Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Telecommunication Engineering

Secure Firmware Upgrade of Embedded Platform

Jan Šimůnek

Supervisor: Ing. Bc. Marek Neruda, Ph.D.
Supervisor–specialist: Ing. Tomáš Zitta
Field of study: Communications, Multimedia, Electronics
Subfield: Network and Information Technology
May 2018

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Šimůnek Jan**	Personal ID number:	**457155**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Telecommunications Engineering**		
Study program:	**Communications, Multimedia, Electronics**		
Branch of study:	**Network and Information Technology**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Secure Firmware Upgrade of Embedded Platform

Bachelor’s thesis title in Czech:

Bezpečný upgrade firmware embedded platformy

Guidelines:

Design a process procedure for embedded platform firmware management (secure upgrade). Implement and test secure firmware transfer from remote storage to the embedded platform, including firmware authentication. Document the proposed solution.

Bibliography / sources:

[1] Kvarda, L.; Hnyk, P.; Vojtěch, L.; Lokaj, Z.; Neruda, M.; Zitta, T.: Software Implementation of Secure Firmware Update in IOT Concept. In: Advances in Electrical and Electronic Engineering. 2016, vol. 14, iss. 4, pp. 389-396.
[2] EEE, Internet of Things (IoT) Security and Privacy Best Practices, February 2017. Dostupný na http://internetinitiative.ieee.org/images/files/resources/white_papers/internet_of_things_feb2017.pdf [on-line]

Name and workplace of bachelor’s thesis supervisor:

Ing. Marek Neruda, Ph.D., Department of Telecommunications Engineering, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Ing. Tomáš Zitta, Department of Telecommunications En, FEE

Date of bachelor’s thesis assignment: **04.01.2018** Deadline for bachelor thesis submission: **25.05.2018**

Assignment valid until: **30.09.2019**

Ing. Marek Neruda, Ph.D.	Head of department’s signature	prof. Ing. Pavel Ripka, CSc.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt   Student’s signature

Acknowledgements

I would like to thank my supervisor Ing. Bc. Marek Neruda, Ph.D. and supervisor-specialist Ing. Tomáš Zitta for their kind guidance and patience. They supported me, provided me with valuable advice and always steered me to the relevant source of information.

I must express my gratitude to my parents, sister, other family members and my girlfriend for supporting me not only during my studies but throughout my whole life.

Declaration

I hereby declare that the presented work was carried out independently and that I have listed all information sources used in accordance with the Methodical Guidelines on Maintaining Ethical Principles During the Preparation of Higher Education Theses. Furthermore, I declare that the borrowing and publishing of my thesis or part of it is allowed with the agreement of department.

In Prague, 25. May 2018

. . . .

Abstract

This bachelor thesis deals with the design of process procedure for embedded platform firmware management (secure upgrade). The theoretical part consists of characteristics of vulnerabilities in embedded systems and design of commonly used secure upgrade procedure. Further, the required components for the design of secure upgrade are described. The practical part specifies an advanced approach for secure upgrade with firmware OpenWrt. The proposed procedure is described and tested on embedded platform Vocore2 to provide the secure and authenticated transfer of firmware to the device. Testing is measured to evaluate how the procedure is computationally demanding.

Keywords: firmware, upgrade, embedded platform, OpenWrt, Vocore2, AES

Supervisor: Ing. Bc. Marek Neruda, Ph.D.

Abstrakt

Tato bakalářská práce se zabývá návrhem procesního postupu bezpečného upgrade firmware pro embedded platformu. Teoretická část obsahuje popis soudobých bezpečnostních hrozeb pro embedded systémy a popisuje současné postupy upgrade firmware pro tyto zařízení. Dále jsou zde také představeny podklady pro návrh bezpečného upgrade firmware. V praktické části je popsán pokročilý postup, který využívá firmware OpenWrt a embedded platformu Vocore2. Procesní postup je vysvětlen a otestován na platformě Vocore2 s cílem bezpečného přenosu a autentizace firmware do zařízení. Pro zjištění výpočetní náročnosti tohoto procesu je testování změřeno a vyhodnoceno.

Klíčová slova: firmware, upgrade, embedded platform, OpenWrt, Vocore2, AES

Překlad názvu: Bezpečný upgrade firmware embedded platformy

Contents

Figures

Tables

0.1 List of Abbreviations

ACK	Acknowledgement
AES	Advanced Encryption Standard
AMT	Active Management Technology
ANSI	American National Standards Institute
AP	Access Point
ARM	Advanced Reduction Instruction Set Computer Machine
BC	Before Christ
CA	Certification Authority
CBC	Cipher Block Chaining
CC	Control and Command server
CCM	Counter with Cipher Block Chaining Message Authentication Code Mode
CIA	Confidentiality, Integrity, Authenticity
CISC	Complex Instruction Set Computer
CPS	Cyber-Physical Systems
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSPRNG	Cryptographically Secure Pseudorandom Number Generators
CTR	Counter Mode
DDoS	Distributed Denial-of-Service Attack
DES	Data Encryption Standard
DNS	Domain Name System
DVR	Digital Video Recorder
e.g	exempli gratia (for example)
ELF	Executable and Linking Format
GCM	Galois/Counter Mode
GPU	Graphics Processing Unit
grep	Global Regular Expression Print
HP	Hewlett-Packard
ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
IDS	Intrusion Detection System
IoT	Internet of Things
IP	Internet Protocol

IPS	Intrusion Prevention System
ISP	Internet Service Provider
IV	Initialization Vector
JFFS2	log-structured file system
JSON	JavaScript Object Notation
LED	Light-Emitting Diode
LEDE	Linux Embedded Development Environment
LPWAN	Low-Power Wide-Area Network
MAC	Message Authentication Checksum
MD5	Merkle–Damgård hash function
ME	Management Engine
MINIX	mini-Unix
MIC	Message Integrity Check
MITM	Man in the Middle Attack
MIPS	Microprocessor without Interlocked Pipeline Stages
mtd	memory technology device
NAND	Not And (electronic logic gate)
NFQUEUE	Netfilter Queue
NFI	New Firmware
Nmap	Network Mapper
nonce	number that can only be used once
NOR	Not Or (electronic logic gate)
OISF	Open Information Security Foundation
OS	Operating System
PC	Personal Computer
PCB	Printed Circuit Board
PID	Process Identification
PRN	Pseudo Random Number
PRNG	Pseudorandom Number Generators
RAM	Random-Access Memory
RFID	Radio-Frequency Identification
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RSA	Rivest–Shamir–Adleman
S-Box	Substitution-Box
SD	Secure Digital

SHA	Secure Hash Algorithm
SoC	System on a Chip
SOC	Security Operations Centers
SPI	Serial Peripheral Interface Bus
SquashFS	compressed read-only file system
SNO	Selected Numbers Orders
SSD	Solid-State Drive
SSH	Secure Shell
STA	Station
SYN	Synchronization
sysupgrade	system upgrade
TAM	Total Available Market
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
top	Table of Processes
TRNG	True Random Number Generators
USB	Universal Serial Bus
UDP	User Datagram Protocol
VSZ	Virtual Memory Size
WAN	Wide Area Network
XOR	Exclusive Or (electronic logic gate)

Chapter 1

Introduction

Internet of Things (IoT) is a popular theme which is regarded as the future remarkable technology. It is considered a network consisting of physical devices especially embedded platforms. These devices can communicate and send data among themselves in order to monitor or control some activities. The term IoT is also defined as a collection of embedded systems with sensors that are connected to the Internet [1]. IoT market rapidly grows due to developments in hardware, increased availability, reduced cost and high demand.

The fourth industrial revolution known as Industrial 4.0 includes IoT as well. Together with Cyber-Physical Systems (CPS), factories will expand in a more complex environment with the ability to be more efficient. However, relatively purely protected devices are effortless victims of hackers. Security level of devices is obviously underestimated in many cases and an increasing number of successfully performed cyber attacks is the reason to consider. In 2015, Hewlett Packard conducted a research and 100 % of home IoT devices were marked as significantly vulnerable [2]. The vulnerabilities formed weak passwords, lack of encryption while communicating through network and an account enumeration.

In the theoretical part of this thesis, vulnerabilities of embedded systems are mentioned. As a result of purely applied security on embedded devices, various cyber attacks against the embedded systems are committed. These attacks are described to understand an alarming condition of most embedded systems. To protect the systems involved in the IoT, an unsolicited modification of devices cannot be allowed. Hence, the secure upgrade firmware has to be accomplished. In the practical part of this thesis, a process of a secure firmware upgrade using an encryption and a monitoring of network traffic is introduced. The whole process is described in detail and measured to provide a level of consumption of computing resources.

Part I

Theoretical Part

Chapter 2

Embedded Systems

Embedded systems create an essential part of the world of IoT. The devices are small-sized computers with dedicated functions. They are widely used to collect data, monitor environment and communicate with each other. The systems are hardware optimized for predefined utilization and they are regarded as low power computer devices. The low power consumption and high availability are the leading advantages.

The first made estimation of the whole number of IoT was made by Ericsson and the company predicts 50 billion devices in 2020 [3]. However in 2017, more precise estimation of the Gartner company was made. They examines to be 8.4 billion devices in 2017. The number excludes smartphones, tablets, and computers. The Gartner company also predicts 20.4 billion devices in 2020. In 2017, the consumer segment forms 63 %. The application mostly used by consumers are smart TVs and many automotive systems. The rest of 37 % consists of enterprise devices especially used for monitoring and data collecting like security cameras or smart electric meters.

Most companies are involved in a development of the embedded systems which are considered as the major products in the growing IoT market [4]. Nevertheless, embedded systems are currently not as embedded as they used to be. Higher computer performance, components shrinking and lower price influenced nowadays form of these devices. Embedded systems still tend to be dedicated. However, many functions are added. A cell phone is one of the finest examples of evolution in the embedded systems. The main idea of this invention was using voice calls and text messages as a form of communication. Through time, the cell phone converged into a smartphone – device covering an extended number of provided services as a music player, digital camera or datalogger.

2.1 Firmware of Embedded Systems

A firmware is a crucial software running on devices. It is located between hardware and an application layer. The main purpose of firmware is to connect hardware resources with the running processes. Generally, the firmware could be divided into an embedded firmware and an operating system-based firmware.

2.1.1 Embedded Firmware

Embedded systems are hardware limited devices using custom-build firmware. The firmware tends to be cost-effective and closely written to interact with hardware which could be a difficult task for developers. Developers need to have deep knowledge of hardware. Moreover, they also assemble software that cooperates with a user interface. Directly written embedded firmware targets devices with the minimum maintenance and the firmware is rarely upgraded.

2.1.2 Operating System-Based Firmware

As the embedded systems become more complex, demand for more complicated software managing more capabilities has grown. Very similar process as in personal computer evolution could be seen in the embedded devices. Computing power of devices grows and an urgency in using more complicated software is required. An operating system can deliver to user more than just the embedded firmware. The most popular choice for manufacturers is a stripped down version of the Unix-like operating system – BusyBox [5]. It uses the most useful capabilities found in the Unix system in a single executable file. In summary, more than 300 commands could be found in the BusyBox [6]. On the other hand, more complex firmware with more functionalities increases the possibility of vulnerabilities released in the firmware. Maintenance of these devices is necessary and a firmware upgrade should be released frequently. The following diagram shows a survey conducted by VDC Research company in 2017 [7]. It covers representation of operating systems across the embedded and IoT market.

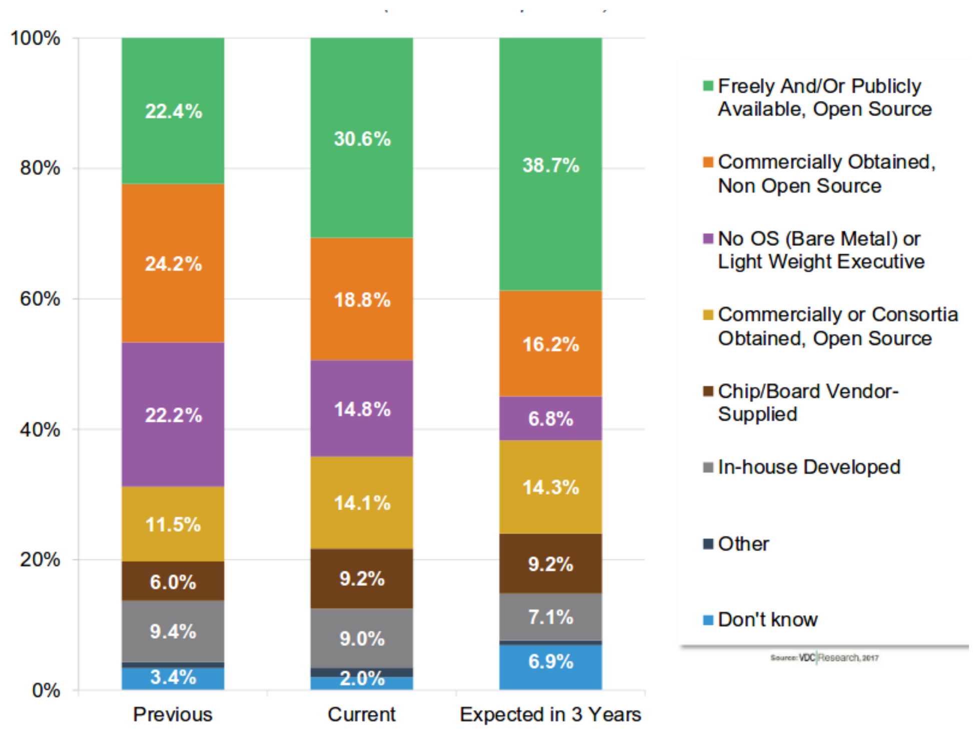


Figure 2.1: Previous, Current, and Expected Primary OS, by OS Source [7]

2.2 Architecture of Embedded Systems

A scalability is necessary while designing embedded systems. The design of platform is divided into three sections – a hardware layer, a system software layer and an application software layer. The hardware layer consists of whole components assembled on a printed circuit board (PCB). The system software layer covers device drivers communicating with the hardware of an embedded device. The application layer is an upper layer commonly known as an userspace. The most running applications are executed on the application layer which interacts with the user. To design the embedded platform some high-level capabilities should be characterized.

- Data acquisition and control
- Data processing and storage
- Connectivity
- Power management

Data acquisition refers to measuring real-world conditions and converting them into a readable digital or analog output. Sensors are the input components involved in measuring physical variables. Physical variables are converted to electrical signals. The sensors can measure a lot of physical variables including temperature, pressure, humidity, smoke, speed, proximity and others. One of the most important characteristics of sensors is the resolution which helps to measure the variable. The output is represented by an appropriate technology which human can interact with, e.g. LEDs, screens, speaker.

The embedded systems involved in IoT require data processing and storage capabilities to analyze or store collected data. The processes could be executed directly on a device. It is also possible to send data to other devices, services or applications like cloud services. This is so called centralized analysis. An analysis of data is provided by another device or service. On the other hand, an edge analysis is done on the embedded device collecting data. It could discard unused data and reduce processing and output in the form of upstream. For this purpose, more processing capabilities and larger storage is needed.

A network connectivity is necessary for an interaction of devices with each other. An operating range and a volume of transmitted data have to be considered. The devices can communicate wirelessly using 802.11 set of specifications, RFID (Radio-Frequency Identification) or LPWAN (Low-Power Wide-Area Network) like LoRa and SigFox technologies. When designing the device it must be considered that a power consumption differs with regard to used technology.

A portable embedded systems rely on batteries or on power sources like solar. The devices tend to be economical. However, depending on power requirements like used sensors, computing power, data storage or network capabilities, suitable power source has to be added. For example, Raspberry Pi 3 has the power consumption around 700 – 1000 mA of current. With every sensor added, consumption is increased, e.g. using a camera module requires extra 250 mA [8].

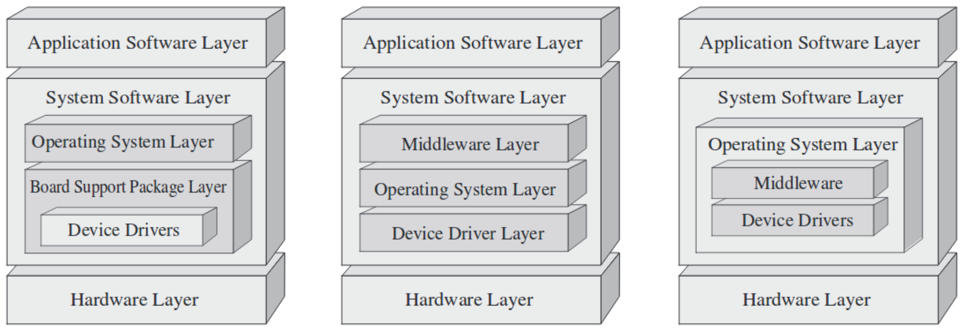


Figure 2.2: Embedded System Model [9]

2.2.1 System on a Chip

A system on a Chip (SoC) integrates a microcontroller or a microprocessor with a graphical processor unit (GPU) or a Wi-Fi module. A Reduced Instruction Set Computing (RISC) is widely spread microprocessor architecture in the embedded devices. An Instruction Set is reduced and consists of fewer transistors. Moreover, power consumption is lower than using Complex Instruction Computing (CICS) architecture. CISC is mostly used in personal computers (PCs) and x86 architecture. The leading position on the market of the microprocessor RISC belongs to MIPS (Microprocessor without Interlocked Pipeline Stages) and ARM (Advanced RISC Machine) architecture. A diversity of microprocessor could be found in the widely known commercial embedded systems as Raspberry Pi 3 using ARM architecture. On the other hand, a computer with higher computing performance such as Up Board is based on x86 architecture and small-sized router Vocore2 used MIPS architecture. Every embedded platform has almost unique hardware design. Firmware for each device is designed to cooperate only with the desired platform. As a result of complex development for each type of device, firmware errors could occur. It happens more frequently comparing to the microprocessor from Intel and AMD used in PCs. The following diagram shows representation of architectures used in the embedded systems with the total available market (TAM) in 2013 [10].

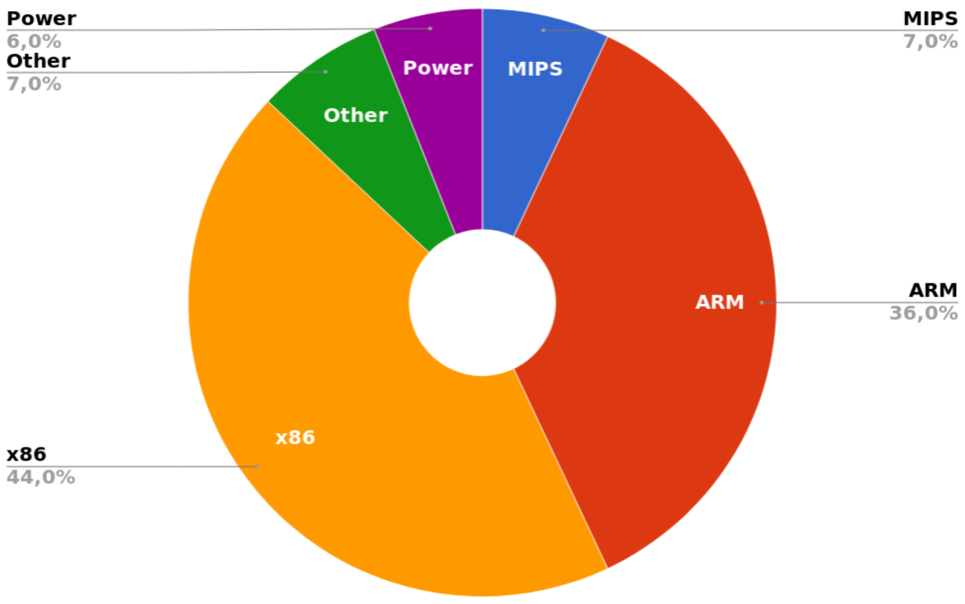


Figure 2.3: The embedded systems market with $11.3B of total available market in 2013 [10]

2.2.2 Memory Management

The most used memory in embedded devices is a flash memory. It is a non-volatile memory organized in blocks. The blocks are filled with data separately and independence of block is provided. If data have to be removed, only blocks containing data will be cleared. The rest of data remain invulnerable. This is an advantage comparing to the older technology ROM (Read Only Memory) which needs to be fully removed to store new data. The flash memory uses two technologies NOR (not or - electronic logic gate) and NAND (not and - electronic logic gate). The memory storage of NOR is usually 4 – 16 Mbit. The newest NAND obtains 32 – 256 Mbit [11]. Two methods of connecting flash memory to SoC are devised.

- Raw flash (host managed)
- Flash translation level (self-managed)

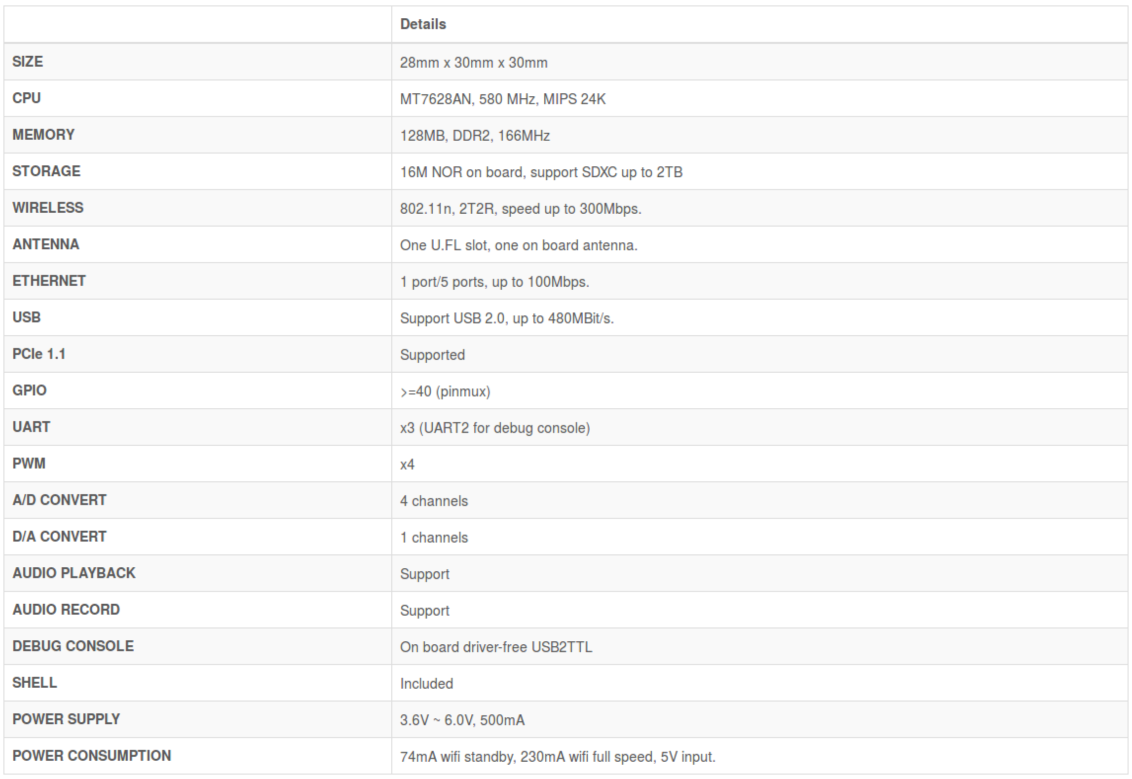
A memory directly connected to SoC is known as host managed. If memory is connected via a controller to SoC, self-managed is used instead. The embedded systems solely use host-managed. An example of self-managed is connection of SSD (Solid-State Drives) to PC.

Chapter 3

Vocore2

For the testing purposes in the practical part of this thesis, the testing device needs to be chosen. The main criteria are the computing performance which represents general embedded platform and an ability to run the embedded firmware OpenWrt. The embedded platform Vocore2 was chosen as the best candidate. It is a small-sized device with an advanced computing performance and it runs embedded firmware OpenWrt. Other candidates were Up Board and Raspberry Pi. However, these candidates were not satisfactory. The Up Board gives powerful computing performance which is much higher than a typical embedded platform and the device can be almost compared with the computing performance of PCs. Moreover, the Up Board does not support OpenWrt and a firmware management is already ensured by the default operating system Ubuntu or Windows. The Raspberry Pi is considered a device for testing purposes and it can run OpenWrt. Nevertheless, the system uses the SD card as the primary memory storage. To upgrade the device, SD card has to be formatted and physically accessible. This means that the remote upgrade is not possible and only the Vocore2 is the suitable candidate.

The Vocore is a small-sized Linux computer developed by Qin Wei, Tong Wu and Thomas Hommers in China in October 2014. It is a low-cost computer running OpenWrt. There are some modifications of Vocore such as Vocore, Vocore2, Vocore2 Lite or Vocore2 Ultimate. The Vocore2 Ultimate is the most expensive modification costs $44.99 and this modification is used for an experimentation. It provides additional modules covering USB, Ethernet and more computing performance comparing older versions. The complete specification can be seen below.



	Details
SIZE	28mm x 30mm x 30mm
CPU	MT7628AN, 580 MHz, MIPS 24K
MEMORY	128MB, DDR2, 166MHz
STORAGE	16M NOR on board, support SDXC up to 2TB
WIRELESS	802.11n, 2T2R, speed up to 300Mbps.
ANTENNA	One U.FL slot, one on board antenna.
ETHERNET	1 port/5 ports, up to 100Mbps.
USB	Support USB 2.0, up to 480MBit/s.
PCIe 1.1	Supported
GPIO	>=40 (pinmux)
UART	x3 (UART2 for debug console)
PWM	x4
A/D CONVERT	4 channels
D/A CONVERT	1 channels
AUDIO PLAYBACK	Support
AUDIO RECORD	Support
DEBUG CONSOLE	On board driver-free USB2TTL
SHELL	Included
POWER SUPPLY	3.6V ~ 6.0V, 500mA
POWER CONSUMPTION	74mA wifi standby, 230mA wifi full speed, 5V input.

Table 3.1: Vocore2 Ultimate specification [12]

Chapter 4

IoT Security Threats

The concept of IoT is based on connected embedded systems through WAN (Wide Area Network) or LPWAN [13]. A network is considered as an untrusted area and security requirements need to be accomplished. Known vulnerabilities such as open TCP/UDP ports, open serial ports or use of poor password should be protected. Therefore, the IoT devices are hardware limited and strength of security could be insufficient. An increasing demand for IoT causes releases of vulnerable devices. Companies tend to supply customers as quickly as possible due to high demand and revenue. However, this is the reason for deploying undeveloped products containing critical errors. Pre-testing is underestimated, and critical errors could be accessible for cyber attack causing steal of data or loss of control.

High probability of already released and corrupted firmwares is a complication for manufacturers and a reaction has to be made. It is also worth saying, that manufacturers' revenues come from the sale of devices, not from the maintenance. Hence, if problems are not alarming, updates are not in the main company's attention. Nevertheless, a frequent upgrade to a new improved firmware is the finest solution. Redistributing new firmware into an enormous amount of devices is not possible. Instead of physical access to a device, remote access for a frequent upgrade is necessary. Transfer of data over untrusted area like WAN or LPWAN is secure exposure. A traffic needs to be protected with an adequate mechanism in the mean of providing services for the vast number of devices. The following examples are the most frequent problem of using IoT devices.

- weak passwords
- lack of encryption
- backdoors
- Internet exposure

Weak passwords are in the most cases connected with using default credentials. It is very important not to use default passwords. An encryption is also a significant part of embedded systems, because not every manufacturer releases firmware supporting cryptography. The reason for developing backdoors in the released devices is that devices can be easily supported by the manufacturer. A default user or an open port on the device is the most often case. Checking open ports on a device is the base rule to secure a device from easily accessible ports like Telnet. A scanner of potentially vulnerable devices could also be found as online tools like Shodan or IoT scanner Bull Guard. Finally, device exposed to the Internet which accepts incoming traffic could very likely be a target of an attacker. If previous points will not be underestimated, the device could be strongly resistant against attacks. However, the embedded devices covering the world of IoT and collected data from an environment should not be directly exposed to the Internet.

4.1 Security Incidents

Not surprisingly, embedded systems became an alluring playground for spreading malicious software. The extensive number of devices is dispersed around the world and attracts attackers to use them for their main goal – assembling botnets to perform Distributed Denial of Service (DDoS) attack on a predefined target. In the early stage, Botnets flooded their target via UDP, TCP or HTTP protocols. According to researchers at IBM's X-force, new attempts of Botnets namely ELF/Mirai were aimed to use victims for mining cryptocurrency [14]. It turned out to be effective only with cooperation with every node acting as one miner consortium. Another example is an attack aiming bootloaders of embedded system like UbootKit, which aims to modify bootloader. If the attempt is successful, the device can load a malicious firmware or can be filled with junk data to cause denial of service like IoT Brickerbot [15]. Complex analysis of IoT botnets is useful tool to avoid vulnerabilities in a security design.

4.2 Botnet Landscape

The supremacy of botnets belongs to botnet Mirai. His presence was recorded in 2016 for the first time. Mirai is classified as Executable and Linkable Format multi-platform worm also known as ELF Linux/Mirai. According to estimations, more than 1.5 million IoT devices were involved in Mirai botnet after the publication of his source code. Mirai takes part in the most catastrophic DDoS attack targeted on computer security journalist Brian Krebs's website with the peak of 620 Gbps. Most of the IP addresses were put together worldwide and most of the bots were DVRs (Digital Video Recorder), WebIP cameras on BusyBox or purely secured Linux servers. Afterwards, when the source code was published, more derivation of Mirai occurred such as BrickerBot or LuaBot [16] .

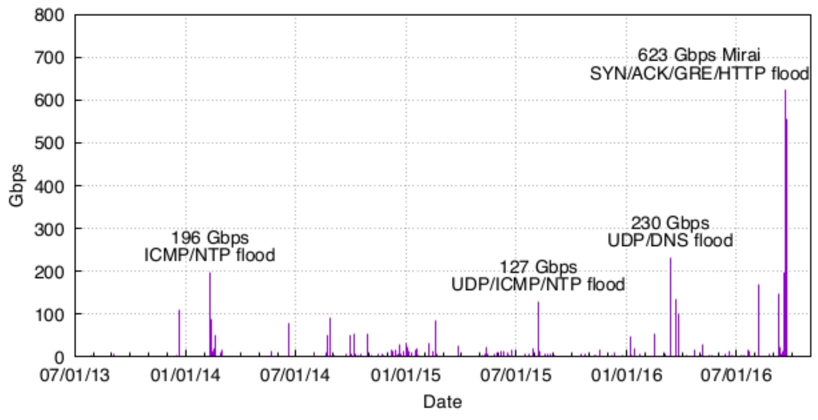


Figure 4.1: Historical DDoS Attacks Targeting Krebs on Security [17]

4.2.1 Infiltration and Infection

The process of spreading malware software begins by scanning random public IP addresses followed by executing port scanning. It is usually the most vulnerable Telnet port 23 (97 % of attempts). Some Mirai variants also targeted to TCP ports 20, 21, 2323 or 7547 which is well-known as ISP's remote management tool for customers broadband routers. Another Mirai descendant Radware aims at purely protected SSH port 22. Basically, malicious malware discovers services running on a device. Afterwards, it performs brute-force-attack by guessing passwords based on the hard-coded list to gain remote access to the device. In case of guessing correct credentials and gaining shell access, report message with credentials is sent to the report server. Having access to the device, malware starts a network lookup and decides whether spreading to other devices is possible or not. The malware attempts to execute commands wget or TFTP which are able to download binary data. Afterwards, it checks for writable parts of memory executing *cat /proc/mounts* and hardware platform information *cat proc/cpuinfo*. Next, the binary data are executed and removed immediately. The procedure initializes a new bot (device under the control of attacker) which is able to resolve IP address of Control and Command server (CC). An infected device is then controlled by CC and can receive command including attack types, attack duration or list of targets. It ought to be mentioned that the malware examines system processes (PIDs) and kills all processes using remote access. The reason for process killing is an elimination of other competitive malicious software related to Mirai. The newly initialized bot has the ability to form application and network layer flood attack such as SYN (Synchronization), ACK (Acknowledgement), UDP (User Datagram Protocol) or HTTP (Hypertext Transfer Protocol) floods.

4.2.2 Detection of Malware

An early stage of infection could be recognized by monitoring ports 23, 2323 and 22. These ports are assaulted with recursive authorization attempts. Another pattern could be identified with enormous TCP packets streams on an egress port of a device. According to experiments with Mirai – infected Raspberry Pi 3, SYN flooding produces 1 500 000 SYN packets in a minute. Fortunately, packet fields are mostly not randomized and monitored data are significant [18].

CWMP (28.30%)		**Telnet (26.44%)**		**HTTPS (19.13%)**		**FTP (17.82%)**		**SSH (8.31%)**	
Router	4.7%	Router	17.4%	Camera/DVR	36.8%	Router	49.5%	Router	4.0%
		Camera/DVR	9.4%	Router	6.3%	Storage	1.0%	Storage	0.2%
				Storage	0.2%	Camera/DVR	0.4%	Firewall	0.2%
				Firewall	0.1%	Media	0.1%	Security	0.1%
Other	0.0%	Other	0.1%	Other	0.2%	Other	0.0%	Other	0.0%
Unknown	95.3%	Unknown	73.1%	Unknown	56.4%	Unknown	49.0%	Unknown	95.6%

Table 4.1: Top Mirai Device Types - infected devices labeled by active scanning [17]

4.2.3 Prevention

The vulnerability of the systems is mainly caused by vendors. The shipped devices usually run remote administrator capabilities such as Telnet or SSH. Documentation is underestimated and customers are not accurately instructed to change default passwords or disable a remote access. Frequent firmware upgrade to apply patches is necessary and vendors should aware customers.

CWMP (28.30%)		**Telnet (26.44%)**		**HTTPS (19.13%)**		**FTP (17.82%)**		**SSH (8.31%)**	
Huawei	3.6%	Dahua	9.1%	Dahua	36.4%	D-Link	37.9%	MikroTik	3.4%
ZTE	1.0%	ZTE	6.7%	MultiTech	26.8%	MikroTik	2.5%		
		Phicomm	1.2%	ZTE	4.3%	ipTIME	1.3%		
				ZyXEL	2.9%				
				Huawei	1.6%				
Other	2.3%	Other	3.3%	Other	7.3%	Other	3.8%	Other	1.8%
Unknown	93.1%	Unknown	79.6%	Unknown	20.6%	Unknown	54.8%	Unknown	94.8%

Table 4.2: Top Mirai Device Vendors - infected devices labeled by active scanning [17]

4.3 Intel Management Engine

Intel Management Engine (Intel ME) is an accurate example of potential security vulnerabilities of features running below the kernel of the devices' main operating system [19]. Intel ME can be only found in embedded system with a higher computing performance.

Intel Corporation is the world's second-largest semiconductor chip maker and inventor of x86 architecture. It has the largest share in the microprocessor market of personal computers. Not surprisingly, Intel is interested in the growing market of IoT with their processors like Intel Quark, Intel Atom or 7^{th} generation of Intel Core. However, most of these chips have their own operating system based on MINIX 3, the Unix-like distribution adjusted in need of Intel. The purpose is to monitor a device and send data to a manufacturer. The main concern is that the whole activity is not visible to the user of the main operating system. The possibility of a remote attack is likely raised to a higher level. The first publicly released document about a critical vulnerability in a remote management tool Intel Active Management Technology (AMT) inserted in Intel ME appeared in May 2017. Some researchers from Positive Technologies discovered more vulnerabilities and published them in November 2017. As a reaction, Intel released information about additional bugs in Intel ME and confirmed affecting millions of endpoints and servers worldwide [20]. As a result, many hardware vendors like Dell or Purism disabled Intel ME from their systems. However, distribution of firmware upgrade to every vulnerable device will be a long-term event. Meanwhile, there will be numerous potential targets for attackers.

4.4 UbootKit: A Worm Attack for the Bootloader of IoT Devices

U-boot is a widely used open source bootloader implemented in most embedded systems. In 2017 Tencent Anti-Virus Laboratory demonstrated a new worm targeting bootloaders in IoT devices, to indicate propagation between variable infected devices using ARM and MIPS architecture. The worm tries to rewrite bootloader. Then the malicious code which existed in the bootloader is executed after rebooting. At the end of the process, worm downloads the whole part of itself and it will continue spreading to other devices by scanning and remote code execution exploits. Moreover, it is very complicated to remove the malicious worm, because it persists in bootloader and the worm is executed before the start of Linux kernel or any userspace code. Due to this fact, the worm receives the highest privileges and removing him becomes complicated [21].

4.4.1 Analysis

In most of the IoT devices, flash memory can be updated with root privilege. The reason is that firmware contains Linux kernel and file system needs privilege to be updated. The bootloader is situated at the beginning of the flash memory following firmware. Hence, it is possible to rewrite bootloader by the *mtd_write tool*. By modifying bootloader, a piece of a malicious code can be injected. The best opportunity for running the worm is before the Linux kernel takes control. The piece of instructions in Linux Kernel is modified and added to the location of init. Init is the first process while booting and its probable location is */etc/init.d*. If an auto-run shell script including the malicious code runs, it downloads the additional malicious binary file to launch and scan the network for vulnerable devices.

4.4.2 Bypass Security Methods

The first solution coming to mind could be resetting the device. However, formatting configuration in a file system will not affect malicious code implemented in the bootloader area. Verification of kernel image is also not sufficient. Uboot typically supports CRC checksum and in some cases, advanced bootloaders support even RSA signature verification of an image. On the other hand, when UbootKit attacks the bootloader, kernel image is not affected. It means that UbootKit can patch verification function to return True value constantly. Moreover, the kernel image will not be modified in the memory before booting. An injected script will be executed after the verification procedure. As the last possible protection can be considered a protection of the flash memory. Nevertheless, this protection is rather a method preventing unintentional writing instead of protection against attacks. Writing to flash memory is not difficult to unlock.

4.4.3 Prevention

In order to prevent UbootKit, it necessary to ensure that bootloader was not modified. The security method can be implemented on a chip before the infected bootloader is executed. To verify bootloader integrity, when the first boot is made, a hash value of bootloader is calculated and stored in flash memory. SHA256 hash algorithm can be used. When the device is then rebooted the hash value will be calculated on bootloader and it will be compared with the value obtained during the first boot. If a content of bootloader is altered, respond to IoT device should be made. The best solution is to abort process to avoid propagation of the worm attack.

Chapter 5

Recommendation for Securing of Embedded Systems

The firmware upgrade is the most critical [22] and weak phase for embedded systems. There will be millions of small embedded systems diffused across the world [23]. Researchers have observed that an advancement of IoT and a deploying of vulnerable devices will increase exponentially. In most cases, it is impossible to be physically touched. Devices need to be upgraded remotely which gives a playground for an unsolicited action [24]. Upgrades should be released frequently and the possibility to upgrade the device has to be accomplished. Otherwise, the device could be exposed to malicious code or in the worst cases to firmware modification attack. The firmware modification attack to HP Laser Jet was conducted by students from the Columbia University to scan network traffic. An infected printer scans a network and tries to expand further to another vulnerable device. It is also important to mention that signed firmware is not secure firmware. There is always the possibility to securely sign an image to verify if a firmware integrity and authenticity was not changed. However, a certainty that firmware was released without any critical security vulnerability is not possible to ensure [25].

- Crypto processors
- Signed bootloaders
- Crypto bootloaders
- Network traffic detection

One possibility of securing upgrade of firmware is to use Crypto processor or verified bootloaders. Crypto processors can help with the provision of the key. It is necessary to assure that the device is in the correct state and cannot be threatened. The main reason for using the Crypto processors is the use of a securely stored key instead of user-predefined passwords.

Initialization of ROM is the first process in the chain of booting and then ROM executes the bootloader. Using ROM, which is able to verify the signature of bootloader, provides assurance that bootloader was not customized by an attacker. To avoid attacks like UbootKit described in the chapter IoT Security Threats, signed bootloader is a solution.

Crypto bootloaders implemented in the microcontrollers are also an effective solution. To increase the security level, it verifies the signature of an image and decides whether the image is modified or not. Texas Instruments developed Crypto bootloaders for MSP microcontroller [26]. The implementation is very light occupying only 3.2 KB of code and less than 1 KB of data. Another example is Atmel which is not using bootloaders. However, they use Ethernet AES - GCM (Advanced Encryption Standard in Galois/Counter Mode) encryption to secure booting process of the firmware. [27].

The designs presented above are highly connected with the lowest software level of embedded systems. On the other hand, embedded systems are becoming more complex with more data capabilities and greater data processing. As a result, the operating system-based firmware can be run on devices. Introducing the IoT threats, most attacks try to scan devices and tend to find vulnerable ports, before the whole process of attack begins. Detection of an unauthorized access and a control of a system is the fundamental issue. Network traffic can be recorded through Intrusion Detection and Prevention Systems (IDS/IPS). Scientific community also studied data mining methods to establish behaviour patterns of constructed attacks [28]. It can help with the detection of an advanced packet flooding attacks which are constructed with an unpredictable algorithm to successfully flood the target device.

Chapter 6

Application of Cryptography

Communication through the network in presence of other parties has to be encrypted to ensure confidentiality, integrity and availability of data also known as the CIA triad. Applying the cryptography to transferred information is essential for every secure communication. Its history is dated around 2000 BC. As the first documented reference is considered secretly encrypted writing also known as Caesar cipher. This chapter introduces main principles of the cryptography used in the practical part. It covers encryption, authentication of data and key generating.

The cryptography is divided into Symmetric and Asymmetric. The symmetric cryptography is based on sharing the same key for two parties of communication. On the other hand, the asymmetric algorithms use private and shared public key for a process of decryption/encryption. For the purposes of the practical part, the symmetric cryptography will be described in more detail.

6.1 Symmetric Cryptography

The symmetric cryptography is divided into stream ciphers and block ciphers. The stream ciphers encrypt every bit individually, whereas block ciphers encrypt information to block in an identical length of bits. The result of a block cipher is that every bit of encrypted information is determined on the key and every bit contained in an entry block. In an early stage of the cryptography, the stream ciphers were considered as an agile solution for an easy implementation and speed of encryption or decryption. However, the advantage of the stream ciphers is nowadays omitted due to a computing power [29].

Definition 6.1. Let x_i be a bit of message, X be a block of message, K be a key and y_i be a bit of ciphertext.

- Stream cipher: $y_i = E(x_i, K)$
- Block cipher: $y_i = E_i(X, K)$

6.1.1 Stream Cipher

The classic digital stream cipher is Vernam cipher, developed in the 20^{th} century for encrypting telecommunication traffic. Encryption of every individual bit is achieved by adding a bit from a keystream to a plaintext bit. The encryption and decryption functions are the same procedure. Security of the stream ciphers completely depend on the keystream – values s_i. If an attacker solves one plaintext from the ciphertext, it is possible to deduce the keystream and decrypt communication.

Definition 6.2. Let x_i be a bit of message, s_i be a bit of a key stream, y_i be a bit of ciphertext, where $x_i, y_i, s_i \in \{0, 1\}$

- Encryption: $y_i = (x_i \oplus s_i)$
- Decryption: $x_i = (y_i \oplus s_i)$

Generating the keystream is the general issue for the security. Otherwise, an attacker could predict the stream and decrypt a message. The keystream is generated by a pseudorandom number generator discussed later and the given key serves as a seed.

6.1.2 Block Cipher

An algorithm for the block cipher takes data blocks of particular size of n bits and encrypts these blocks with a key of a particular size. The result is the ciphertext of a particular size. Most notably implementation is the DES cipher in early 1970s at IBM and the AES cipher which will be discussed in detail.

6.2 Advanced Encryption Standart (AES)

The Advanced Encryption Standard (AES) is the most widely used symmetric standard. It was invented by Vincent Rijmen and Joan Daemen as a Rijndael cipher in 1998 for the U.S National Institute of Standards and Technology. It was recognized as a replacement for the Data Encryption Standart (DES). The AES block cipher is formed by block and key size between 128, 192 and 256 bits. However, for demonstration, 128 bit key will be used. Based on the key size, a number of internal rounds of cipher is distinguished [30].

key lengths	# rounds = n_r
128 bit	10
192 bit	12
256 bit	14

Figure 6.1: AES key length [31]

Each round comprise of four sub-processes:

- Byte Substitution (SubBytes)
- Shift Rows
- Mix Collumns
- Add Round Key

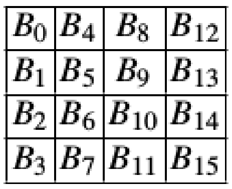
The process for encryption and decryption is inverse. At the beginning, Byte Substitution is conducted. It takes 128 bites from a message and a sequence of bits is transferred into 16 bytes matrix (4x4). The 16 input bytes in matrix are substituted by looking up a fixed table (S-Box). As example, the input byte to the S-Box is $A_i = 53_{(16)}$ then the output byte will be $ED_{(16)}$

								y								
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
x 8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 6.2: AES S-Box: Substitution values in hexademical notation for input byte (xy) [31]

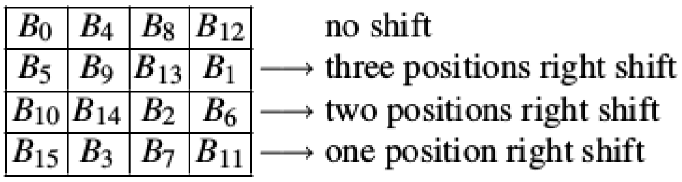
Shift Rows is the next step. Created matrix is used as input. Following shift is made:

- First row is not shifted.
- Second row is shifted one byte position to the left.
- Third row is shifted two byte positions to the left.
- Fourth row is shifted three byte positions to the left.



B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}

Figure 6.3: the input of Shift Rows [31]



B_0	B_4	B_8	B_{12}	no shift
B_5	B_9	B_{13}	B_1	$\longrightarrow$ three positions right shift
B_{10}	B_{14}	B_2	B_6	$\longrightarrow$ two positions right shift
B_{15}	B_3	B_7	B_{11}	$\longrightarrow$ one position right shift

Figure 6.4: the output of Shift Rows [31]

The major of the Mix Columns is diffusion. Each column of four bytes is transformed using a fixed matrix. The result is that every byte in a column of an output matrix depends on a value of four bytes in columns of input matrix.

At the end, Add Round Key has to be performed. The main sub-process is the Key schedule. It takes the original input key of length 128 bits and extracts the subkeys used in the AES according to the number of rounds. The subkey is XORed to 16 bytes matrix also known as AddRoundKey. The picture below illustrates an operation included in one round [29].

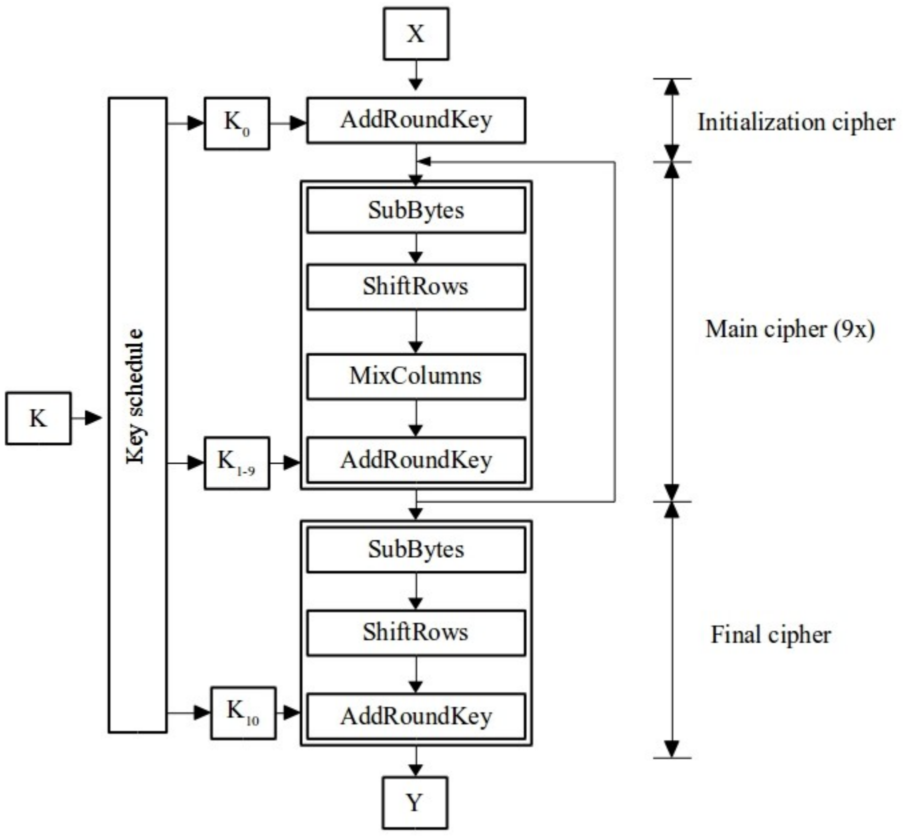


Figure 6.5: Diagram of one round AES encryption [29]

6.3 Modes of Operation

An encryption of data consisting of an arbitrary number of bits will be explained. The AES encryption of blocks is used to protect keys. Chaining is the process to encrypt data with the AES algorithm and thus confidentiality is provided. It results in the use of different modes of operation in the meaning of blocks chaining to form encrypted data. Above that, it is also notable to know if the sender was really the person who shares the key for communication. Hence, authentication is implemented in some modes of operation like CCM (Counter with Cipher Block Chaining Message Authentication Code Mode) or GCM (Galois/Counter Mode). Moreover, authentication also detects whether a message was altered during transmission or not. Every mode requires the exact length of a multiple block size. It puts together the whole plaintext. Nevertheless, the plaintext can have an arbitrary size. Due to this fact, plaintext has to be padded to fulfil the blocks. A possible method is appending “1” bit to the plaintext and the append “0” to the desired size (reach a multiple of the block length). If the plaintext is an exact multiple of the block length, an extra block consisting only of padding bits is appended. The following described modes are used in the practical part. However, the AES can operate in other modes as well.

6.3.1 Cipher-Block Chaining (CBC)

The Cipher-Block Chaining method (CBC) is based on the idea of chaining encrypted blocks. It means that every block depends on the previous encrypted block. The first block is randomized with an initialization vector IV which is an unique random number of 16 bytes. The initialization vector is not secret and it is delivered together with the encrypted data. Consequently, the IV is used as a nonce – a number used only once. As seen in the following formula, every encrypted block is XORed with following block consisting of a plaintext. Then the AES encryption is made with usually 16 bytes long secret key. This method is suitable for reliable transfer medium using optics or metallics. Block chaining is an appropriate solution with regard to security reasons. On the other hand, if an error bit occurs, the decrypted message will contain $n/2$ more bit errors on average.

Definition 6.3. Let $e()$ be a block cipher of block size b; let x_i and y_i be bit strings of length b; and IV be a nonce of length b.

- Encryption: $y_i = e_k(x_i \oplus y_{i-1}), i \geq 2, y_o = IV$
- Decryption: $x_i = e_k^{-1}(y_i) \oplus y_{i-1}, i \geq 2, y_o = IV$

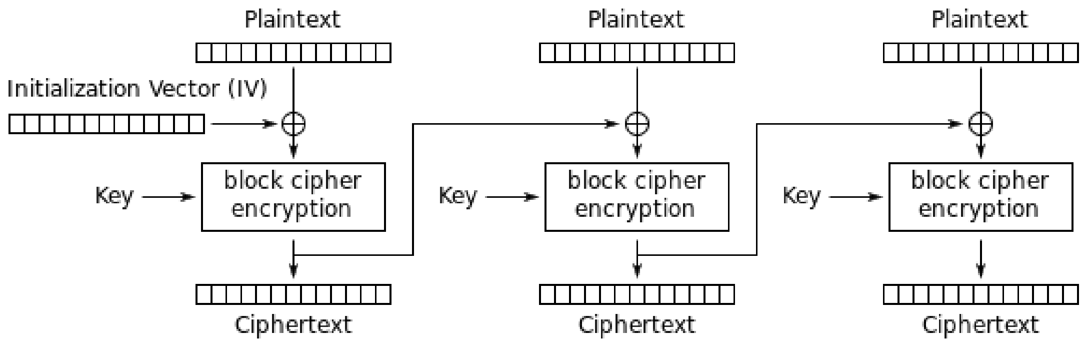


Figure 6.6: Cipher Block Chaining (CBC) mode encryption [32]

6.3.2 Counter Mode (CTR)

The Counter Mode is not a typical block cipher, because it uses a block cipher as a stream cipher. In other words, the block cipher is computed first. Its inputs are a counter - a number of n bits incrementing in every step and a key. The block cipher encryption (AES) is realized with these inputs. Afterwards, the created block cipher is XORed with an exact size of a plaintext with the result of a ciphertext. The input to the block cipher has to be chosen very carefully and cannot be reproduced. In order to achieve this, an example will be showed. A block cipher has an input width of 128 bits (the shorten block of an AES). Firstly, 96 bits length IV is chosen as a nonce. The remaining 32 bits are then used by a counter. For every block which is encrypted during the session, the IV is still the same. Nevertheless, 32 bits length counter is incremented. This solution gives 2^{32} blocks to encrypt with one IV. If every block consists of 8 bytes, a maximum of $2^3 * 2^{32} = 2^{35}$ bytes – approximately 32 Gigabytes can be encrypted before a new IV is generated [31].

Definition 6.4. Let $e()$ be a block cipher of block size b, and let x_i and y_i be a bit strings of length b. The concatenation of the initialization value IV and the counter CTR_i is denoted by $(IV||CTR_i)$ and is a bit string of length b.

- Encryption: $y_i = e_k(IV \parallel CTR_i) \oplus x_i, i \geq 1$
- Decryption: $x_i = e_k(IV \parallel CTR_i) \oplus y_i, i \geq 1$

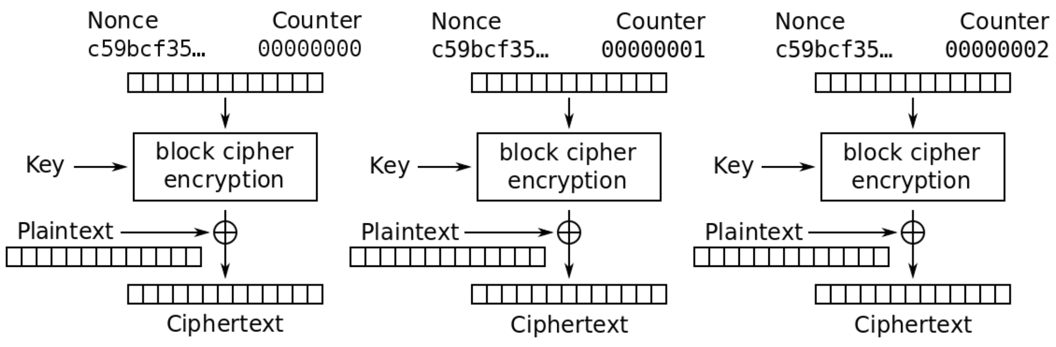


Figure 6.7: Counter (CTR) mode encryption [32]

6.3.3 Counter with Cipher Block Chaining-Message Authentication Code (CCM) Mode

An algorithm is based on the Counter mode (CTR) and the Cipher Block Chaining Authentication Code (CBC-MAC). It is a generic mode providing an authentication and a confidentiality. The two CCM processes are called generation-encryption and decryption-verification. For both CTR and CBC-MAC is used the same key K within the CCM [33]. The first step is to compute the authentication field using the CBC-MAC. The CBC-MAC is applied to the whole plaintext with the secret key and zero initialization vector. The result is the last block of the Cipher Block Chaining which is considered as a Message Authentication Checksum (MAC) sometimes also called a Message Integrity Check (MIC) or a Tag. It provides authenticity of data and ensures that data have not been modified in transit. Moreover, it means that an integrity of data is accomplished as well. The input data for the CTR Mode are 7 – 13 bytes nonce, 16 bytes key K, a plaintext and the MAC. The Counter mode is applied to the formatted plaintext and the MAC separately. The resulting data – the ciphertext is the output of the generation-encryption process [34].

The input into a decryption - verification process is a ciphertext and additionally added data, that are not encrypted but authenticated (usually header). The CTR mode is applied to the ciphertext to produce a corresponding MAC – “expected tag” and the payload. If the nonce, the associated data string and the payload are valid, the strings are formatted into appropriate blocks and the CBC-MAC is applied to verify the MAC. If the MAC - “calculated tag” is identical as received, then the decryption - verification process is successful. Otherwise, the payload and the MAC should not be revealed [32].

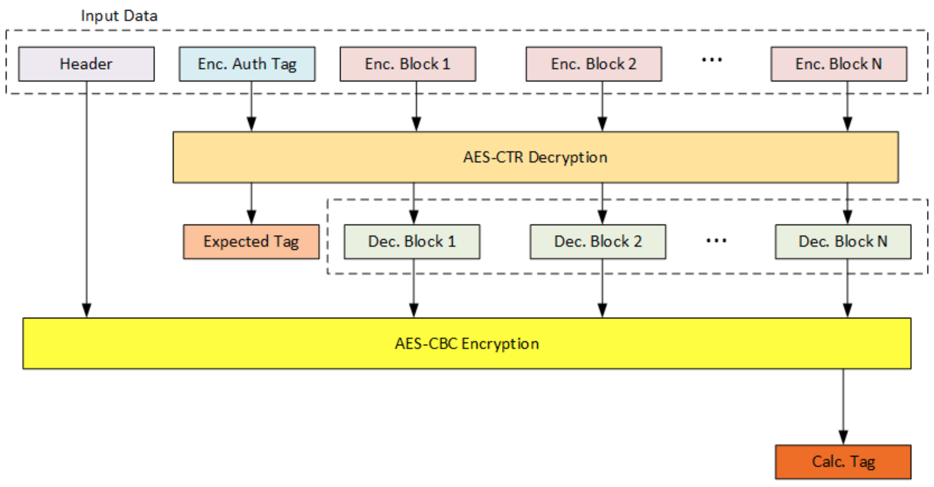


Figure 6.8: CCM mode encryption [32]

6.4 Random Numbers Generators

Random number generators are necessary to construct key streams used in stream ciphers. It is differentiated between three types of the random number generators which will be described in more detail.

6.4.1 True Random Number Generators (TRNG)

These types of random number generators are also called “Non-deterministic Random Bit Generators”. Their characteristic is based on an unpredictable output that cannot be reproduced. The TRNGs depend on physical processes and the final output is modified to be statistically independent. The Intel Pentium III uses for the TRNGs a temperature noise of resistors [35]. Other examples like a semiconductor noise, a clock jitter in digital circuits and a radioactive decay are also used for the generators.

6.4.2 Pseudorandom Number Generators (PRNG)

Pseudorandom Number generators are considered as a deterministic approach to construct a pseudorandom sequence of bits. Generating of a bits sequence is computed from an initial seed value.

$$s_0 = seed$$
$$s_{i+1} = f(s_i),\ i = 0, 1, ...$$

A popular example is the linear congruential generator:

$$s_0 = seed$$
$$s_{i+1} = a\ s_i + b \, mod \, m,\ i = 0, 1, ...$$

Integers a, b, m are constants. It should be noted that the PRNGs generators are not truly randomized because they could be computed. Widely used *rand()* function is a part of the ANSI C with parameters:

$$s_0 = 12345$$
$$s_{i+1} = 1103515245\, s_i + 12345 \, mod \, 2^{31},\ i = 0, 1, ...$$

Favorable statistical properties should maintain the PRNGs. Many mathematical tests on the PRNGs can be conducted. The tests can verify the statistical behaviour of sequences. However, using the asynchronous PRNGs is more secure, regarding their robustness comparing linear systems. An example includes a A5/1 cipher used to encrypt voice calls on telecommunication networks.

6.4.3 Cryptographically Secure Pseudorandom Number Generators (CSPRNG)

It is a special type of the PRNG used for cryptographic purposes. The general purpose is that the output is unpredictable. It means that the output n of the keystream $s_i,\ s_{i+1}, \dots, s_{i+n-1}$ is computationally infeasible to compute the subsequent of bits $s_{i+n}, s_{i+n+1}, \dots$ According to Knuth [36], there are two requirements for cryptographically strong PRNGs:

- There should be no obvious mathematical relationship between numbers in the sequence, like common multiples, ordering of values, or patterns of values.
- Any given key may be generated as part of variety of different key sequences.

The second statement describes a problem with the PRNGs, because they tend to produce the same sequences repeatedly. An importance of the unpredictability of the CSPRNGs is the general issue for the cryptography.

6.5 Hash Function

A hash function can be considered as a fingerprint of a message. For a particular message, a hash digest or a hash value is calculated. An output is a unique fixed-length-bit string of length n (usually 128-512 bits) representing the input message. The essential parts of the hash function are digital signatures. They are also widely used to store passwords or keys as in the case of this thesis. However, there is no need to use keys as in symmetric algorithms. The Hash function is a one-way function $z = h(x)$ and a given hash output must be computationally infeasible to compute the input message x. The advantage of the one-way function is that if the attacker obtains the hash function z, then a regeneration $h^{(-1)}(z) = x$ of a message is not possible. To achieve this, security requirements for hash functions are necessary. Three central properties need to be possessed in order to be secure:

- preimage resistance or one-wayness
- second preimage resistance or weak collision resistance
- collision detection or strong collision resistance

The preimage resistance is discussed above. The most crucial part to accomplish one-wayness is the key derivation. Theoretically, there are possibilities to find correct hash. A weak collision detection and a strong collision detection assure that for two messages, the same hash value is not calculated . The weak collision detection tries to find x_2 with the same hash value as x_1. On the other hand, the strong collision detection referees to free choice of both messages x_1 and x_2. The collision detection is the most critical security requirement.

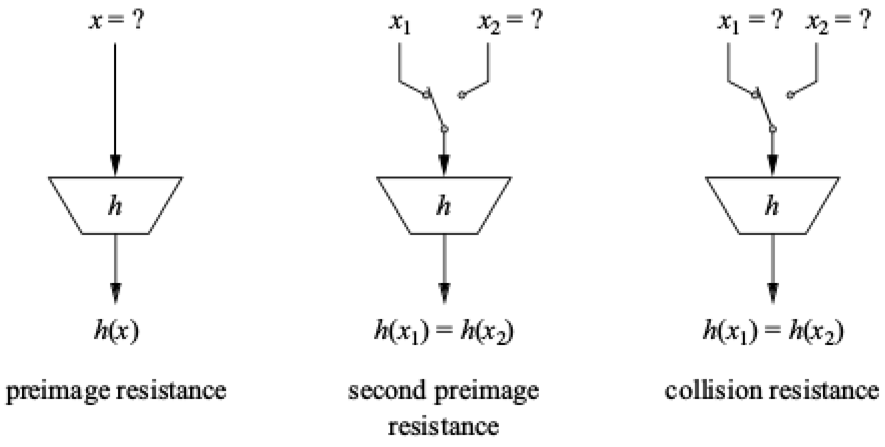


Figure 6.9: Three security properties of hash functions [31]

6.5.1 Secure Hash Algorithm 256 (SHA-256)

SHA-256 is a frequently used message digest function. The SHA is a family of cryptographic hash functions including four versions: SHA-0, SHA-1, SHA-2, SHA-3 [37]. It is supposed to be highly non-linear. In this chapter, attention is given to the SHA-256 which is used in the practical part later. The SHA is based on the Merkle–Damgård construction and it is quite similar to a block cipher. It can be used for a message M of length l bits, where $0 \leq l \leq 2^{64}$. First is the message M padded with a binary '1' followed by a sufficient number of zeros and an additional block of 64-bits containing the binary length of the original message. The maximum length of a message is 2^{64}. The length of a padding is situated to create the blocks multiple of 512 bits. After the message has been padded, it must be cut to N blocks of 512 bits illustrated as $M^{(1)}, M^{(2)}, ..., M^{(N)}$. Every block can be then expressed as sixteen 32-bit words, the first 32 bits of messages of block i are denoted $M_0^{(i)}$, the next 32 bits are $M_1^{(i)}$ up to the fifteenth block. From these sub blocks an expansion of the block is conducted [38]. Together sixty-four 32-bit words are created. A round on these words is constituted and the results are words $W_0, W_1, \ldots, W_{63}$ as a compression function. Consequently, Eight 32-bit words are created $H_0^{(i)}, H_1^{(i)}, ..., H_7^{(i)}$. They are used as an initialization hash value that are necessary to compute the final message digest through eight hash values $H^{(i)} = H^{(i-1)} + C_{M^i}\left(H^{(i-1)}\right)$ to receive the final hash value consisting of $H_0^{(N)} \parallel H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)} \parallel H_5^{(N)} \parallel H_6^{(N)} \parallel H_7^{(N)}$. Where C is the SHA-256 *compression function* and + means word-wise $mod\, 2^{32}$ addition and an input message M.

Chapter 7

Monitoring Network Activity

Analysis tools for monitoring network traffic are useful for detecting abnormalities. A prevention against a malicious activity is the basis for securing systems. The key benefit is an improvement of a security incident detection with a quick response time. To strengthen and to expand the security, the Security Operations Centers (SOC) taking care of monitoring can be seen in companies [39].

7.1 Intrusion Prevention/Detection (IPS/IDS) System

The Intrusion Prevention/Detection System (IPS/IDS) is a software application monitoring a network traffic of a device in both directions [40]. It protects devices against the malicious activity or violation of signatures known as rules. The rules can be created and edited by a user. However, in most cases the existing rules sets are used like the Emergency Threats founded on the database of Proofpoint [41]. Therefore, the rules based on the complex patterns identify the malicious activity. The network traffic is logged into a specific logging file. Not to be confused about an output, a logging file formatted in JavaScript Object Notation (JSON) is simple to be displayed with graphic tools like Logstash. The system can be usually used in the IPS or the IDS mode. However, there is a difference between them. The IDS system detects only a network traffic on a specific interface and it alerts the user. On the other hand, the IPS system is able to manipulate with packets, e.g. dropping them.

7.2 Suricata

The Suricata is a high-performance network security monitoring engine offering IPS and IDS modes of operation. It is an open source owned by a non-profit foundation – the Open Information Security Foundation (OISF). The most important part is creating and understanding of rules. A rule consists of the action, header and rule-options.

An **action signature** determines an activity of signatures' matches. There are four types of the Action.

- Pass
- Drop
- Reject
- Alert

The signature with a *pass* action is an alarm for Suricata which will stop scanning the packet and skip to the end of all rules for current packet. A *drop* action is only concerned in a IPS/inline mode. If the Suricata matches the signature with the *drop* action, the packet is not sent any further. In addition, a receiver does not receive a status message reporting an error. To reject a packet a *reject* action is used. In compare with the *drop* action, a receiver and a sender receive the packet. If offending packet concerns TCP a reset packet will be sent. The rest of the used packets will result in the ICMP-error packet. Lastly, an *alert* action does not threaten any packets. However, if signatures match, the alert will be generated by the Suricata and it will be recorded in a logging file as a possible danger. Due to different priorities of signatures, processing of rules should not be in order in which they appear in a file. The most significant signature is scanned first. The order is following: *pass, drop, reject, alert.*

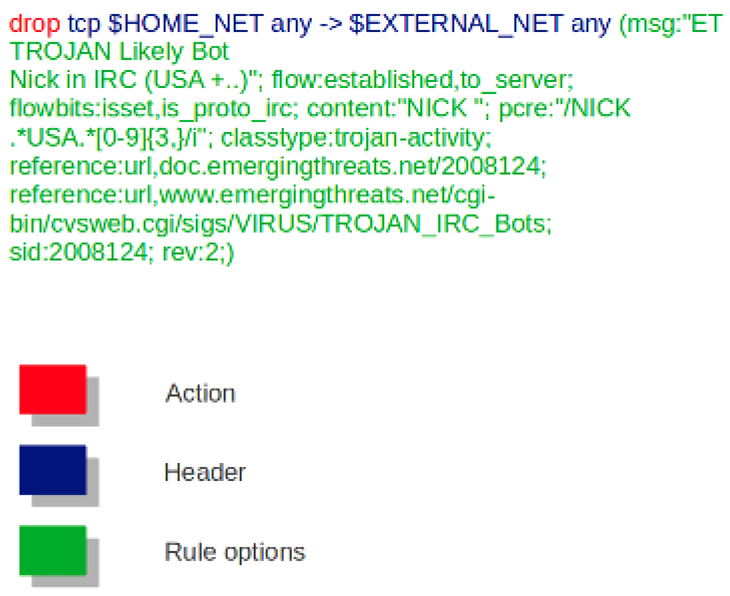
A **header signature** contains a protocol, destination and source IP addresses and ports. The protocol is used by the Suricata to determine which protocols need to be concerned. It is possible to use options *tcp, udp, icmp* and *ip*. The *ip* option stands for 'any' and every protocol match this signature. In newly released version of the Suricata, it is also possible to determine few added protocols like *http, ftp, tls, smb* and *dns*. Source and destination of IP addresses can be combined in more modification as shown in the example below. Instead of adding IP address directly to rules, it is possible to declare them in a configuration file *suricata.yaml*.

!1.1.1.1	(Every IP address but 1.1.1.1)
![1.1.1.1, 1.1.1.2]	(Every IP address but 1.1.1.1 and 1.1.1.2)
$HOME_NET	(Your setting of HOME_NET in yaml)
[$EXTERNAL_NET, !$HOME_NET]	(EXTERNAL_NET and not HOME_NET)
[10.0.0.0/24, !10.0.0.5]	(10.0.0.0/24 except for 10.0.0.5)

Source and destination ports have similar requisites as IP addresses. According to the type of traffic and received packets, different ports are used. The HTTP port uses port 80 or the encrypted version of HTTP uses port 443. If port is set to 'any', the rules are fully influenced by the protocol. Allowing control in both direction gives the Suricata feasibility to restrict outgoing packets. The example shows how the direction works.

```
source -> destination
source <> destination (both directions)
```

A **rule option** has additional settings for meta-information, headers, payloads and flows. However, most of the network traffic is encrypted and some settings can be used only on unencrypted packets. If an investigation of packet has to be deeper, then using rule options instead of only header signatures is appropriate. One of the examples is investigation of payload.



```
drop tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET
TROJAN Likely Bot
Nick in IRC (USA +..)"; flow:established,to_server;
flowbits:isset,is_proto_irc; content:"NICK "; pcre:"/NICK
.*USA.*[0-9]{3,}/i"; classtype:trojan-activity;
reference:url,doc.emergingthreats.net/2008124;
reference:url,www.emergingthreats.net/cgi-
bin/cvsweb.cgi/sigs/VIRUS/TROJAN_IRC_Bots;
sid:2008124; rev:2;)
```

🟥	Action
🟦	Header
🟩	Rule options

Figure 7.1: An example of a signature [41]

7.2.1 IPS/ Inline Mode

First, to use an Inline Mode, the Suricata has to be properly compiled and cooperation with iptables and NFQUEUE (Netfilter Queue) has to be accomplished. The iptables is a utility program for configuration tables provided by Linux kernel firewall. On most

Linux distributions iptables are fundamental tools. They specify how will be the packets in a system threatened [42]. The syntax of iptables is following:

```
iptables <option> <chain> <matching criteria> <target>
sudo iptables -I FORWARD -j NFQUEUE
```

NFQUEUE is a target of iptables which delegates the decision on packets to a userspace software - Suricata. There are three modes using the NFQUEUE - accept, repeat and route. If the mode is set to accept, the packets are not inspected by iptables anymore. The mode route sends the packet to another tool after being processed. In the mode repeat, packets are marked by the Suricata and they are re-injected at the first rule of iptables [43].

Afterwards, it is important to determine how the device operates. In other words, a device can behave like a gateway or like a client. If the Suricata runs as the gateway – devices in an inner network have to be secured. Scenario 1 illustrates this situation as *forward_ing*. To correctly apply the iptables, the following command is executed.

```
sudo iptables -I FORWARD -j NFQUEUE
```

In the case of the client situation illustrated in scenario 2, two iptables rules are created.

```
sudo iptables -I INPUT -j NFQUEUE
sudo iptables -I OUTPUT -j NFQUEUE
```

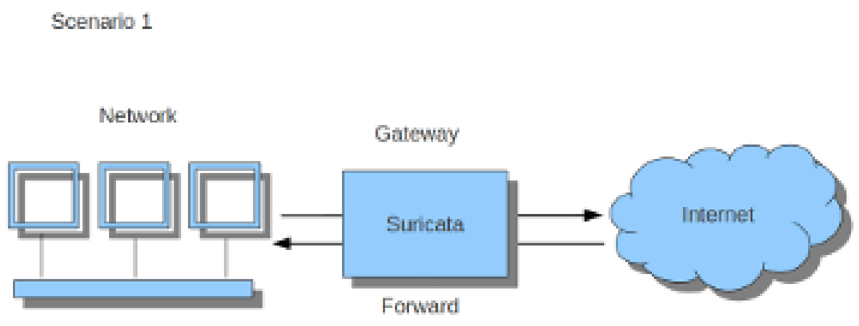


Figure 7.2: Suricata running on a gateway [43]

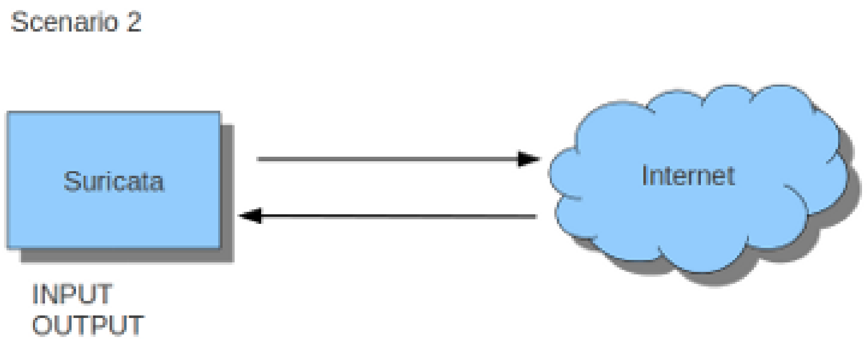


Figure 7.3: Suricata running on a client [43]

Part II

Practical Part

Chapter 8

Design of Firmware Management

An implementation of a secure firmware upgrade can be achieved using different approaches. The most spread procedure is likely a use of the crypto bootloader, because it is an efficient solution to secure embedded devices. Nevertheless, vulnerabilities of the bootloader implementation have already been discovered. Using a different method which is more computationally demanding is also possible. The implementation should be accessible and efficient to be widely spread. The proposed design is focused on an upper layer secure implementation of an operating system-based firmware. The operating system-based firmware is a part of the extensive amount of advanced embedded platforms which are responsible for more computationally demanding applications. The Vocore2 represents this type of devices, hence for the testing is used.

A design of an upgrade process bears in mind today security threats and need for increasing a prevention against the attacks. The approach is designed to not just secure the process of the upgrade but also to monitor an incoming traffic. The network traffic is analyzed and it is a crucial prevention of potential attacks. The prevention is performed with the IPS system Suricata running in an Inline Mode. The whole process will consist of an auto-run script written in the Python and the IPS system Suricata. These two elements will cooperate and defend the device with help of a hardware USB key. The hardware USB key will be a necessary part of an upgrade because using hardware security is much stronger than using a software-defined security element. For the firmware management, three basic features of cryptography have to be provided: confidentiality, integrity and authenticity also known as CIA. Two approaches using different modes of operation of AES are proposed to transfer the firmware securely. However, monitoring of the network traffic and a management of a decrypted firmware is the same for both approaches.

The firmware used in the practical part is the OpenWrt. Next chapters are focused on a compilation of an adjusted distribution of the OpenWrt, the testing of an upgrade procedure, the program explanation and a measurement on an embedded device Vocore2.

Chapter 9

Compilation of the OpenWrt

The OpenWrt project is a Linux distribution targeting embedded devices. It is an open source firmware allowing to compile own distribution with the specific customization through package management. The main targets generally cover OpenWrt routers, however, it can be also ported to the client side device. The main advantage is an availability of numerous packages covering programming languages like Python used for scripting, network packages like Suricata or other storage packages. In May 2016, the OpenWrt project was forked and the Linux Embedded Development Environment (LEDE) was released. Nevertheless, in 2018 LEDE and OpenWrt joined under the name of OpenWrt. The current stable version of OpenWrt is 17.01 which is a copy of the newest LEDE firmware. OpenWrt 18.01 is in the development, nevertheless, distribution for Vocore2 is stable and it contains numerous packages in comparison to the older version. Therefore, OpenWrt 18.01 is used [44].

9.1 Build System

A source code of a firmware can be found at the website of GitHub in the official OpenWrt repository. It is a frequently upgraded repository and the newest one containing OpenWrt 18.01 will be used for a compilation. After cloning the repository to a computer in *home/Documents/openwrt* directory, updating of packages have to take place. Executing following commands, OpenWrt will be cloned to a directory and all available packages will be updated and installed. It is important to use the newest version because solely the latest releases contain plentiful content of packages.

```
git clone https://www.github.com/openwrt/openwrt.git
./scripts/feeds update -a
./scripts/feeds install -a
```

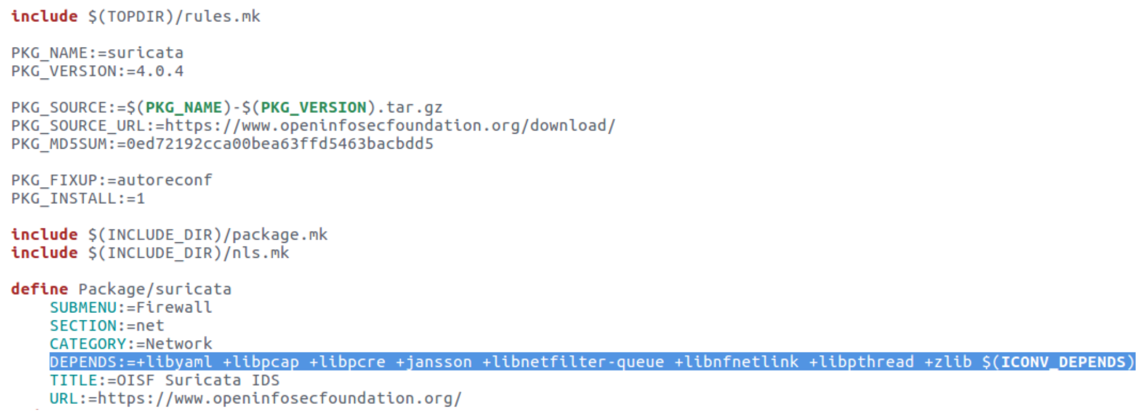
9.1.1 Suricata Modules

Suricata packages are still not under the official distribution of OpenWrt. Only a similar system Snort in IDS mode is available. However, a light version of the Suricata for the OpenWrt could be found as an open source at GitHub from the Sean Lin developer [45]. The Package is based on Suricata 4.0.4 and supports desired inline IPS mode. Building Suricata is more difficult compared to official packages. Firstly, it is required to separately copy downloaded package into */openwrt/package/feeds* directory of cloned OpenWrt. Next, running the Suricata needs to use iptables with NFQUEUE which is not compiled by default. Hence, kernel modules and libraries for using iptables and NFQUEUE have to be added. To find the Suricata in the OpenWrt configuration it is necessary to update copied files and then open a configuration.

```
./scripts/feeds update -i suricata
./scripts/feeds install suricata
make menuconfig
```

Dependencies for Suricata like *libyaml, libmagic, libpcap, libpcre, jansson, libnetfilter-queue, libnfnetlink, libpthread, zlib* have to be imported through configuration in */Libraries.* For iptables and NFQUEUE support, it is essential to import Kernel Modules *kmod-nfnetlink-queue, kmod-nfnetlink, kmod-ipt-nfqueue* in */Netfilter/Extensions.* Moreover, in *Network/iptables* a package *iptables-mod-nfqueue* is imported and in */Libraries, libmnl, libnfnetlink* have to be added.

Dependencies are prepared, however, these settings cause troubles with Suricata while testing on a Vocore2. Suricata Makefile has to be adjusted and a library *libmagic* has to be removed because it is an apparent reason for troubles. The mentioned library is officially specified on a Suricata website as a necessary dependency. On the other hand, on some architectures of embedded devices it showed up that *libmagic* is problematic and running Suricata is suddenly aborted. The most probable reason is used MIPS processor.

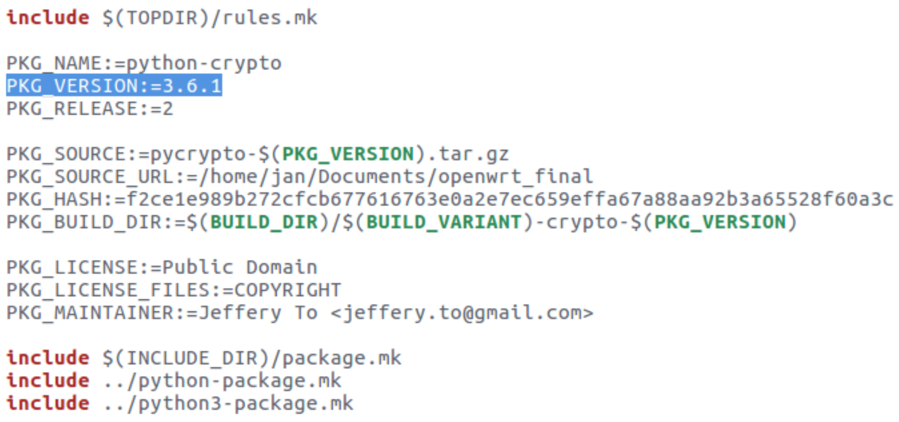


```
include $(TOPDIR)/rules.mk

PKG_NAME:=suricata
PKG_VERSION:=4.0.4

PKG_SOURCE:=$(PKG_NAME)-$(PKG_VERSION).tar.gz
PKG_SOURCE_URL:=https://www.openinfosecfoundation.org/download/
PKG_MD5SUM:=0ed72192cca00bea63ffd5463bacbdd5

PKG_FIXUP:=autoreconf
PKG_INSTALL:=1

include $(INCLUDE_DIR)/package.mk
include $(INCLUDE_DIR)/nls.mk

define Package/suricata
    SUBMENU:=Firewall
    SECTION:=net
    CATEGORY:=Network
    DEPENDS:=+libyaml +libpcap +libpcre +jansson +libnetfilter-queue +libnfnetlink +libpthread +zlib $(ICONV_DEPENDS)
    TITLE:=OISF Suricata IDS
    URL:=https://www.openinfosecfoundation.org/
```

Figure 9.1: Suricata Makefile

9.1.2 Python modules

A script written in the Python language is used to control the whole process of the secure firmware upgrade with the help of Python modules. PyCryptodome module includes plentiful AES modes such as required AES-CCM mode responsible for the authentication and the encryption or the decryption procedure. A hash function used for storing keys and data integrity is also available. The Python version 3.6.5 is added in an official repository. However, PyCryptodome module is not available in the official repository of the OpenWrt. Only an older version of PyCrypto module is available. The PyCrypto module does not include AES-CCM mode which is required. On the other hand, the structure of the PyCryptodome and the PyCrypto is almost identical. After updating all feeds, a Makefile used for compilation of PyCrypto can be founded in */Documents/openwrt/feeds/packages/lang/python/python-crypto*. To replace the PyCrypto module with the PyCryptodome module, Makefile needs to be modified. A package version is changed to 3.6.1. The most important part is to modify the correct path to the PyCryptodome 3.6.1 module. The module is downloaded and the path to the online repository is replaced with the PyCryptodome module which can be found in a computer. The following figure shows the modified Makefile.



```
include $(TOPDIR)/rules.mk

PKG_NAME:=python-crypto
PKG_VERSION:=3.6.1
PKG_RELEASE:=2

PKG_SOURCE:=pycrypto-$(PKG_VERSION).tar.gz
PKG_SOURCE_URL:=/home/jan/Documents/openwrt_final
PKG_HASH:=f2ce1e989b272cfcb677616763e0a2e7ec659effa67a88aa92b3a65528f60a3c
PKG_BUILD_DIR:=$(BUILD_DIR)/$(BUILD_VARIANT)-crypto-$(PKG_VERSION)

PKG_LICENSE:=Public Domain
PKG_LICENSE_FILES:=COPYRIGHT
PKG_MAINTAINER:=Jeffery To <jeffery.to@gmail.com>

include $(INCLUDE_DIR)/package.mk
include ../python-package.mk
include ../python3-package.mk
```

Figure 9.2: PyCryptodome Makefile

9.1.3 External Memory

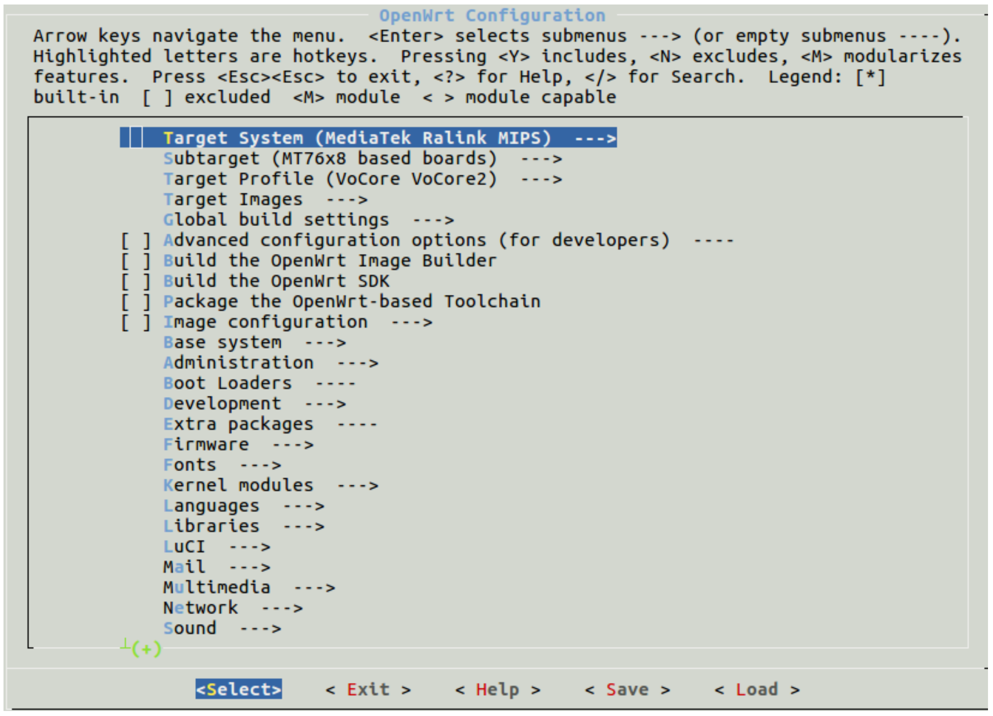
The Suricata and the Python modules are larger that the size of a Vocore2's memory. The main issue is the size of PyCryptodome module. Using the older PyCrypto module, the Vocore2's memory would be sufficient. In this procedure, an external memory is necessary to add the desired module. During the compilation, packages supporting USB device has to be added. The Kernel modules and packages supporting file system ext4 are *block-mount, kmod-fs-ext4* and */kmod-usb-storage* located in */Base System* and *Kernel/Modules*. The advantage of an external memory is enough space for logs that Suricata produces.

9.2 Compilation

Packages for the adjusted distribution are currently prepared. Next step is selecting architecture, the target device and essential packages. While configuring Suricata, check of an option 'Compile with full language support' guarantee that Suricata will be built properly, otherwise compilation would be unsuccessful. It is also necessary to append dependencies and packages like Python, Suricata or Nmap used for a port access check. The complete configuration can be seen in the attachment. Having prepared everything, configuration is saved and compilation can begins with the following command to see error messages if they occur.

```
make -j V=s
```

After a successful compilation, a binary file is created and it can be found in *openwrt/bin/targets/ramips/mt76x8* location. The binary file – created firmware with the adjusted distribution of OpenWrt *openwrt-ramips-mt76x8-vocore2-squashfs-sysupgrade.bin* is prepared for the use. Only a sysupgrade firmware can be used for an upgrade through a shell environment of the embedded device. It ought to be mentioned that the compiled firmware cannot be bigger than a flash memory of the target device which is 16 MB in Vocore2. Otherwise, the sysupgrade firmware would not be created.



```
                        OpenWrt Configuration
 Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).
 Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
 features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*]
 built-in  [ ] excluded  <M> module  < > module capable

             Target System (MediaTek Ralink MIPS)  --->
             Subtarget (MT76x8 based boards)  --->
             Target Profile (VoCore VoCore2)  --->
             Target Images  --->
             Global build settings  --->
         [ ] Advanced configuration options (for developers)  ----
         [ ] Build the OpenWrt Image Builder
         [ ] Build the OpenWrt SDK
         [ ] Package the OpenWrt-based Toolchain
         [ ] Image configuration  --->
             Base system  --->
             Administration  --->
             Boot Loaders  ----
             Development  --->
             Extra packages  ----
             Firmware  --->
             Fonts  --->
             Kernel modules  --->
             Languages  --->
             Libraries  --->
             LuCI  --->
             Mail  --->
             Multimedia  --->
             Network  --->
             Sound  --->
         (+)

             <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 9.3: Configuration of OpenWrt

Chapter 10

Testing of Secure Firmware Upgrade

The procedure of the secure upgrade consists of an auto-run script which is present on the Vocore2 and a terminal program on a computer which manage the process. All programming is written in Python. The script on the Vocore2 cooperates with the Suricata and it can modify Suricata rules. At the beginning of the process, the rules are set to drop every attempt to get a remote access of a Vocore2. Going through the successful authentication and sending the firmware for the upgrade, the rules can be modified. The modification allows the remote access for the corresponding IP address of an authenticated terminal. Then the management of the firmware is possible and the upgrade of the firmware can be performed.

10.1 OpenWrt Configuration

Before the procedure of a remote upgrade of the firmware, the OpenWrt system needs to be configured to be remotely accessible and have enough storage space. The connection to the network has to be established. The Vocore2 is a memory limited device. Consequently, the memory of a device is increased by adding USB drive and mounting it as an */overlay*. Finally, before the system can be used for the procedure, the vulnerable ports on the Vocore2 are scanned to show the services which are opened to the network.

10.1.1 Network Settings

The Vocore2 can be run in two modes, an access point (AP) or a workstation (STA). The AP mode is applied, if the Vocore2 is set up as a hotspot. In this case, the device is connected to the gateway. Through connection to the Vocore2, it is possible to reach the Internet. In the AP mode, the Vocore2 routes the traffic into the gateway which forward packets further to the destination IP address.

On the other hand, the STA mode works as a client. In this mode, the Vocore2 operates as an end device which can communicate through the network. The STA mode is used in this procedure. The configuration can be set for a wireless connection or the Ethernet connection. Having connection more reliable, settings for the Ethernet connection is set with the following commands. The IP address is added as the static 192.168.2.110. Then the IP address of a gateway is set with a DNS server responsible for assigning domain names with the corresponding IP addresses.

```
uci set network.lan.ipaddr=192.168.2.110
uci set network.lan.gateway=192.168.2.1
uci set network.lan.dns=8.8.8.8
uci commit
service network restart
```

Another settings involve dependencies for running Suricata in IPS inline mode. Hence, iptables have to be configured. Consequently, a network traffic in both directions will pass the Suricata. Iptables are useful security tools, in the case of aborting Suricata intentionally. The device will be disconnected from the network because all the traffic is set to go through the Suricata which is not running.

```
iptables -I INPUT -j NFQUEUE
iptables -I OUTPUT -j NFQUEUE
```

10.1.2 Storage Management

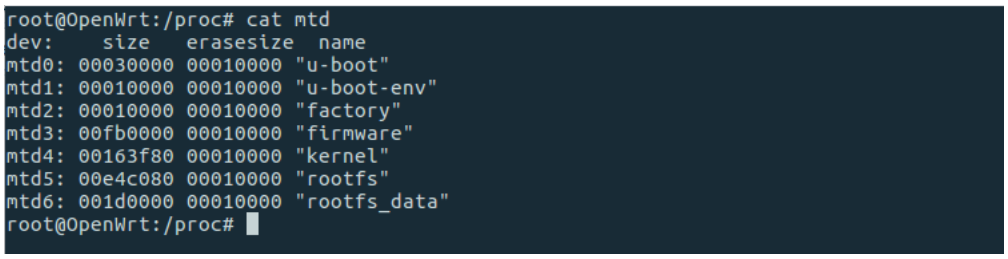
The main reason for using an external memory mounted as */overlay* is a size of a Python PyCryptodome library. The non-volatile memory of Vocore2 is 16 MB. However, a size of a compiled version of the PyCryptodome is 7 MB and together with other packages, the memory of the Vocore2 is exceeded.

A flash memory of Vocore2 is separated into layers consisting of partitions. Layer 0 is a flashchip which is connected to the SoC via SPI (Serial Peripheral Interface Bus). Layer 1 consists of a bootloader U-boot which loads the firmware. Factory settings, the OpenWrt firmware and U-boot environment variables cover Layer 1 as well. On Layer 2, the firmware is separated into the kernel and the rootfs. During the booting procedure, a kernel is copied to the temporary RAM and the booting procedure is executed. The rootfs is a root (the user with the highest privilege) file system which consists of configuration files and applications which are necessary to run operating system. Layer 3 divides the file system into the SquashFS file system as */rom* and the JFFS2 filesystem as */overlay*. SquashFS is only readable and it is used in the safe mode. The next, JFFS2 is a writable part of the file system containing every file in a memory written after an installation of the system e.g. configuration files and an additional packages [11].

The following commands increase the size of the rootfs filesystem in the size of an additional USB drive [46]. Firstly, the driver has to be formatted to ext4 file system. Then file system table is enabled and partition of the USB driver is mounted to */overlay*. At the end of the process, files from existing flash memory in */overlay* are copied into the newly created */overlay* in the external storage.

```
/etc/init.d/fstab enable
block detect > /etc/config/fstab
vi /etc/config/fstab
    option target '/overlay'
    option enabled '1'
mount –bind / /tmp/extroot
mkdir /tmp/system
mount /dev/sda1 /tmp/system
tar -C /tmp/extroot -cvf - . | tar -C /tmp/system -xf -
```

The Vocore2 needs to be rebooted, then an external memory appears. The external memory is also important to store captured Suricata and system logs persisting in a device memory. Suricata logs are stored in JSON file format which can be used for the visual output.

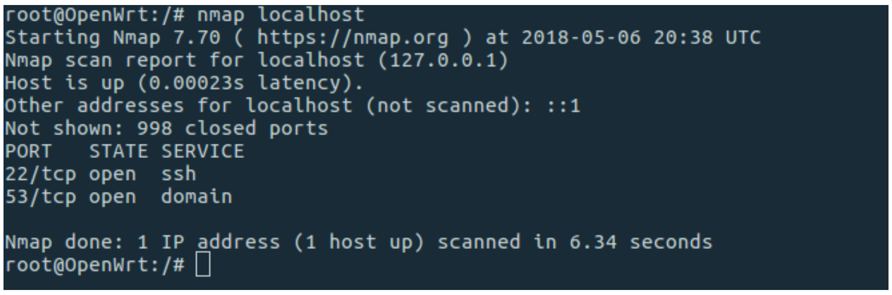


```
root@OpenWrt:/proc# cat mtd
dev:    size   erasesize  name
mtd0: 00030000 00010000 "u-boot"
mtd1: 00010000 00010000 "u-boot-env"
mtd2: 00010000 00010000 "factory"
mtd3: 00fb0000 00010000 "firmware"
mtd4: 00163f80 00010000 "kernel"
mtd5: 00e4c080 00010000 "rootfs"
mtd6: 001d0000 00010000 "rootfs_data"
root@OpenWrt:/proc# 
```

Figure 10.1: Memory partition on Vocore2

10.1.3 Port Scanning

Lastly, it is necessary to ensure that vulnerable ports on the embedded device are disabled and a password to a get remote access to the device is set. The setting of a password is accomplished by command *passwd*. The analysis of ports can be provided with *Nmap* tool which can scan device and show all open ports on a device. It can be seen that only port 22 (SSH) and 53 (DNS) are opened on Vocore2. DNS serves to a domain name translation. To properly resolve the domain name with the corresponding IP address, DNS needs to be opened.



```
root@OpenWrt:/# nmap localhost
Starting Nmap 7.70 ( https://nmap.org ) at 2018-05-06 20:38 UTC
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00023s latency).
Other addresses for localhost (not scanned): ::1
Not shown: 998 closed ports
PORT   STATE SERVICE
22/tcp open  ssh
53/tcp open  domain

Nmap done: 1 IP address (1 host up) scanned in 6.34 seconds
root@OpenWrt:/# 
```

Figure 10.2: Nmap tool used on a localhost

Only port 22 (SSH) is used for a management and it should be opened if exact conditions are accomplished. Through the SSH can be managed the device and a firmware is upgraded.

10.2 Suricata Settings

The procedure of firmware upgrade begins with the modification of Suricata rules. The auto-run script on the Vocore2 adjusts the Suricata rules to drop all communication targeting a SSH port without an authentication. Consequently, Suricata rules in *etc/suricata/rules/test.rules* location are created with the following command: *drop ip any any -> any 22.* The command ensures that every attempt to get the access SSH without authentication will be dropped and logged into a specific file found in */var/log/suricata.*

10.3 AES-CCM and CBC Mode Approach

When the default rules of the Suricata are set, an implementation of separate authentication and encryption is conducted. The AES-CBC mode is used for an encryption however, only confidentiality and integrity is provided. Authenticity has to be achieved as well. Digital signatures are not suitable due to demanding asymmetric encryption and need for third-party Certification Authority (CA) which is a paid service. On the other hand, it is satisfying solution avoiding Man in the Middle Attack (MITM). Instead of using digital signatures, AES-CCM mode of operation resistant to MITM attack is used to provide authenticity. Two different keys are necessary for authentication and encryption. The keys are safely stored and protected against reading in a flash memory of the device. The size of the keys can be various according to the used AES-128 or AES-256 mode. In the implementation, 32 bytes long key created with the SHA-256 is used.

Given that, hardware security is stronger than software security, a USB hardware key is used. Due to this implementation, only an owner of a hardware key is able to upgrade the device with an encrypted firmware from the manufacturer. Hardware key contains safely stored a Pseudo Random Number (PRN). Size of the PRN is chosen for 512 bits. The PRN is used as a seed for an initialization vector (IV) used for the CBC mode and for nonce used in the CCM mode. The IV and the nonce are unique for every session and must not be repeated again. From the PRN, the IV is declared by randomly selected bits in length of 128 bits (16 bytes). Using the CCM mode, nonce selects 104 bits (13 bytes) from PRN [47].

10.3.1 Authentication

An authentication is provided in the AES-CCM mode with an authentication key and a nonce as its inputs. Positions of selected bits from the PRN while creating the nonce are stored as a Selected Numbers Order (SNO). A terminal performs a calculation of a Message Authentication Checksum (MAC) and encrypts the MAC with AES-CTR mode which is a part of AES-CCM mode. Together the MAC and the SNO are sent to the device.

Device receives the MAC and the SNO. A valid authentication is only if a received MAC is equal to a calculated MAC in a device. The device knows the secret authentication key, only element which is unknown for decryption an calculation MAC is the nonce. Via the PRN, a calculation of a correct nonce is possible because of the received SNO, which stores positions of selected bits for the nonce. To accomplish that, a hardware USB key storing the PRN has to be connected to device. A calculation of the MAC on the device is currently possible. Finally, the device send an authentication packet to a terminal with a valid access to send the firmware.

10.3.2 Encryption and Decryption

An encryption of a new firmware (NFI) is performed with the AES-CBC mode with inputs of a secret key and created IV. The key is a different key than used during authentication. The same mechanism for creating SNO and IV is used. However, size of IV is 16 bytes. The encrypted firmware is separated into frames of 1024 bytes and send together via TCP packets to the device to provide assurance that data were not lost during transmission. In previous authentication, UDP packets were used instead to provide an authentication as quick as possible.

For a decryption, secure key and IV is mandatory. Generating of IV is the same process as regenerating nonce during authentication. Due to PRN stored on a hardware USB key and the SNO, there is enough information to regenerate IV. Consequently, the encrypted firmware is stored in a temporary memory and a hash function SHA256 is performed to check a data integrity. If the hash is valid with a hash received from a terminal, the integrity is correct and the decryption of firmware can start. The final decrypted firmware is prepared to be managed.

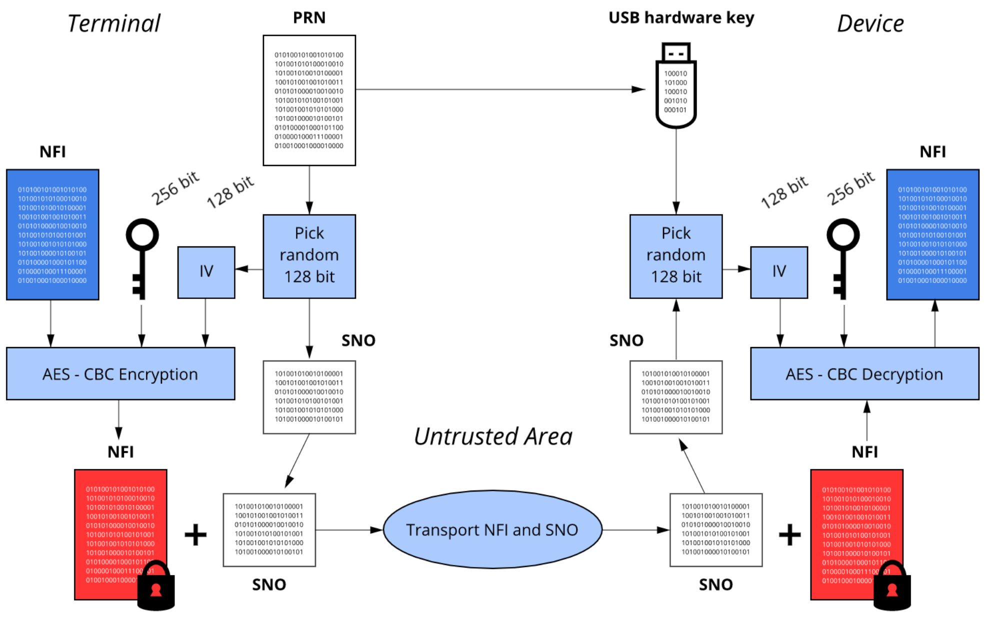
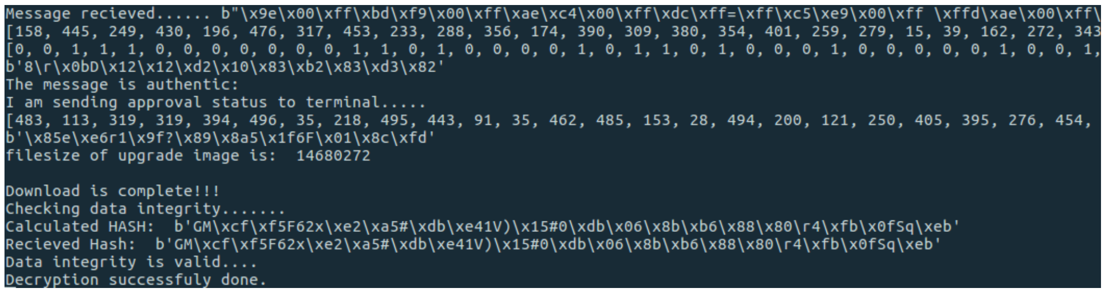


Figure 10.3: The procedure of encryption/decryption in AES-CBC mode

While testing the output of procedure on Vocore2 is reported into *report.txt* file which can be found on the Vocore2. The following figure shows the procedure until the successful decryption and allowing the remote access.



```
Message recieved...... b"\x9e\x00\xff\xbd\xf9\x00\xff\xae\xc4\x00\xff\xdc\xff=\xff\xc5\xe9\x00\xff \xffd\xae\x00\xff\
[158, 445, 249, 430, 196, 476, 317, 453, 233, 288, 356, 174, 390, 309, 380, 354, 401, 259, 279, 15, 39, 162, 272, 343
[0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1,
b'8\r\x0bD\x12\x12\xd2\x10\x83\xb2\x83\xd3\x82'
The message is authentic:
I am sending approval status to terminal.....
[483, 113, 319, 319, 394, 496, 35, 218, 495, 443, 91, 35, 462, 485, 153, 28, 494, 200, 121, 250, 405, 395, 276, 454,
b'\x85e\xe6r1\x9f?\x89\x8a5\x1f6F\x01\x8c\xfd'
filesize of upgrade image is:  14680272

Download is complete!!!
Checking data integrity.......
Calculated HASH:  b'GM\xcf\xf5F62x\xe2\xa5#\xdb\xe41V)\x15#0\xdb\x06\x8b\xb6\x88\x80\r4\xfb\x0fSq\xeb'
Recieved Hash:  b'GM\xcf\xf5F62x\xe2\xa5#\xdb\xe41V)\x15#0\xdb\x06\x8b\xb6\x88\x80\r4\xfb\x0fSq\xeb'
Data integrity is valid....
Decryption successfuly done.
```

Figure 10.4: The report of the procedure on Vocore2

10.4 AES-CCM Approach

There is another option which is an alternative to AES-CCM and CBC Mode Approach. An adaptation of the AES-CCM mode for the whole procedure can replace separate authentication and encryption. This adaptation is more demanding for computing resources. However, the process of authentication, encryption and a data integrity check is put together due to characteristics of the AES-CCM mode. The authentication and the data integrity is provided with MAC which is constructed from the firmware. Confidentiality is ensured with AES-CTR mode which is a part of AES-CCM mode. During the process, only one key is necessary and it should be safely stored in the flash memory of a device.

The same hardware security as in the previous method is used. PRN is 512 bits and the nonce (13 bytes) is randomly selected from the PRN. An initialization vector (IV) for AES CCM is usually declared as the zero value. Suricata settings and a management of the device remain the same during the procedure.

10.4.1 Authentication and Encryption/Decryption

The AES-CCM mode needs as inputs a nonce, a key and the firmware. First, 13 bytes nonce is created and its positions from PRN are stored as SNO. Then the MAC calculation is conducted. The MAC is the last block of the AES-CBC mode with inputs of zero IV, the key and the firmware in the form of a plaintext. Finally, the AES-CTR mode is conducted on the MAC and firmware separately. The encrypted data in the AES-CCM mode consisted of the encrypted firmware and the encrypted MAC are prepared for transfer. The SNO and the data are sent from terminal to the Vocore2 via TCP connection to avoid the loss of packets.

The device receives SNO and encrypted data. It is also able to regenerate the nonce. If a hardware USB key is available, SNO will pick the correct positions from PRN to regenerate the nonce. The same key as in the terminal is stored in the memory of a device. Together nonce and the key are two necessary parts for decryption. Due to the known size of MAC which is the size of the last block - 16 bytes, encrypted data could be divided into an encrypted MAC and an encrypted new firmware (NFI). The authentication and the integrity check of the firmware are performed in one step. Decrypted MAC is considered as the "expected tag". The rest of data belongs to the firmware. These data are decrypted with AES-CTR and the output is used for AES-CBC encryption with the result of the last block considered as the "calculated tag". The expected tag and the calculated tag are compared. If the tags are matched, the authentication is valid and moreover, the data integrity of firmware is guaranteed, because the firmware acts as an input for the MAC. Validation is correct and the decrypted firmware can be used for a firmware upgrade by management of the firmware. To inform the terminal, the UDP packet with validation for remote access is sent. Otherwise, the MACs are not matched, the decrypted firmware is immediately deleted. Consequently, the terminal is informed with unsuccessful attempt of an upgrade.

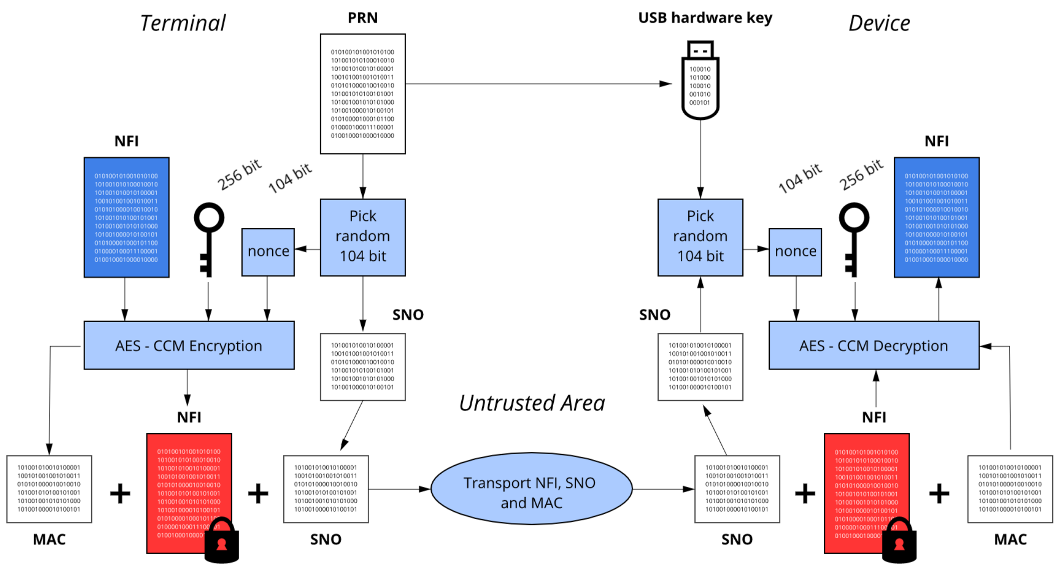
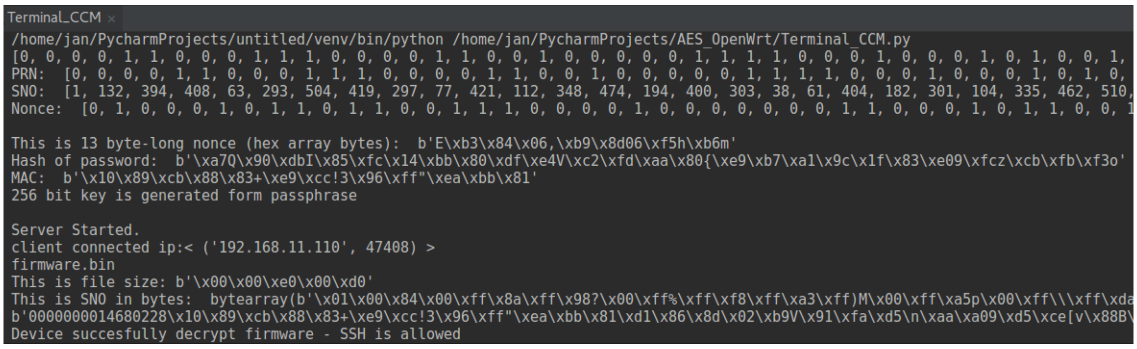


Figure 10.5: The procedure of firmware upgrade in AES-CCM mode

The output of the procedure is also reported into a *report.txt* file. However, the structure is almost the same as in AES-CCM and CBC Mode Approach.

10.5 Management of the Firmware

When a decryption is successful despite the approach which was used for the authentication and encryption, the device rewrites Suricata rules. These rules allow SSH connection of an authenticated terminal. The rules in *etc/suricata/rules/test.rules* are changed to the following form: *drop ip any any -> !source_ip_address 22*. Where *source_ip_address* is a IP address of terminal. The entry instructs that every packet targeting port 22 will be dropped except the IP address of the terminal. The following figure shows the output of the terminal script which informs that the decryption of firmware was successful and the remote access from the terminal to the device is now possible.



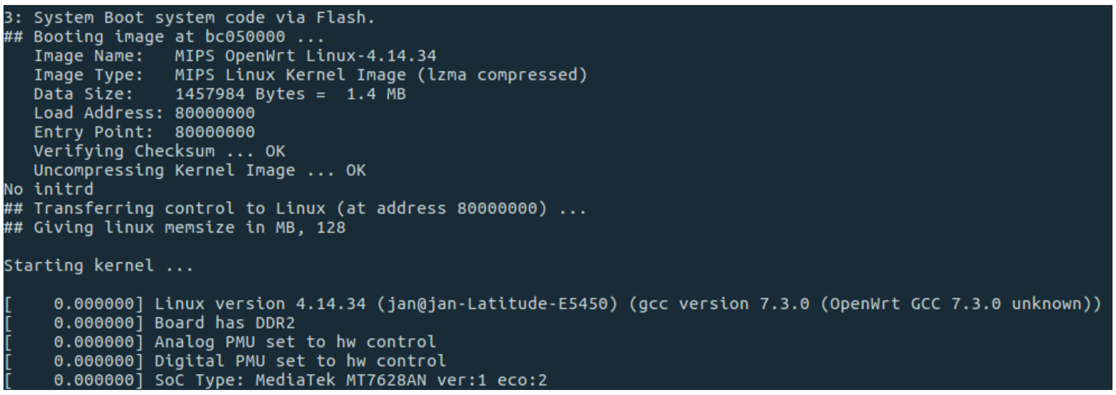
```
/home/jan/PycharmProjects/untitled/venv/bin/python /home/jan/PycharmProjects/AES_OpenWrt/Terminal_CCM.py
[0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1,
PRN:  [0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0,
SNO:  [1, 132, 394, 408, 63, 293, 504, 419, 297, 77, 421, 112, 348, 474, 194, 400, 303, 38, 61, 404, 182, 301, 104, 335, 462, 510,
Nonce:  [0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1

This is 13 byte-long nonce (hex array bytes):  b'E\xb3\x84\x06,\xb9\x8d06\xf5h\xb6m'
Hash of password:  b'\xa7Q\x90\xdbI\x85\xfc\x14\xbb\x80\xdf\xe4V\xc2\xfd\xaa\x80{\xe9\xb7\xa1\x9c\x1f\x83\xe09\xfcz\xcb\xfb\xf3o'
MAC:  b'\x10\x89\xcb\x88\x83+\xe9\xcc!3\x96\xff"\xea\xbb\x81'
256 bit key is generated form passphrase

Server Started.
client connected ip:< ('192.168.11.110', 47408) >
firmware.bin
This is file size: b'\x00\x00\xe0\x00\xd0'
This is SNO in bytes:  bytearray(b'\x01\x00\x84\x00\xff\x8a\xff\x98?\x00\xff%\xff\xf8\xff\xa3\xff)M\x00\xff\xa5p\x00\xff\\\xff\xda
b'0000000014680228\x10\x89\xcb\x88\x83+\xe9\xcc!3\x96\xff"\xea\xbb\x81\xd1\x86\x8d\x02\xb9V\x91\xfa\xd5\n\xaa\xa09\xd5\xce[v\x88B\
Device succesfully decrypt firmware - SSH is allowed
```

Figure 10.6: The output of terminal script for AES-CCM approach

The terminal can now be connected via SSH to the device and the process of upgrade firmware can be finished. The decrypted firmware will be found in the temporary memory */tmp*. According to the OpenWrt there are two possibilities of upgrading firmware via command line. The tool sysupgrade and the mtd_write tool.

```
ssh root@192.168.2.110
sysupgrade -v /tmp/[specified firmware].bin
mtd -r write /tmp/[specified firmware].bin firmware
```

After executing sysupgrade or the mtd_write tool, the configuration settings are saved and the firmware is written to memory. At the end, the system is rebooted and firmware is uncompressed. If the firmware is not damaged, the kernel initializes all components of the new firmware and the OpenWrt is ready to use.



```
3: System Boot system code via Flash.
## Booting image at bc050000 ...
   Image Name:   MIPS OpenWrt Linux-4.14.34
   Image Type:   MIPS Linux Kernel Image (lzma compressed)
   Data Size:    1457984 Bytes =  1.4 MB
   Load Address: 80000000
   Entry Point:  80000000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
No initrd
## Transferring control to Linux (at address 80000000) ...
## Giving linux memsize in MB, 128

Starting kernel ...

[    0.000000] Linux version 4.14.34 (jan@jan-Latitude-E5450) (gcc version 7.3.0 (OpenWrt GCC 7.3.0 unknown))
[    0.000000] Board has DDR2
[    0.000000] Analog PMU set to hw control
[    0.000000] Digital PMU set to hw control
[    0.000000] SoC Type: MediaTek MT7628AN ver:1 eco:2
```

Figure 10.7: Successful uncompressing of the firmware

Chapter 11

Implementation details

An auto-run script is written in the Python version 3.6.5 which is supported by OpenWrt. The two scripts for an implementation are required. The first script is written for the terminal located on a computer. It encrypts and sends the firmware to the device. The second script is running on Vocore2 and cooperates with Suricata to avoid unsolicited actions on the network interface. It is necessary to import default Python modules to establish a connection or to manipulate with files. Moreover, PyCryptodome module is additionally added. This module contains Pseudo Random Number Generators, numerous list of AES modes and most of the commonly used hash functions. The following chapter gives the most significant examples of the program.

11.1 Creating variables

At the beginning, declaration of variables has to take part. The PRN is the initial variable and it has to be randomly generated with the random function from PyCryptodome module. The PRN is created as 512 long integer array consisted of '1' and '0'. The array is created to easily pick random bits while construction of IV or nonce.

```
PRN = [ ]
prn_bits =512
def generatePRN(prn_bits):
   for i in range(0,prn_bits):
      value = random.randint(0,1)
      PRN.insert(i,value)
   a = bytearray(PRN)
   return a
```

A constructed PRN is saved as binary data into a file. After that, it is stored in the terminal and in the USB hardware key which is delivered to a device. Every upgrade sessions, PRN is read as shown in the next example.

```
def Read_PRN_file():
    with open(filename, 'rb') as infile:
        prn = infile.read(1024)
        a = list(prn)
```

PRN serves as a seed for generating nonce in authentication and IV in encryption. The Difference is only in the size of these variables – nonce is 13 bytes long and IV is 16 bytes long. As seen in the next example SNO and nonce of 13 bytes is created. Index stands for the position of chosen bit from PRN.

```
nonce_bits = 104
sno =[ ]
nonce = [ ]
def generateSNOandNonce(nonce_bits):
    for i in range(iv_bits):
        index = random.randrange(len(PRN))
        sno.insert(i,index)
        pick = PRN[index]
        nonce.insert(i,pick)
```

The last important step while creating variables is to set the appropriate form and the right size of a variable. Using integers is simple for manipulation. However, for every variable that will be transmitted, the correct length in the form of the binary data has to be performed. The following example transforms integers into binary data with correct size such as 16 bytes for IV.

```
string_iv = ''.join(str(e) for e in iv)
hexa_iv= '%0*X' % ((len(string_iv) + 3) // 4, int(string_iv, 2))
bytes_iv = bytes.fromhex(hexa_iv)
```

11.2 Network connection

A network communication is ensured by a socket module. The procedure uses two types of protocols. The first UDP is used in an authentication and a confirmation of status on Vocore2. The UDP is a connectionless protocol and it can be easily implemented. The following example is a part of AES-CCM + CBC approach. It shows communication between the terminal and the Vocore2, while Vocore2 successfully check MAC and send to the terminal value ’x00’ in the payload. The value ’x00’ means for the terminal that the authentication was validated and the procedure can continue with sending an encrypted firmware.

```
terminal = 192.168.2.190
port =5005
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto(b’x00’, (terminal, port))
sock.close()
```

Terminal receives the UDP packet. According to the content, the authentication message is considered as valid or invalid.

```
port = 5005
localhost = 192.168.2.190
sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
sock.bind((localhost, port))
while True:
   message, addr = sock.recvfrom(1024)
   if message == b’x00’:
      print("Authentication Valid......")
   else:
      print("Authentication is not valid :(")
   break
sock.close()
```

On the other hand, TCP connection is considered as a reliable communication especially used for the data transfer. The terminal operates as a server and the Vocore2 as a client. The server needs to bind the IP address and the port to listen on it. Then the connection has to be established with the three-way handshake to be synchronized and ensure that packets will not be lost during the transmission. The following example is a part of AES-CCM approach and it shows how an encrypted firmware declared as data is sent to Vocore2. Terminal listens on the binded port to connect one client. Then the encrypted firmware is send with *SendFile* function to Vocore2. The *AcceptMessage* function informs the terminal about the status of decryption on the Vocore. If the decryption is successful, a specific message 'x00' translated as 'allow' is sent from Vocore2 and the socket is closed.

```
port= 5010
terminal = 192.168.2.190
s = socket.socket()
s.bind((terminal,port))
s.listen(1)
while True:
   c, addr = s.accept()
   print("client connected ip:<", str(addr), ">")
   t = threading.Thread(target=SendFile,args=("retrThread",c))
   t.start()
   if AcceptMessage() == "allow":
      s.close()
   break
```

Vocore2 receives data with an encrypted firmware and save them into a file.

```
port= 5010
terminal = 192.168.2.190
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((terminal, port))
data = s.recv(1024)
s.close
```

11.3 Cryptography

An application of cryptography is ensured with the PyCryptodome module. The process of encryption and decryption is almost identical. The following example shows function for encryption for the firmware with the AES-CBC mode.

```
def encrypt(key, filename):
    chunksize = 64*1024
    outputFile = "encrypted"+filename
    filesize = str(os.path.getsize(filename)).zfill(16)
    encryptor = AES.new(key, AES.MODE_CBC, hex_array_iv_cbc)
    with open(filename, 'rb') as infile:
        with open(outputFile, 'wb') as outfile:
            outfile.write(filesize.encode('utf-8'))
            outfile.write(hex_array_iv_cbc)
            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                elif len(chunk) % 16 !=0:
                    chunk += b' ' * (16 - (len(chunk) % 16))
                outfile.write(encryptor.encrypt(chunk))
```

The process reads data from the firmware and encrypts them with the object encryptor which needs to declare outputs as the key, the IV and the AES mode. The encrypted data consists of the file size of an original firmware and the firmware itself. Together it forms an encrypted firmware.

The Decryption process is inverse to the encryption. The object decryptor has to be created and the encrypted file is read and then it is written into a new file consist of original firmware. The file size which is written at the beginning of an encrypted firmware serves to decrypt the correct size of the firmware.

11.4 Command Management

The script located on the Vocore2 needs to manage processes and execute commands to cooperate with the Suricata and hardware USB key. The script has to be able to run Suricata and rewrite rules to match signatures if the authentication and firmware decryption is validated. Cooperation with the hardware USB key consists of mounting the device to the system and reading the PRN from the hardware key. The following example shows a creation of the default signatures and running the Suricata. The variable of the subprocess is created for terminating the Suricata while the new rules are created.

Otherwise, if the Suricata is not rebooted, the newly created signatures are not initialized.

```
with open(completename, ’w’) as outfile:
   outfile.write(’drop ip any any -> any 22
   (msg: "Default settings for upgrade testing";)’)
cmd = "suricata -c /etc/suricata/suricata.yaml -q0"
p = subprocess.Popen(cmd, shell=True)
```

Chapter 12

Measurement

Embedded systems tend to be low power devices. The Vocore2 can be considered as an embedded device with more computing resources. The usage of computing resources is measured to picture how demanding is the proposed method of a firmware upgrade. The measurement is conducted on the Vocore2 and it consists of time for an upgrade, CPU (Central Processing Unit) usage and VSZ (Virtual Memory Size) usage. The measured approaches are AES-CCM mode and AES-CCM+CBC mode with separate authentication and encryption. The compared result also shows the difference between the used key size. The key is 128 bits length constructed with MD5 hash function or 256 bits length constructed with SHA-256 hash function.

Time for the upgrade is measured on the terminal in the Integrated Development Environment (IDE) Pycharm. Import of the time module is required to see the time output. The measured time covers the whole process to get the access to the device, where the decrypted firmware is prepared for an upgrade. It does not include a management of the firmware because the management is controlled by a user and it is independent on a used mode of an operation. The following results show the average time from 10 values measured for each method. The longest time of an upgrade using AES-256 CCM mode takes 12.972 seconds. It can be seen that use of the smaller key size can slightly make the upgrade quicker.

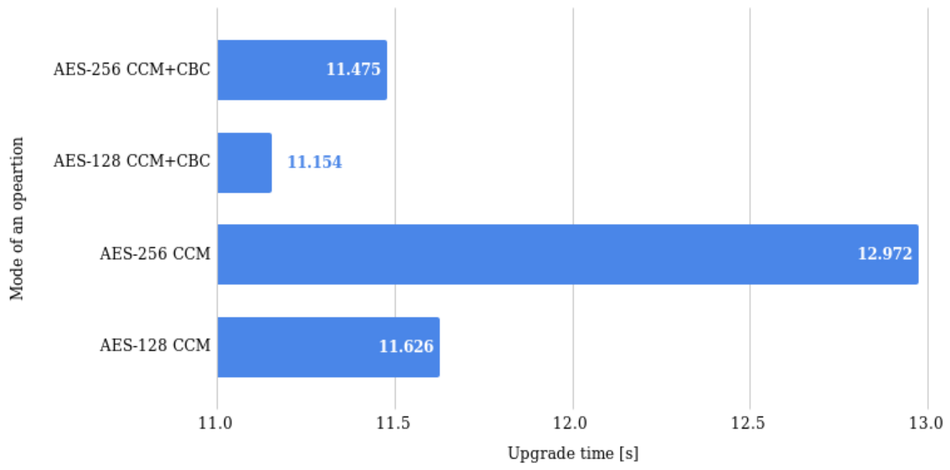


Table 12.1: Firmware upgrade time

The CPU usage is measured on the Vocore2. The measurement is conducted until the settings of a new access policy when the firmware is prepared to be managed. The method of measurement consists of using *top* program which provides a real-time view of a running system. The *top* program captures data and the output is redirected into the *measurement_ top.txt* file in Vocore2. The capturing is executed every second and the command is added to the Python auto-run script. At the end of the measurement, the captured data have to be filtered to obtain only information about Suricata and Python script. A *grep* is another tool that helps to accomplish that. The following example is responsible for creating the *top* output containing CPU values used by Suricata

```
top -b -d 1 > measurement_top.txt
cat measurement_top.txt | grep Suricata
```

The VSZ usage shows how the tasks use the available memory. Virtual memory maps memory addresses used by a task. The addresses can be targeted to the RAM or to a flash memory of a Vocore2. The same process for the measurement as for the CPU usage is used.

The first chart shows computing resources usage of the Suricata. The result of using another AES modes is still with the same progress. It can be seen that running Suricata is not CPU demanding. At the beginning of the task, the higher CPU usage can be detected. During the upgrade process, the level of CPU usage is under 10 %. The decline of CPU and VSZ is caused by creating new Suricata rules and re-running the Suricata.

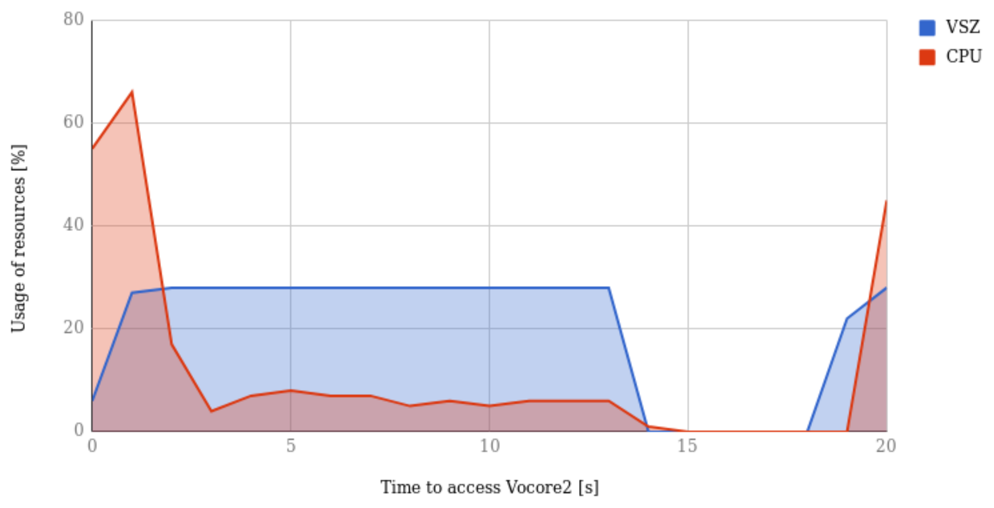


Table 12.2: Suricata AES-256 CCM+CBC

The more computing demanding is an auto-run script which does not affect the system during an inactive run. However, during the firmware upgrade, there is a substantial increase to the level of 90 % of CPU usage. The first peak of CPU usage shows an authentication process which is performed in a more demanding CCM mode. Then the decrease shows the time while the encrypted firmware is being transferred. The second peak shows the decryption of the firmware in the CBC mode which consumes the same amount of CPU. On the other hand, the time of CBC decryption is lower than the time during authentication. At the end, the scripts behave like an inactive and the usage of CPU is not notable. VSZ remains on almost the same level for the whole process.

The second example of an auto-run script shows AES-256 CCM mode. The inactive mode is not correctly adjusted and it shows 30 % of CPU usage which should be lowered in the real usage. The issue is a TCP connection which tries to establish a connection in a loop between the terminal and the Vocore2. Nevertheless, the process of upgrade shows different continuity than AES-CCM+CBC. The authentication and decryption are put together and the chart clearly shows this fact. The slight increase shows decryption with CTR mode to obtain MAC and firmware. The highest level of computing usage represents calculating of MAC with CBC mode and a comparison of values. This procedure reaches the maximum of 94 % of CPU usage. After the process, CPU usage decline to the level of an inactive mode. The VSZ is the constant in this case at the level of 15 % of the VSZ usage.

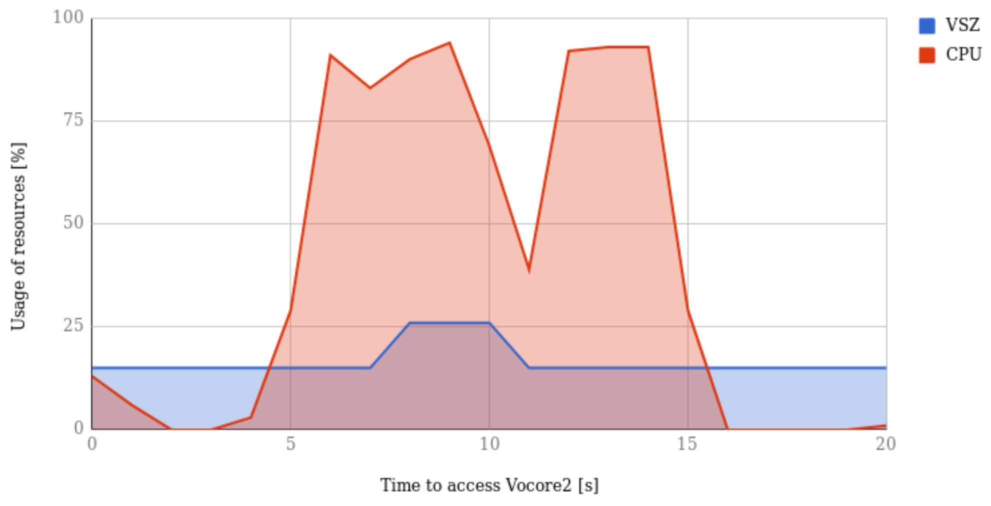


Table 12.3: Python script AES-256 CCM+CBC

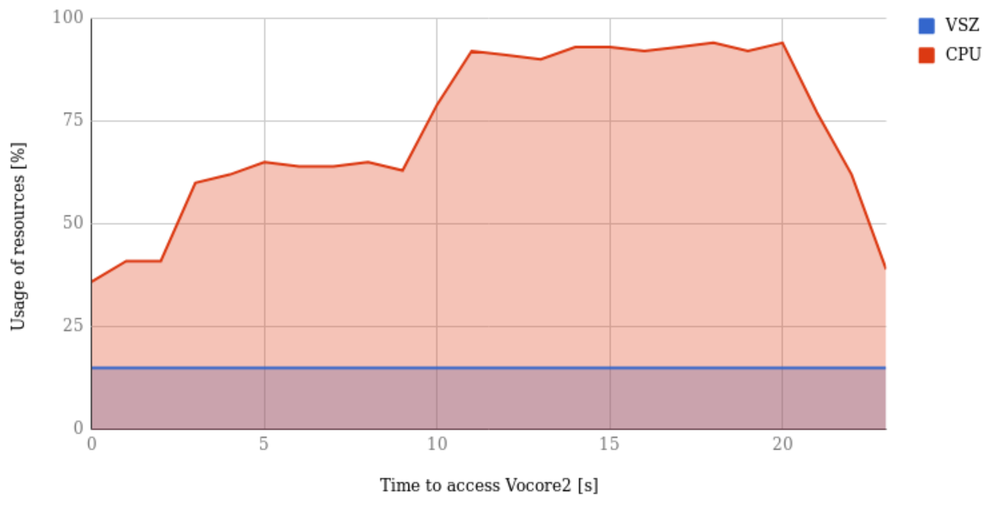


Table 12.4: Python script AES-256 CCM

Chapter 13

Conclusions

The research of the IoT security vulnerabilities showed that the most exposed devices to unsolicited actions are mostly home devices. The end users are usually not concerned about the security of their devices which are connected to the Internet. Manufacturers should put more effort to accurately inform customers about the security vulnerabilities of their systems. The performed attacks can affect the target diversely, nevertheless the most critical threat is the firmware modification. If users were aware of using the default password and using open ports, most of the attacks would be unsuccessful and more complex attacks have to be devised up. On the other hand, the security of firmware management of embedded devices used in an industrial area is commonly resistant to the attacks. To avoid modification of the firmware, implementations of secure firmware upgrade are discovered. The most applied implementations are bootloaders which verify the firmware before the booting process. Nevertheless, an analysis of these bootloaders also known as the crypto bootloaders was conducted and the weakness of this implementation was discovered.

The vast amount of security elements on embedded devices tend to secure the device with the strong cryptographic procedure. However, an embedded device has computing power limitations and the security has to bear in mind the hardware used for the device. To improve the security of the embedded device, the prevention against the attack can be a solution. The devices are remotely controlled and they are exposed to the Internet. The prevention consists of controlling network traffic going through the network interface of the embedded device. For the prevention, IPS system Suricata is used. It controls the network traffic with the signatures and it can be considered as the prevention mechanism which can recognize a possible attack. In the practical part of the thesis, a secure firmware upgrade procedure was proposed. The proposed design consists of the prevention system which can cooperate with the cryptographic procedure formed with the hardware security. Together it can ensure the highest level of security.

Generally, the embedded devices can be very diverse. The hardware resources and the type of operations are different. Some embedded devices collecting data do not need a higher computing power, because the power consumption is the most crucial. Therefore, an optimization and stability of the proposed secure upgrade have to be performed. The measurement of firmware upgrade procedure was conducted on the embedded platform Vocore2 to see how the process is demanding to power resources. The result showed that the firmware upgrade process considered as an active mode is very demanding for Vocore2's processor with the frequency of 580 MHz. However, the computing performance was sufficient. The inactive mode of the security elements consisted of Suricata and auto-run script which run in the background showed the usage of the processor under 10 % and usage of virtual memory about 43 %. The hardware resources of Vocore2 were satisfactory, only the higher usage of virtual memory during the inactive mode can be the reason for higher power consumption. As a result, the proposed firmware procedure can target the embedded devices with higher computing resources as gateways. The very low power devices such as dataloggers would be probably not sufficient. Nevertheless, these devices can be considered as an end devices. They should be hidden in the network behind the gateway to not be directly exposed to the Internet. Then the less computing demanding procedure of the upgrade firmware can be applied to them.

Bibliography

[1] IEEE Internet Initiative. Internet of things research study, May 2015. [Online]. Available at: `https://iot.ieee.org/images/files/pdf/IEEE_IoT_Towards_Definition_Internet_of_Things_Revision1_27MAY15.pdf` [Accessed: 25-February-2018].

[2] Hewlett Packard Enterprise. Towards a definition of the Internet of Things (IoT), November 2015. [Online]. Available at: `http://files.asset.microfocus.com/4aa5-4759/en/4aa5-4759.pdf` [Accessed: 27-February-2018].

[3] Amy Nordrum. Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated, August 2016. [Online]. Available at: `https://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated` [Accessed: 02-March-2018].

[4] Telecom Engine. The Future of IoT is Embedded Systems and Real-time Operating Systems (RTOS), June 2016. [Online]. Available at: `http://www.telecomengine.com/the-future-of-iot-is-embedded-systems-and-real-time-operating-systems-rtos-2/` [Accessed: 02-March-2018].

[5] BusyBox. [Online]. Available at: `https://busybox.net/about.html` [Accessed: 10-March-2018].

[6] Steven Perry. Anatomy of an IoT malware attack (and what to do to prevent one), October 2017. [Online]. Available at: `http://www.ibm.com/developerworks/library/iot-anatomy-iot-malware-attack/index.html` [Accessed: 10-March-2018].

[7] VDC Research. The Global Market for IoT Embedded Operating Systems, 2017. [Online]. Available at: `https://www.vdcresearch.com/_documents/briefs/IoT/17-embedded-OS.pdf` [Accessed: 11-March-2018].

[8] Anna Gerber. Choosing the best hardware for your next IoT project, January 2018. [Online]. Available at: `http://www.ibm.com/developerworks/library/iot-lp101-best-hardware-devices-iot-project/index.html` [Accessed: 11-March-2018].

[9] T. Noergaard. *Embedded systems architecture: a comprehensive guide for engineers and programmers*. Elsevier/Newnes, 2005.

[10] Rick Lehrbaum. AMD reveals roadmap for ARM and x86 SoCs, September 2013. [Online]. Available at: `http://linuxgizmos.com/amd-reveals-arm-and-x86-soc-and-apu-plans/` [Accessed: 15-March-2018].

[11] The OpenWrt Flash Layout [OpenWrt Wiki]. [Online]. Available at: `https://wiki.openwrt.org/doc/techref/flash.layout` [Accessed: 15-March-2018].

[12] VoCore Studio. VoCore2 | Coin-sized Linux Computer. [Online]. Available at: `http://vocore.io/v2u.html` [Accessed: 10-March-2018].

[13] M. Scott S. J. Johnston and S. J. Cox. Recommendations for securing Internet of Things devices using commodity hardware, December 2016. [Online]. Available at: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7845410` [Accessed: 18-March-2018].

[14] Dave McMillen. Mirai IoT Botnet: Mining for Bitcoins?, April 2017. [Online]. Available at: `https://securityintelligence.com/mirai-iot-botnet-mining-for-bitcoins/` [Accessed: 18-March-2018].

[15] A. Stavrou C. Kolias, G. Kambourakis and J. Voas. DDoS in the IoT: Mirai and Other Botnets, July 2017. [Online]. Available at: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7971869` [Accessed: 18-March-2018].

[16] Michael Mimoso. Mirai Bots More Than Double Since Source Code Release, October 2016. [Online]. Available at: `https://threatpost.com/mirai-bots-more-than-double-since-source-code-release/121368/` [Accessed: 18-March-2018].

[17] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis

Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai Botnet, 26th USENIX Security Symposium (USENIX Security 17), 2017. [Online]. Available at: `https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-antonakakis.pdf` [Accessed: 14-March-2018].

[18] C. Kolias G. Kambourakis and A. Stavrou. The Mirai botnet and the IoT Zombie Armies, 2017. [Online]. Available at: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8170867` [Accessed: 18-March-2018].

[19] Thomas Claburn. Intel Management Engine pwned by buffer overflow, December 2017. [Online]. Available at: `https://www.theregister.co.uk/2017/12/06/intel_management_engine_pwned_by_buffer_overflow/` [Accessed: 21-March-2018].

[20] Limor Kessem. Managing Security Risk in the Face of Intel ME Vulnerabilities, November 2017. [Online]. Available at: `https://securityintelligence.com/news/ibm-security-advisory-on-intel-management-engine-vulnerability/` [Accessed: 21-March-2018].

[21] J. MA J. YANG, G. LIU and W. YANG. UbootKit: A Worm Attack for the Bootloader of IoT Devices Authors, 2017. [Online]. Available at: `https://www.blackhat.com/docs/asia-18/asia-18-Yang-UbootKit-A-Worm-Attack-for-the-Bootloader-of-IoT-Devices-wp.pdf` [Accessed: 21-March-2018].

[22] M. Scott S. J. Johnston and S. J. Cox. Recommendations for securing Internet of Things devices using commodity hardware, 2016. [Online]. Available at: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7845410` [Accessed: 22-March-2018].

[23] L. Z. Cai and M. F. Zuhairi. Security Challenges for Open Embedded Systems , 2017. [Online]. Available at: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8215952` [Accessed: 25-March-2018].

[24] George Corser. INTERNET OF THINGS (IOT) SECURITY BEST PRACTICES, February 2017. [Online]. Available at: `https://internetinitiative.ieee.org/images/files/resources/white_papers/internet_of_things_feb2017.pdf` [Accessed: 25-March-2018].

[25] M. Costello A. Cui and S. Stolfo. When Firmware Modifications Attack: A Case Study of Embedded Exploitation, 2013. [Online]. Available at: `https://academiccommons.columbia.edu/catalog/ac:198970` [Accessed: 27-March-2018].

[26] L. Reynoso O. Guillen, B. Nisarga and R. Brederlow. Crypto-Bootloader - Secure In-Field Firmware Updates for Ultra-Low Power MCUs, September 2015. [Online]. Available at: `http://www.ti.com/lit/wp/slay041/slay041.pdf` [Accessed: 29-March-2018].

[27] AT11787: Safe and Secure Firmware Upgrade via Ethernet, 2015. [Online]. Available at: `http://ww1.microchip.com/downloads/en/AppNotes/Atmel-42492-Safe-and-Secure-Firmware-Upgrade-via-Ethernet_ApplicationNote_AT11787.pdf` [Accessed: 29-March-2018].

[28] W. Fuertes G. Guerrero M. Macas, L. Lagla and T. Toulkeridis. Data Mining model in the discovery of trends and patterns of intruder attacks on the data network as a public-sector innovation, 2017. [Online]. Available at: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7962513` [Accessed: 29-March-2018].

[29] Karel Burda. *Aplikovaná Kryptografie.* VUTIUM, 2013.

[30] S. M. Matyas X. Wang, D. Coppersmith and C. H. Meyer. Cryptography, 2014. [Online]. Available at: `http://www.accessscience.com.ezproxy.techlib.cz/content/170600` [Accessed: 03-March-2018].

[31] Ch. Paar and J. Pelz. *Understanding Cryptography.* Springer Heidelberg Dordrecht London New York, 2010.

[32] AES-CCM Attack - ChipWhisperer Wiki, March 2017. [Online]. Available at: `https://wiki.newae.com/AES-CCM_Attack` [Accessed: 05-March-2018].

[33] N. Ferguson D. Whiting and R. Housley. Counter with CBC-MAC (CCM), September 2013. [Online]. Available at: `https://tools.ietf.org/html/rfc3610` [Accessed: 20-February-2018].

[34] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, 2007. [Online]. Available at: `https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38c.pdf` [Accessed: 03-March-2018].

[35] Karel Burda. *Úvod do Kryptografie.* AKADEMICKÉ NAKLADATELSTVÍ CERM, s.r.o, 2015.

[36] Richard E. Smith. *Internet Cryptography.* Addison Weley Longman, Inc, 1997.

[37] National Institute of Standards and Technology. SECURE HASH STANDARD, August 2002. [Online]. Available at: `https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf` [Accessed: 05-March-2018].

[38] National Institute of Standards and Technology. Descriptions of SHA-256, SHA-384, and SHA-512. [Online]. Available at: `http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf` [Accessed: 05-March-2018].

[39] What is a Security Operations Center (SOC)?, November 2016. [Online]. Available at: `https://digitalguardian.com/blog/what-security-operations-center-soc` [Accessed: 04-April-2018].

[40] K. A. Scarfone and P. M. Mell. Guide to Intrusion Detection and Prevention Systems (IDPS), 2007. [Online]. Available at: `https://ws680.nist.gov/publication/get_pdf.cfm?pub_id=50951` [Accessed: 04-April-2018].

[41] Suricata Rules - Suricata - Open Information Security Foundation, 2011. [Online]. Available at: `https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Suricata_Rules` [Accessed: 04-April-2018].

[42] R. Ziegler and S. Suehring. iptables: The Linux Firewall Administration Program, November 2005. [Online]. Available at: `http://www.informit.com/articles/article.aspx?p=421057&seqNum=2` [Accessed: 04-April-2018].

[43] Open Information Security Foundation. Setting up IPS/inline for Linux — Suricata 4.0.0-dev documentation, 2016. [Online]. Available at: `http://suricata.readthedocs.io/en/suricata-4.0.4/setting-up-ipsinline-for-linux.html`[Accessed: 04-April-2018].

[44] OpenWrt Project: About the OpenWrt/LEDE project, February 2018. [Online]. Available at: `https://openwrt.org/about` [Accessed: 07-April-2018].

[45] S. Lin. suricata: OpenWRT Suricata package, February 2018. [Online]. Available at: `https://github.com/seanlinmt/suricata` [Accessed: 14-March-2018].

[46] Rootfs on External Storage (extroot) [OpenWrt Wiki], April 2017. [Online]. Available at: `https://wiki.openwrt.org/doc/howto/extroot` [Accessed: 15-April-2018].

[47] L. Vojtech L. Kvarda, P. Hnyk and M. Neruda. Software Implementation of Secure Firmware Update in IoT Concept, 2017. [Online]. Available at: `http://advances.utc.sk/index.php/AEEE/article/view/2467` [Accessed: 03-February-2018].

Appendices

Appendix A

DVD contents

The following table shows directories which are contained on the DVD.

Directory name	**Description**
compiled_openwrt	Compiled firmware
measurement	Tables and charts of measured data
photo_documentation	Photo documentation of Vocore2
scripts_terminal	Python scripts which manage the procedure on a computer
scripts_vocore2	Python scripts which manage the procedure on Vocore2
thesis	This bachelor thesis

Table A.1: DVD content