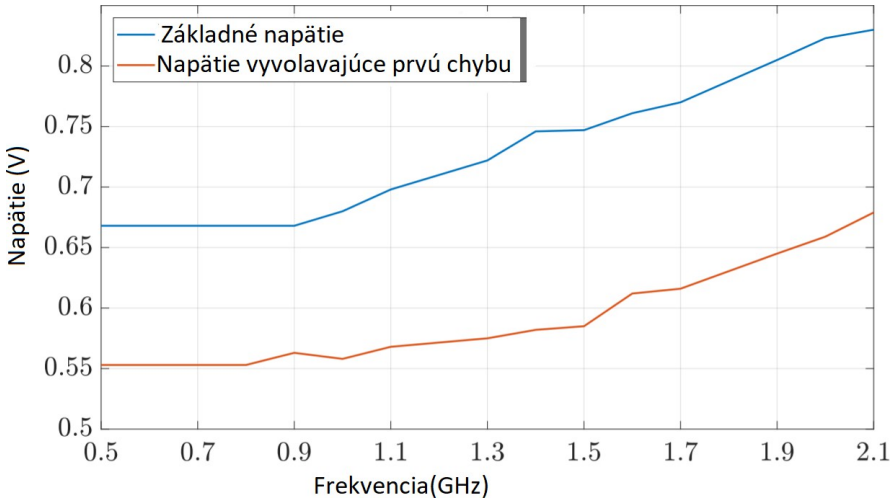

Fig. 6.3: Measured relationship between frequency, normal voltage (blue), and required undervoltage (orange) to achieve an erroneous multiplication result inside the SGX enclave for the i3-7100U-A processor [40]

Analysis of the undervoltage effect

Using the MSR 0x198 register, it is possible to determine the current voltage in normal operating mode and also to record it when calculating the incorrect result. The measurements in this register may not be absolutely accurate, but they accurately reflect the relative undervoltage.

After testing different values for multiplication operands, erroneous results can be divided into two categories:

- Flipping of one to five (consecutive) bits
- Flipping of all most significant bits

6.1.3 Extraction of keys from the enclave using error injection

After demonstrating the feasibility of injecting errors into the SGX enclave in section 6.1.2, undervolting techniques are applied to the cryptographic libraries used in enclaves.

Extracting keys from RSA-CRT decryption/signing in SGX using IPP Crypto

Intel SGX-SDK *Tcrypto* API exposes only a limited number of cryptographic primitives. However, developers can also call IPP Crypto functions directly when additional functions are needed. One function available through this API is **decryption** or **signature generation** using RSA with the commonly used CRT (Chinese Remainder Theorem) optimization. In IPP Crypto terminology, this is referred to as "type 2" keys initialized using the `ippsRSA_InitPrivateKeyType2()` function.

RSA-CRT private key operations (decryption and signing) are known to be being vulnerable to **Bellcore** and **Lenstra** attacks [15]. These attacks require an error in exactly one of the two powers of the RSA operation kernel, with no further requirements on the nature or location of the error.

The first step in practically implementing this attack for SGX was to inject a bug into the *ippsRSA_Decrypt()* function running within the SGX enclave throughout the entire duration of the RSA operation. However, this resulted in unusable errors, probably because both partial exponentiations were incorrect. Therefore, a second thread (in untrusted code) was introduced, which sets the voltage to a stable value after one third of the total ECALL duration. The errors obtained with this could be used to obtain a 2048-bit RSA modulus using Lenstra and Bellcore attacks. This means that it is possible to recover the entire key with a single incorrect decryption or signature and negligible computational effort. The exact procedure for recovering this key is described in 5.1.

Differential error analysis in AES-NI encryption in SGX

The Intel processor instruction set provides efficient hardware implementation for AES key scheduling and round computations. For example, on Skylake architecture, the AES instruction for round computation has a latency of only four clock cycles and a throughput of one cycle per instruction[1]. AES-NI is widely used in cryptographic libraries, including the SGX `Tcrypto` API, which exposes functions for AES in Galois mode (GCM), normal counter mode, and CMAC construction. These cryptographic primitives are then used within the Intel SGX-SDK, including critical operations such as sealing and unsealing enclave data.

Experiments have shown that *the* AES-NI encryption instruction for round computation is **vulnerable** to Plundervolt attacks. The errors recorded were always a single bit flip in the second byte from the left. A single bit flip is an ideal error for differential fault analysis (DFA). Example output:

```
[Enclave] plaintext: 697DBA24B0885D4E120FFCAB82DDEC25
[Enclave] round key: F8BD0C43844E4B4F28A6D3539F3A73E5
[Enclave] ciphertext1: C9210B59333A07A922DE59788D7AA1A7 [Enclave]
ciphertext2: C9230B59333A07A922DE59788D7AA1A7 [Enclave] plaintext:
4C96DD4E44B4278E6F49FCFC8FCFF5C9 [Enclave] round key:
BE7ED6DB9171EBBF9EA51569425D6DDE
[Enclave] ciphertext1: 0D42753C23026D11884385F373EAC66C [Enclave]
ciphertext2: 0D40753C23026D11884385F373EAC66
```

These single-cycle errors were then used to create a key recovery attack against the entire AES cipher. The canonical implementation of AES using AES-NI2 instructions was run in the enclave, with undervolting, of course. However, the probability that the error will affect a specific round instruction is approximately 1/10, which indicates a uniform distribution of probability across each of the ten AES rounds. By repeating the operation frequently (on average 5 times), an error was achieved in the 8th round. An example of the output (using key 0*x* 000102030405060708090*a* 0*b* 0*c* 0*d* 0*e* 0*f*) is as follows:

```
[Enclave] plaintext: 5ABB97CCFE5081A4598A90E1CEF1BC39 [Enclave]
CT1: DE49E9284A625F72DB87B4A559E814C4 &lt;- incorrect [Enclave]
CT2: BDFADCE3333976AD53BB1D718DFC4D5A &lt;- correct
Entry into round 10:
[Enclave] 1: CD58F457 A9F61565 2880132E 14C32401
```

[1] https://software.intel.com/sites/landingpage/IntrinsicsGuide/#expand=233&text=_mm_aesenc_si128

```
[Enclave] 2: AEEBC19C D0AD3CBA A0BCBAFA C0D77D9F
Entry into round 9:
[Enclave] 1: 6F6356F9 26F8071F 9D90C6B2 E6884534 [Enclave]
2: 6F6356C7 26F8D01F 9DF7C6B2 A4884534
Entry into round 8:
[Enclave] 1: 1C274B5B 2DFD8544 1D8AEAC0 643E70A1 [Enclave]
2: 1C274B5B 2DFD8544 1D8AEAC0 646670A
```

To understand what error occurred, both the correct and incorrect ciphertexts were taken and then decrypted round by round while comparing the intermediate states. The result can be seen in the output above. Note the byte marked in red in **round 8**. It changed from $0x$ 66 to $0x$ $3E$. This erroneous byte was caused by the XOR operation with $0x$ 02 (i.e., a single-bit flip). Based on the error in round 8, it was possible to use Tunstall's **differential error analysis** technique [53] and its implementation by Jovanovic([2]).

Using this attack, it is possible to recover the correct and erroneous ciphertext for a given pair, the entire 128-bit AES key with an average computational complexity of 2^3 2 + 256 ciphers. In practice, extracting the entire AES key from the enclave took only a few minutes, including error injection and key calculation.

6.1.4 Intel's response

Intel responded almost immediately after the attack was reported and **disabled** voltage changes via the MSR register with a BIOS update (CVE-2019-11157). This means that if your computer has a BIOS version from late 2019 or later, this attack is no longer possible.

6.2 VoltPillager

All information in this section is drawn from a study of this attack [18].

VoltPillager is **a hardware attack** targeting Intel x86 processors with SGX support, which uses low-cost equipment to inject messages onto the **SVID** (Serial Voltage Identification) bus between the CPU and the voltage regulator on the motherboard. This allows precise control of the CPU core voltage. With this equipment, an attack was successfully carried out to inject a bug that compromises **the confidentiality and integrity** of Intel SGX enclaves. VoltPillager also includes `proof-of-concept` attacks to **recover keys** against cryptographic algorithms running inside SGX. This attack is even more dangerous than software attacks against SGX (such as PlunderVolt) because it works even on patched systems with the latest countermeasures, where software attacks are no longer possible. Protection against VoltPillager attacks is not easy and may require rethinking the SGX model, where the cloud provider is untrustworthy and the attacker has physical access to the hardware.

However, VoltPillager requires obvious **hardware access** to the system's motherboard and a thorough connection to the voltage regulator, so even though the Plundervolt attack was limited by the need for root/admin access to the system, VoltPillager is much more limited.

The cost of all the equipment needed for the attack is approximately €30. The equipment consists of:

- Teensy 4.0 Development Board[3] , which controls the entire attack

[2] https://github.com/Daeinar/dfa-aes [3] https://www.pjrc.com/store/teensy40.html

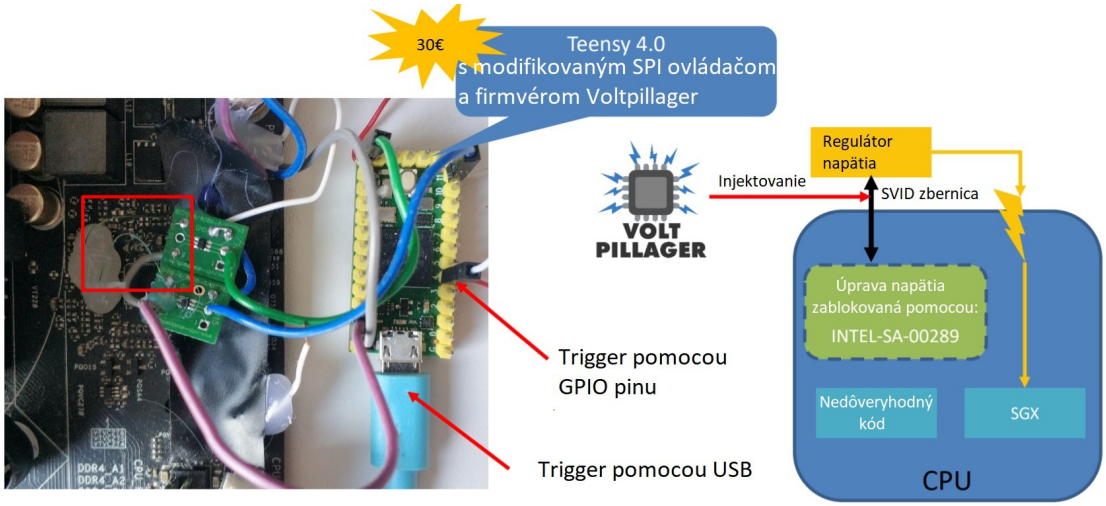

Fig. 6.4: Injecting a bug into an Intel processor with SGX support using the VoltPil-lager attack

- 2xSOT IC adapter[4]
- 2xNL17SZ07XV5T2G bus controller[5]

6.2.1 Interfaces for CPU voltage control

In modern computers, there are usually one or more voltage regulators connected to the CPU on the motherboard. They are used to control the power and energy consumption of the system by changing the core voltage (and other voltages) supplied to the CPU. When the CPU is running at lower frequencies or is idle, it sends commands to the voltage regulator to lower the voltage. Conversely, when the CPU is working under heavy load and/or at high frequency, it requests the voltage regulator to increase the voltage. Two VR interfaces were found on the motherboard that can be used to change the CPU voltage and thus carry out an undervolting attack:

- **SVID** interface
- **SMBus** interface

Although error injection via the SMBus is possible, the SVID bus was preferred for several reasons:

- higher SVID frequency, which allows for more accurate error injection
- SVID commands are the same for all tested systems, while on the SMBus they differed depending on the voltage regulator used
- SVID is used by all modern motherboards and CPUs, while SMBus is only used by specific motherboards and voltage regulators.

[4] https://www.farnell.com/datasheets/1934984.pdf [5] https://www.farnell.com/datasheets/2007350.pdf

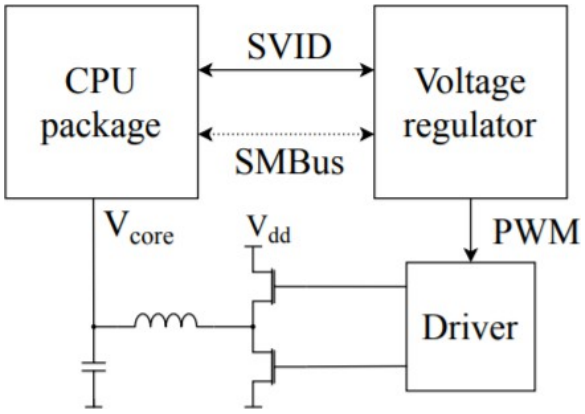

Fig. 6.5: Power supply architecture on x86 systems. The CPU is connected to the VR via SVID and SMBus (optional, dotted) to the VR. The VR uses pulse width modulation (PWM) to control the output to generate the required voltage for the processor

SVID (Serial Voltage Identification) bus

SVID is an interface used to send the voltage required by the processor to an external voltage regulator (VR). Intel does not provide any detailed documentation for this bus, but the CPU documentation shows that SVID uses 3 pins:

- **VCLK** - clock signal
- **VDIO** - data
- **ALERT#** - VR confirms that the voltage change has been completed

The VCLK and VDIO pins are used for bidirectional serial communication similar to common serial protocols such as I2C or SPI. SVID uses voltage levels of 0V (logic 0) and 1V (logic 1). The clock signal has a frequency of **25MHz**. Both the clock (VCLK) and data (VDIO) lines are connected via **pull-up resistors** to 1V and are actively controlled to logical 0 by the CPU or VR during data exchange. This allows multiple devices to be connected to SVID. This is used to connect the Voltpillager board for command injection.

After successfully identifying the clock and data lines on the motherboard, a DSlogic logic analyzer was connected to the SVID bus. By observing the bus, setting known values for voltage, and using a logic analyzer set to the SVID protocol, reverse engineering was used to construct the relevant commands used to configure the voltage output by the voltage regulator. Based on this, the command that configures the voltage and its response were analyzed (Figure 6.6).

6.2.2 Causing a multiplication error

To trigger a multiplication error, source code very similar to that used in the PlunderVolt attack (see Listing 6.1) was used: two multiplication operations (compiled into the *imul* instruction) are used, which are performed with the same input in a loop and in close proximity. The result of the calculation is compared after each operation. However, before entering the loop, a *trigger* signal is generated to start hardware undervolting using VoltPillager.

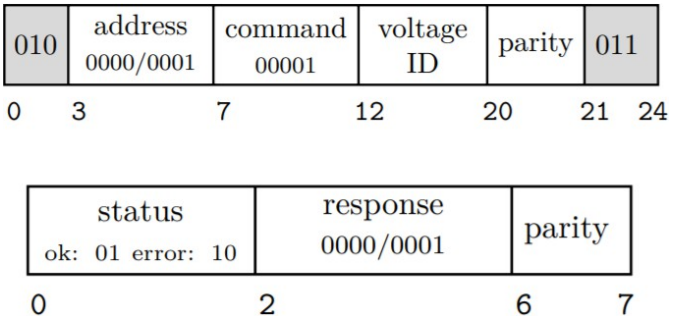

Fig. 6.6: Format of a 24-bit SVID command from the CPU to the VR for voltage setting (top) and a 7-bit response from the VR to the CPU (bottom)

```
TRIGGER_SET // Trigger signal setting
do {
    i++;
    correct_a = operand1 * operand2 ;
    correct_b = operand1 * operand2 ;
    if ( correct_a != correct_b ) {
        faulty = 1;
    }
} while ( faulty == 0 &amp;amp;&amp;amp; i &amp;lt; iterations ) ;
TRIGGER_RST // Reset trigger signal
```

Listing 6.2: C code used to demonstrate the multiplication error

When the multiplication operands were set to 0xAE0000 and 0x18, the same incorrect result was obtained on all tested CPUs (0xC500000 instead of 0x10500000).

6.2.3 Extracting the key from RSA-CRT decryption/signature in SGX

This attack is again based on the PlunderVolt *sgx_crt_rsa*[6]attack, which has been slightly modified to enable a hardware attack. This program calculates RSA signature/decryption inside the SGX enclave using standard *ipps* library functions. This attack was successfully executed, obtaining incorrect signatures, and it was also proven that these incorrect values can be used to obtain the private key using the **Lenstra** attack [15].

6.2.4 Comparison with software attacks

Compared to software attacks using the MSR 0x150 register, the Volt-Pillager hardware attack has several advantages. The first and most important is that the VoltPillager attack is still feasible even after the release of CVE-2019-11157 security measures, which limited software attacks. This means that even the latest Intel CPUs with the latest BIOS **cannot defend** against the VoltPillager attack.

The second advantage is **time accuracy**. The authors of the PlunderVolt attack state that at least 100,000 iterations are required to successfully execute the attack and trigger the multiplication error in the *imul* instruction. In addition, the error cannot be targeted at **a specific iteration of the cycle**. VoltPilla-ger can overcome both limitations. With the appropriate settings, it was always possible to trigger

[6] Available at: https://github.com/KitMurdock/plundervolt/tree/master/sgx_crt_rsa

an error with less than 1680 iterations. Suitable settings were also found for targeting a specific iteration, which resulted in the error occurring in 75% of cases in iterations 14,334 to 14,934.

In software attacks causing errors, **a large delay** was observed between writing to the MSR register and the actual voltage change. This limits the ability to generate short and potentially "deeper" interference pulses. In contrast, VoltPillager is limited in the width of the interference pulse only by the speed of the SR transition. Assuming that the error-inducing voltage is 200xml-ph-0000@deepl.internal
triggering the error is $200mV$ lower than the supply voltage and the typical$SR = 20mV/\mu s$, the minimum value of the disturbance pulse is therefore$T_{min} = 20\mu s$.

6.3 TrustZone-M(eh)

This attack is aimed at breaking the protection of the Trusted Execution Environment (TEE) TrustZone-M on ARMv8-M microcontrollers. The attack deals with several MCUs, and the attack is quite different for each MCU. This paper describes the attack on Microchip SAM-L11, as the attack was also carried out on this microcontroller. All information in this section is taken from the presentation([7]) of this attack.

The goal is to read secure data (from the secure world) from the insecure world, using **an instruction skip** error model. This achieves that potentially non-secure firmware is loaded as secure.

The pulse for triggering the error at the right moment is generated using **a Lattice iCEstick FPGA**. To ensure that the pulse has sufficient current and the correct voltage to power the microcontroller, **an N-channel MOSFET transistor** and **a voltage source** must also be used. The principle of fault injection is shown in Figure 6.7. Under normal circumstances, the correct supply voltage from the source is sent to the microcontroller via the transistor. When the FPGA sends a pulse to the transistor, the transistor opens, causing the supply voltage to be connected to ground. The width of this pulse determines how long the microcontroller will be without power. This means that both **the moment** of undervoltage and how **long** the MCU will be undervolted are controlled by the FPGA. The moment when undervoltage is to occur is usually selected by sending information to one of the FPGA inputs that the MCU has been reset, and the FPGA then sends a pulse to the transistor with **a** precisely defined **delay.**
transistor. This delay must be very **precise** (usuallyμs to tens ofns) in order to skip the correct instruction.

In addition to the aforementioned FPGA, any MCU with sufficient frequency to inject errors at the right time can also be used. For example, the presentation on this attack mentions that the attack was also successfully carried out using an Atmel ATTINY85 MCU, which costs approximately €3.

6.3.1 Connecting the attack hardware to the MCU

The ARMv8-M microcontrollers require a supply voltage of $3.3V$. However, the internal peripherals run on different voltages. **The CPU core**, which is most important for this attack, typically runs at $0.7\,V$ **to** $1.2V$. A regulator located directly on the chip is used to regulate the voltage. However, on a large number of chips, a capacitor outside the chip is connected between the voltage regulator and the CPU core. This capacitor is used to stabilize the voltage because the regulator has a noisy output. However, if this capacitor is removed,

[7] Available from: https://media.ccc.de/v/36c3-10859-trustzone-m_eh_breaking_armv8-m_s_security

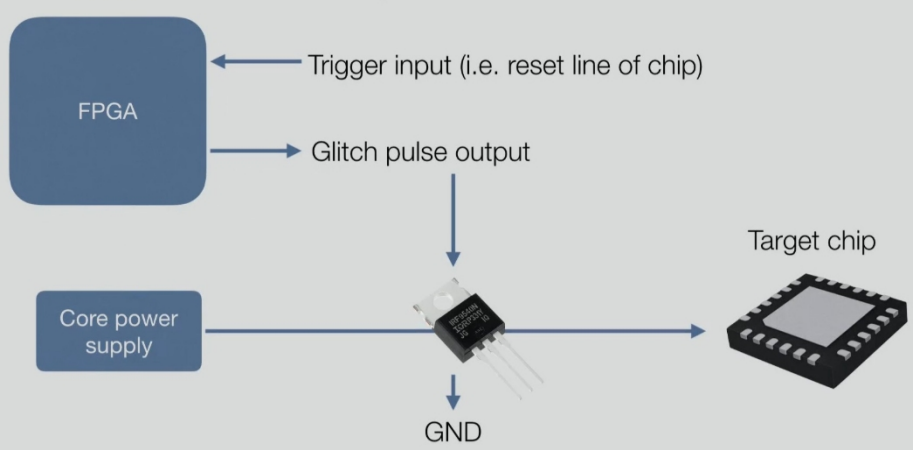

Fig. 6.7: Principle of MCU undervoltage

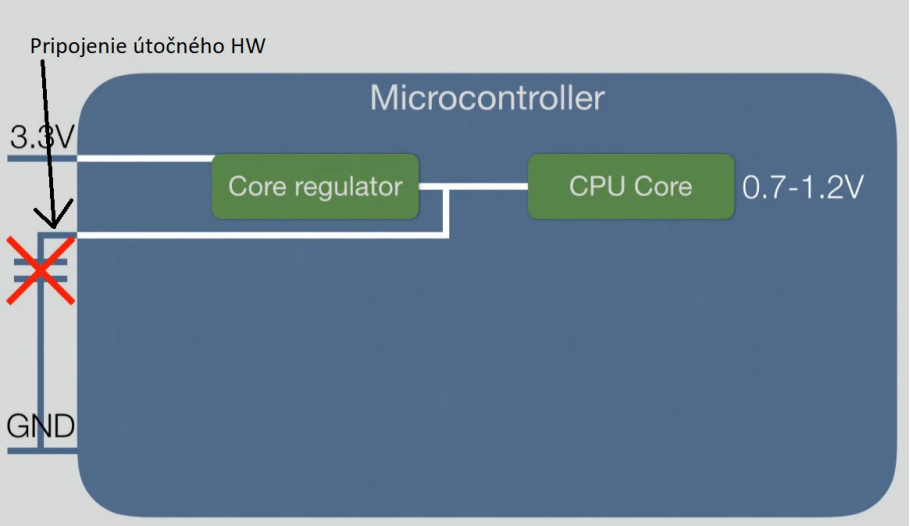

Fig. 6.8: Connecting the attack hardware to the MCU

we have direct access to changing the CPU core voltage. As shown in Figure 6.8, after removing the capacitor, we obtain a new pin to which the attack hardware is connected, which will control the CPU core voltage.

6.3.2 Attack on the SAM-L11 MCU

The flash memory in `SAM-L11` is divided into two parts: a section for the bootloader and a section for applications. As described in section 3.2.2, TEE TrustZone-M further divides each part into three additional parts: **secure**, **unsecured secure** (which can be called from unsecured), and **unsecured**. The size of each section is configured only using the `Implementation Defined Attribution Unit (IDAU)`, because this MCU does not have an SAU unit. The IDAU is located in the configuration flash memory and is configured during ROM boot. Figure 6.9 shows how individual IDAU registers can change individual areas of flash memory. The larger the value in the register, the larger the area. The goal is to set the value in **the AS register** to 0, because then the entire section

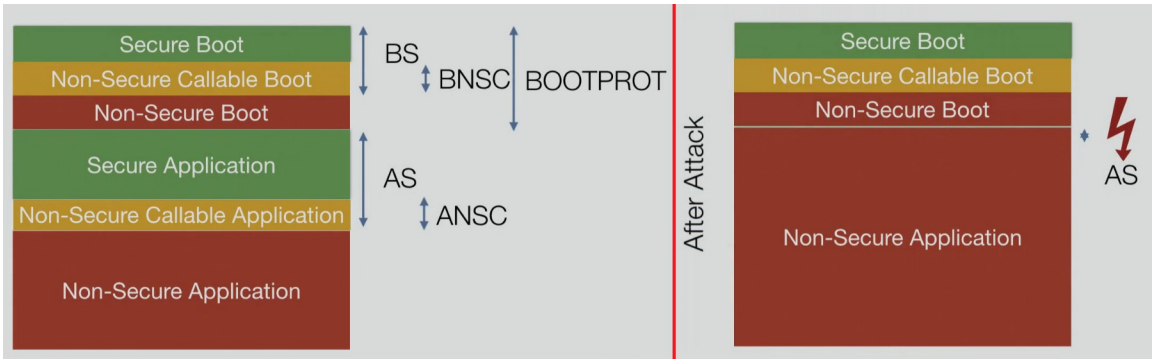

Fig. 6.9: Status of individual flash memory sections before the attack (left) and after the attack (right)

application memory is marked as unsecured, which means that memory that should have been secured can be read from the unsecured part.

Since the register values are initially set to 0, it is sufficient to trigger an error when reading the value of this register, which causes this instruction to be **skipped** and the value to remain zero.

Chapter 7

Attack replication protocol

As mentioned in Chapter 6, attacks are divided into hardware and software attacks. Hardware attacks are always more difficult to carry out and also require advanced knowledge of electrical engineering. Overall, they differ significantly from software attacks. Therefore, a separate procedure has been created for both types of attacks, which describes in detail how attacks are replicated in practice.

7.1 Replication of a SW attack

To replicate a software attack, you need to:

1. Obtain the necessary equipment for the attack
2. Find out how to change the voltage of the purchased CPU/MCU using software
3. Verify whether the MCU/CPU is vulnerable to attacks by changing the voltage
4. Activate the Trusted Execution Environment (TEE)
5. Trigger an error during code execution in the TEE and analyze the error

7.1.1 Equipment needed for the attack

In the case of software attacks, only devices vulnerable to attack are involved. This means that for attacks on ARM microcontrollers, all you need is **a microcontroller with TrustZone support** and a device that can be used **to program** the MCU. In most cases, it is possible to purchase an entire development board with an MCU, which already has everything you need. For attacks on smartphones with ARM architecture, **the smartphone** itself is sufficient**.**

For computers with Intel x86 architecture, **a complete PC** is required, where the most important component is the CPU, which must support **TEE Software Guard Extensions (SGX)**. In addition, **the motherboard** is also very important**,** as it cannot have a BIOS version with `CVE-2019-11157` or newer, which prohibited changing the CPU voltage using software. This means that newer CPUs cannot be used either, because even if they fit into the motherboard, the CPU requires a BIOS update on an older motherboard. The configuration of other components is irrelevant.

In addition, it is advisable to have a voltmeter or oscilloscope ready in case of problems with setting the voltage on the CPU/MCU.

7.1.2 SW voltage change

The easiest way to find out how voltage changes using software is to search for existing studies of software attacks on a given CPU/MCU, where this method is described. The **VoltJockey** attack [47] describes very well how to change the voltage on **smartphones** using software, and the **PlunderVolt** attack, which is also described in section 6.1, describes very well attacks on various series of **Intel** processors.

However, if such an attack does not yet exist, it is possible to try one of the attacks on the same architecture. If this method does not help either, it remains to find out how the CPU/MCU voltage changes using software from the technical documentation for the CPU/MCU, if at all possible.

7.1.3 Verification of susceptibility to voltage changes

To verify whether it is possible to cause an error by changing the voltage, it is common practice **to cause an error during multiplication**. The code for this was already described in section 6.1.2 in listing 6.1. This code will cause an error regardless of which error model was used, or if it is unknown at this point, it can be determined based on the error that was caused. This is because if the error is successfully triggered multiple times during the `var *= multiplier` operation, it is possible to determine from the result whether multiple bits **have been set/reset** or whether the bits **have been flipped**, whether it is possible to determine the location based on the size/time of the undervoltage, or whether the location of the error is **random.** Or, even if this instruction or the `var = 0xdeadbeef` instruction is **skipped**, it is clear from the result that a skip has occurred. In addition to verifying whether it is possible to trigger errors in the CPU, the code can also be used to find **the** best **parameters** for triggering an error. The following parameters are searched for:

- Undervolting amount
- How long should the processor be undervolted
- CPU frequency

7.1.4 TEE activation

To activate TEE on an MCU with TrustZone support, it is necessary **to select** whether the firmware is secure or not before loading. If unsecured firmware is used, **secure firmware** required for proper operation is also selected before loading, and the necessary information is obtained from it via the NSC (non-secure callable) world. To prevent the user from introducing errors into the secure world, some MCUs only support the loading of unsecured firmware. In this case, only predefined functions of the secure world can be called via the NSC world.

To create a trusted execution environment on smartphones with ARM architecture running the Android operating system, [an] isolated OS **Trusty TEE**[1]is used, which is **always active**, meaning that it does not need to be specially activated.

Processors with Intel x86 architecture require Intel SGX TEE to be enabled in **the BIOS settings**. Typically, there are three options to choose from: enabled, disabled, or software-controlled, which means that the software chooses whether TEE will be active or not. If this option is not available in the BIOS setup, the motherboard/processor probably does not support SGX, or

[1] https://source.android.com/security/trusty

it is software-controlled. In the case of software-controlled SGX, it is possible to use the `sgx-software-enable` application[2] to enable SGX on Linux operating systems.

7.1.5 Causing an error in TEE and analyzing it

The goal of software attacks is typically to trigger an error in the **RSA** or **AES** encryption used by TEE. The source code in which it is possible to trigger the desired error has a very similar concept to triggering multiplication, and the principle can be described as follows:

```
cipher_text_correct = encrypt(plaintex); cipher_text
= encrypt(plaintex); while(cipher_text ==
cipher_text_correct){
    undervolt(delay);
    cipher_text = encrypt(plaintex);
}
```

Listing 7.1: C code suitable for demonstrating the triggering of an encryption error

The `encrypt()` function encrypts plain text using RSA or AES within the TEE. When using the same encryption key every time, the cycle should never end unless an error is triggered. The `undervolt()` function serves as a trigger for the undervolting command with `a delay,` which must run on a different thread. The CPU frequency, undervolting value, and undervolting time can be used from the step in which the multiplication error was triggered.

The **Bellcore** attack described in section 5.1 is used to analyze the induced error in RSA. There are also freely available implementations, such as the Python implementation of the CRT[3]variant in the freely available source code for the PlunderVolt attack. **Differential error analysis** described in section 5.2 is used to analyze errors in AES encryption. A well-known freely available DFA implementation is from Jovanovic[4]To use this implementation, it is necessary to induce an error in **the 8th round.** In addition, it is useful to know in which byte the error was triggered, as this significantly speeds up the process of finding the encryption key. Therefore, when implementing the error trigger, it is advisable to write out the correctly encrypted text and the encrypted text with an error for each round of encryption.

7.2 Replication of HW attack

A hardware attack is similar to a software attack in terms of TEE activation. Although some other steps have the same or similar names, there are differences between them, which are described below. To replicate a hardware attack, you need to:

1. Obtain the necessary equipment for the attack
2. Assemble the attack hardware
3. Find out how the voltage of the acquired CPU/MCU changes in hardware

[2] Available from: https://github.com/intel/sgx-software-enable/tree/master
[3] Available from: https://github.com/KitMurdock/plundervolt/blob/master/sgx_crt_rsa/Evaluation/ eval.py
[4] Available from: https://github.com/Daeinar/dfa-aes

4. Create/use existing firmware and upload this firmware to change the CPU-/MCU voltage change to the attack hardware
5. Connect the attack hardware to the motherboard/MCU pin
6. Verify that the MCU/CPU is vulnerable to voltage change attacks
7. Activate the Trusted Execution Environment (TEE)
8. Trigger an error during code execution in TEE and analyze the error

7.2.1 Equipment needed for the attack

In addition to the equipment mentioned in software attacks, it is necessary to obtain attack hardware. Typically, this is **a** high-frequency **microcontroller** so that undervolting commands can be sent with **high precision**, which is crucial in voltage change attacks. Typically, the undervolting commands sent must have a different voltage than the microcontroller is capable of sending (for example, I/O MCUs operate at 3.3V and the CPU/MCU core voltage is 0.9V to 1.2V). Therefore, **a power supply with** the required voltage and an N-channel MOSFET **transistor** or **a** high-speed **multiplexer** or a component specialized for these purposes is required. In addition, it is advisable to have **an oscilloscope** or **logic analyzer with** a sufficiently high sampling frequency ready to help with troubleshooting.

7.2.2 Assembling the attack hardware

Unless it is specific hardware for a given attack that can be purchased as a single component, the attack hardware must also be assembled. **A wiring diagram**, which is usually available for the attack, is used for this purpose. However, if the diagram is not available, it must be derived from the attack description. In most cases, apart from the attack microcontroller, only commonly used components such as resistors, capacitors, or transistors are used, from which it is easy to create a diagram with basic knowledge of electrical engineering.

7.2.3 The principle of voltage change

Various studies, such as **VoltPillager**, a well-known attack on Intel CPUs described in section 6.2, are again best suited for this purpose. In this case, the voltage change consists of sending voltage regulator commands via the bus.

Attacks on various ARM MCU architectures are being investigated by, for example, the **chip.fail**[5] research group, where undervolting commands directly control the voltage of the MCU core. All of the studies reviewed used one of these two approaches to undervolt the CPU/MCU.

7.2.4 Creating and uploading firmware to change the voltage

Since two approaches are used to change the voltage, the attack firmware also has two forms. Creating firmware to send values to the bus is much more difficult because it first requires **reverse engineering** and **a logic analyzer** to determine how the command to change

[5] Available at: https://chip.fail/

voltage looks like. In most cases, it is also necessary **to detect the bus** on the motherboard using **an oscilloscope** or logic analyzer, because the technical documentation for the voltage regulator is not available. However, in the VoltPillager attack on Intel CPUs, for example, everything necessary has already been done, so it is sufficient to upload the existing firmware([6]) to the Teensy 4.0 attack microcontroller.

In the case of attacks on ARM architecture microcontrollers, undervolting can be achieved using simple commands to **set the digital output** to logical 0 and logical 1. A delay is inserted between these commands, the time of which determines how long the undervolting will last. An example of attack firmware is shown in Listing 7.2.

```
bool end = false;
while(end == false){
  for(unsigned int j = GLITCH_WIDTH_MIN; j &lt; GLITCH_WIDTH_MAX; j++){
    set_digital_pin(PIN_GLITCH, LOW);
    delay(j); set_digital_pin(PIN_GLITCH,
    HIGH); if (read_digital_pin(PIN_READ))
    {
      Serial.print("Glitch width: "); Serial.println(j);
      end = true;
      break;
    }
  }
}
```

Listing 7.2: Code created in the Arduino environment suitable for finding the ideal parameters for triggering an error

`GLITCH_WIDTH_MIN` and `GLITCH_WIDTH_MAX` determine the minimum/maximum undervoltage time, i.e., the pulse width. For vulnerable high-frequency MCUs, these time values are typically only a few nanoseconds. The attack will be successful if a logical 1 appears at the `PIN_READ` pin input. In this case, information about the pulse width will be written to the serial output.

7.2.5 Connecting the attack hardware to the motherboard/MCU pin

The bus to which the attack hardware is connected typically operates at **a different voltage** than the commands sent. Therefore, **a bus controller** is used for this purpose, which serves as a buffer but also reduces the voltage of the commands from the MCU to the value at which the bus operates. In the VoltPillager attack, the SN74LVC1G07DRLR([7]) is used for this purpose. If ready-made source code is used for voltage control, bus detection using **the voltage regulator documentation** or an oscilloscope/logic analyzer must be performed in this step.

When attacking an MCU, the technical documentation for that MCU is used to determine the `VDDCORE` pin to which the attack hardware is connected. When using a development board**,** it is necessary **to remove the capacitors** that would smooth out the temporary voltage fluctuation on the MCU core. The attack hardware also cannot be connected directly to the VDDCORE pin, as it operates at a different voltage. **A multiplexer** or **transistor** is used for this purpose**.** To the transistor-

[6] Available at: https://github.com/zt-chen/voltpillager/tree/master/voltpillager-firmware

[7] Technical documentation available from: https://www.ti.com/lit/ds/symlink/sn74lvc1g07.pdf?ts=1591037569523

A **power supply** is also connected to the multiplexer to replace the removed capacitors. It must be set to the core voltage, which is typically 0.9V to 1.2V.

7.2.6 Verification of susceptibility to voltage changes

The same code used to cause a multiplication error in a software attack is used on the vulnerable CPU/MCU. The attacking MCU is then used to find the undervolting value, pulse width, and delay before undervolting. Finding the correct delay can be significantly accelerated if the vulnerable CPU/MCU sends information about when the multiplication instruction begins. This is possible with Intel architecture CPUs via **the USB bus** or **RS232,** which has better response, meaning that RS232 also achieves better accuracy. **A digital I/O pin** can be used for attacks on microcontrollers**.**

7.2.7 Causing an error in TEE and its analysis

For attacks on RSA or AES encryption, the source code in the software error call from Listing 7.1 can be used. The only difference is in the `undervolt()` function, which in this case sends information to the attacking MCU before encryption begins. The analysis in this case also matches the software attack.

If the goal is to skip an instruction, which causes unsecured firmware to be loaded as secure, there are two options for triggering an error in the TEE. As described in section 6.3.2, the goal is to skip the instruction that sets the size of the secured world in the relevant register. The first option is **performance analysis.** When reading different register values, the consumption also changes. Therefore, different values are written to the register that determines the size of the secure world and the change in consumption is compared. Writing different values to the AS register is done by programming fuses. Since the rest of the code is the same, the moment when the consumption changes most significantly determines the exact time when the instruction to read the value of the given register occurs. However, if performance analysis cannot be performed, **the moment** when undervoltage is to occur must **be chosen randomly**, which can take a very long time.

Chapter 8

Implementation

Since a total of three attacks were performed, the implementation is divided into three parts. The first part describes the replication of the PlunderVolt software attack. The second part describes the replication of the VoltPillager hardware attack, and the last part describes the TrustZone-M(eh) hardware attack on the SAM-L11 microcontroller. All parts follow the protocol for replicating hardware or software attacks described in Chapter 7.

8.1 Replication of the PlunderVolt attack

Replication of the PlunderVolt attack consists of **five steps**, described in the software attack replication protocol. All the necessary information was found in the study itself, and even the source codes for triggering errors are freely available. This attack was carried out on two **different** computers with Intel processors. The only problem that arose during replication was an unsuitable operating system. This problem is described in more detail in section 8.1.3. Compiling, installing all the necessary dependencies, and running the freely available codes with the correct parameters was also not trivial, as it required studying the codes themselves, since there are no instructions for running them.

8.1.1 Required equipment

A PC with an Intel CPU supporting TEE is sufficient to replicate the PlunderVolt attack. The attack was replicated on two computers with Intel Core i5 6500 and i5 7400 processors. The complete specifications of the computers are shown in Table 8.1. Both PCs had a BIOS version on the motherboard that predated the `CVE-2019-11157` patch, which means that the CPU voltage should be changeable via software.

An ELMA BM257s multimeter was used to measure the voltage on the CPU.

Motherboard	ASRock Z170 EXTREME4	GIGABYTE B250M-Wind
Processor	Intel Core i5 6500	Intel Core i5 7400
RAM	2xKingston 4 GB DDR4 2133 MHz	Tigo 8 GB DDR4 2400 MHz
Disk	Seagate ST1000DX001	Colorful SL300 120 GB SSD
Power	SeaSonic SSP-350GT	Segotep Wasrship S7
OS	Microsoft Windows 7	Fedora 32

Table 8.1: Specifications of computers used for attacks

8.1.2 Voltage change

The voltage of Intel processors is changed using the MSR 0x150 register. A more detailed explanation of how to enter the desired value into this register is provided in section 6.1. Ready-made code from the PlunderVolt `undervolt.c`[1] attack was used to test the voltage change. All freely available codes are prepared for UNIX-type operating systems, so `Ubuntu 20.04` was added to the PC running `Windows 7`, on which all attacks were performed.

To read and write values to the MSR register, the command `sudo modprobe msr` command. Then, the code was simply compiled with the command:

```
gcc undervolt.c -Wall -lncurses -o test_voltage
```

and then run `test_voltage` with the voltage parameter. Whether the CPU voltage had changed was verified using the ready-made script `0x198_read_msr.sh`[2], which displays the current CPU voltage value. A multimeter was also used for additional verification of the CPU voltage. The CPU voltage can be measured on some capacitors near the CPU. The voltage value was gradually measured on all capacitors near the CPU, and the voltage closest to the value shown by the script was selected as the CPU voltage. On both PCs, when the voltage was changed using the `test_voltage` command, there was also a change in the selected capacitor. When the voltage was set too low, the PC froze, which is expected behavior.

8.1.3 Verification of susceptibility to voltage change

To verify susceptibility to voltage change by triggering a multiplication error, the ready-made code in C language operation.c[3] was used. It was sufficient to compile it with the `make` command, which creates the `undervolt` file. The PlunderVolt attack states that in order to successfully trigger a multiplication error, it is necessary to find a voltage-frequency pair. To set the CPU frequency to 1.1GHz, for example, and to see if the frequency has been successfully set, the following commands can be used:

```
sudo cpupower -c all frequency-set -u 1.1GHz
sudo cpupower -c all frequency-set -d 1.1GHz
grep "cpu MHz" /proc/cpuinfo
```

Unfortunately, the CPU frequency setting was not successful on either PC. It turned out that on newer UNIX OSs, setting the frequency to the desired value is **disabled**. Even after a long search, a functional solution that would work on the OS used could not be found. The solution may be to overclock the CPU to the desired frequency in the BIOS settings or to install an older OS. We chose to install an older OS, specifically **Ubuntu 18.04** was installed on both PCs, where there was no longer a problem with setting the frequency. Even in the PlunderVolt attack, researchers report that the attacks were successfully carried out on this OS, so there should be no further problems.

After using the command to access the MSR registry read and write, compiling the code, and setting the frequency, the `undervolt` file was run with the following parameters:

- Number of CPU threads: -t 4
- Number of iterations: -i 1000000

[1] Available from: https://github.com/KitMurdock/plundervolt/tree/master/utils
[2] Available from: https://github.com/KitMurdock/plundervolt/blob/master/utils/0x198_read_msr.sh
[3] Available at: https://github.com/KitMurdock/plundervolt/blob/master/faulting_multiplications/operation.c

- Multiplication operands: -1 0xFFFFFFFFFF -2 0xFFFFFFFFFF
- Setting operands as variable: -z max -x max
- Initial undervoltage size of 30mV: -s -30
- Final undervoltage value of 280mV: -e -280
- Number of undervoltage steps: -X 250
- Undervoltage step size set to 1mV: -v 1

These parameters were consulted with the creator of the attack and are most likely to cause an error. This setting uses **different** values for multiplication operands, where the maximum value is 0xFFFFFFFFFF, which significantly speeds up the search for suitable operands. To make the multiplication parameters specify a fixed value rather than the maximum, simply set the -z and -x parameters to `fixed.` The final undervolting value is set so that either an error is triggered at a given frequency or the PC freezes. All processor frequencies from 1.0GHz to their maximum were tested sequentially. On the i5 6500 CPU, errors were triggered at frequencies of 1.6GHz, 1.7GHz, and 2.0GHz, and on the i5 7400 CPU, they were triggered at frequencies of 2.4GHz, 2.6GHz, and 2.7GHz. The output after triggering an error looks like this.

```
    ------     CALCULATION ERROR DETECTED      ------
 > Iterations      : 00000260
 > Operand 1       : 0000000057e19499
 > Operand 2       : 00000000ff1a223b
 > Correct         : 5792abadb5439143
 > Result          : 5792abada5439143
 > xor result      : 0000000010000000
 > undervoltage: -240
 > Frequency       : 1700MHz
Done.
```

The attached example shows that 1 bit was flipped, which occurred most frequently. This is exactly the type of error that needs to be induced in RSA or AES encryption.

Finding the optimal parameters

It was not difficult to find the ideal parameters for a PC with an i5 7400. It was a frequency **of 2.6GHz** and undervolting in the range of **215mV to 220mV**. These parameters achieved success in approximately **90% of cases** and later proved themselves in attacks on RSA and AES ciphers.

Finding the optimal parameters for the i5 6500 CPU was much more difficult, as the highest success rate **of 50%** was achieved at a frequency of **2.0GHz** with undervolting in the range of **232mV to 238mV**. In addition, the PC often froze, after which the only option was to manually reset the computer. Later, it turned out that these parameters were not suitable for triggering an error in AES encryption. Therefore, comprehensive testing of all frequencies was performed. However, manually restarting the PC and always running the script manually would take several days.

A custom script, **test_PV_VP.sh**, was created to store all necessary information in files. It stores information about success at a given frequency and any errors that occur. In addition, it automatically moves to the next frequency after 51 attempts. For success

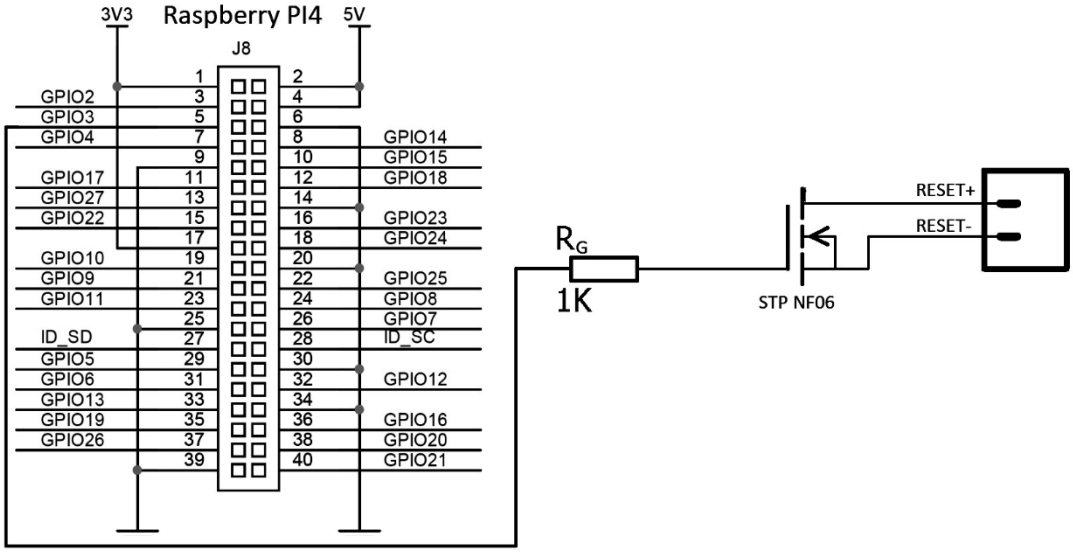

Fig. 8.1: Diagram of the connection of the Raspberry PI4 to a computer to control the restart

is considered an error if it causes a crash, and a failure if it causes the PC to freeze. The `crontab -e` command was used to automatically run the script at OS startup, which opens a file where you just need to add the path to the script to be run. However, it would still be necessary to restart the computer manually. This problem was solved using **a Raspberry Pi 4.0 and** an N-channel MOSFET **transistor**, which was connected to the PC instead of the restart button. The PC connection diagram is shown in Figure 8.1. The value of the resistorR_G is also very important, as it must be at least 206.25Ω, otherwise the RaspberryPi could be damaged. This value is derived from Ohm's law. Since the output voltage of the Raspberry Pi is $3.3V$ and the maximum current is $16mA$[4], the minimum value of the resistorR_G will be:

$$R_G = \frac{U}{I} = \frac{3.3}{0.016} = 206.25\Omega$$

A program in Python **called reset_PC.py** was then created for RaspberryPi, which checked every 10 seconds whether the PC had frozen using the `ping` command. If no response was received, a logical 1 was sent to the GPIO3 output pin, which opened the transistor and restarted the PC. Then, it was sufficient to leave the PC on for a few days until all attempts to trigger an error had been made. The test data can be found in the `data` folder. The file named 10 indicates the measured data at a frequency of 1.0GHz, 11 at a frequency of 1.1GHz, etc. The results are very surprising, as up to **100% efficiency** was achieved at several frequencies**.** Manual testing achieved a maximum success rate of 50% at the same frequencies. This may be due to the automatic execution of the script when the PC starts up.

Automatic testing could not be performed on the second PC because it was only a borrowed PC without the possibility of hardware intervention. To verify whether there would be an increase in success, at least the automatic execution of the script was set up at PC startup at frequencies that had previously successfully triggered an error. In this case, however, there was **a** slight **deterioration.**

[4] RaspberryPi specifications obtained from: https://www.tomshardware.com/reviews/raspberry-pi-gpio-pinout,6122.html

CPU	Frequency (GHz)	Undervolting	Success rate	Test type
i5 7400	2.6	from 215mV to 220mV	90%/8%/2	Manual
i5 6500	2.0	from 236mV to 240mV	50%/15%/35%	Manual
i5 6500	1.8, 2.0, 2.1, 2.3 2.4, 2.5, 2.8	from 30mV to 280mV	100%/0%/0	Automatic

Table 8.2: Undervolting parameters and success rate of inducing multiplication errors

Detailed information on the success rate of the best parameters is shown in Table 8.2, where the success column shows three numbers representing successfully induced multiplication error/no effect/PC freeze.

8.1.4 TEE activation

On both PCs, it was possible to activate Intel SGX TEE in the BIOS settings. Before performing an attack on RSA and AES encryption using freely available source codes from the PlunderVolt attack, the `sgx-step`[5]library must be downloaded and installed. Commands

```
make clean load
source /opt/intel/sgxsdk/environment
```

must be performed in the `kernel` folder after each computer restart.

8.1.5 Causing an error in TEE

The goal of the PlunderVolt attack is to trigger an error in RSA and AES encryption running within the enclave of the trusted execution environment.

First, the RSA decryption attack was replicated. Freely available code[6]attempts to trigger an error in RSA decryption in the RSA variant that uses Chinese remainder optimization.
trigger an error in RSA decryption in the RSA variant that uses Chinese remainder optimization. Decryption within the enclave is based on an example from Intel[7] , from which the encryption parameters were also used, meaning that the prime numbers `p` and `q` are also known. This means that it is sufficient to perform the Bellcore/Lenstra attack itself, which, if successful, will obtain one o f these numbers, from which it is trivial to derive the original unencrypted text.

On a CPU i5 7400, using the best parameters obtained from the multiplication error, the error was successfully triggered **in 40% of cases**. The CPU i5 6500 was slightly less successful, and in manual testing, the best results were achieved at a frequency of **2.0GHz**, with **a success rate of 25**%. However, this only refers to triggering the error itself, not its usability.

The `eval.py`[8]script was used to derive the key from the triggered error, which contains the implementation of both the Bellcore and Lenstra attacks. It was sufficient to enter the incorrectly and correctly decrypted text into the script for Bellcore analysis, or the incorrectly decrypted text and correctly encrypted text for Lenstra analysis. On both CPUs, the key was successfully obtained from incorrect decryption in less than **40% of cases.** This means that the overall success rate is **even lower.** On a PC with an i5 6500 CPU, an automatic test of all frequencies was also performed using the script

[5] Available from: https://github.com/jovanbulck/sgx-step

[6] Available from: https://github.com/KitMurdock/plundervolt/tree/master/sgx_crt_rsa

[7] Available from: https://www.intel.com/content/www/us/en/develop/documentation/ipp-crypto-reference/top/public-key-cryptography-functions/rsa-algorithm-functions/rsa-primitives/ example-of-using-rsa-primitive-functions.html

[8] Available at: https://github.com/KitMurdock/plundervolt/blob/master/sgx_crt_rsa/Evaluation/ eval.py

CPU	F(GHz)	Undervolting	Success rate	Test type
i5 7400	2.6	from 215mV to 220mV	45%/45%/10%/15	Manual
i5 6500	2.0	from 236mV to 240mV	25%/25%/50%/10	Manual
i5 6500	1.4	from 30mV to 280mV	43%/0%/57%/39%	Automatic

Table 8.3: Undervolting parameters and success rate of triggering an error in RSA algorithm decryption

test_PV_VP.sh, which was extended to include comprehensive RSA testing. This yielded very surprising results, as the most successful frequency was **1.4GHz**, which was one of the less successful frequencies in terms of multiplication errors.

More detailed statistics are described in Table 8.3, where F indicates the CPU frequency and, compared to Table 8.2, the success rate is supplemented by a fourth value, which indicates **the overall success rate**. This means how often it was possible to trigger an error that could be exploited to obtain the RSA decryption key. So if the success rate of triggering any error is 40%, from which it was possible to obtain the decryption key with a 50% success rate, the overall success rate will be 20%. To calculate the overall success rate**,** a Python script called **success_rate.py** was created**,** which lists the overall
success rate for each file located in the specified folder. The script also includes the implementation of the Bellcore attack, which is taken from the aforementioned `eval.py` script.

The attack on the AES[9]cipher was even less successful. In the case of the i5 7400 CPU, only **15% In some cases**, errors were triggered and the PC even crashed frequently, which had not been a problem with this CPU before. The number of iterations and changes in undervoltage values only exacerbated the problem. A total of 20 encryption errors were triggered on this PC. After the error was triggered, the program displayed all 10 rounds of encryption of the correctly and incorrectly encrypted text, but in no case **did** the required error **occur** in the 8th round of encryption. In fact, in most cases, these two encrypted texts were completely different in each round of encryption. In other cases, it was clear that the error occurred in the 10th round and always resulted in the same bits being flipped, but this is not enough for complete key extraction.

With the i5 6500 processor, there was an even bigger problem at 2.0GHz, which had been the most successful frequency for triggering errors in both multiplication and RSA in manual tests. The error could not be triggered, even at frequencies that achieved high success rates in automatic tests. So the only option was to add the ability to automatically trigger errors in AES encryption to the **test_PV_VP.sh** script.

Finally, we managed to trigger the error, but with very **low success**. However, the triggered errors were of the same nature as with the i5 7400 CPU. In total, only **25 errors** were triggered **out of** 1071 attempts, of which not a single **error occurred in** the required 8th round, only one error occurred in the 6th round, and the other errors were of unknown nature.

Detailed statistics on AES encryption attacks are shown in Table 8.4. The table contains the same parameters as the table showing the success rate of inducing errors in RSA decryption.

Summary

After acquiring all the necessary equipment, we tested whether it was possible to change the CPU voltage. After a successful change, we proceeded to induce an error in multiplication. To successfully trigger the error, it was necessary to reinstall the OS on which the CPU frequency could be set. A custom script was created to find the optimal parameters. Since there were frequent

[9] Source codes used available from: https://github.com/KitMurdock/plundervolt/tree/master/ sgx_aes_ni

CPU	F(GHz)	Undervolting	Success rate	Test type
i5 7400	2.6	from 215mV to 220mV	15%/45%/40%/0	Manual
i5 6500	-	-	0%/0%/0%/0%	Manual
i5 6500	1.0, 1.1, and 1.4	from 30mV to 280mV	8%/0%/92%/0	Automatic

Table 8.4: Undervolting parameters and success rate of triggering an error in the AES encryption algorithm

To prevent CPU freezing, a custom system was created to restart the PC in case of freezing using Raspberry PI 4.0, which achieved a **100%** success rate in triggering a multiplication error. After activating TEE, we moved on to inducing errors in RSA decryption and AES encryption. The optimal parameters obtained when inducing multiplication errors did not achieve sufficient success, so the script was extended to find the optimal parameters for inducing errors in AES and RSA. In RSA decryption, in the best case, it was possible to induce an error in **39% of cases**, which could be exploited to obtain the secret key. In AES encryption, it was also possible to induce an error, but none of the errors **could be exploited** to derive the encryption key.

8.2 Replication of the VoltPillager attack

Since this is a hardware attack, replication in this case is governed by the hardware attack replication protocol. All source codes[10]based on the PlunderVolt attack are freely available for the attack. The only significant difference is the voltage setting using the attack hardware. Replicating this attack is **more difficult** than the PlunderVolt attack because it requires knowledge of electronics to work with an oscilloscope, micro soldering iron, and understanding of circuit diagrams. The TEE activation step is omitted because it is the same as the procedure described in the software attack replication.

8.2.1 Equipment needed for the attack and assembly of attack hardware

The attack was carried out on 1 PC with an i5 6500 CPU. Detailed specifications for this PC are provided in the table below.
8.1. The attack hardware consisted of three components:

- Teensy 4.0 Development Board
- 2x NL17SZ07XV5T2G - used to change the voltage of the clock and data signals sent from the Teensy board to the SVID bus. (from 3.3V to 1.1V).
- 2x SOT IC adapter

The SOT IC adapter is used to convert pins from SOT to DIP. This means that the NL17SZ07XV5T2G bus controller (Figure 8.2) is soldered to it so that the Te-ensy board can be connected to the NL17SZ07XV5T2G and from there to the SVID bus. **A micro soldering iron with** a very thin tip was used for this purpose. The tip must be thinner than the bus controller pins, which are $0.2mm$, otherwise the conductive paths leading to the pins can easily be damaged. It is recommended to use **solder paste** for soldering**.** However, this was not available, so regular tin was used, which is much more difficult and less likely to be successful. Between the conductive paths to which

[10] Available at: https://github.com/zt-chen/voltpillager

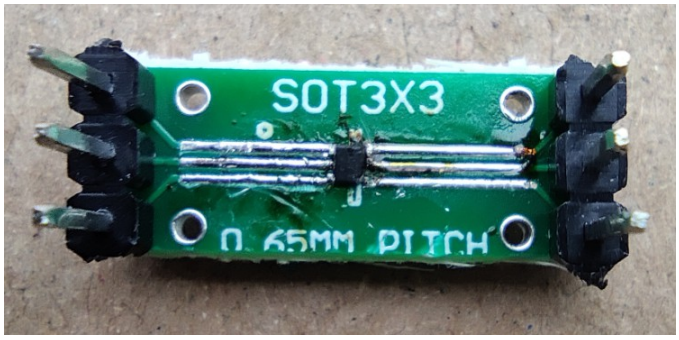

Fig. 8.2: Soldered NL17SZ07XV5T2G on SOT IC adapter

NL17SZ07XV5T2G is soldered, the difference is only 0.45*mm* , which means that a short circuit can easily occur. Therefore, after soldering, a check was performed using a multimeter to ensure that no short circuits or cold joints had occurred. One short circuit was found and subsequently removed.

In addition, the following devices and components were used to carry out the attack:

- RIGOL DS1074Z oscilloscope
- ELMA BM257s multimeter
- ZD-931 micro soldering iron
- MAX232
- Contact field with jumper set

Since the most accurate method for timing the undervoltage in this attack is to send commands from the computer to the Teensy board via RS232, it was necessary to reduce the voltage from the serial port to the voltage at which the Teensy board operates (logical 1 12V to 3.3V, logic 0 from -12V to 0V). A board with an integrated MAX232 circuit was used for this purpose. The ASRock Z170 motherboard does not have an RS232 output on the back of the board, as was typical on older boards, but the technical documentation for the ASROCK motherboard([11]) shows that the serial port is marked directly on the board as COM1. The connector from the MAX232 board cannot be connected directly to the COM1 connector because the MAX232 only modifies the RX and TX signals, but the undervolting command is sent to the DTR pin. This means that the RX pin of the MAX232 board was connected to the DTR pin of the serial line.

The wiring diagram according to which the attack hardware was assembled is shown in Figure 8.3. Compared to the original wiring diagram obtained from the VoltPillager attack study[12], it has been expanded to include the aforementioned MAX232 integrated circuit and its connection to the PC and Teensy board.

8.2.2 Principle of voltage change

The VoltPillager attack uses the SVID bus on the motherboard to change the voltage, to which the attack hardware is connected, sending commands to change the voltage. A more detailed description can be found in section 6.2.

[11] Available at: https://download.asrock.com/Manual/Z170%20Extreme4.pdf

[12] Available from: https://github.com/zt-chen/voltpillager/blob/master/VoltPillager-board/VoltPillager-board.png

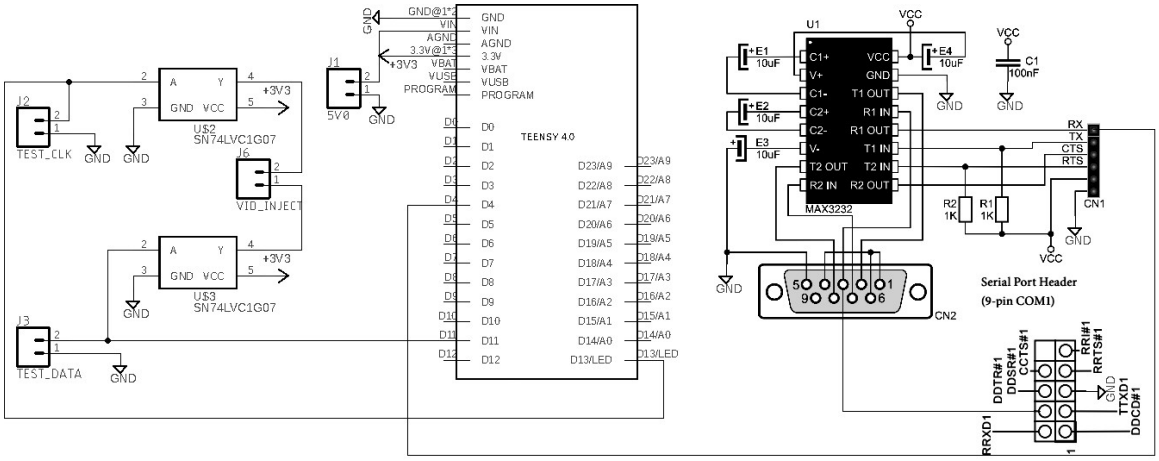

Fig. 8.3: Diagram of the connection of attack hardware to perform a VoltPillager attack

8.2.3 Uploading firmware to change the voltage

Ready-made voltpillager-firmware[13] was used to change the voltage.

In order to upload the firmware to the board, it is first necessary to download and install the following programs:

- Arduino (supported versions 1.8.5, 1.8.9, 1.8.11, 1.8.12, 1.8.13)
- Teensydrino
- TeensyTimerTool library v0.1.8 (Using the Arduino library manager)

After installing all dependencies, simply compile and upload the correct firmware for the attack to the Teensy board in the Arduino environment.

8.2.4 Connecting the attack hardware to the base board/MCU pin

Before connecting the attack hardware, the SVID bus on the motherboard had to be **detected**. The first step is to detect **the voltage regulator (VR)** on the motherboard. Some vendors provide schematic diagrams of their boards, which greatly simplify the process. Unfortunately, most vendors do not publish such detailed documentation about their hardware. VRs are usually located in close proximity to the CPU and large switching transistors and inductors, making them easy to identify visually. Each component has a name on it, so it is possible to search for the component by name to see if it is indeed a VR. It is usually not a problem to find at least a basic description of the component. On the motherboard used for the attack, the VR is located directly above the processor. It is an **ISL95824** voltage regulator**.**

After identifying the VR, it was also possible to identify **the SVID bus**. The technical documentation for the VR should contain all the necessary information about which pins are connected to the VR. However, in most cases, the documentation is not available. Therefore, the bus was identified using **an oscilloscope** by gradually connecting the oscilloscope to the VR pins and then

[13] Available at:https://github.com/zt-chen/voltpillager/tree/master/voltpillager-firmware

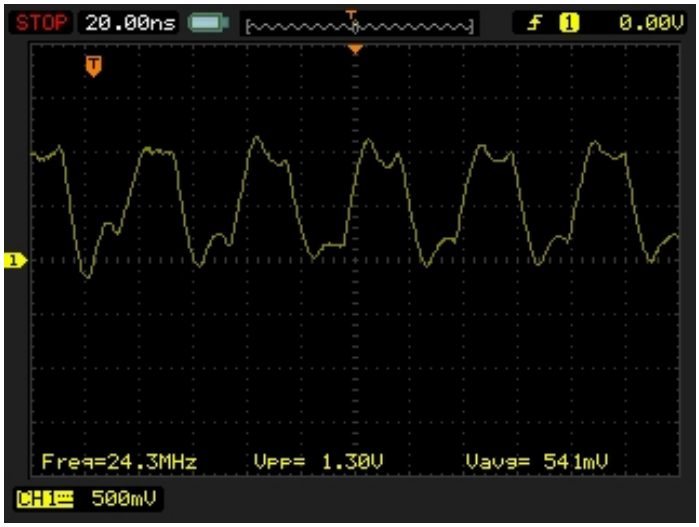

Fig. 8.4: VCLK signal displayed on an oscilloscope

analyzed to see if the signals at a given output behave as they should on the SVID bus. Only the VDIO and VCLK signals of the SVID bus are used for the attack, so the ALERT# signal was not identified.

Detecting the SVID bus using an oscilloscope

SVID must be connected to **pull-up resistors** that supply logic 1 to the bus when nothing is being transmitted on the bus. So the VR pins that were connected to the resistors were checked first. The desired waveform that should be displayed on the oscilloscope for the VCLK signal is **a** typical **clock signal with** a frequency of 25MHz. For the VDIO signal, the data of the desired signal is based on Figure 6.6 with a sampling frequency of 25MHz. Commands are sent to the bus even if a fixed voltage for the CPU is set in the BIOS, although in a limited number compared to other modes. For the fastest possible detection, a mode that dynamically changes the voltage on the processor - 'auto' - was set in the BIOS. Access to the VR pins using an oscilloscope was very limited in the PC case, and with such small pins, there was a high risk of a short circuit. Therefore, for better access, the motherboard had to be removed from the case. This also facilitates the possible connection of the Voltpillager board to the SVID bus.

Before searching for the bus, it was necessary to set up the oscilloscope to detect the signals being searched for. The oscilloscope was therefore set to display 0.5V per division and a time interval of 20ns per division, which should be sufficient to display signals operating at a voltage of 1V and a frequency of 25MHz.

After the correct settings were made, it was only necessary to find the ground (GND) on the motherboard, where 1 pin of the oscilloscope is always connected. For this purpose, the technical documentation for the motherboard was used again and the ground of the COM1 serial port, shown in the wiring diagram in Figure 8.3, was used.

It was then possible to search for the signals themselves. The **VCLK** clock signal shown in Figure 8.4 was easy to find with the oscilloscope set up correctly.

Finding the VDIO signal is much more difficult than finding the clock signal, which has a typical periodic waveform. A logic analyzer and SVID data analysis software can help with detection. However, this was not available, but this signal can also be detected using an oscilloscope. It is sufficient to take a larger portion of the data and analyze whether it contains

Fig. 8.5: Analysis of the data signal

a 24-bit SVID command from the CPU or a 7-bit response from the VR. Therefore, a larger portion of the signal was displayed for this case than when searching for VCLK (200 ns/division). In addition, the fact that this signal is typically very close to the clock signal is quite helpful. After a while, a candidate for the searched signal was found. The data of this candidate was then analyzed.

In the data signal candidate shown in Figure 8.5, a 24-bit SVID command from the CPU to the VR was found during analysis, which exactly matches the expected command according to Figure 6.6:

- First 3 bits - 010
- Address - 0000
- Command - 00111
- Voltage ID - 00010101
- Parity - 0
- Last 3 bits - 011

It follows that this candidate can be designated as the SVID data signal of the bus - **VDIO**.

Figure 8.6 shows where the SVID bus data and clock signals were detected on the motherboard. Between the resistors on which the VDIO and VCLK signals were measured, there is another resistor to which the ALERT# signal will probably be connected, because these three resistors are connected to three adjacent processor pins. This is also confirmed by the block diagram of the voltage regulator ([14]), where the VCLK, ALERT#, and VDIO pins are next to each other in exactly the expected order. However, since the VoltPillager board is not connected to this pin, it was not analyzed whether it is actually the ALERT# signal.

After detecting the bus, it was sufficient to connect the attack hardware to the bus, i.e., the `TEST_CLK (J2)` and `TEST_DATA (J1)` connectors marked on the wiring diagram in Figure 8.3. The connection of all attack hardware, including preparation for automatic testing, which requires a restart using RaspberryPI, can be seen in Appendix C.1.

[14] https://www.renesas.com/sites/default/files/isl95824_0.png

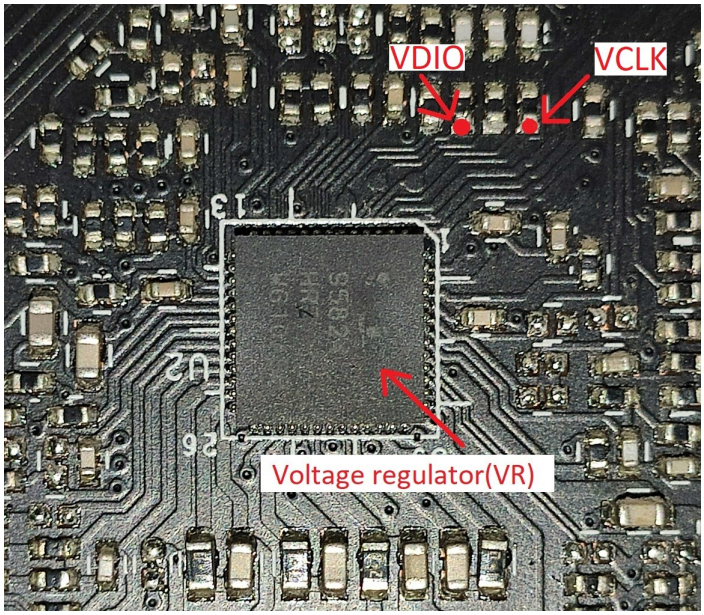

Fig. 8.6: Detection of SVID bus pins on the motherboard

8.2.5 Verification of susceptibility to voltage changes

Before inducing the multiplication error, it was verified whether it was possible to successfully change the CPU voltage. To determine whether this was successful, it is necessary to monitor the voltage on the processor. This can be done, for example, using **monitoring tools (** such as lm-sensors) or an oscilloscope connected to the motherboard. Since undervolting lasts only a few microseconds, the best option is to use **an oscilloscope,** which will definitely detect undervolting.

After uploading the correct firmware to the Teensy board and connecting the attack hardware to the bus, simply connect the Teensy board to a PC via USB, which will enable communication with the board. This means receiving and sending commands to it.To communicate with the Teensy board's serial port, you can use Arduino software (Tools->Serial Monitor), for example. Using the serial link, it is now possible to send undervolting parameters via command. The command has the following form:

$$\<N\ \>\<V_p\ \>\<T_p\ \>\<V_f\ \>\<T_f\ \>\<V_n\ \>$$

where:

- N - Number of undervoltage repetitions
- V_p - Voltage before undervoltage. Typically ≤ V_n
- T_p - Switching time to voltageV_p and the time for which voltageV_p is maintained
- V_f - Voltage used to inject the fault
- T_f - Switching time to voltageV_f and time for which voltageV_f is maintained
- V_n -Stable operating voltage after V_f

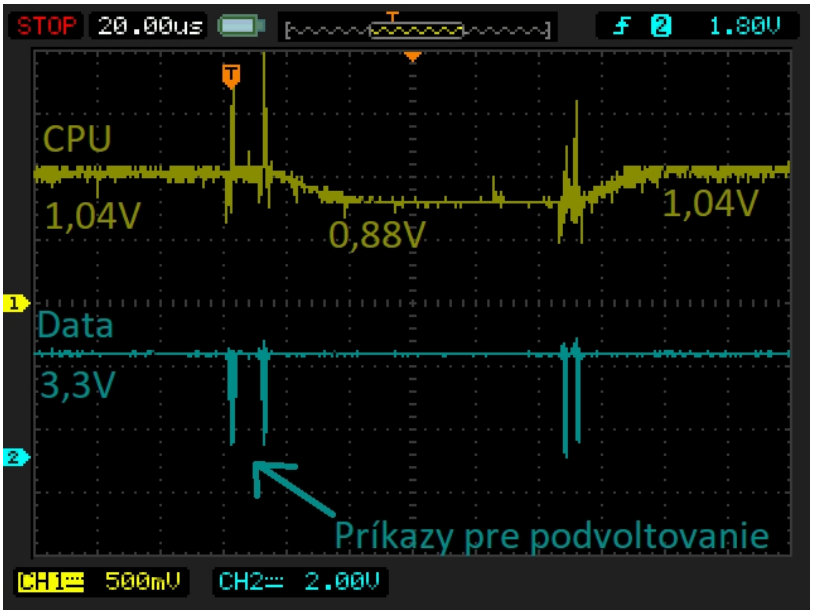

Fig. 8.7: CPU undervoltage recorded using an oscilloscope

Incorrect settings of these parameters will affect system stability. This means that if, for example, the voltage is set too low, the CPU will freeze or crash. The parameters were set to:

$$N = 10.V_p = V_n = V_{cc} = 1.050V, V_f = 0.88V, T_p = 10\mu s, T_f = 32\mu s$$

Following their application, the `arm` command was called on the serial line and the `trigger` signal was sent. As explained above, a USB bus or RS232 serial line is used for this purpose. Both options have been tested. In the case of RS232, it was sufficient to send a logical 0 to the DTR pin of the RS232 line, and in the case of the USB bus, a `delay n` command is sent to the serial line of the Teensy board, where n determines the time (in μs) for which the board receives the `trigger` signal.

Figure 8.7 shows how CPU undervolting works. The oscilloscope measures the voltage at CPU and data sent by Teensy to NL17SZ07XV5T2G, which then sends them to the SVID bus. In the SVID bus, this signal represents a request from the CPU to the VR to reduce the voltage to 0.88V. The VR reduces the voltage because, as can be seen in Figure 8.7, the voltage has dropped to the desired value. After a few microseconds (set according to the parameters), the Teensy board's data pin sends a request to increase the voltage to its original stable value. This is necessary because the CPU freezes when undervolted for a longer period of time. Moreover, this approach is required to trigger errors.

To trigger a multiplication error freely available code from the voltpillager[15] attack was used. which contains both the underclocking commands and the multiplication itself. This is possible because the attack hardware is connected to the same PC that is being attacked, as this achieves the most accurate timing, since the PC always sends the `Trigger` signal before multiplication. In the makefile, the `trigger` signal type was first selected, then the code was compiled with the `make` command, then the CPU frequency was set to **2GHz, and** then the code was run with the following parameters, which managed to trigger the first error:

- Number of CPU threads: –calc_thread_num 4

[15] Available at: https://github.com/zt-chen/voltpillager/tree/master/poc/mul

- Number of iterations: –iter 1000000
- Number of attempts: –retries 10
- Baud rate: -b 115200
- Teensy serial port: -p /dev/ttyACM0
- Multiplication operands: –calc_op1 0xae0000 –calc_op2 0x18 -g
- Voltage value before undervolting: –pre_volt 0.859
- Undervoltage value: –glitch_voltage 0.655
- Voltage value after undervolting: –rst_volt 0.859
- Delay: –pre_delay 35
- Pulse width: –rst_delay -30

The **fault_VP.sh** script was created for comprehensive testing of the multiplication error. The best frequency **was no** longer **sought**, as it can be used from the PlunderVolt attack. The search was for a suitable delay from the `Trigger` command, when the Teensy board should send a command to the SVID bus. The delay was tested from 5μs to 100μs in increments of 5μs . A significant deterioration in results was observed at longer delays. For each delay, a separate file is created in the `data` folder, which contains data on success and errors encountered. The files are named according to the delay, for example, file 5 contains data for a delay of 5μs .

The tests were performed when sending the `Trigger` command via the **USB** bus, but also via the **RS232** serial line. The results achieved are shown in Table 8.5. The success rate indicates the probability of triggering an error. In the remaining cases, the PC crashed, as the automatic test is designed so that either an error will be triggered at a given delay or a crash will occur. Both tests achieved **100% success,** confirming that the best frequency obtained from the PlunderVolt attack is also suitable for the VoltPillager attack, and therefore no others were tested.

μs	**Undervolting**	**Success rate**	**Trigger**
60, 85	from 190mV to 210mV	100	USB
25	from 190mV to 210mV	100	RS232

Table 8.5: Undervolting parameters and success rate of triggering an error in the Volt-Pillager attack for the i5 6500 CPU with a frequency set to 2GHz

8.2.6 Inducing an error in TEE and its analysis

An automatic success test was also performed on RSA decryption, this time at a frequency of **1.4GHz**, which was the most successful in the PlunderVolt attack. Again, **a 100% success rate** was achieved in triggering the error for both methods of sending the `Trigger` signal. In fact, a **100% success rate** was achieved in **obtaining the secret key from** the triggered error.

The results achieved are described in Table 8.6

The automatic testing script was also extended to AES encryption. The success rate of triggering the error was lower than for the previous algorithms, and the triggered error **was not** even

μs	Undervolting	Overall success rate	Trigger
10, 25, 30, 70	from 170mV to 190mV	100	USB
10, 25, 35, 50, 55, 60, 85	from 170mV to 190mV	100	RS232

Table 8.6: Undervolting parameters and success rate of inducing errors in the RSA algorithm using the VoltPillager attack for a CPU i5 6500 with a frequency set to 1.4GHz

possible to exploit. Using the best parameters, another 200 errors were triggered, but even so, no error occurred in the 8th round. The success statistics are shown in Table 8.7. The success column shows three numbers representing the success rate of triggering an error in AES/PC freeze/overall success, which indicates the success rate of triggering an error from which it was possible to obtain the encryption key.

μs	Undervolting	Overall success rate	Trigger
20	from 130mV to 160mV	19%/80%/0%	USB
70	from 130mV to 160mV	33%/67%/0%	RS232

Table 8.7: Undervolting parameters and success rate of triggering errors in the AES algorithm using the VoltPillager attack for the i5 6500 CPU with a frequency set to 1.1GHz

Summary

After procuring all the necessary equipment for the attack, the attack hardware was assembled according to the wiring diagram in Figure 8.3. To change the CPU voltage, it was first necessary to determine the principle of voltage change, upload the correct firmware to the attack hardware, and connect the attack hardware to the motherboard. To connect to the motherboard, it was necessary to detect the voltage regulator and then the SVID bus using an oscilloscope. The optimal frequency for triggering the given error had already been found using a previous attack, but this time the ideal delay for sending undervolting commands was also sought, for which the created script **test_PV_PV.sh** was used again**.** In the case of triggering an error in RSA multiplication and decryption, parameters with **100% success rate** were found**,** including the extraction of the RSA secret key. When an error was triggered in AES encryption, the same problem occurred as in the PlunderVolt attack, i.e., **it was not** possible to obtain the encryption key from the triggered error.

8.3 Replication of the TrustZone-M(eh) attack

The attack was implemented by replicating hardware attacks. Only a presentation is available for the attack itself, with all the necessary information described in section 6.3. This means that all source codes for the attack had to be **created** in this case. The step of analyzing the triggered error is skipped in this case, because it will be clear whether the attack was successful or not as soon as the error is triggered.

8.3.1 Equipment needed for the attack

The main piece of equipment is the **SAM L11** microcontroller, which is being attacked. For this purpose,[a]SAM L11 XPLAINED PRO[16]development board was purchased, which has a built-in debugger that can be used to program and debug the MCU. The presentation can be found at

[16] More detailed information about the board: https://www.microchip.com/en-us/development-tool/dm320205

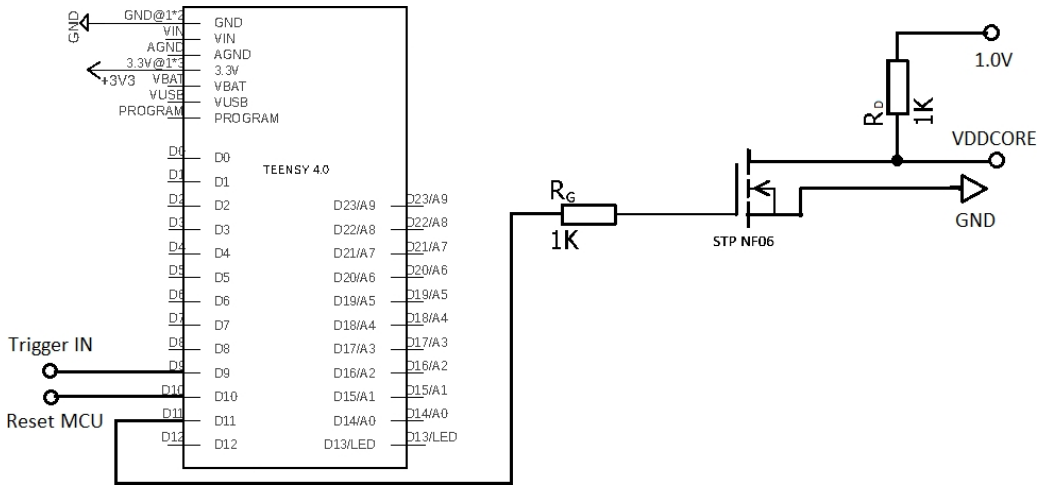

Fig. 8.8: Attack hardware connection diagram

mentions that an FPGA with a frequency of 100MHz was used to control the attack, which can generate sufficiently accurate pulses for undervolting. However, the VoltPillager attack used a **Teensy 4.0** board**,** which has an MCU with a frequency **of 600 MHz.** So instead of the aforementioned FPGA, this board was used, which should achieve even greater accuracy. The entire attack hardware consists of the following components:

- Power supply
- Teensy 4.0 board
- MOSFET N STP 16NF06 transistor
- Contact field with jumpers
- 1-K Ω, 100Ω, and 4.6Ω resistors

8.3.2 Assembling the attack hardware

The basic connection of components is based on Figure 6.7 from the presentation, but the actual wiring diagram for the attack is not available. Therefore, **a custom one** was created **(**Figure 8 .8), which consists of a MOSFET transistor, a Teensy board, and resistors. TheR_G resistor serves to limit the maximum current of the Teensy board. This is $4mA$, so the value of the resistor must must be higher than:$R_G = \frac{U}{I} = \frac{3.3}{0.04} = 825\Omega$ The resistorR_D serves as a pull-up resistor. `VDDCORE` is an output that connects to the voltage of the SAM L11 MCU core. The SAM L11 MCU sends a logical 1 to the `Trigger IN` input if the attack is successful. `The MCU reset` controls the restart of the SAM L11 microcontroller.

8.3.3 Voltage change principle

Undervoltage commands directly control the MCU core voltage. No specific voltage is set, only **ground** is sent for a certain period of time. To protect against undervoltage on pin

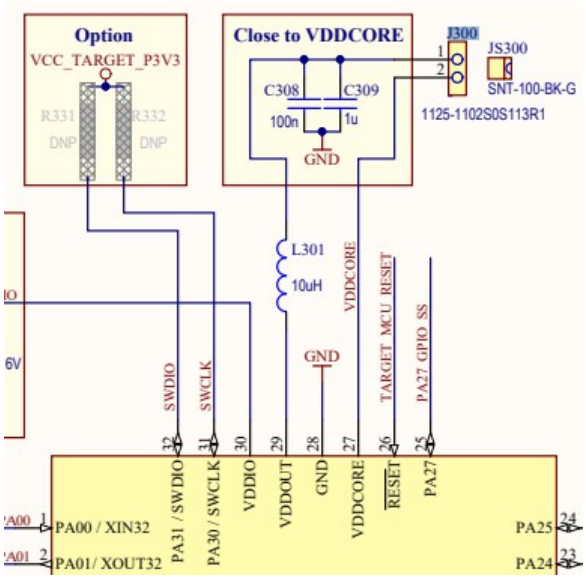

Fig. 8.9: Connection of capacitors C308 and C309 to the VDDCORE pin, which must be removed[19]

VDDCORE, the SAM L11 MCU contains a BOD12 detector, which restarts the microcontroller when undervoltage is detected. However, BOD12 **cannot respond** to very short pulses (tens of*ns*).

8.3.4 Creating and uploading firmware to change the voltage

The attack firmware for the Teensy board to trigger the multiplication error **mul_err.ino**, created in the Arduino environment, is based on listing 7.2. In addition to triggering the error, the goal is to find **the ideal** pulse **width**. To trigger an error that causes unsecured firmware to be uploaded as secured, the **boot_nonsec_as_sec.ino** firmware was created**,** which extends the original code by resetting the SAM L11 MCU after each undervoltage and by one cycle that searched for the ideal time to trigger the error after restarting the MCU.

8.3.5 Connecting the attack hardware to the motherboard/MCU pin

To connect the attack hardware to the SAM L11 MCU, the VDDCORE pin had to be **detected** first using the technical documentation[17]for the MCU. Next, the capacitors connected to the VDDCORE pin had to be detected and removed. This was aided by the wiring diagram for the SAM L11 development board, shown in Figure 8.9. It was not necessary to physically remove the capacitors, but it was sufficient **to remove jumper** J300, where the attack hardware was connected to pin 2.

The connection of the attack hardware to the SAM L11 using the contact field is shown in Figure 8.10.

[17] Page 1102: https://ww1.microchip.com/downloads/en/DeviceDoc/SAM-L10L11-Family-DataSheet-DS60001513F.pdf

[19] Schematic available from (SAM L11 Xplained Pro Design Documentation): https://www.microchip.com/en-us/development-tool/dm320205

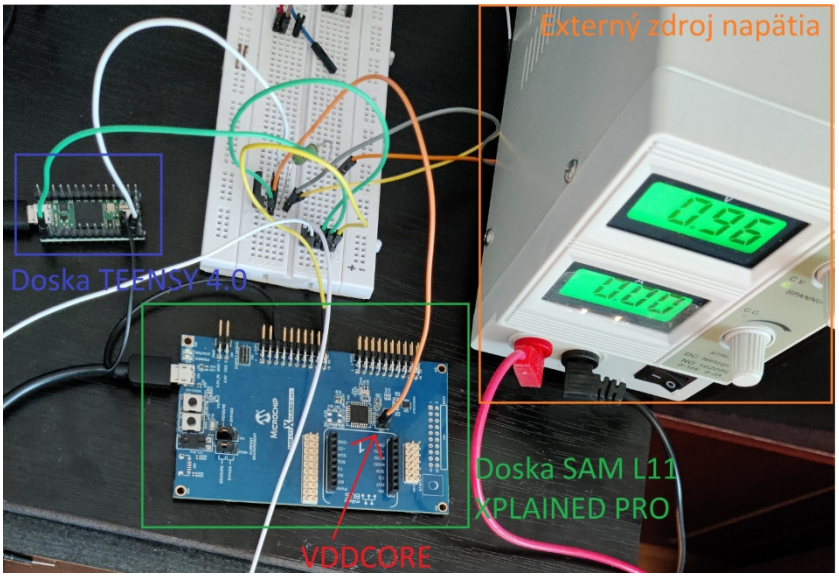

Fig. 8.10: Connection of attack hardware to SAM L11 MCU

8.3.6 Verification of susceptibility to voltage changes

To verify susceptibility to voltage changes, a typical example of a multiplication error was used, sending a logical 1 to the Teensy board if the error could be triggered. In addition, the code was extended to flash the LED every 2 seconds upon success, because sometimes instead of triggering an error, the microcontroller was reset and an error was probably triggered in the MCU initialization, which caused a logical 1 to be sent to the Teensy board, but the LED did not flash at regular intervals. **Atmel Studio** was used for the implementation**,** where there was already an example ([20]of how to flash an LED on the SAM L11 MCU at regular intervals. So the example was only supplemented with an infinite loop with multiplication and sending a logical 1 after the loop. The implementation can be found in the **led_flasher_main.c** file**,** which replaces the file of the same name in the LED flasher example. Atmel Studio also directly supports the SAM L11 XPLAINED PRO board, so it is possible to automatically upload the code to the microcontroller after compilation.

With the connection according to the wiring diagram shown in Figure 8.8 and a pulse width range of $10ns$ to $1000ns$, **no error could be induced** even after several hours. Therefore, the signal sent to the VDDCORE pin was analyzed using an oscilloscope. Already behind the resistor R_G , the transmitted signal looked completely different than it should have. Since the maximum current could also be limited at the power supply, the resistor was removed. However, the signal on the VDDCORE pin still did not look as expected, so the pulse width was extended to $2\mu S$, which revealed the problem. As can be seen on the left in Figure 8.11, even at $2\mu S$, the pulse lasts another $2\mu S$ until the VDDCORE voltage returns to the desired value. This means that the originally wide pulse, for example $100ns$, is up to $2100ns$, which **is** easily **detected by the** BOD12 **detector** and resets the MCU. Since the presentation shows a circuit completely without resistors, the second resistorR_D was also removed, but as expected, no undervoltage occurred in this case. Therefore, resistors of various sizesR_D were tested, which showed that the smaller the resistance, the shorter the return to the original value, but if the resistance is too small

[20] An example can be constructed according to the instructions at:
http://ww1.microchip.com/downloads/en/Appnotes/ Getting-Started-with-SAM%20L10L11-Xplained-Pro-DS00002722A.pdf

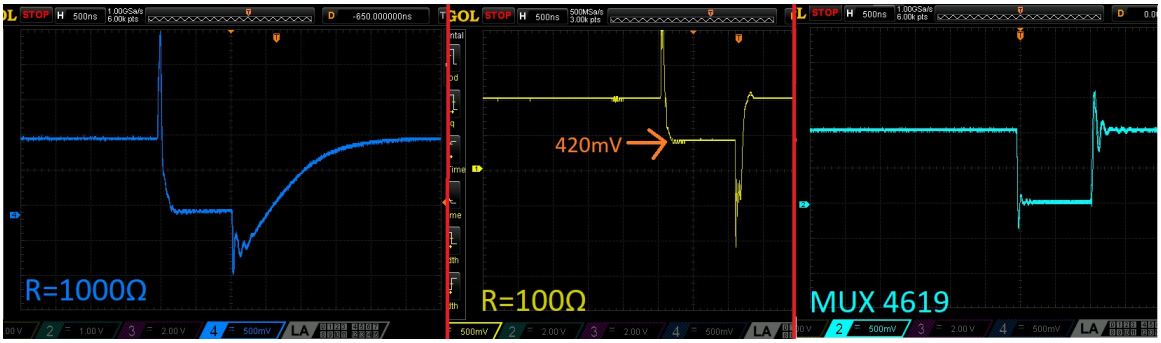

Fig. 8.11: Measured waveform at the output of the VDDCORE pin attack hardware using a transistor and a 1000Ω resistor (left, blue), a transistor and a 100Ω resistor (center, yellow), a multiplexer (right, turquoise)

there is only a fewmV undervoltage. The best results were achieved with a 100Ω resistor, where there was sufficient undervoltage and at the same time a quick restoration of the voltage to its original value. However, even this did not help, so the only option left was to use the MAX4619 multiplexer, which replaced the transistor and was connected to the attack hardware according to the wiring diagram shown in the appendix in Figure D.2. The waveform achieved using the multiplexer is shown on the right in Figure 8.11 in turquoise. However, even the multiplexer did not help, from which it was concluded that the problem **was not** unwanted pulse prolongation.

When debugging the multiplication error step by step, it was found that it was not even possible to enter the multiplication cycle. The value of the multiplication result had already been calculated and stored in a variable before the actual calculation. The program behaved as if it were performing an infinite empty cycle. Therefore, it was not possible to trigger the error, as there was none. The solution was to disable the optimizations, which were originally set to -O1. After debugging step by step again, everything went as expected. Using the MAX4619 multiple XOR, it was possible to trigger the error after a few seconds. After triggering the error, the pulse width was gradually optimized. At a width above 120nS , the microcontroller restarted, which means that the BOD12 detector recorded undervoltage. Therefore, the pulse width was examined from 2ns , which is the minimum that the Teensy board can generate, to 120ns . Subsequently, versions of the attack hardware with a transistor and 100Ω and 1000Ω resistors were also tested. With a 100Ω resistor, errors were also **successfully** triggered with a similar number of attempts as with the multiplexer. What was surprising about this setting was that errors were also triggered at a pulse width of 300ns , where the MCU had previously restarted regularly. Even at tens of microseconds, it was possible to trigger an error with a larger number of attempts. This is because the undervoltage only reaches a value of approximately $420mV$ (at 0V for the multiplexer), which the undervoltage detector cannot detect at all. With a resistance of 1000Ω, **it was not** even **possible** to trigger the error. This confirms that the time it takes for the MCU core voltage to return to normal after undervoltage has a significant impact on triggering the error.

The graph representing the number of attempts required to cause an error for the multiplexer and 100Ω resistor is shown in Figure 8.12.

8.3.7 TEE activation

Two separate programs are used to activate TEE, one considered secure and the other insecure, which are then linked together. Atmel Studio already contains a ready-made example of secure TrustZone-GetStart-S firmware and insecure firmware

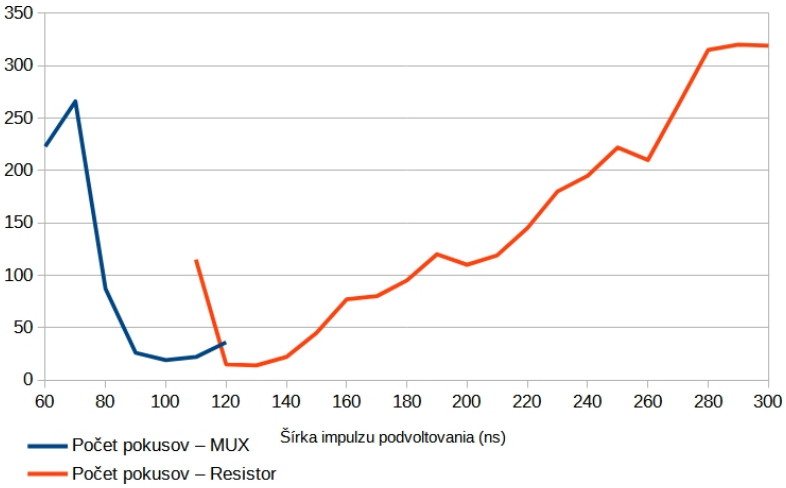

Fig. 8.12: Graph representing the number of attempts to successfully trigger an error using a multiplexer or a transistor with a 100Ω resistor

`TrustZone-GetStart-NS`. To connect these examples, their correct translation and uploading instructions from Microchip for the SAM L11 XPLAINED PRO board were used[21]. Subsequently, it was tested whether the TEE actually works by using a function from the unsecured part to light up an LED. The LED did not light up. However, when a function was created and then called from the NSC (non-secure callable) part, in which the LED lighting was implemented, the LED lit up as expected.

8.3.8 Calling an error in TEE

The goal in the example used in the previous section was **to flash an LED** at regular intervals and **send a logical 1** to the Teensy board from pin PA10, directly from the unsecured part, which is normally **impossible**. The created source code is located in the `boot_nonsec_as_sec` folder, which contains the **main.c** file**,** which replaces the file of the same name in the created example. As already described in section 6.3, this is achieved by skipping the instruction that reads the size of the AS register, which determines the size of the secure part. This causes its size to remain zero and all code to be in one section. This means that it should be possible to access any function, including working with I/O outputs, which are the target of the attack.

First, the time interval **from the** MCU **restart** to **the setting of logical 1** on one of the microcontroller outputs was determined. An oscilloscope was used for this purpose, and its measurement is shown in Figure 8.13, from which it can be easily read that this time is approximately $3.8ms$. This means that undervoltage must occur no later than $3.8ms$ after restart. However, this is still too long a time interval, as the instruction runs for a few tens of nanoseconds and, moreover, the error may not be triggered on the first attempt. **Performance analysis** was used to determine a more precise point of error injection.

[21] Available from (pages 15-34): http://ww1.microchip.com/downloads/en/Appnotes/Getting-Started-with-SAM%20L10L11-Xplained-Pro-DS00002722A.pdf

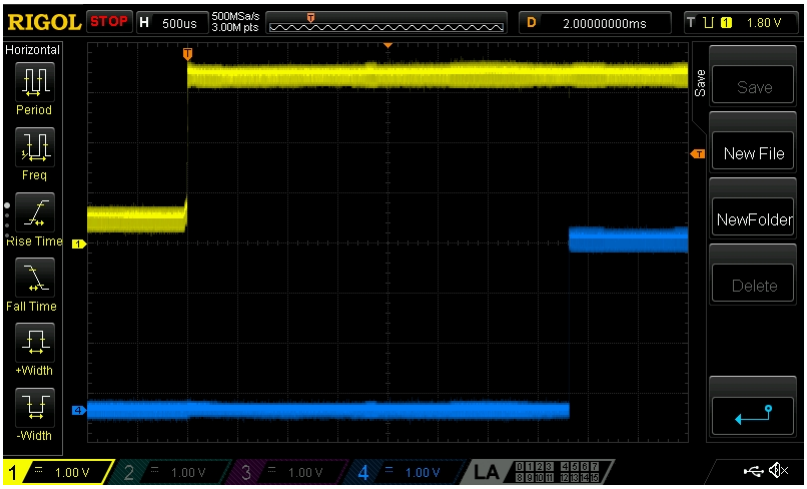

Fig. 8.13: Measuring the time using an oscilloscope from the restart of the microcontroller (yellow signal) to the execution of the first program instruction (setting the blue signal to logical 1)

Performance analysis

Performance analysis is used to measure consumption when reading the **highest** value of the AS register and the **lowest** value, because consumption should **differ** when reading different register values. Since everything else in the source code remains unchanged, the most significant difference in consumption will determine the exact moment when the AS register value is read. The values written to the registers (change of fuses) that determine the sizes of the secured and unsecured parts are stored in the `trustzone_config` file, where the value of the `CONF_UROW_IDAU_AS` macro has been changed. The largest and smallest values of the AS register so that the code used fits into the given part were `0xF7` and `0x20`. Using the SAM L11([22])security reference manual, it was found that, in addition, it is necessary to set new sizes for the changed memories in the `saml11_nonsecure.ld` and `saml11_secure.ld` files. In the `saml11_nonsecure.ld` file, only the `ORIGIN` and `LENGTH` parameters of the rom memory are set to the value AS * 0x 100. In the `saml11_secure.ld` file, the `LENGTH` parameter of the rom memory and the `ORIGIN` parameter of the rom_nsc memory are set to the value AS * 100 − $ASNC$ * 0x 20. For example, with a register value of AS `0xF7` and a register value of ASNC `0x10`, the value is `0xF500`.

The shunt method was used for the performance analysis, which means that a low-ohm resistor is connected in series to the load circuit. An oscilloscope is then used to measure **the voltage drop** across the resistor. For this purpose, a special differential oscilloscope probe is commonly used, which is connected to points A and B shown in Figure 8.14. However, this was not available, so two standard probes were used, one connected to points A and C and the other to points B and C. The difference in the measured voltage at the probes indicates the voltage drop across the shunt. To achieve the highest possible accuracy**,** the value of resistor R must be as **small as possible.** Several resistor values from 0.6Ω to 100Ω were tested, and it was confirmed that the best results were achieved with the smallest resistance, which in this case was 0.6Ω.

The SAM L11 XPLAINED PRO board has pins on the board where you can connect hardware that performs power analysis, or in this case, a shunt resistor. For measuring I/O consumption, this is jumper J102, and for measuring MCU consumption, which in this case is

[22] Available at: http://ww1.microchip.com/downloads/en/AppNotes/SAM-L11-Security-ReferenceGuide-AN-DS70005365A.pdf

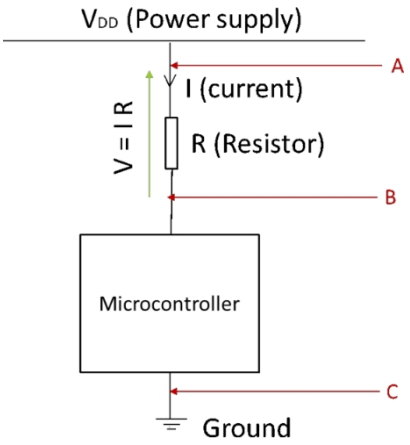

Fig. 8.14: Options for connecting an oscilloscope to perform MCU power analysis using a shunt [22]

Essentially, this involves removing jumper J103 and connecting the shunt to it. In addition, it was necessary to connect a probe to the oscilloscope to monitor when the microcontroller restarts, which serves as **a trigger** for measuring consumption. The measurement result can be seen in Figure 8.15.

A total of **10** consumption **measurements** were performed with the AS register set to 0x20 and **10** consumption **measurements** with the AS register set to 0xF7. The measurement results can be found in the `power_analysis` folder. Subsequently, the consumption for both values was averaged and the difference between these averages was calculated and plotted in the graph shown in Figure 8.16. From the graph**,** six **candidates** can be identified where the AS register value may be read. The best candidate, which was tested first, is around sample number
400. At this point, the highest difference in consumption occurs, and in addition, neighboring samples also achieve a greater difference in consumption than neighboring samples of other candidates. The oscilloscope generates a sample every $5\mu s$, which means that the range of the first candidate, where a significant difference in consumption occurs, is approximately $1900\mu s$ to $2300\mu s$ after restarting the MCU. These parameters were entered into the Teensy board source code, with a step of $1\mu s$. First, an attack was performed using a transistor with a 100Ω resistor, but this **did not** cause the desired error. When using a multiplexer, after approximately **90 minutes**, a logical 1 was successfully sent to the Teensy board and the LED started flashing. The time after the restart when the error was correctly triggered was $2180\mu s$. This environment was examined more closely with an accuracy of $2ns$, but the error still occurred **randomly in** the interval $2175\mu s$ to $2189\mu s$ after restart. The best parameters for triggering the error were the interval $2180\mu s$ to $2189\mu s$ with a step of $50ns$, which triggered the desired error on average every 5 **minutes.**

Summary

The first step was to procure the necessary equipment and assemble the attack hardware according to the wiring diagram in Figure 8.8. To change the voltage, it was necessary to determine how the voltage changes, create and upload firmware to the attack hardware, and connect the attack hardware to the SAM L11 MCU. To verify the susceptibility to voltage changes, special firmware was created and uploaded to the SAM L11 MCU to cause a multiplication error. Original wiring diagram

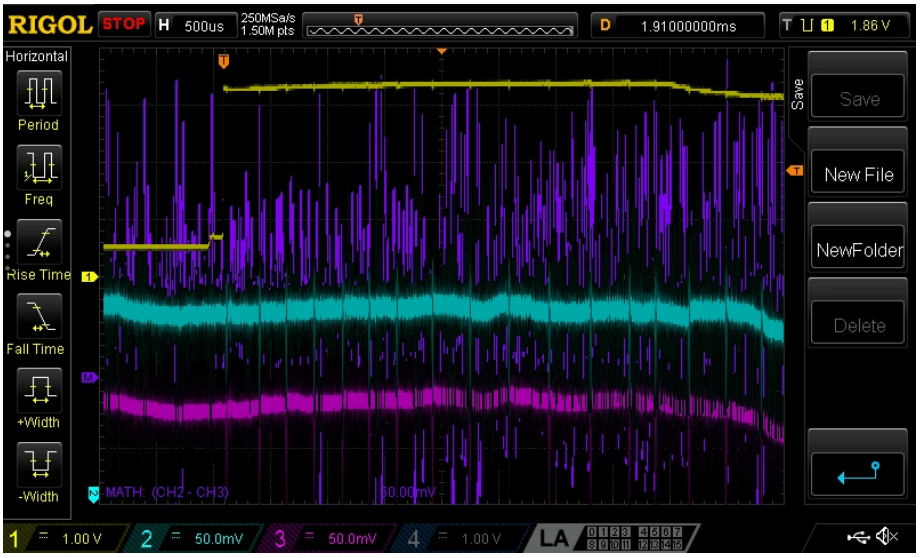

Fig. 8.15: Performance analysis using an oscilloscope. Channel 1 (yellow signal) measures the value at the MCU reset pin, channel 2 (turquoise signal) measures the voltage before the shunt, channel 3 (pink signal) measures the voltage after the shunt, and the purple signal determines the voltage drop across the shunt.

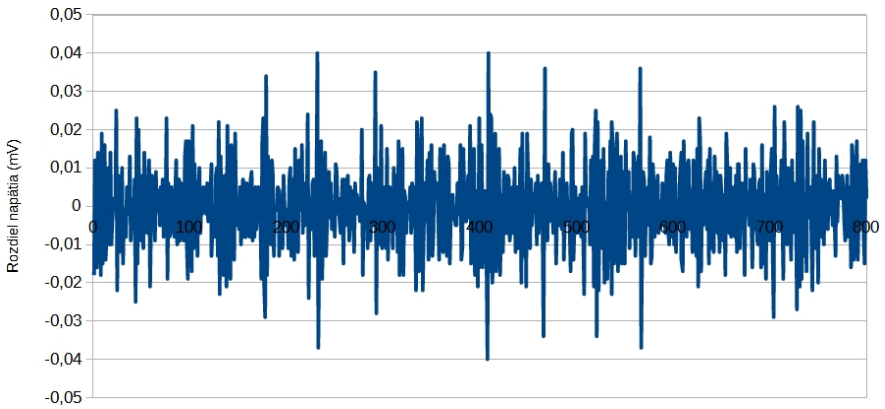

Fig. 8.16: Resulting graph from performance analysis to determine the value written to the AS register

from Figure 8.8 proved to be unsuitable, so two additional options were created (Appendix D), which successfully triggered the error. After activating the TEE, firmware was created for the attack hardware to trigger the error in the TEE. Performance analysis was used to find the delay at which the error should be triggered, which revealed the execution time of the instruction that must be skipped by undervolting. This makes the original trusted code untrusted, and its instructions can also be executed from the original untrusted part. To verify that this error could be triggered, firmware was created for the MCU SAM L11, where untrusted code attempts to flash an LED, which is normally prohibited. With the best parameters (i.e., a delay of 2180μs to 2189μs), the LED flashed on average once every 5 minutes.

Chapter 9

Discussion

The discussion is divided into three parts, just like the implementation. Each part discusses the results achieved in comparison with the results achieved by the attackers and possible types of defenses against the attack.

9.1 PlunderVolt

The PlunderVolt software attack was successfully replicated on two different CPUs, each from a different generation. An automatic comprehensive test was also performed on the i5 6500 CPU, and the i5 7400 CPU was tested to see if the bug could be triggered, which it was **20 times** at the most successful frequency. However, the results are quite **different** from what the attackers achieved. The point at which the error was successfully triggered caused frequent **crashes.** This problem was to be solved by a larger number **of iterations,** which would reduce the undervolting value and make it no longer critical. However, even when triggering a multiplication error, increasing the number of iterations had **no effect** on the undervoltage value. Even in an extensive test of all frequencies, 1,000 iterations achieved greater success than 1,000,000 iterations.

With the multiplication error, it was possible to trigger the desired fault, i.e., the flipping of 1 bit in the multiplication, which caused the infinite loop to be terminated. Since at frequencies from 1.1GHz to 3.0GHz always managed to cause at least 1 error, and a graph was also created showing when the first error occurred at a given frequency. As can be seen in Figure 9.1, this corresponds to the findings of the creators (Figure 6.3), i.e., the higher the frequency, the higher the undervoltage.

The attack on the RSA decryption algorithm was also **successful** and the secret key was obtained, but triggering the error did not automatically mean obtaining the decryption key. This was quite dependent on the frequency at which the error was triggered. In some cases, only **10% of errors** could be exploited, but the most successful (1.4GHz), which caused the most faults, caused over **90% of errors** that could be exploited. The overall success rate was therefore still quite high and poses a real danger to the processor on which the error was triggered.

The results of the attack on AES encryption were **unsatisfactory**. Not only was the error rate very low even after comprehensive testing, but **it was not possible to trigger** an error that would allow the encryption key to be obtained. Therefore, over a thousand additional attempts to trigger a failure were made at the most successful frequencies, but still not a single one triggered the desired error. In other words, out of more than 100 errors induced**, the success rate** was still **0%,** which is nowhere near the 10% success rate reported by the authors. However, errors were induced in **the fourth**

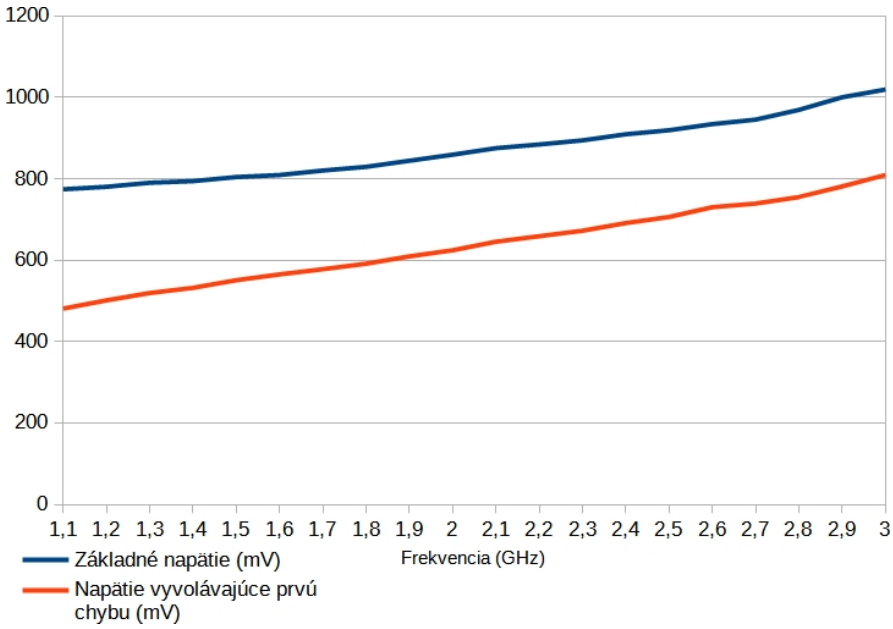

Fig. 9.1: Measured relationship between frequency, normal voltage (blue), and required undervoltage (orange) to achieve an incorrect multiplication result for the i5-6500 processor

and the **sixth** round, meaning that theoretically it would be possible to cause an error in the required eighth round. The entire test, which was ultimately unsuccessful, took over a week of continuous PC operation with hundreds of restarts. It is therefore quite unlikely that the victim would not notice the strange behavior of the PC, and it was therefore concluded that this attack on a specific processor (i5 6500) does not pose a threat. However, this behavior was also confirmed by a second CPU, where none of the 20 errors triggered **could be exploited.**

Defensive techniques

However, PlunderVolt attacks can be easily prevented. All you need to do is **update your BIOS** to the 2019 version or newer. In the case of CPUs manufactured since 2019, this is not even necessary, as they only support BIOS versions where protection is already implemented. The defense is that changing the CPU voltage by writing a value to the MSR register has been **disabled**. In addition to prohibiting writing values to the MSR register, Intel could have solved the problem, for example**,** by automatically **scaling back** any voltage change after entering the enclave**.** Alternatively, the solution could be to allow only verified and **safe values** to be changed for each frequency**.** At the software level**,** security can be increased by performing critical operations **multiple times.** This means, for example, that RSA decryption is performed three times a n d compared to see if the same result was always achieved. However, it is still possible to cause an error in all decryptions. However, there is a very small chance of causing the same error in all three decryptions.

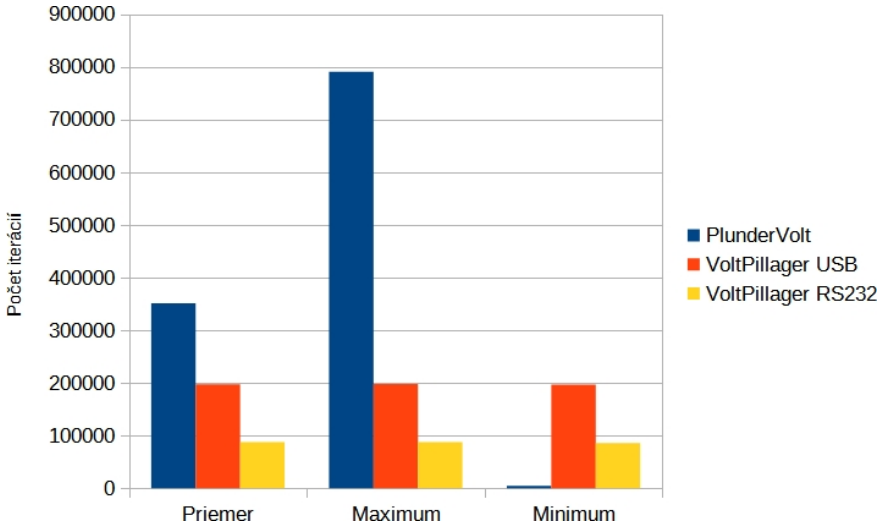

Fig. 9.2: Average, maximum, and minimum number of iterations required to cause a multiplication error in a CPU i5 6500 at a frequency of 2.0GHz

9.2 VoltPillager

The VoltPillager hardware attack was **successfully** replicated on an i5 6500 CPU, where a comprehensive test was performed again, but this time to find the ideal delay value for sending the undervolting command. This time, however, the results achieved are very similar to those achieved by the creators. Although the undervolting value was very close to the value at which the PC froze, with ideal parameters, the error was successfully triggered with **100% success** without freezing the PC. This was also the case for **RSA multiplication and decryption.** Even with RSA decryption, under optimal parameters**,** it was possible to trigger an error that could **always be exploited.** So the overall success rate was also 100%. Inducing an error in AES encryption also achieves satisfactory success, but it was not possible to induce an error from which the encryption key could be derived using differential error analysis. So, if the attack can be classified as a threat model, the attack poses a real danger only to the RSA algorithm.

The creators also claim that, compared to the PlunderVolt attack, fewer iterations are needed. This was **confirmed** by the graph in Figure 9.2, which was created based on the data obtained. In addition, the iterations in which an error occurred during the VoltPillager attack differed by a maximum of 2000. In the PlunderVolt attack, the iteration of the triggered error was purely **random, and** the differences were in the hundreds of thousands of iterations. The graph in Figure 9.2 also confirms that when sending the `Trigger` signal via the RS232 serial link, a **significantly smaller** number of iterations is sufficient.

Defensive techniques

The protection applied by Intel against the PlunderVolt attack, where changing the value written to the MSR register was prohibited, is ineffective in this case. Even the latest generation of processors **are unable** to defend against the attack. One possible solution, which is not definitive, is **to encrypt communication** on the SVID bus. This would prevent reverse engineering, which has been used to discover commands for undervolting to a specific value. However, VR essentially

converts SVID commands into a PWM signal that controls the transistors that generate the actual core voltage. Instead of sending commands to SVID, an attacker could disconnect the SVID bus and supply their **own** (malicious) PWM signals, **bypassing** any SVID authentication. A well-equipped attacker could even completely replace the VR with their own. However, a solution could be for the CPU **to monitor its voltage and**, if it falls below a safe value, interrupt execution within the enclave. Although Intel processors support CPU voltage monitoring (for example, by reading the value of the MSR 0x198 register), all known methods have **a low sampling frequency,** which would not always be able to detect undervolting. This means that new generations of CPUs could include specialized circuits for monitoring voltage with **a high refresh rate.** From the customer's point of view, a definitive solution is not known. Security can be increased **by redundancy of critical operations,** as already mentioned in the defense against PlunderVolt attacks.

9.3 TrustZone-M(eh)

The implementation proved that the attack can **be replicated** with similar success as in the attack presentation. When triggering the multiplication error, the error occurred almost **immediately**, but when triggering the error in TEE, the success rate was **slightly lower**. In the presentation demonstrating the triggering of this error, the error occurs after a few seconds. In some cases, it was also possible to skip the required instruction within a few seconds, but on average, even with the best parameters, it took up to 5 **minutes.** However, the author does not state whether this is the best result or a normal one. It was originally thought that this might be because the author managed to find the time to trigger the error from restart with an accuracy of $10ns$. However, this was not confirmed because the Teensy board would be able to find the time to inject the error with an accuracy of $2ns$, as the interval had already been sufficiently narrowed down by performance analysis. However, the error still occurred randomly in the interval from $2175\mu s$ to $2189\mu s$, and even the accuracy of $2ns$ did not achieve the best results.

The result of the performance analysis **is not** entirely convincing. This may be due to **the** performance analysis **method**, **insufficient number of measurements**, or **low sampling frequency**. The method should not cause a problem, as it has been successfully used, for example, to decrypt AES on various MCUs [22], where greater accuracy is certainly required. Few measurements were taken because the fastest way to save the results from the oscilloscope was **to manually save** each measurement **to a USB drive** in csv format. A higher sampling frequency was also tested, with voltage measurements taken every $200ns$, but the results were even worse. In any case**,** it would definitely be better to use **a power analyzer** directly**,** which would allow hundreds of measurements to be taken in a matter of moments and then averaged. For example, Keil ULINKplus is also supported with the SAM L11 XPLAINED PRO board, which has a connector for connecting it. Even the development board has a built-in power analyzer, whose data can be displayed in Atmel Studio, but when measuring the microcontroller's power consumption, the analyzer was only able to measure a maximum of 15,000 samples per second. That is only 1 sample every approximately $67,\mu s$, which is **insufficient.**

Defensive techniques

Software defense against this attack at the user level **is not possible** because the AS register value is read during **ROM** memory **bootup**. This means that the time and method of writing the value to the AS register cannot be influenced, and the method is not even known. Therefore, only **the manufacturer** can increase the security of the microcontroller. To increase security, the manufacturer could implement

read the AS register value multiple times at **random times**. It would then be possible to determine whether the values read in the register match. However, with high-quality performance analysis and a large number of attempts, it would still be possible to carry out the attack. Some MCUs with TrustZone-M, such as the NXP LPC55s69**,** use a special **BLXNS** instruction to transition from the secure world to the unsecured world**,** which prevents unintentional jumps to the unsecured world. An attack on this MCU is still possible, but it is much **more difficult,** as it is necessary to bypass this instruction as well. At the chip design level, the solution would be not to output the VDDCORE pin, to which undervolting commands are sent. However, this would cause the MCU core voltage to become unstable, and smoothing capacitors would need to be added directly to the design. It would still be possible to use error injection, for example using **an electromagnetic pulse,** to skip the required instructions**.**

Chapter 10

Conclusion

The aim of the work was to replicate attacks that cause errors in CPU and MCU calculations by changing the voltage. We successfully replicated the PlunderVolt software attack and the VoltPillager hardware attack on the Intel i5 6500 CPU. In addition, the TrustZone-M(eh) hardware attack on the SAM L11 MCU was also replicated. Successful replication first required studying the principle of attacks that cause errors and the trusted execution environment, which is CPU and MCU protection used to prevent primarily hardware attacks. However, the attacks that were replicated successfully bypassed this environment.

The goal of the PlunderVolt and VoltPillager attacks is to cause errors by flipping bits in the RSA and AES encryption algorithms running within the Intel SGX trusted execution environment. Exploiting this error first required studying the encryption algorithms themselves and analysis techniques to obtain the secret key. Both attacks were only successful in decrypting RSA.

The TrustZone-M(eh) hardware attack even managed to bypass the entire TrustZone-M trusted execution environment. The intention was to inject a bug that would skip the instruction that determines the size of the secure part of the TrustZone-M environment. This caused its size to be zero, and all code was therefore located in the unsecured part. This meant that unsecured code could access all the functions of the originally secured code.

The last part of the work is a discussion of these attacks, where the results achieved and suggestions for possible protection against these attacks are discussed in detail. These could not be implemented because they are effective at the hardware or microcode level, so it is up to the manufacturer to implement some of them.

The work could be extended by testing other Intel processors and attempting to trigger an error in AES encryption that would allow the encryption key to be obtained, as it has not been confirmed that this is possible. Replicating the TrustZone-M(eh) attack on the SAM L11 MCU provides a good basis for how to trigger errors in MCUs with ARM architecture. In the future, it would be possible to carry out attacks on the Nuvoton NuMicro M2351 or NXP LPC55S69 MCUs, which use a similar principle to the attack on the SAM L11 MCU. These microcontrollers already have a simple defense against these attacks implemented, but this protection can be bypassed. Attacks on the Trezor One or Ledger Nano S hardware wallet microcontrollers have a similar basis, making it possible to gradually work your way up to obtaining the wallet's PIN code.

Literature

[1] ABDULLAH, A. Advanced Encryption Standard (AES) Algorithm to Encrypt and Decrypt Data. June 2017, [cited 2021-12-10].

[2] ALI, S. S. and MUKHOPADHYAY, D. Differential Fault Analysis of AES-128 Key Schedule Using a Single Multi-byte Fault. In: PROUFF, E., ed. *10th Smart Card Research and Advanced Applications (CARDIS)*. Leuven, Belgium: Springer, September 2011, LNCS-7079, pp. 50–64 [cited 2021-12-15]. Smart Card Research and Advanced Applications. DOI: 10.1007/978-3-642-27257-8_4. Part 2 : Invasive Attacks. Available from: https://hal.inria.fr/hal-01596300.

[3] AMBEDKAR, B. and BEDI, S. A New Factorization Method to Factorize RSA Public Key Encryption. *International Journal of Computer Science Issues*. November 2011, vol. 8, [cited 2021-12-03].

[4] ANATI, I., GUERON, S., JOHNSON, S., and SCARLATA, V. Innovative Technology for CPU Based Attestation and Sealing. In:. 2013 [cited 2022-05-11]. Available f r o m : https://www.intel.com/content/www/us/en/developer/articles/technical/innovative-technology-for-cpu-based-attestation-and-sealing.html.

[5] ARAMAKRISHNAN, V., VENUGOPAL, P., and MUKHERJEE, T. Proceedings of the International Conference on Information Engineering. In: Association of Scientists, Developers and Faculties (ASDF), 2015, p. 9 [cited 2021-11-15]. ISBN 9788192974279. Available from: https://books.google.cz/books?id=Gw9pCwAAQBAJ&pg=PA9&redir_esc=y#v=onepage&q&f=false.

[6] ARFAOUI, G., GHAROUT, S., and TRAORÉ, J. Trusted Execution Environments: A Look under the Hood. In: *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. 2014, pp. 259–266 [cited 2022-03-17]. DOI: 10.1109/MobileCloud.2014.47.

[7] ASJAD, S. The RSA Algorithm. December 2019, [cited 2021-12-01].

[8] AWANG, N. F. B. Trusted computing - opportunities amp; risks. In: *2009 5th International Conference on Collaborative Computing: Networking, Applications and Worksharing*. 2009, pp. 1–5 [cited 2022-03-15]. DOI: 10.4108/ICST.COLLABORATECOM2009.8402.

[9] BAGHERI, N., EBRAHIMPOUR, R., and GHAEDI BARDEH, N. New differential fault analysis on PRESENT. *EURASIP Journal on Applied Signal Processing*. December 2013, vol. 2013, pp. 145–, [cited 2021-12-01]. DOI: 10.1186/1687-6180-2013-145.

[10] BAKSI, A. et al. *Fault Attacks In Symmetric Key Cryptosystems* [online]. 2020 [cited 2021-11-17]. Available from: https://eprint.iacr.org/2020/1267.pdf.

[11] BARENGHI, A., BERTONI, G., PARRINELLO, E. and PELOSI, G. Low Voltage Fault Attacks on the RSA Cryptosystem. In:. September 2009, pp. 23–31 [cited 2021-11-16]. DOI: 10.1109/FDTC.2009.30.

[12] BENOT, O. *Fault Attack*. Encyclopedia of Cryptography and Security, 2011 [cited 2021-11-12]. ISBN 978-1-4419-5905-8.

[13] BIHAM, E. and SHAMIR, A. Differential fault analysis of secret key cryptosystems. In: KALISKI, B. S., ed. *Advances in Cryptology — CRYPTO '97*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 513–525 [cited 2021-12-15]. ISBN 978-3-540-69528-8.

[14] BLÖEMER, J. and SEIFERT, J.-P. *Fault based cryptanalysis of the Advanced Encryption Standard* [Cryptology ePrint Archive, Report 2002/075]. 2002 [cited 2021-11-28]. Available from: https://ia.cr/2002/075.

[15] BONEH, D., DEMILLO, R., and LIPTON, R. On the Importance of Checking Computations. July 1997, [cited 2021-12-18].

[16] BREIER, J., HOU, X., and LIU, Y. Fault Attacks Made Easy: Differential Fault Analysis Automation on Assembly Code. *IACR Transactions on Cryptographic Hardware and Embedded Systems*. May 2018, vol. 2018, no. 2, pp. 96–122, [cited 2021-12-17]. DOI: 10.13154/tches.v2018.i2.96-122. Available from: https://tches.iacr.org/index.php/TCHES/article/view/876.

[17] CAI, Y., GHOSE, S., LUO, Y., MAI, K., MUTLU, O. et al. Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques. In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 49–60 [cited 2021-11-10]. DOI: 10.1109/HPCA.2017.61.

[18] CHEN, Z., VASILAKIS, G., MURDOCK, K., DEAN, E., OSWALD, D. et al. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In: *30th USENIX Security Symposium (USENIX Security 21)*. Vancouver, B .C.: USENIX Association, August 2021 [cited 2021-10-20]. Available from: https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai.

[19] CONDÉ, R. C. R., MAZIERO, C. A., and WILL, N. C. Using Intel SGX to Protect Authentication Credentials in an Untrusted Operating System. In: *2018 IEEE Symposium on Computers and Communications (ISCC)*. 2018, pp. 00158–00163 [cited 2022-03-18]. DOI: 10.1109/ISCC.2018.8538470.

[20] DAEMEN, J. and RIJMEN, V. *The Design of Rijndael: AES — The Advanced Encryption Standard*. Springer, Berlin, Heidelberg, 2002 [cited 2021-12-03]. ISBN 978-3-642-07646-6.

[21] ELBELTAGY, M. *StegoCrypt3D: 3D Object and Blowfish*. Dissertation.

[22] GAMAARACHCHI, H. and GANEGODA, H. *Power Analysis Based Side Channel Attack*. arXiv, 2018 [cited 2022-04-27]. DOI: 10.48550/ARXIV.1801.00932. Available from: https://arxiv.org/abs/1801.00932.

[23] GUERON, S. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptol. ePrint Arch.* 2016, vol. 2016, p. 204, [cited 2021-12-17].

[24] HUA, Z., GU, J., XIA, Y., CHEN, H., ZANG, B. et al. vTZ: Virtualizing ARM TrustZone. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, August 2017, pp. 541–556 [cited 2022-03-20]. ISBN 978-1-931971-40-9. Available from: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hua.

[25] JAKUB, B., JAP, D., and CHEN, C.-N. Laser Profiling for the Back-Side Fault Attacks: With a Practical Laser Skip Instruction Attack on AES. *CPSS 2015 - Proceedings of the 1st ACM Workshop on Cyber-Physical System Security, Part of ASIACCS 2015*. April 2015, pp. 99–103, [cited 2021-11-30]. DOI: 10.1145/2732198.2732206.

[26] JOVANOVIC, P., KREUZER, M., and POLIAN, I. A Fault Attack on the LED Block Cipher. In: SCHINDLER, W. and HUSS, S. A., eds. *Constructive Side-Channel Analysis and Secure Design*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 120–134 [cited 2021-11-20]. ISBN 978-3-642-29912-4.

[27] KENJAR, Z., FRASSETTO, T., GENS, D., FRANZ, M., and SADEGHI, A. V0LTpwn: Attacking x86 Processor Integrity from Software. *CoRR*. 2019, abs/1912.04870, [cited 2021-10-22]. Available from: http://arxiv.org/abs/1912.04870.

[28] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H. et al. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014, pp. 361–372 [cited 2021-10-25]. DOI: 10.1109/ISCA.2014.6853210.

[29] KORKIKIAN, R. *Side-channel and fault analysis in the presence of countermeasures: tools, theory, and practice*. 2016. [cited 2021-11-17]. Dissertation. Université Paris sciences et lettres. Available from: https://tel.archives-ouvertes.fr/tel-01762404/document.

[30] KORKIKIAN, R., PELISSIER, S., and NACCACHE, D. Blind Fault Attack against SPN Ciphers. In: *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2014, pp. 94–103 [cited 2021-11-28]. DOI: 10.1109/FDTC.2014.19.

[31] KÜÇÜK, K. A., PAVERD, A., MARTIN, A., ASOKAN, N., SIMPSON, A. et al. Exploring the use of Intel SGX for Secure Many-Party Applications. In:. December 2016, pp. 1–6 [cited 2022-03-17]. DOI: 10.1145/3007788.3007793.

[32] LEE, H., LEE, K., and SHIN, Y. AES Implementation and Performance Evaluation on 8-bit Microcontrollers. *CoRR*. 2009, abs/0911.0482, [cited 2021-12-10]. Available from: http://arxiv.org/abs/0911.0482.

[33] LENSTRA, A. K. Memo on RSA signature generation in the presence of faults. 1996, [cited 2021-11-30]. manuscript. Available from: http://infoscience.epfl.ch/record/164524.

[34] MARTINÁSEK, Z. *Cryptanalysis using side channels*. 2013. [cited 2021-11-14]. 129 p. Dissertation. Brno University of Technology, Faculty of Electrical Engineering and Communication Technologies. Available from: https://www.vut.cz/www_base/zav_prace_soubor_verejne.php?file_id=62844.

[35] METULA, E. Chapter 9 - Defending against MCRs. In: METULA, E., ed. *Managed Code Rootkits*. Boston: Syngress, 2011, pp. 261–290 [cited 2022-03-15]. DOI: https://doi.org/10.1016/B978-1-59749-574-5.00009-X. ISBN 978-1-59749-574-5. Available from: https://www.sciencedirect.com/science/article/pii/B978159749574500009X.

[36] MINNI, R., SULTANIA, K., MISHRA, S., and VINCENT, D. R. An algorithm to enhance security in RSA. In: *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*. 2013, pp. 1–4 [cited 2021-12-02]. DOI: 10.1109/ICCCNT.2013.6726517.

[37] MOHAMED, A. A. and MADIAN, A. H. A modified Rijndael algorithm and its implementation using FPGA. In: *2010 17th IEEE International Conference on Electronics, Circuits and Systems*. 2010, pp. 335–338 [cited 2021-12-03]. DOI: 10.1109/ICECS.2010.5724521.

[38] MORO, N., DEHBAOUI, A., HEYDEMANN, K., ROBISSON, B., and ENCRENAZ, E. Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In: *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2013, pp. 77–88 [cited 2021-11-28]. DOI: 10.1109/FDTC.2013.9.

[39] MORO, N., HEYDEMANN, K., DEHBAOUI, A., ROBISSON, B., and ENCRENAZ, E. Experimental evaluation of two software countermeasures against fault attacks. In: *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 2014, pp. 112–117 [cited 2021-11-29]. DOI: 10.1109/HST.2014.6855580.

[40] MURDOCK, K., OSWALD, D., GARCIA, F. D., VAN BULCK, J., GRUSS, D. et al. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In: *41st IEEE Symposium on Security and Privacy (S&P'20)*. 2020 [cited 2021-10-25].

[41] NGABONZIZA, B., MARTIN, D., BAILEY, A., CHO, H., and MARTIN, S. TrustZone Explained: Architectural Features and Use Cases. In:. November 2016, pp. 445–451 [cited 2022-03-23]. DOI: 10.1109/CIC.2016.065.

[42] OTTO, M. *Fault Attacks and Countermeasures*. 2004. [cited 2021-11-13]. Dissertation. University of Paderborn, Faculty of Electrical Engineering, Computer Science and Mathematics. Available from: https://digital.ub.uni-paderborn.de/ubpb/urn/urn:nbn:de:hbz:466-20040101308.

[43] PADATE, R., PATEL, A., and RODRIGUES, C. Encryption and Decryption of Text using AES Algorithm. In:. 2014 [cited 2021-12-11].

[44] PINTO, S. and SANTOS, N. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.* New York, NY, USA: Association for Computing Machinery. Jan 2019, vol. 51, no. 6, [cited 2022-03-23]. DOI: 10.1145/3291047. ISSN 0360-0300. Available from: https://doi.org/10.1145/3291047.

[45] PINTO, S., ARAUJO, H., OLIVEIRA, D., MARTINS, J., and TAVARES, A. Virtualization on TrustZone-Enabled Microcontrollers? Voilà! In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2019, pp. 293–304 [cited 2022-03-20]. DOI: 10.1109/RTAS.2019.00032.

[46] PIRET, G. and QUISQUATER, J.-J. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad. In: WALTER, C. D., Koç, Ç. K. and PAAR, C., eds. *Cryptographic Hardware and Embedded Systems - CHES 2003*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 77–88 [cited 2021-11-15]. ISBN 978-3-540-45238-6.

[47] QIU, P., WANG, D., LYU, Y., and QU, G. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-Core Frequencies. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 195–209 [cited 2022-04-20]. CCS '19. DOI: 10.1145/3319535.3354201. ISBN 9781450367479. Available from:

[48] RIVEST, R. L., SHAMIR, A., and ADLEMAN, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. Feb 1978, vol. 21, no. 2, pp. 120–126, [cited 2021-12-03]. DOI: 10.1145/359340.359342. ISSN 0001-0782. Available from:

[49] RIVIÈRE, L., NAJM, Z., RAUZY, P., DANGER, J.-L., BRINGER, J. et al. High precision fault injections on the instruction cache of ARMv7-M architectures. In: *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2015, pp. 62–67 [cited 2021-11-29]. DOI: 10.1109/HST.2015.7140238.

[50] ROSCIAN, C., SARAFIANOS, A., DUTERTRE, J.-M. and TRIA, A. Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells. In: *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2013, pp. 89–98 [cited 2021-11-28]. DOI: 10.1109/FDTC.2013.17.

[51] SHEPHERD, C., AKRAM, R. N., and MARKANTONAKIS, K. Establishing Mutually Trusted Channels for Remote Sensing Devices with Trusted Execution Environments. In:. August 2017 [cited 2022-03-16]. DOI: 10.1145/3098954.3098971.

[52] TANG, A., SETHUMADHAVAN, S., and STOLFO, S. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 1057–1074 [cited 2021-11-09]. ISBN 978-1-931971-40-9. Available from: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

[53] TUNSTALL, M., MUKHOPADHYAY, D., and SUBIDH ALI, S. Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. In:. January 2011, pp. 224–233 [cited 2021-12-18].

[54] WANG, J., HONG, Z., ZHANG, Y., and JIN, Y. Enabling Security-Enhanced Attestation With Intel SGX for Remote Terminal and IoT. *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems. 2018, vol. 37, no. 1, pp. 88–96, [cited 2022-05-11]. DOI: 10.1109/TCAD.2017.2750067.

[55] WU, C.-H., HONG, J.-H., and WU, C.-W. RSA cryptosystem design based on the Chinese remainder theorem. In: *Proceedings of the ASP-DAC 2001. Asia and South Pacific Design Automation Conference 2001 (Cat. No.01EX455)*. 2001, pp. 391–395 [cited 2021-12-02]. DOI: 10.1109/ASPDAC.2001.913338.

[56] YUCE, B., SCHAUMONT, P., and WITTEMAN, M. Fault Attacks on Secure Embedded Software: Threats, Design and Evaluation. *CoRR*. 2020, abs/2003.10513, [cited 2021-11-09]. Available from: https://arxiv.org/abs/2003.10513.

[57] ZUSSA, L., DUTERTRE, J.-M., CLEDIERE, J., and ROBISSON, B. Analysis of the fault injection mechanism related to negative and positive power supply glitches using an on-chip voltmeter. In: *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 2014, pp. 130–135 [cited 2021-11-17]. DOI: 10.1109/HST.2014.6855583.

Appendix A

Contents of the attached storage media

```
/
├── auto_reset_PC_RPI
│   ├── Readme.md reset_PC.py
│   └── scheme_reset.png
├── PlunderVolt
│   ├── data
│   ├── Readme.md
│   └── RSA_success.py
├── test_scipt
│   ├── Readme.md
│   └── test_PV_VP.sh
├── thesis
│   ├──  thesis.pdf
│   └──  thesis.zip
├── TrustZone_meh
│   ├── Arduino Atmel_studio
│   ├── data_power_analysis
│   ├── data_mul_error.ods
│   ├── Readme.md
│   ├── tz_meh_hw_scheme.png
│   └── tz_meh_hw_scheme_2.png
└── VoltPillager
    ├── data
    └── Readme.md
```

- The **auto_reset_PC_RPI** directory contains a script for automatically restarting a PC using Raspberry PI 4.0 in case of a freeze. In addition, the directory contains instructions for connecting Raspberry PI to a PC, including a wiring diagram.

- The **PlunderVolt** directory contains measured data from the PlunderVolt attack and a script for obtaining the success rate of key extraction from error data generated in RSA decryption.

- The **test_scipt** directory contains a script for performing VoltPillager and PlunderVolt attacks using the source codes for the attack. The script also collects data on success rates and saves the errors generated to files.
- The **thesis** directory contains the text of the thesis and the source files needed to create it.
- The **TrustZone_meh** directory contains:
  - The **Arduino** subdirectory containing attack firmware to trigger a multiplication error and an error that causes secure code to be considered insecure.
  - The **Atmel_studio** subdirectory contains firmware for SAM L11 to verify that the desired errors were successfully triggered by the attack hardware.
  - The **data_power_analysis** subdirectory contains data from the power analysis.
  - The **data_mul_error.ods** file contains data obtained from triggering a multiplication error in the TrustZone-M(eh) attack.
  - Instructions for compiling source codes and assembling attack hardware using the attached wiring diagrams.
- The **VoltPillager** directory contains measured data from the VoltPillager attack.

Appendix B

S-box substitution table

	00	**01**	**0**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0a**	**0b**	**0c**	**0d**	**0e**	**0f**
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Table B.1: S-box table[34]

Appendix C

VoltPillager board including extensions

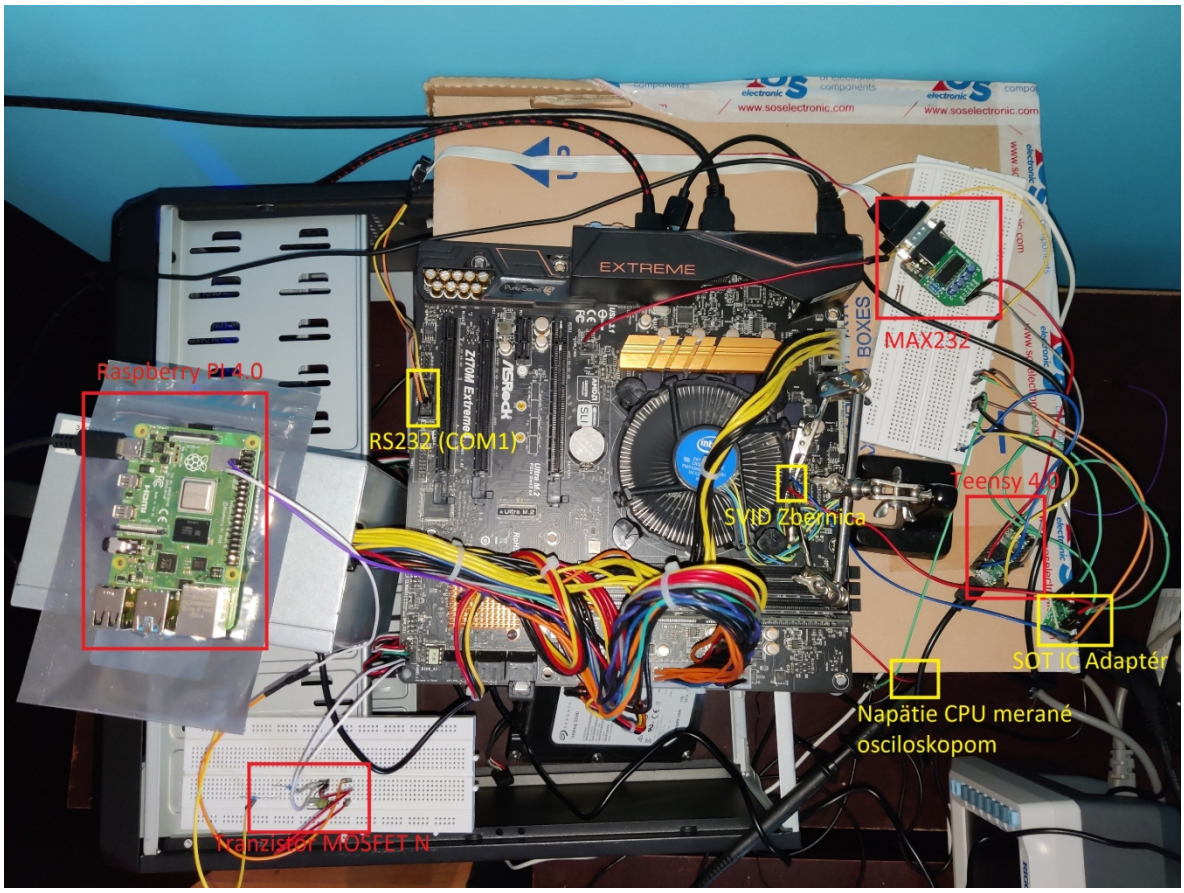

Fig. C.1: Connecting the VoltPillager board and Raspberry PI to a PC

Appendix D

Connection options for performing a TrustZone-M(eh) attack

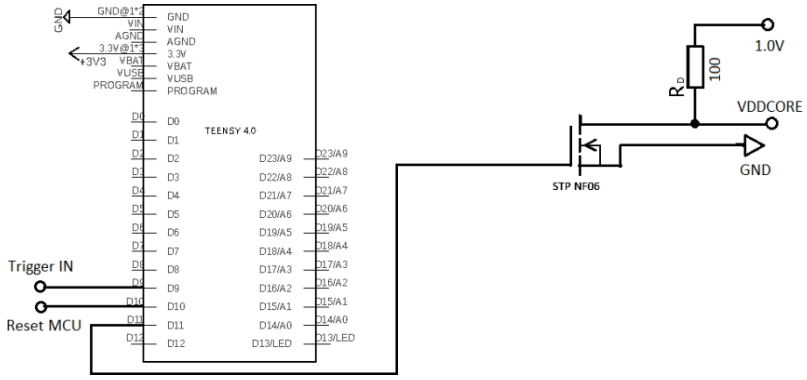

Fig. D.1: Option 1 – Connection diagram for performing a TrustZone-M(eh) attack using a transistor

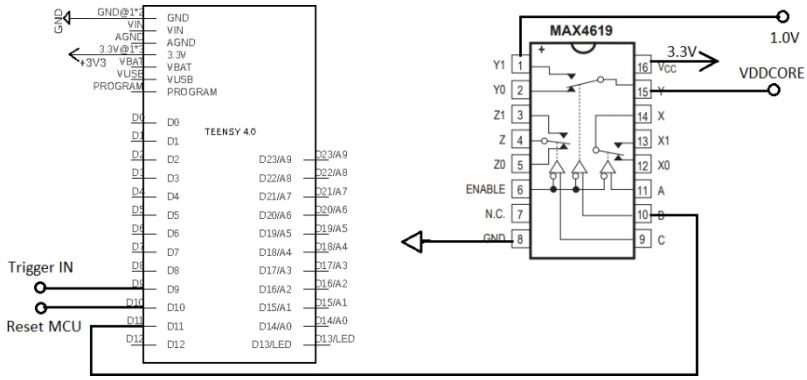

Fig. D.2: Option 2 – Connection diagram for performing a TrustZone-M(eh) attack using a multiplexer