BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

MULTIPLATFORM LINUX SANDBOX FOR ANALYZING IOT MALWARE

LINUXOVÉ VÍCEPLATFORMNÍ ODDĚLENÉ BĚHOVÉ PROSTŘEDÍ PRO ANALÝZU

MALWARE V IOT

BACHELOR’S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **DANIEL UHŘÍČEK**
AUTOR PRÁCE

SUPERVISOR **Doc. Dr. Ing. DUŠAN KOLÁŘ**
VEDOUCÍ PRÁCE

BRNO 2019

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2018/2019

Bachelor's Thesis Specification

22120

Student:	**Uhříček Daniel**
Programme:	Information Technology
Title:	**Multiplatform Linux Sandbox for Analyzing IoT Malware**
Category:	Security

Assignment:

1. Get acquainted with the IoT malware and state of the art in ELF binary analysis. Focus on sandboxing methods, static and network analysis.
2. Analyze existing solutions of *nix sandboxes and ELF analysis methods.
3. Propose an implementation of a sandbox system to analyze ELF binaries. Design crucial components and consider use of known tools.
4. Implement proposed methods of analysis and prepare a sandbox environment.
5. Thoroughly test your solutions on real IoT malware samples.
6. Verify and validate your solution if it meets requirements described in point (3).
7. Discuss the achieved results and possible future work.

Recommended literature:

- Cozzi, E., Graziano, M., Fratantonio, Y., & Balzarotti, D. (2018). Understanding Linux Malware. In: 2018 IEEE Symposium on Security and Privacy (SP), 161-175.
- According to the recommendation of the supervisor and the consultant

Requirements for the first semester:

- First three items and elaboration of the fourth item.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor:	**Kolář Dušan, doc. Dr. Ing.**
Consultant:	Jursa David, Avast
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	November 1, 2018
Submission deadline:	May 15, 2019
Approval date:	October 24, 2018

Abstract

Diversity of processor architectures used by IoT devices complicates IoT malware analysis. This thesis summarizes current state of static, dynamic, and network analysis and it evaluates existing open source solutions of sandboxes providing automated analysis. It proposes a design of a modular system that is easy-to-use, has available REST API, and web interface. The implementation supports five processor architectures. It was tested on current IoT malware samples.

Abstrakt

Analýza IoT malwaru je problematická zejména pro množství a rozlišnost architektur procesorů používaných IoT zařízeními. Práce shrnuje možnosti statické, dynamické a síťové analýzy Linuxového malwaru a hodnotí existující open source řešení oddělených běhových prostředí pro automatizovanou analýzu. Práce navrhuje modulární, rozšířitelný systém s jednoduchými možnostmi nasazení, dostupnou API a webovým rozhraním. Výsledná implementace podporuje pět architektur a byla testována na vzorcích IoT malwaru.

Keywords

Linux sandbox, IoT malware, static analysis, dynamic analysis, network analysis, YARA, SystemTap

Klíčová slova

Linuxové oddělené běhové prostředí, IoT malware, statická analýza, dynamická analýza, síťová analýza, YARA, SystemTap

Reference

UHŘÍČEK, Daniel. *Multiplatform Linux Sandbox for Analyzing IoT Malware.* Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Dr. Ing. Dušan Kolář

Rozšířený abstrakt

Dlouhou dobu se antivirové společnosti a výzkumné skupiny zabývající se IT bezpečností soustředily primárně na malware pro platformy Windows, Android a iOS. Tyto platformy mají většinový podíl trhu, tudíž jsou i žádaným cílem hackerů. S rozvojem Internetu věcí (angl. *Internet of Things, IoT*) přibývá množství zařízeních připojených k Internetu. Tyto zařízení – SOHO routery, IP kamery, chytré lednice apod. – mívají typicky menší výpočetní výkon. Jejich firmware bývá postaven na variantách operačního systému Linux. IoT zařízení jsou často slabě zabezpečené, používají slabá hesla či obsahují zranitelnosti přímo ve firmwaru. To má za důsledek zvýšený zájem útočníků o tvorbu malwaru pro tyto zařízení. Hlavní motivací je zejména tvorba tzv. *botnetů*, které jsou nabízeny jako služba provádějící útoky typu DDoS. Největší zaznamenané botnety byly tvořeny až 600 000 IoT zařízeními. Ty byly součástí útoků např. na majoritního poskytovatele DNS Dyn nebo telekomunikační společnost Deutsche Telekom. Malware Mirai, který botnet vytvořil, má veřejný zdrojový kód, což vede ke vzniku nových variant právě této rodiny.

Detekci a ochraně před škodlivým softwarem předchází jeho důkladná analýza. Na základní úrovni můžeme analýzu rozdělit na statickou a dynamickou – dle toho zda pozorujeme statické rysy bez potřeby program spustit, nebo samotné chování programu za běhu. Základní statická analýza zahrnuje zkoumání formátu spustitelného souboru – pro platformu Linux to je ELF (Executable and Linkable Format). Nápomocná může být i pouhá analýza tisknutelných řetězců obsažených v souboru. Ke statické analýze přispívají také metody reverzního inženýrství, které se snaží získat co nejpřesnější reprezentaci analyzovaného programu v některém z vyšších programovacích jazyků. Práce zmiňuje hlavní dostupné nástroje pro reverzní inženýrství a forenzní analýzu malwaru. Za zmínku stojí zveřejnění zdrojových kódu nástrojů RetDec (Avast Software, prosinec 2017) a Ghidra (NSA, duben 2019). V neposlední řadě je dnes hojně využíván nástroj pro detekci vzorů YARA. YARA umožňuje v jednoduchém formátu definovat pravidla pro detekci a klasifikaci malwaru. Statická analýza však může být problematická v případě, že autor malwaru využil technik obfuskace. Poté je třeba přejít k analýze programu za běhu.

Systémová volání jsou prostředek pro komunikaci uživatelských programů s jádrem operačního systému. Typický nástroj pro monitorování systémových volání strace, podobně jako ladící nástroje, je implementován systémovým voláním *ptrace*. Volání ptrace je však lehce odhalitelné. Druhou možností je monitorování přímo na úrovni jádra. Jelikož chování IoT malwaru je dnes charakterizováno zejména síťovou komunikací, je síťová analýza podstatná složka analýzy malwaru. Podstatné aplikační protokoly pro analýzu IoT malwaru jsou DNS, IRC, Telnet a HTTP. DNS je využíváno k rezoluci domén, v případě botnetů např. k vyhledávání IP adres kontrolních serverů. Protokol IRC je využíván k ovládání botů. Zachycenými informacemi pak mohou být např. IRC kanály nebo příkazy zasílané botům. Telnet je využíván k připojení k IoT zařízením. Útok na nezabezpečené Telnet služby je hlavní metoda šíření IoT malwaru. Speciální HTTP požadavky bývají prostředek k útokům na zranitelnosti webových rozhraní IoT zařízení. Nezabezpečené parametry požadavku mohou útočníkovi poskytnout možnosti vzdáleného spuštění kódu. Spouštění programu a následná dynamická a síťová analýza musí probíhat v odděleném běhovém prostředí.

Práce shrnuje klady a zápory existujících open source řešení oddělených běhových prostředí pro automatizovanou analýzu. Hlavními nevýhodami analyzovaných řešení byla nedostatečně podrobná analýza a nedostatečné množství podporovaných architektur procesorů. Slibné možnosti automatizované analýzy nabízí Cuckoo Sandbox, ač je primárně určen pro analýzu na platformě Windows. Cuckoo Sandbox je však příliš robustní a neplní podstatnou úlohu přípravy běhového prostředí, která je ponechána na uživateli. Díky

zjištěným poznatkům o existujících řešeních byly stanoveny funkční požadavky pro navrhované řešení.

Celkové řešení by se dalo rozdělit do tří částí. Za prvé je to příprava odděleného běhového prostředí, ve kterém bude malware spouštěn. Druhou část tvoří jednotlivé analýzy. Poslední částí je celková architektura systému. V rámci přípravy běhového prostředí se řeší konfigurace jádra operačního systému. Správná konfigurace je nutná pro možnost monitorovat na úrovni jádra a tedy i předejít odhalení malwerem. Jádro Linuxu společně s kořenovým souborovým systémem muselo být přeloženo napříč architekturami. Pro samotný překlad byl využit nástroj Buildroot, který slouží k vytváření vestavěných Linuxových distribucí. Emulace procesorů a systému je prováděna skrz emulátor QEMU. QEMU obrazy jsou automatizovány přes sériové rozhraní. Části analýzy jsou navrženy, aby byly rozšiřitelné. Výstup celkové analýzy je ve formátu JSON. Tento výstup je vytvořen z mezivýstupů jednotlivých modulů. Základní navržené moduly zahrnujou statickou analýzu, dynamickou analýzu, síťovou analýzu a modul pro komunikaci s veřejnou API služby VirusTotal. Pro zpracování výstupů ve formátu JSON bylo navrženo a implementováno rozšíření pro nástroj YARA. Díky tomuto rozšíření mohou YARA pravidla obsahovat podmínky týkající se manipulovaných souborů, systémových volání, DNS rezolucí, HTTP požadavků, IRC zpráv atp. Výsledná architektura systému kromě sady analyzačních modulů zahrnuje i napojení na databázi, REST API a webové rozhraní. Systém povoluje současný běh více analyzačních jednotek, které pak zpracovávají požadavky z interních front.

Pro jednotlivé analyzační moduly byly připraveny sady jednotkových testů. Testována byla jak funkčnost zpracování získaných dat, tak správnost emulování na všech architekturách. Výsledný systém byl testován na datasetu čítající 150 vybraných vzorků IoT malwaru. Projekt byl zveřejněn v dubnu 2019 na portálu GitHub, kde získal první uživatele z komunity bezpečnostních výzkumníků. Mimo to byl také prezentován na studentské konferenci inovací, technologií a vědy v IT Excel@FIT.

Multiplatform Linux Sandbox for Analyzing IoT Malware

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of doc. Dušan Kolář and David Jursa. All the relevant sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

....
Daniel Uhříček
May 15, 2019

Acknowledgements

I want to thank doc. Dušan Kolář for guidance during writing this thesis and for providing valuable feedback. I want to thank David Jursa for all technical consultations. I would also like to thank Dr. Jakub Křoustek for his support and inspiration caused by his immense work ethics. Last but not least, I would like to express my thanks to whole Avast intern squad (especially Matej Kašťák).

Contents

1 Introduction **3**

2 IoT Malware and Its Analysis **4**
2.1 Malware . . . 4
2.2 Executable and Linkable Format . . . 6
2.3 Static Analysis . . . 6
2.4 Dynamic Analysis . . . 9
2.5 Network Analysis . . . 12
2.6 Sandboxing . . . 15

3 Existing Solutions **16**
3.1 REMnux . . . 16
3.2 Detux . . . 16
3.3 Limon . . . 17
3.4 Cuckoo Sandbox . . . 17
3.5 Padawan Sandbox . . . 18
3.6 Summary and Proposing General Requirements . . . 18

4 Sandboxed Linux Environment **20**
4.1 QEMU . . . 20
4.2 Considering Existing Systems . . . 20
4.3 Linux Environment Preparation . . . 21
4.4 Communication and Automation . . . 22

5 Designing Framework for Automated Linux Malware Analysis **24**
5.1 Analysis Pipeline . . . 24
5.2 Analysis Output and Further Processing . . . 29
5.3 Microservices Architecture . . . 30

6 Implementation **32**
6.1 Analysis Implementation . . . 32
6.2 Architecture Implementation and Conteinerization . . . 33

7 Verification and Validation **35**
7.1 Testing . . . 35
7.2 Summary . . . 36

8 Possible Future Work **38**

9 Conclusion **39**

Bibliography **40**

A CD Content **42**

Chapter 1

Introduction

Over the past few years, we could witness rise of Internet of Things (IoT) as a concept of connecting the world of embedded devices to the Internet. SOHO routers, IP cameras, smart fridges, and many other IoT devices usually run some kind of embedded Linux firmware. It is quite common that IoT devices have weak security standards. Whether it is because of the low available processing power of embedded devices, or because of the cheaper price and thus cheaper development. System vulnerabilities, exposed telnet and ssh services, or outdated firmware – all of those allow attackers to compromise more devices. Malware researchers were primarily focused on Windows malware analysis for many years. First large-scale comprehensive study of Linux malware was conducted by researches from Eurocom and Cisco [8]. Their dataset consisted of 10 548 samples of various malware strains.

Manual analysis of potential malware is a time expensive task. The goal of this thesis is to design and implement a solution for automated IoT malware analysis. The solution should be extensible with an available plugin system. Extensibility and modularity are crucial because of the dynamics of information security field. The solution should be easy to setup and manage to be suitable also for small malware analysis teams. The solution should be able to reliably emulate multiple target architectures. This reliable emulation overcomes anti-debug protections. Overall analysis results contain static analysis, dynamic analysis, and network analysis related information.

This thesis is organized as follows: Chapter 2 focuses on the current state of IoT malware and its analysis. It outlines necessary background for automated malware analysis, and it states some possible problems. It analyzes common tools used in static, dynamic, and network analysis. Chapter 3 evaluates existing solutions implementing sandboxed malware analysis. Freely available projects REMnux, Detux, Limon, and Cuckoo Sandbox are considered. The chapter then summarizes all of the existing solutions, and it defines functional requirements. Chapter 4 describes preparation of sandboxed Linux environment. It solves cross-compilation and sandbox communication and automation problems. Chapter 5 proposes a design of the sandbox system. It describes an analysis pipeline, its individual parts, and overall system architecture. Chapter 6 illustrates an implementation of the system. Chapter 7 describes process of testing, verification, and validation. Finally, Chapter 8 summarizes future project goals.

Chapter 2

IoT Malware and Its Analysis

This chapter focuses on the current state of IoT malware and on possible ways to analyze IoT malware. Firstly, the chapter takes into account different malware types, and it states some important differences between previously known malware and IoT malware. Next, it briefly describes Executable and Linkable Format – the file format of IoT malware that was considered in this thesis. Afterwards, sandboxing and multiple analysis techniques are described. The end of this chapter shows examples of IoT malware strains.

2.1 Malware

Malicious software (malware) [27] is a software that might cause any type of damage to a target user, computer, or network. Malware is being created and used by attackers for financial profit, educational purposes, or for the sole purpose of causing harm. We can classify malware according to its typical behavior [27, 10].

- **Virus** is a self-replicating program that usually attaches itself onto executable files without user's attention. Once these binaries are somehow transferred to some other system, virus again self-replicates and infects the system.
- **Worm** replicates using network. Worm might exploit application or system vulnerabilities, or it can spread via e-mail or social networks.
- **Backdoor** compromises target systems and sets an entry point for attacker. Commonly used backdoors use the technique called reverse shell. To setup a reverse shell, malware connects to attackers machine to forward its shell session.
- **Botnet** is created by infecting many targets. After the infection, these targets, known as bots or zombies, are controlled by one entity. This entity is called command-and-control server (C&C). C&C servers are managed by attacker and, later, they are used to send individual commands (e.g. to set specific target of their attack) to individual bots.
- **Downloaders'** sole purpose is to download files from the Internet – commonly other malicious programs.
- **Information-stealing malware** or spyware collects information from a target computer. Spyware can also act as a keylogger. Keylogger logs pressed keys and analyzes

them or directly forwards them to attacker. Other information-stealing software can be searching for configuration files, web-browser's cookies, etc.

- **Spamware** uses target machines to send spam and generate profit. Spamware sending e-mail spam is the most common kind of spamware. Nowadays, spamware targets also social networks.

- **Trojan horse** is usually a fully functional program that is extended by a malicious part. Users are unaware of a hidden malicious part that infects their system.

2.1.1 IoT Malware

IoT malware [8] is a malicious program targeting any IoT device. In comparison with Windows, Android or iOS malware, IoT malware is not that sophisticated. Malicious programs targeting desktop and mobile platforms have been around for many years, and they evolved thanks to the need of overcoming more and more secured systems. Today, almost every desktop computer has antivirus installed. On the other hand, current IoT devices are known to be poorly secured.

Another difference between IoT malware and malware for other platform is its spreading. On Windows or mobile platforms, there is an ecosystem of applications and packages. User usually gets infected during program installation e.g. from app store. IoT and embedded systems tend to be minimal by design with no package management for a user. Malware on IoT spreads mainly via two paths. Firstly, it spreads by searching for open services (Telnet, SSH) that have weak or even default passwords. Secondly, by exploiting application errors contained in devices' firmware.

2.1.2 IoT Malware Families

This section summarizes few different families of IoT malware. The content of this section is based on articles [1, 2, 9] and leaked malware source code. The earliest IoT malware was detected in year 2008. Since then, different strains of malware evolved. During the past years, many IoT devices were compromised through exposed telnet and ssh services, or through exploited vulnerabilities. IoT malware became much more popular after Mirai attacks [18] in year 2016 when up to 600 000 devices delivered huge denial-of-service (DDoS) attacks on targets as OVH (09/18/2016), Krebs on Security (09/21/2016), Dyn (10/21/2016), and Deutsche Telekom (11/26/2016).

IRC Botnets Derived from Linux/Hydra

Linux/Hydra was released in year 2008. Thus, it is the earliest known IoT malware. It targeted MIPS architecture. It spread using a dictionary attack, or by exploiting authentication vulnerability in D-Link routers. Bots were communicating with a C&C server via IRC. Linux/Hydra was able to perform SYN flood DDoS attack.

Psyb0t is a descendant of Linux/Hydra. It was discovered in 2009. It also targeted MIPS architecture. It was again controlled via IRC. Psyb0t compromises IoT devices using brute force attack. Psyb0t performed UDP and ICMP DDoS attacks. Pretty similar is also another Linux/Hydra descendant that was discovered year later – Chuck Norris.

Tsunami/Kaiten malware is also based on Linux/Hydra. It implemented additional attacks as PUSH flood, ACK flood, and HTTP flood. Surprisingly, this malware spread also via hacked website of Linux Mint distribution.

LightAidra

LightAidra targets MIPSel, MIPS, SuperH, ARM, and PowerPC. Botnet is again controlled via IRC. LightAidra implements classic telnet service scanning. SYN flood and ACK flood are the only available attacks. LightAidra is an open source project. Availability of source code lead to custom modifications. Malware strains that derived from LightAidra are for example BASHLITE, Lizkebab, or Torlus. Malware derived from LightAidra often implemented custom attacks or different communication protocol to eliminate the need of IRC server.

Mirai

As mentioned before, Mirai is the most famous of IoT malware strains. Its botnet was able to deliver up to 1.1 Tbps DDoS attack. Shortly after the attacks, author of Mirai released its source code. We can examine ten different attack vectors including DNS water torture attack. When a device is compromised, it starts randomly generating IP addresses. The source code includes a blacklist of IP addresses belonging to General Electric Company, Hewlett-Packard, US Postal Service, and Department of Defense. Mirai checks port 23. It tries to login with 61 predefined login-password combinations. After successful authentication, loader infects another device.

2.2 Executable and Linkable Format

Executable and Linkable Format (ELF) [5, 20] is the main file format for object files in Linux based systems. ELF is defined and maintained by Tool Interface Standard Committee. ELF object file can be either executable, relocatable, or shared object. Executable can be understood as already linked and prepared binary. This binary can run on a system. Relocatable is a file that was processed with a compiler but it needs to be linked by a linker to determine relocations. Shared library can be processed by a linker to link with other object files but it already has some specified executable code.

ELF file starts with an ELF header. Initial bytes of every ELF file are `0x7fELF`. These bytes, called the magic number, serve for quick recognition of such files. Next two bytes specify a class and a byteorder. Class has values 1 or 2. Value of 1 means that the binary targets 32 bit system and value of 2 means that the binary targets 64 bit system. Byteorder field specifies whether the byte order is little-endian (1) or big-endian (2). ELF files can contain sections and segments. Sections are defined by a section header table. They are used by compilers, assemblers, and linkers. Segments are defined by a program header table. Segments are used by program loader to load executable code, read-only data, and symbols.

2.3 Static Analysis

Static analysis [27, 10] of executable files is a process of analysis without running the executable. We can analyze basic characteristics based on the ELF header, or perform analysis of strings and symbols. More sophisticated ways of static analysis include disassembling and decompilation of analyzed executable.

We can statically identify files by their hashes or fingerprints. We often use hashing algorithms like MD5 (Message-Digest Algorithm 5), SHA-1 (Secure Hash Algorithm 1), and

SHA-256 (Secure Hash Algorithm 256), to search for already analyzed malware or to share and search malware files in databases.

2.3.1 Standard Static Analysis Tools in Linux

Essential command available in Linux distributions is `file`[1]. `File` is part of UNIX since 1973. It's primary purpose is to determine a file type. However, it can also parse ELF headers, determine whether ELF binary was linked statically or dynamically, for which platform it was built, etc.

GNU Binutils[2] (binary utilities) also include multiple tools for basic static analysis as:

- **strings** – A utility to find printable character sequences in files. By default, strings searches for sequences with minimal length of four. It is possible to estimate a potential malicious behavior just by looking at file's strings. Strings can contain URLs, specific requests patterns that are send by malware, file paths etc.
- **readelf** – A program to dissect ELF binaries. Readelf can analyze ELF binary headers, examine segments, sections, dynamic symbols, relocations etc.
- **objdump** – Objdump can also inspect segments and sections of an ELF file. Moreover, it functions as a disassembler. Disassemblers decode machine code into assembly language.

2.3.2 Reverse Engineering, Disassembling, and Decompilation

Reverse engineering (reversing) [10] is a general process of analyzing a product, machine, technology, or software to obtain some kind of knowledge. Reversing is often used by malware analysts and antivirus developers. Through malware reversing, they expose true malware function.

Reversing of a malware can start with disassembling. Disassembling is a process of creating assembly representation of an actual machine code data. Assembly language highly depends on target platform. Individual processor architectures have their own specific instruction set, different operation codes and registers.

Decompilation is a process of obtaining source code out of compiled binaries. Decompilation offers high-level representation of a program. This high-level representation is then much easier to analyze. Obtaining full original source code is usually not possible because compilers strip off many language specific information that are not needed for program execution.

Radare2

Radare2[3] is a framework for reverse engineering and forensic analysis. It can disassemble and debug binaries of various architectures. Moreover, radare2 implements several command-line utilities to help with binary analysis. Few of them are:

- **rabin2** – It extracts detailed information from binaries. These information consist for example of architecture details, compiler info, symbols, segments, sections, or library dependencies.

[1]https://linux.die.net/man/1/file
[2]https://www.gnu.org/software/binutils
[3]https://rada.re/r

- **rasm2** – It is a multiplatform assembler and disassembler. It provides the core functionality of radare2 framework.
- **rafind2** – Rafind2 is a useful tool for finding byte patterns in binaries. Technique of pattern matching is often used when analyzing malware to compare it with known signatures.

With radare2, it is possible to examine control flow graphs or determine functions. Control flow graphs are constructed mostly out of disassembled conditional jump instructions. These instructions often represent if-else statements or loops.

Furthermore, radare2 is able to patch and directly modify programs instructions. This can be used to disable anti-analysis techniques that are implemented by malware authors.

Radare2 has also Application Programming Interfaces (APIs) for many programming languages as Ruby, JavaScript, Python, Go, Java, or Rust.

IDA Pro

IDA Pro[4] is an industrial standard for forensic analysis. It serves mainly as an interactive disassembler and debugger. IDA is packed with almost every feature for reversing, that malware analysts can think of.

It provides excellent analysis of functions. IDA stores cross-references (xrefs) to other parts of the code. Through xrefs, analysts can find from where functions are called and where variables and other data was re-used. IDA is often shipped with Hex-Rays decompiler. IDA's main downside is its price.

RetDec

RetDec[5] (Retargetable Decompiler) is an open source decompiler maintained by Avast. Currently it supports x86, x86-64, ARM, MIPS, PIC32, and PowerPC architectures.

RetDec generates call graphs, control-flow graphs. Its decompilation results are comparable to IDA Pro.

On top of that, RetDec's preprocessing tool **fileinfo** can be used for general static analysis and binary information extraction. Besides standard binary parsing and extracting of headers, symbols, sections or segments, fileinfo implements compiler and packer detections with possibility to unpack binaries. RetDec's core is based on modern tools like LLVM, Capstone, and YARA.

Ghidra

National Security Agency (NSA) publicly released the source code of their reverse engineering framework Ghidra[6] in April 2019. It is a feature-rich toolkit with hundreds of features including disassembly, decompilation, and graphs creation. Ghidra outstands IDA Pro in number of supported architectures.

2.3.3 Pattern matching and YARA

Pattern matching is a really important technique in malware analysis and antivirus development. Thanks to well-written rules, we can quickly detect and classify known malware.

[4]https://www.hex-rays.com/products/ida
[5]https://retdec.com
[6]https://ghidra-sre.org

Currently, the most popular pattern matching engine is YARA[7]. YARA is an open-source tool developed by VirusTotal. When using YARA, malware samples are described through rules in a simple readable format. Each rule consists of:

1. Optional metadata section that might contain description of a sample, author, checksums, and other custom fields. These sections usually depend on company culture, standards in shared databases of rules, etc.

2. Sections of strings defined either as a text, as a hex encoded bytes, or as a regular expression.

3. Condition – boolean expression to match the rule. This expression can use any of boolean operators. Expressions can contain simple rules whether strings are present or not, they also can count strings occurencies or take into account string offsets.

Conditions part and YARA's pattern matching can be extended by modules. Base modules include PE (Portable Executable), ELF, Math, Hash, Magic, and Cuckoo modules. PE module can match characteristics of Windows object files. ELF can match fields available in ELF header. Math and Hash are auxiliary modules that allow calculations in YARA rules. Magic module can help identify file types. Cuckoo module extends YARA so it can match behavioral information from Cuckoo sandbox's output.

YARA is used by many malware research teams and antivirus companies. Some of them share their rules in open-source databases.

2.3.4 Static Analysis Problems

Malware authors sometimes modify their binaries to prevent analysis [27] or at least to make analysis more complicated. Many malware samples is obfuscated. During obfuscation, author is trying to hide real code meaning. Common obfuscation method is binary packing.

When binary is packed, its data is compressed and prepended with wrapper program that unpacks the binary during runtime. This prohibits static analysis since common tools will only detect code of a wrapper program. UPX packer is often used due to being open-source. Malware authors also modify original packer source code to prohibit unpacking by UPX unpacker.

Another way to prevent static analysis is anti-disassembly. Anti-disassembly techniques cause fault disassembly results by some disassemblers. Example mechanics used by malware are inserting and manipulation with jump instructions. Jump instructions representing conditional statements can have the same target and thus confusing disassembler program. Similar effect is achievable by inserting jump instructions with a constant address as a target.

2.4 Dynamic Analysis

Dynamic analysis [27] is a process of inspecting malware during its runtime. It is a great way to analyze malware and determine its function. Several problems must be solved to be able dynamically analyze malware's behavior. We must ensure that our environment for running malware is secured so our system will not get destroyed. The network belonging to

[7]https://virustotal.github.io/yara

environment should be isolated from our local network so that other systems in the network are not infected.

Common approaches of dynamic analysis are program tracing, debugging, analyzing network traffic, tracking file system operations etc.

2.4.1 Program Tracing

Linux, as well as most operating systems, differs between kernel mode execution and user mode execution. The interface to communicate between user mode and kernel mode is implemented by system calls [5, 12]. System calls are requests to the kernel made via software interrupts. These interrupts are then processed by a system call handler and then by corresponding system call service routine. Programmers need system calls in order to carry out highly privileged operation like hardware interaction. In user-level programming, we normally do not call system calls directly, but we use wrapper functions implemented in the standard C library.

Called system calls can describe malware's behavior. System calls occur during file system operations, memory manipulation, process creation, and other important operations. Especially interesting system calls for malware analysis are mentioned in Table 2.1.

system calls	**purpose**
`execve, execveat`	program execution
`clone, fork, vfork`	process creation
`open, openat, creat`	opening files
`unlink, unlinkat`	deleting files
`rename, renameat, renameat2`	file renaming
`write`	writing to a file
`read`	reading from a file
`connect`	initiation of connection on a socket
`send, sendto, sendmsg`	sending network data to a socket
`recv, recvfrom, recvmsg`	recieving network data from a socket
`ptrace`	tracing processes

Table 2.1: Selected system calls and their meaning.

Strace [12] is a well-known utility for tracing system calls in Linux. It can handle parsing of system call arguments, return values. It can output timestamps of individual system calls. It is also able to follow newly created processes. Finally, strace outputs statistics about whole process actions. The statistics show information about system call occurencies, total time spent in system calls, errors of system calls.

Ltrace[8] (library call tracer) is an another utility for tracing processes. Its main purpose is to track library functions calls. It can also trace system calls.

2.4.2 Debugging

Debuggers serve mainly as a help tool during software development. However, they can be used to analyze malware behavior [27]. Debuggers inspect program's execution state, memory, stack, registers, called functions, etc. It is also possible to alter program's execution during debugging.

[8]https://linux.die.net/man/1/ltrace

Using debuggers, we can either single step program's instructions, step over or step into functions. We can also set breakpoints to continue until we reach desired part of the program. In Linux, the most popular interactive debugger is **gdb** (GNU Debugger).

2.4.3 Ptrace and Anti-debugging

Debuggers and execution tracers are implemented using `ptrace` system call. Ptrace [5] is a fairly comprehensive system call with 26 available commands (requests). The most important ones are:

- `PTRACE_ATTACH` – attaches to another process and starts up tracing.
- `PTRACE_GETREGS` – reads registers' values during execution of a process.
- `PTRACE_SETREGS` – sets register value.
- `PTRACE_SINGLESTEP` – single steps one assembly instruction.
- `PTRACE_SYSCALL` – continues execution until a system call occurrence.
- `PTRACE_TRACEME` – starts tracing caller process.

Initial experiments with malware samples proved ptrace based tools to be inefficient. Analyzed sample Satori implemented quite common anti-debugging technique. This anti-debugging technique takes an advantage of ptrace system call with `PTRACE_TRACEME` as an argument. After this call, ptrace system call returns error values if program is already being traced. This serves as a quick detection of debuggers and tracers.

As a solution to this anti-debugging technique, `LD_PRELOAD` environment variable can be used. `LD_PRELOAD` holds a list of shared libraries to be loaded before loading standard libraries. If we reimplement ptrace system call wrapper in our own shared library, we can alter return code and thus not be detected by malware. Reimplementing is fairly straightforward. It is sufficient to just wrap original ptrace implementation from C library. Its address can be found by dlsym. However, this approach of reimplementing ptrace function is only viable if the binary was dynamically linked.

To evade ani-ptrace method in statically linked binary, it is needed to patch actual instructions. IDA Pro, Ghidra, and radare2 can easily patch instructions. The goal is to find appropriate system call instructions of target architecture and consider calling conventions of target architecture. Then we can replace system call instruction with any instruction that sets return value register to 0.

2.4.4 Kernel-level Tracing

Kernel-level tracing [21, 17] is more suitable method than patching every single binary for anti-debugging techniques. Topic of kernel-level tracing and monitoring is very well described on a blog of kernel performance engineer Brendan Gregg[9]. In Linux, we may gather information from three main event sources: tracepoints, dynamic probes and PMCs (performance monitoring counters).

Tracepoints and dynamic probes are most convenient for tracing malware. Tracepoints are statically defined events in kernel. Probes allow to attach our code to both kernel-level and user-level functions. Data from these event sources are accessible via multiple frontends. In this thesis four were considered: perf, ftrace, eBPF and SystemTap.

[9]http://www.brendangregg.com/blog

Perf

Perf tool is a part of the kernel tree at `/tools/perf`. Perf can report scheduler events, hardware metrics, and many other events both on kernel and user level.

Ftrace

Ftrace is also part of the kernel. It is available through virtual filesystems debugfs and tracefs. These are mounted to `/sys/kernel/debug` and `/sys/kernel/debug/tracing`. Ftrace configuration and monitoring is done through filesystem operations (reading and writing to files in `/sys/kernel/debug`). Debugfs and tracefs filesystems are unfortunately quite complex.

eBPF

eBPF (Extended Berkeley Packet Filter) is included in recent kernel versions (most features available in 4.*.*). It is accessed via `bpf` system call. Its architecture consists of small virtual machine with a JIT (just in time) compilation. It is a great option for performance monitoring due to its little overhead.

SystemTap

SystemTap is a robust, powerful and widely programmable monitoring and tracing tool. Unlike previous mentions, SystemTap is not part of the kernel. SystemTap is configured in stap programs. These programs are written in a C-like language. They contain definition of probepoints (individual probes – e.g. for probing kernel functions) and actions (outputs, data storage, accessing probe variables).

2.5 Network Analysis

Network traffic analysis embraces considerable amount of protocol standards, approaches, techniques of network monitoring, and classifications. Analysis approaches differ in the level of network inspection or desired speed. Resources used for this section are [12, 7, 13, 25].

To analyze network traffic, it is possible to gather statistics through SNMP (Simple Network Monitoring Protocol). We use SNMP to obtain information about managed objects. Each category of objects has different values that are useful for monitoring. Information about these values are stored in a database called MIB (Management Information Base).

Another way of network inspection is through Netflow. Netflow is a monitoring protocol operating over flows. Flow can be defined as a set of packets belonging to one conversation or a connection between two clients. Flows are commonly identified based on source IP address, destination IP address, source port, destination port, and transport protocol. Netflow has multiple implementations as Netflowv5, Netflowv9, IPFIX, sFlow, and OpenFlow.

For purpose of this thesis, the most important type of network analysis is network packet analysis and deep packet inspection.

2.5.1 Capturing Network Data

Network analysis begins with data acquisition. Method of capturing data on network host is called packet sniffing. Sniffing starts with selection of a network interface. This interface must be in promiscuous mode if we want to capture all the incoming data.

Popular programs providing network capture are tcpdump and wireshark. **tcpdump** is a popular console program. Tcpdump saves captured packets in a cap or pcap format. Pcap, besides other metadata and actual packets, contains timestamps, interfaces identifications, and packet lengths. Tcpdump also comes with libpcap library that can be used for our own implementation of network sniffing tools. **Wireshark** can also capture data. Moreover, it acts as a packet dissector. Wireshark analyzes both individual packets and communication streams.

2.5.2 Packet Inspection

Next step, after acquiring network data, is packet dissecting and inspection. Majority of modern networking applications operate over TCP/IP model. TCP/IP model is a networking model divided into 4 layers: Link layer (layer 1), Internet layer (layer 2), Transport layer (layer 3), and Application layer (layer 4). Application data is encapsulated in each layer. During encapsulation, data is prefixed with layer protocol header. Sometimes (for example in case of Ethernet), there is also a footer appended after data.

Experiments with IoT malware shown that most relevant application-level protocols for packet analysis are DNS, HTTP, Telnet, and IRC.

DNS

DNS (Domain Name System) [22] is a mechanism for naming resources and translating their names. The core of the DNS architecture consists of name servers and resolvers. The process of quering for DNS answers is called DNS resolution.

Every message starts with a DNS header. Besides other fields like query ID or message flags, DNS header declares number of entries in each section. These sections are: question section, answer section, authority section, and additional section. Question section contains a question for a name server. Answer question contains records answered by the name server. Authority section holds name server records of authoritative name server. Finally, additional section lists additional information that might speed up the resolution process.

Questions are represented in a format containing domain name, query type, and query class. Question can include one or more resource records types. Answer, Authority. and Additional sections all have the same format. Their entries are called resource records.

Each resource record has values of name, type, class, TTL (time to live), resource data length, and resource data. Resource record's name is a list of domain name parts divided with a null byte. This name is further compressed by reusing string parts that were already mentioned previously in the DNS message. Resource record's data format depends on its type. Common resource record types and their data are mentioned in Table 2.2:

HTTP

HTTP (Hypertext Transfer Protocol) [11] is an application protocol implemented over TCP protocol. HTTP is primarily used for exchanging hypertext and other documents in web services. It is a protocol based on requests and responses. During HTTP connection, client sends a request to a web server and the server responses.

Requests begins with a stating of a request method. HTTP supports various request methods. Typical request methods are `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, and `OPTIONS`. After request method, first line of a request specifies a Uniform Resource Identifier (URI). URI is a string that represents a location of a resource. The first line of a request message ends with

RR type	**data**
`A`	IPv4 address
`AAAA`	IPv6 address
`CNAME`	canonical name (alias)
`NS`	name server
`MX`	mail server
`DS`	delegation signer
`DNSKEY`	public key for DNSSEC
`RRSIG`	resource signature
`NSEC`	next secure record

Table 2.2: Common resource records' types.

HTTP version. Nowadays, HTTP/1.1 is used. Next lines hold header fields. Host request-header field is required in HTTP/1.1. It represents address and port of the web server with desired resource. Commonly used header fields are also Accept headers (Accept, Accept-Charset, Accept-Encoding, Accept-Language) for specifying preferred response format. For example, `Accept: application/json` is often used in communication with REST APIs.

Response format starts with HTTP version, status code, and reason phrase on the first line. Status code and reason are used for response classification. Responses with a status code 1xx are informational messages, status code 2xx is sent after a successful operation. Status code starting 3xx signals redirection. Status code starting 4xx reports client error and status code starting 5xx reports server error. Again, following lines contain header fields.

Data of HTTP messages can be encoded using multiple methods. Default "encoding" is identity. Identity encoding does not transform the message content. Available compression methods are: gzip, compress, and deflate. We have to take into account these encodings in the process of HTTP protocol analysis.

Current IoT malware are relatively simple programs and they do not implement HTTPS (Hypertext Transfer Protocol Secure). One occurrence of port 443 was in case of Torii botnet [19]. Even Torii however did not implement TLS nor SSL and it only used port 443 for its own simple XOR-based encryption.

Telnet

Telnet [26] is one of the earliest protocols. Its purpose is to establish connection for controlling remote terminal. Telnet embraces a concept of Network Virtual Terminal (NVT). NVT is implemented by both client and server applications. Thanks to NVT, it is simpler to minimize differences among terminals. Client or server use option negotiation to arrange supported terminal type, character set, modes of operation etc. After the options negotiation, Telnet connection is established.

Then, each host that participates in Telnet connection can send terminal data. To differentiate between Telnet commands and the actual data, we use Interpret as Command escape sequence (IAC). IAC is one byte long and its value is 255. Telnet does not implement any type of encryption, thus it should not be used for transmission of sensitive data.

Manufacturers often use Telnet service as a connection point to IoT devices. This connection point, often with weak or default credentials, might be exploited by malware.

IRC

IRC (Internet Relay Chat) [23] is a client-server protocol for chat communication. IRC is text-based. The protocol specifies messages and command formats. Important messages are listed in Table 2.3.

command	meaning	example
`JOIN`	join specific channel	`JOIN #fitvut`
`PRIVMSG`	send private message	`PRIVMSG daniel :Hi Daniel!`
`NOTICE`	send notice message	`NOTICE daniel :Bot message.`
`PING`	test presence of a user	`PING daniel`
`PONG`	response to `PING`	`PONG fitvut`

Table 2.3: IRC commands.

2.6 Sandboxing

Sandbox [14, 15] creates a safe restricted environment to run programs. Program executed in the sandbox does not have direct access to the system's resources and its network. Thus, it should not be able to damage the system. Application sandboxing is commonly used to protect users for example in web browsers. Microsoft Practical Sandbox implements protected modes of Internet Explorer, Microsoft Office, Google Chrome, and Acrobat Reader X. This thesis focuses on fully virtual sandboxed environments. These environments use virtual machines to run and observe malware's behavior. Unknown software execution could normally harm the system but the execution is often the only option for malware analysis. Chapter 3 analyzes popular open source sandboxes that are used for malware analysis.

Chapter 3

Existing Solutions

This chapter analyzes existing solutions that implement Linux sandboxes. These solutions are evaluated based on several metrics. First of these metrics is their availability. Considered system should be publicly available so that we can use it or extend its implementation. Next, possibilities of automatic malware analysis are evaluated. Ease of use was also considered. Systems that are easy to setup and manage are preferred. Lastly, number of supported architectures and reliability of analysis results was evaluated. Results mentioned in this chapter were acquired from comprehensive study of open source Linux sandbox systems [24].

3.1 REMnux

REMnux is a toolkit for malware analysts. It is implemented as a standalone Linux distribution based on Ubuntu. The REMnux virtual machine creates sandboxed environment. The distribution has pre-installed many malware analysis tools that are able to analyze Linux malware, perform memory forensic or statically examine malware files.

For static analysis, REMnux has built-in radare2 framework. Possibilities of radare2 were already described in Section 2.3.2. REMnux utilizes r2 and rabin binaries from radare2 framework. This way, REMnux is able to examine ELF headers, entry points, or detect dynamic loaders. For dynamic analysis, REMnux primarily uses strace utility. Strace is implemented via `ptrace` system call. As described in Section 2.4.3, this dynamic analysis method can be easily detected. REMnux also provides memory analysis using Volatility[1]. REMnux can also check malware samples in the VirusTotal database.

Analysis in REMnux is not automated. For our purposes, virtual machine automation would have to be implemented. Moreover, REMnux is based on Ubuntu, so it does support only x86 architectures.

3.2 Detux

Detux is a sandbox system providing automated Linux malware analysis. It supports five different architectures (i386, x86-64, ARM, MIPS and MIPSel). Sandbox environment is emulated by QEMU. Its environment is based on Debian images.

Detux outputs JSON reports. Nonetheless, Detux implements only basic features. Static analysis part in Detux extracts strings from binary and it parses ELF information

[1]https://www.volatilityfoundation.org

via readelf command (both of these commands were described in Section 2.3.1). Considering dynamic analysis, Detux sandbox tracks only malware network behavior. It captures network traffic during malware execution. This traffic is then analyzed. Detux tracks DNS requests, IP addresses of endpoints, and accessed ports.

3.3 Limon

Limon is a small Python script that implements a wrapper around several analysis tools. It provides static, dynamic, and memory analysis. Sandbox environment runs as a virtual machine in VMware. Dynamic analysis is again implemented with strace. Strace output is formatted to output individual system calls. Automation of Limon is possible. Limon script orchestrates the virtual machine and it can be scripted to accept multiple files for analysis. VMware virtualization allows only x86 architectures.

One advantage of Limon is its choice of analysis tools. Limon uses for example VirusTotal public API for submitting samples. It also prepares folder with YARA rules to match and classify malware samples or to detect packed malware.

3.4 Cuckoo Sandbox

Cuckoo is a large Python library actively developed since year 2011. It is a leading solution in automated malware analysis field. It has many possibilities for virtual machine preparation. Supported virtualization software is for example VMware, Virtualbox, or QEMU. Cuckoo is popular mainly because of its Windows analysis capabilities. However, Cuckoo's Linux analysis is usable too.

Cuckoo runs static analysis modules, it inspects behavior of analyzed malware and examines dropped files (downloaded or created files). Its network analysis analyzes endpoints and several network protocols. However, on application layer, only HTTP and IRC is analyzed. It also has IDS (Intrusion Detection System) plugins. These plugins show alerts from IDS systems Suricata[2] and Snort[3].

Cuckoo automates its analysis with simple Python script located in a prepared guest system. Dynamic analysis inside the guest system runs as a SystemTap kernel module. This can outline dependencies and requirements for a target system preparation. Complexity of environment preparation is the main downside of Cuckoo sandbox. In comparison with previously mentioned solutions, Cuckoo does not provide Linux target images so users have to prepare images by themselves. This can be complicated and time-consuming task that is usually not doable by hobbyists (as seen in multiple submitted issues considering this problem on Cuckoo's Github page). Another potential disadvantage of Cuckoo is its robustness. Cuckoo includes all of the analysis types and platforms with its primary focus on Windows operating system. All these features might not be desired if we want to analyze only Linux binaries.

One advantage of Cuckoo is its proposed monitoring on kernel level (although users have to prepare and compile kernel modules by themselves). This type of event tracing should not be detectable by analyzed malware. Another big advantage is that Cuckoo implements its own YARA module. Using Cuckoo's YARA module, we can match malware behavior

[2]https://suricata-ids.org
[3]https://www.snort.org

```
cuckoo.network.http_request
cuckoo.network.http_get
cuckoo.network.http_post
cuckoo.network.dns_lookup
cuckoo.filesystem.file_access
cuckoo.sync.mutex
cuckoo.registry.key_access
```

Figure 3.1: YARA functions implemented in Cuckoo module.

by YARA rules. However, official implementation of this module includes only few rules (see Figure 3.1).

Even though these rules are designed for Windows malware behavior detection (Linux does not have registry key abstraction), network rules can be used also for matching Linux malware behavior.

3.5 Padawan Sandbox

Authors of the biggest up-to-date study of Linux malware [8] implemented Padawan sandbox. This sandbox was designed as a part of their Linux malware research and it significantly improved the current state of multiplatform Linux malware analysis. Padawan sandbox supports x86, ARM, MIPS, and PowerPC architectures. System can be virtualizes with KVM or QEMU. It can be configured for different type of execution. Users can choose glibc or uClibc implementations of standard C library.

Padawan sandbox traces kernel and user probes through SystemTap. Authors mention that they had to patch SystemTap's code to support ARM and MIPS architectures. Patches details are not specified in the paper. Unfortunately, Padawan exists only as a service. Authors did not release any source code.

3.6 Summary and Proposing General Requirements

All discussed solutions can provide usable analysis results. Every solution except REMnux is able to perform automated analysis. Limon implements only minimal architecture. It also does not support any standardized output format. Detux and Cuckoo Sandbox can output their results in JSON format. Detux and Limon implement strace based analysis. On the other hand, Cuckoo and Padawan implement SystemTap kernel modules for more reliable dynamic analysis results. Cuckoo seems as a viable solution but it does not solve one of the most important parts – the preparation of a sandboxed environment. On the other hand, it is very robust. For our purposes of IoT malware analysis, standalone Linux analysis system as Padawan sandbox would be ideal. Unfortunately, Padawan sandbox was not publicly released. This thesis proposes implementation of new IoT malware analysis sandbox. The final system should fulfill the following requirements.

3.6.1 Functional Requirements

Functional requirements of the final system can be divided into groups based on their priority. Firstly, there are *must have* requirements. These include the most crucial functionality of the proposed system. Secondly, there are *should have* requirements. They specify functionality that would significantly improve overall quality of the project. Thirdly, there are *could have* requirements. They outline small improvements that can be implemented in the final system.

Must Have

Must have requirements include all the functionality of architecture emulation. Final system must implement proper sandbox environment. It must support at least three architectures to cover large portion of malware samples. It must be able to statically analyze malware samples and run them in sandboxed environment. Finally, network analysis is also expected.

Should Have

Should have requirements target dynamic analysis implementation. Analysis of malware behavior would significantly improve overall analysis results.

Could Have

Could have requirements address user accessibility improvements. One of these requirements is some type of graphical user interface. Another improvement could be implementation of custom YARA module. This would make possible to detect and classify malware based on system's analysis results.

Chapter 4

Sandboxed Linux Environment

This chapter describes the preparation of a sandboxed Linux environment. The Linux environment is the core of proposed sandbox. It is the place where malware runs and where most of analysis happens. This chapter considers existing Linux environments or distributions. Afterwards, it shows the preparation process of minimalist Linux distribution suitable for embedded systems. In the end, it covers possible communication protocols to automate sandbox analysis.

4.1 QEMU

QEMU [3] is a processor emulator. It has two operational modes: user mode and full system emulation mode. QEMU supports many CPU architectures including x86, ARM, AArch64, MIPS, MIPSel, PowerPC, SPARC, or SH4. Even though other virtualization engines are popular (Virtualbox, VMware), QEMU is the only that supports actual emulation of other architectures, thus it is the only viable option.

4.2 Considering Existing Systems

Main requirement and drawback when considering existing Linux systems for preparing our environment is the need of multiple platforms. Most Linux distributions target desktop users on x86 processor architectures.

Debian

Debian is one great example of multiplatform development. It has official ports[1] for ARM architectures (ARMel, ARMhf, AArch64), MIPS (both little-endian and big-endian), PowerPC-64, and System Z. Moreover, debian's web page lists other 21 non-official ports.

In comparison with embedded Linux firmwares on real IoT devices, Debian is much more complex. Experimenting with Debian in multiple architectures emulation shows the drawback of this complexity. Booting of emulated Debian takes up to 60 seconds. This is more than the expected average analysis time.

[1]https://www.debian.org/ports

Aboriginal Linux

Aboriginal Linux is a set of Linux images bootable under QEMU. These images are designed to contain minimal build environment. Aboriginal Linux runs on every platform that is supported by QEMU. Its build environment can be an alternative to cross-compilation. Instead of setting up cross-compiler, we can compile natively in Aboriginal's emulated environment. Reports by Symantec [30] security company state that Aboriginal Linux is being used by malware authors as a way to distribute their malware for multiple platforms. Leveraging Aboriginal Linux by malware authors as a build environment could enhance motivation for using it also for running this malware. Unfortunately, Aboriginal's development has ended in 2017 and it is not currently maintained.

4.3 Linux Environment Preparation

Drawbacks of existing multiplatform Linux systems motivated the creation of custom Linux images. These images should be minimal, they should boot quickly and have low size. They should also resemble embedded firmware so that malware samples can run smoothly in their expected environment.

4.3.1 Cross-compilation

Cross-compilation is a process of compiling on a host platform to produce a binary that can be executed on a different target platform. Cross-compilation is used when target platform does not have prepared native compiler. Another reason is that a host platform typically have more processing power and thus the compilation process is much faster. This process is often used in the process of designing embedded systems.

Two main projects are used to speed up and automate embedded Linux development and cross-compilation – Yocto and Buildroot. These project have similar output (embedded Linux system) but different approach.

Yocto

Yocto[2] is a robust project that implements its utility BitBake for building and setting up Linux images. Yocto builds final system in layers and these layers are reusable if we want to change configuration or build upon a base image. It handles package management and creates a solid, stable base for a default system. Yocto is used by many industry-level companies like Xilinx or NXP.

Buildroot

Buildroot[3] aims at simplicity. It leverages use of Makefiles and menuconfig tools. Whole image build is defined by one configuration file. By default it has more than 2000 available packages. It is also really simple to extend build with our own packages via configuration files. It was chosen for usage in this thesis for its minimalism and low image sizes.

Buildroot supports three standard C libraries for its images – uClibc-ng, glibc and musl. Process of image preparation using Buildroot starts with selecting a predefined configuration file as a base. Buildroot has a predefined configuration file (defconfig) for each architecture

[2]https://www.yoctoproject.org
[3]https://buildroot.org

supported by QEMU. Next, we configure Linux kernel using kernel's menuconfig. Finally, we run buildroot's menuconfig and configure target system, packages, make options, etc. Buildroot then can output kernel image, bootloader image and root filesystem.

4.3.2 Kernel Configuration

As stated previously, Linux kernel build is configured via symbols. Kernel config is a text file containing keys and values. Config keys may be enabled, disabled (not set), enabled as a module or set to a literal value. Since config has thousands of lines, it is helpful to set symbols through menuconfig. Menuconfig simplifies view on individual kernel settings, it provides documentation for symbols and categorizes them for quick look up.

Firstly, it is needed to configure some drivers suitable for emulation. Almost all of needed symbols are already set by buildroot defconfig for target QEMU architecture, but for image setup it was needed to enable Gigabit Ethernet driver `CONFIG_E1000` and its dependent symbols.

Afterwards, it is needed to prepare kernel for debugging (see Section 2.4.4 and Section 6.1.2). Needed configuration can be seen in Table 4.1.

symbols	**meaning**
`MODULES, MODULE_UNLOAD`	enable loading of kernel modules
`KPROBES, KRETPROBES`	kernel probes
`UPROBES`	user probes
`PERF_EVENTS`	performace monitoring counters support
`EVENT_TRACING, TRACERS`	tracing support
`BPF`	Berkeley Packet Filter support
`DEBUG_FS`	debug file system
`DEBUG_INFO`	debug information
`KALLSYMS`	load symbols for debugging

Table 4.1: Kernel configuration symbols used for kernel-level monitoring.

4.3.3 Root Filesystem

Embedded firmware does not have complicated file systems. Many of embedded Linux environments in IoT devices are based on busybox. Busybox packs standard UNIX utilities into one executable. Besides busybox, root filesystem contains tools needed for analysis (see Section 6.1).

4.4 Communication and Automation

Proposed environment will take part in automated malware analysis system. It is needed to consider appropriate communication protocol for automating the sandbox environment during analysis process. Afterwards, it is required to specify the way of transferring individual files between host and guest.

4.4.1 Automation and Sending Commands

One option to communicate with emulated system is networking. Multiple implementations of network protocols are possible. We can simply use Telnet or SSH to control machines and send commands. It is also possible to implement simple text protocol over TCP. However, all networking solutions have the same problem. They can be interrupted during execution of malware. Some malware samples close open ports and isolate the machine after infection.

Another option is to use serial console. QEMU is able to virtualize serial port (e.g. `ttyS0`) and redirect it to standard input and output (stdio). Stdio is then easily accessible by many automation tools.

4.4.2 File Transfer

In virtualized environments, concept of shared folders is adopted. One implementation of shared folders is VirtFS [16]. VirtFS is paravirtualized filesystem. It is implemented in QEMU to provide shared folders between hosts and guests. Host exports a folder (part of its filesystem) through QEMU server. Guest can then mount the filesystem using 9P2000.L protocol. VirtFS can be used in QEMU thanks to implementation of virtio-9p-pci device.

There are two essential file transfers in proposed analysis system. Firstly, it is the insertion of analyzed sample into guest's filesystem. Secondly, it is copying out the output of analysis. Since we do not need to transfer files during analysis, it is beneficial to omit usage of VirtFS. Analyzed binary file can be inserted into guest's filesystem before analysis (and before starting the emulation). The output can be extracted from filesystem right after guest machine has shutdowned.

We can use e2tools[4] to manage filesystem directly without mounting. E2tools is a set of utilities that allow copying, moving, removing, listing of files, creating links or directories and outputting the end of files. `e2cp` tool was used to transfer files and to set appropriate permissions.

[4]https://aur.archlinux.org/packages/e2tools

Chapter 5

Designing Framework for Automated Linux Malware Analysis

This chapter proposes a universal design of a sandbox system for malware analysis. We are designing robust, yet easy-to-use analysis system suitable for usage by both individuals and malware research teams. The proposed design follows functional requirements set in Section 3.6.

It is also very important that the final system can be extended because the field of malware analysis is dynamic and new malware strains are discovered every day. Thus, modularity and maintainability are vital variables for our design.

Proposed sandbox system consists of multiple parts. Core of the system is an analysis pipeline. Analysis pipeline implements actual malware analysis. It uses prepared Linux environment described in Chapter 4. Core of the system may be used either individually (from command line interface), or as integrated solution implementing tasks queuing, REST API or web user interface.

5.1 Analysis Pipeline

Analysis Pipeline (see Figure 5.1) is a set of sub-analysis modules. These modules work individually and all of them are managed by Top Level Analysis module. They are organized in plugin system. Plugin system ensures that the system is extensible with newly desired functionality. Plugins are designed according to Strategy Design Pattern. This design defines set of algorithms and encapsulates implementation details so that there are no problems when number of different implementations (derived classes, plugins) changes.

5.1.1 Common Interface

Architecture of the pipeline is extensible and it consists of individual sub-analysis modules. All sub-analysis modules inherit its interface from Abstract SubAnalyzer. Every analyzer derived from Abstract SubAnalyzer has two inherited data members:

- `file` data member holds Analyzed File object. This object has all the information about analyzed binary files. It is designed to store information about file's path, directory. Next it contains specification of file's architecture, endianness, and information

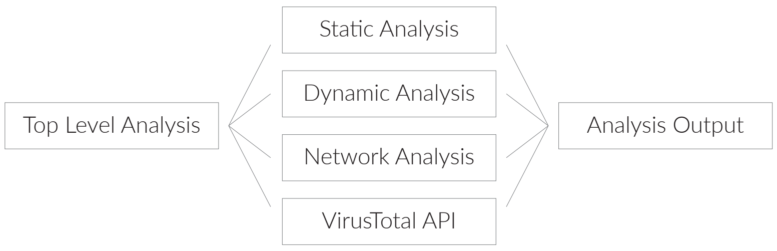

Figure 5.1: Proposed analysis pipeline.

whether binary is 32-bit or 64-bit. These files are identified with an `md5` hash with possibility to access also `sha1` and `sha256` hashes.

- `output` data member is a dictionary containing everything that module wants to add into full analysis output. It is expected that this dictionary is complex and nested with various data types, structures, and arrays. Top Level Analysis appends every output dictionary to final output. Every segment of analysis output must be serializable.

Analyzers also implement one common method – `run_analysis`. This method is called by Top Level Analysis module. Implementation of `run_analysis` method is specific for each sub-analysis. Common behavior of this method is that it fills `output` data member.

5.1.2 Top Level Analysis

Top Level Analysis module creates the beginning of the pipeline. As mentioned before, Top Level Analysis manages all sub-analysis modules and it acts as a director of whole analysis. In the beginning of the analysis, it initializes Analyzed File object, file's paths and execution time of analysis. Information stored in Analyzed File object is accessible in every sub-analysis module. After setting up Analyzed File object, it prepares metadata about analysis as the start time of specific analysis.

Top Level Analysis module loads sub-analysis modules (plugins) according to the configuration file. Plugins are specified in a list and they are identified by their full path (e.g. `lisa.analysis.network_analysis.NetworkAnalyzer`). When plugins are loaded Top Level Analysis module calls `run_analysis` module of every plugin. For each plugin, it gathers its output and combines them.

5.1.3 Static Analysis

This design proposes static analysis module that searches for relevant static patterns in ELF binary files. The main purpose of this module is providing information about file using any of the tools described in Section 2.3.

Basic ELF Information

Static Analysis module goes through ELF header and it gathers information about the architecture of analyzed binary, endianness, machine etc. Architecture is specified by architecture identifier – e.g. x86_64, i386, ARM, ARMel, MIPS, MIPSel, AArch64, PowerPC. Next information taken from ELF header is entry point. Entry point tells us the address where code of the binary begins.

Linkage and Symbols

The module recognizes whether binary was linked statically or dynamically. According to the study [8], more than 80 % of samples were statically linked. Another malware related information is presence of symbols. A lot of information in ELF format might be stripped and stripping is quite a common practice for malware.

This module analyzes imported and exported symbols. These symbols are mostly functions. Imported functions can determine some functionality that is taken from imported libraries.

Other Information

Finally, Static Analyzer module leverages implemented static analysis tools and outputs special information offered by chosen static analysis framework. This output can contain disassembled or decompiled parts of the binary (e.g. first instructions after entry point), it can contain compiler and interpret information, language of implementation, etc. This is highly dependent on final implementation that is described in Section 6.1.1.

5.1.4 Dynamic Analysis

Main tasks of proposed Dynamic Analysis modules are setting up QEMU virtual machine depending on the architecture, starting the emulation, automating emulated machine, running the analysis and extracting captured behavior.

Emulation Design

In the beginning of its analysis, Dynamic Analysis module initializes QEMU Guest Manager. QEMU Guest needs to know about binary architecture in order to provide proper emulation.

QEMU Manager selects Linux images according to sandbox configuration file. Every architecture in the configuration file has a definition of run command, prompt, and rootfs. Run command serves for starting the virtual machine and it takes one argument – path to the root filesystem. Command prompt serves as a delimiter during interaction with virtual machine. Root filesystem contained in rootfs configuration serves as a filesystem base. This base is snapshotted during analysis so that original filesystem is not modified nor destroyed.

Analyzed binary is then copied into the filesystem and QEMU Manager starts the virtual machine chosen in previous step. Machine boots, QEMU Manager automatically logins as a root user into virtual machine and waits for other commands.

QEMU Manager provides interface to start a virtual machine, send commands into the virtual machine, run specified analysis inside the virtual machine or to power off the machine.

Running Analysis

Dynamic Analysis module firstly starts capturing network traffic. Network captures are then provided for latter analysis (either for Network Analysis module or manual inspection). Then, the module starts executing the binary. Meanwhile, it captures called system calls. It also creates a process tree of created processes. This process tree serves also as a filter for system calls called by different processes.

Information captured during dynamic analysis (system calls, processes, and opened files) are stored in an intermediate representation. When system call is captured, `SYSCALL` text is inserted into intermediate representation followed by five lines.

- First line contains name of the process that executed system call. In the beginning, this will be always `analyzed_binary`. This is because every analyzed file inserted into target file system is renamed. Further, this name can change. One way of changing process name is by calling `prctl` function with `PR_SET_NAME` argument.
- Second line contains actual system call name (e.g. `open`, `clone`, `sendto`).
- Third line contains integer that specifies PID (process ID) of the process that called traced system call.
- Fourth line contains all the arguments that were passed to the system call. These arguments depend on the actual system calls.
- Fifth line specifies returned value.

When module notices opened file, `OPENFILE` text is inserted into intermediate representation with only one following information on the next line. This information is obviously file path. When the analyzed binary creates new process, `PROCESS` text is inserted into intermediate representation with PIDs of parent and child processes separated by a colon.

Data Extraction and Output Preparation

After the execution, virtual machine is powered off. QEMU Manager module extracts network capture file and the file containing intermediate representation of dynamic analysis results from machine's filesystem. Intermediate representation is then parsed by Dynamic Analysis module. While parsing, it propagates desired information into module's output.

5.1.5 Network Analysis

Network Analysis module is designed to accept PCAP file as its input. This file is then processed packet by packet. Actual network analysis might be divided into four parts: statistic extraction, endpoint analysis, layer 7 analysis, and anomalies detection.

Statistic

Network Analysis module creates overall statistic about network capture file. Observed values are accessed ports, TCP SYN packets, and TCP FIN packets. Accessed ports are divided according to their transport protocol (TCP, UDP) and sorted by frequency. The statistic containing TCP SYN and TCP FIN packets might help to detect IP or port scanning behavior.

Endpoints

Networking module keeps information about every contacted endpoint. Each endpoint is identified by its IP address. Beside IP address, the module keeps track of ports associated with the endpoint, amount of data send to the endpoint and received from the endpoint.

This module also searches for endpoint's geolocation information. If this geolocation information is available, the module outputs info about endpoint's country, city, organization, and its ASN (Autonomous System Number).

Network Analysis module also searches IP addresses in provided blacklists. System accepts blacklists in ipset or netset format. Files in ipset format contain individual IP addresses on separate lines. Files in netset format can contain both individual IP addresses and whole networks with defined net mask (e.g. `180.153.160.0/23`). All ipset and netset files are then merged into one universal blacklist. This blacklist is used during endpoint identification.

Layer 7 Analysis

Application level analysis is proposed for protocols DNS, HTTP, IRC, and Telnet (see 2.5.2). Each preprocessed packet is passed to the layer 7 analysis part of the module. The module checks presence of application protocol headers.

DNS analysis outputs formatted DNS questions. DNS questions format contains full domain name resolved by analyzed binary (e.g. `www.fit.vutbr.cz`) and type of resource record (e.g. `A`). HTTP analysis looks for HTTP requests. It stores HTTP method (e.g. `GET` or `POST`), requested URI, HTTP version and key-value pairs of headers. The module outputs list of IRC messages. These messages are constructed out of IRC command and all of its parameters. Finally, Telnet related data is analyzed. Telnet commands (data starting with IAC byte `0xff`) are ignored because these commands mainly serve for setting up terminal and they do not contain valuable information for malware analysis.

Anomalies

Every part of Network Analysis module can trigger anomaly detection. Anomalies have unified format. These format includes name of the anomaly, its description, and data where the anomaly was detected. It is expected that users will add anomalies when using the system. Initially, five anomaly types are proposed:

- Port scanning anomaly analyzes earlier calculated port statistics. Port scanning anomaly is reported when number of ports exceeds given threshold. Reported data contains number of ports for TCP and for UDP.
- TCP SYN and TCP FIN scanning anomalies are reported when network capture contains too many TCP SYN or TCP FIN packets. These packets often signal Internet scanning. Reported data contains total number of TCP SYN (FIN) packets, number of packets scanning local network and number of packets scanning the Internet.
- Anomaly is also reported when analyzed binary accesses blacklisted IP address.
- DNS anomaly is reported when malformed DNS packet is found. These malformed packets imply in their headers that they contain either big number of questions, answers, authorities, or additionals.

5.1.6 VirusTotal API

VirusTotal has public API[1] that can be used for uploading and scanning files. Uploaded files are scanned by various antivirus scanners and results are then shared with the community. Proposed design suggests retrieving past results from VirusTotal API.

VirusTotal sandbox module simply requests scan report. Requested report can be identified by either `md5`, `sha1`, or `sha256` hash. This module returns individual scan results as its output.

5.2 Analysis Output and Further Processing

As mentioned above, full analysis output is constructed by Top Level Analysis module by merging its own file's metadata with individual outputs taken from sub-analysis modules. This final output is serialized in JSON[2]. JSON output is stored for further examination. First option is manual examination by malware analysts. For these purposes, graphical user interface was designed. This interface should be intuitive for the target group and it should resemble other graphical user interfaces that are present in other tools used by malware analysts. Second option is further automated analysis. Malware samples can be automatically detected or classified based on fields in JSON output. The design proposes implementation of custom YARA module. As described in Section 2.3.3, YARA is a pattern matching engine implementing the concept of rules. Proposed module defines new set of rules that are compatible with sandbox output. This module extends available conditions for malware detection with behavior and network based rules. Following information may be included in condition part of the rule:

Number of Processes

Suggested rule gives access to number of processes created by an analyzed binary. Number of processes can be a hint when searching for daemonized services.

```
lisa.behavior.number_of_processes() == 5
```

Opened Files

Opened files condition function has one parameter. This parameter is a regular expression specifying path to the opened file.

```
lisa.behavior.file_open(/^/dev/.*/)
```

System Calls

Matching function for system calls has two parameters. First one is a string containing the name of a system call. Second one is a regular expression specifying matched system call's parameters.

```
lisa.behavior.syscall("connect", /{AF_INET, 8.8.8.8, 53}/)
```

[1]https://www.virustotal.com/en/documentation/public-api
[2]https://www.json.org

Endpoints

Endpoints can be matched based on two fields. Firstly, it is specific IP address. Secondly, it is the country of an endpoint.

```
lisa.network.endpoint_ip("8.8.8.8")
lisa.network.endpoint_country("China")
```

DNS Questions

Matching function for DNS questions has two arguments. First one specifies the domain name with a regular expression. Second one contains specific type of the resource record (see Table 2.2) or `ANY`.

```
lisa.network.dns_question(/gov.cn/, "ANY")
```

HTTP Requests

HTTP requests can be matched using function `http_request`. It has two parameters – request method and a regular expression for URI.

```
lisa.network.http_request("POST", /.*\.cgi/)
```

Telnet and IRC

Telnet and IRC rules work similarly. They are both designed with one regular expression parameter.

```
lisa.network.irc(/^JOIN #randomserver/)
lisa.network.telnet(/root/)
```

Anomalies

Proposed anomaly rules give access to quick behavior anomaly detection in YARA.

```
lisa.network.syn_scan()
lisa.network.blacklisted_ip_access()
```

5.3 Microservices Architecture

As stated above, analysis system can be also used as an integrated solution. Proposed analysis workflow of this integrated environment my be seen in Figure 5.2. Individual parts of the system are designed as microservices. Microservice architecture [28] proposes that system should be divided into small independent parts. Each of these parts can be then developed and tested individually. The key is to assign single responsibility to each microservice. The design proposes five main parts: frontend interface, web API, message broker, analysis worker, and database.

Frontend service forms the presentation layer. It allows user to display analyzed data and to submit new binary files for analysis. Frontend communicates with available REST API. API microservice can be accessed directly or via mentioned frontend interface. Available API endpoints are listed in Table 5.1. Web application logic sends analysis tasks to message broker. Message broker service gathers analysis tasks and puts them into queues.

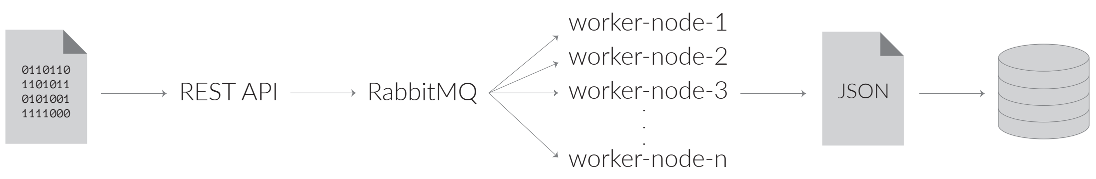

Figure 5.2: Analysis process workflow.

Queues can be allocated depending on the system needs. If the system is deployed as private service it does not make sense to set up more than one queue. However, some type of systems can benefit out of multiple queue types and priorities. Analysis worker microservice is the part when whole malware analysis happens. This part implements analysis pipeline (described in Section 5.1). Analysis worker communicates with message broker and it reserves tasks from subscribed queues. Number of workers can be configured so that multiple analysis processes can run concurrently. This makes analysis service scalable. Information about analysis and tasks is stored in the database. Database contains information about successful and failed tasks and their task IDs. JSON results are stored on the disk. Their location is derived from task IDs.

method	**endpoint**	**action**
`POST`	`/tasks/create/file`	Submits full analysis task.
`POST`	`/tasks/create/pcap`	Submits PCAP analysis task.
`GET`	`/tasks`	Lists all tasks.
`GET`	`/tasks/finished`	Lists successfully finished tasks.
`GET`	`/tasks/failed`	Lists failed tasks.
`GET`	`/tasks/pending`	Lists pending tasks.
`GET`	`/tasks/view/<task_id>`	Returns tasks status.
`GET`	`/report/<task_id>`	Returns analysis report.
`GET`	`/pcap/<task_id>`	Returns pcap captured during analysis.
`GET`	`/machinelog/<task_id>`	Returns QEMU machine log.
`GET`	`/output/<task_id>`	Returns analyzed program's stdout output.

Table 5.1: API endpoints.

Chapter 6

Implementation

This chapter describes the process of sandbox implementation. Implementation was based on the design proposed in Chapter 5. It was vital to focus on choosing right technologies for individual tasks. First of all, this chapter describes implementation of the analysis pipeline and its particular modules. After it summarizes implementation of other parts of the sandbox system. Description of sandboxed Linux environment preparation is omitted in this chapter as it was already described in Chapter 4.

6.1 Analysis Implementation

As a primary implementation language was chosen Python. Python is a popular general purpose, interpreted language. Python has solid standard library. Thanks to its popularity and big community, it has also many user libraries. For resource-heavy operations, C++ was chosen as an implementation language. C++, in comparison with Python, is compiled language and it can be highly optimized. It is also possible to create bindings from C or C++ language to Python. We can then compile fast and efficient modules and use these modules in Python code.

6.1.1 Static Analysis

Section 2.3 mentions several tools for static analysis. For final implementation, radare2 was used. Radare2 was primarily chosen because of its bindings to Python (with `r2pipe` module). Radare2 implements all requested features. Moreover, it offers scriptable editing of binaries and their instructions. Standard utility `strings`, that is available in UNIX systems, is used for strings acquisition.

6.1.2 Dynamic Analysis

Core of the dynamic analysis implementation is SystemTap. Out of all kernel tracing options, SystemTap is the most adaptable one. SystemTap's C-like language can be quickly updated to add new functionality. Problematic part of SystemTap is its compilation. Process of compilation mentioned in SystemTap wiki is:

1. Finding probes, creating stap script and translating it to C language,
2. cross-compiling C code to kernel module,

3. loading and running kernel module on target machine.

However, this process caused problems when compiling SystemTap kernel modules for some platforms (as MIPS or ARM). Final solution of cross-compilation toolchain is implemented as a container based on Fedora. Fedora proved to be the most stable system considering that big part of SystemTap's source code is managed by Red Hat developers. However, Fedora's implementation of elfutils[1] library does not support MIPS platform. This was solved by applying Debian elfutils patches. These patches are not part of elfutils upstream. After resolving cross-compilation problems, SystemTap script was implemented.

SystemTap script traces system call probes. It implements mechanism of process tree creation so that only system calls called by the analyzed process or by its descendants are traced. PIDs of processes are stored in an associative array together with their parents. The array is filled based on return values of `clone` and `fork` system calls. Other information is traced and reported as designed in Section 5.1.4.

6.1.3 Network Analysis

Two Python libraries for packet parsing were used in the early implementation – scapy and dpkt. Both of these libraries work similarly, they are able to parse PCAP files or live data, and they fill Python objects with data from parsed headers. Neither of these solutions proved to be sufficient. Both scapy and dpkt were slow while opening bigger PCAP files. Moreover, they were not able to even load some of the files because of their RAM usage.

As a solution, C++ packet parsing library was implemented. This library aimed to be fast, minimalist and able to parse large PCAP files. This library is compiled into `.so` shared library for Linux. The library has also Python bindings. It has similar interface as scapy and dpkt. The library builds on top of libpcap. Libpcap handles PCAP loading, opening and sniffing of live traffic. Loaded data is then parsed by a pipeline of separate parsers. Each protocol parser is implemented in its own class that is initialized with data pointer and data length. Network Analysis module of the final sandbox is implemented according to the design proposed in Section 5.1.5.

6.2 Architecture Implementation and Conteinerization

Chapter 5 proposed different parts of microservices architecture. Individual microservices were implemented as Docker containers. Docker[2] is currently the most popular containerization platform. Docker utilizes concept of Dockerfiles. Dockerfiles are build recipes for containerized systems. Configuration and connection of microservices are defined in `docker-compose.yml` file.

Full architecture may be seen in Figure 6.1. REST API was implemented with Flask Python library. Frontend uses Javascript framework ReactJS. For setting up task queues was used RabbitMQ messaging broker. Implementations details of API or GUI are omitted because they exceed the main topic of this thesis.

[1]Elfutils is ELF and DWARF format library and it is heavily used by SystemTap.
[2]https://www.docker.com

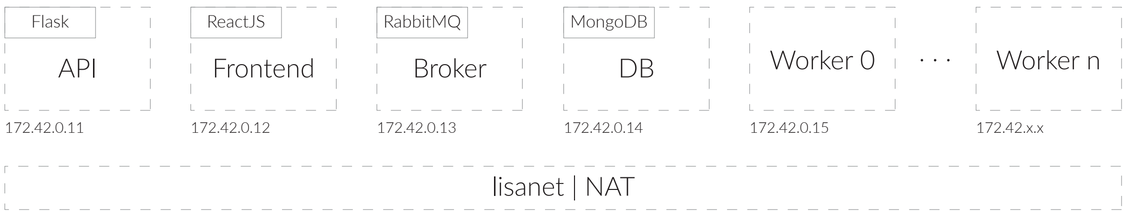

Figure 6.1: Containerized microservice architecture.

Chapter 7

Verification and Validation

Verification and validation [31] are crucial parts of software development life cycle. Verification methods evaluates software during development if it meets stated conditions. This helps to discover early faults caused by wrong design. At the end of development cycle, software must be also validated against requirements. This chapter summarizes process of verification and validation and its implemented methods.

7.1 Testing

The system was tested during its whole development to ensure proper functioning. Testing process was divided into two parts: unit testing and testing on malware analysis samples.

7.1.1 Unit testing

Unit tests are designed to test parts (units) of the software. During any module development, test suite was being prepared meanwhile. Most of the tests were implemented using `pytest` framework. This framework allows to simply define `test_*` files in the project's hierarchy. Pytest framework collects and runs all test functions. It also allows to specify pytest fixtures. Fixtures are used for data initialization e.g. on module level.

Static Analysis Unit Tests

Unit tests for static analysis loads Static Analyzer and runs its analysis on a test file. Then, each static data extraction function of Static Analysis module is tested. Tests simply define expected values for imports, exports, libraries, symbols, etc.

Dynamic Analysis Unit Tests

Unit tests for dynamic analysis test whether every architecture environment can run sample binary and produce analysis output. Other unit tests verify functionality of the methods for intermediate representation extraction.

Network Analysis Unit Tests

Two levels of unit tests were prepared for network analysis. Firstly, there are unit tests for verifying parser functionality. PCAPs were crafted for each tested network protocol.

Secondly, there are unit tests for functions of Network Analysis module that should report anomalies, analyze endpoints, analyze L7 protocols and report statistics.

YARA Module Unit Tests

Unit tests verifying YARA module are implemented via YARA rules. Each YARA function has a predefined test rules that either should or should not match. Test suite then checks if all *should match* rules were successfully matched and that output does not contain any *should not* match rules.

7.1.2 Testing on Malware Samples

In order to experiment and evaluate analysis results, 150 IoT malware samples were chosen to analyze. See Table 7.1 for information about samples' architecture distribution. More then 85 % of samples in the dataset were statically linked.

targeted machine	**amount**
ARM	57
MIPS R3000	52
Intel 80386	23
x86-64	13
AArch64	5

Table 7.1: Architecture of samples in dataset.

Test results shown that emulating ARM is the most problematic part. Some tested ARM binaries were not able to execute properly. System call log showed only error results of not implemented system calls. This is because ARM architecture has got many variants. This issue may be solved by preparing more ARM images for different ARM versions.

Two anti-analysis techniques were found in the dataset. Firstly, it was already discussed ptrace anti-debugging technique. This technique did not influence analysis because of the kernel-level tracing implementation. Secondly, it was checking system information in `/etc/os-release`. `os-release` file of the prepared environment contains buildroot's version. As buildroot is commonly used for embedded Linux development, this also did not influence analysis results.

Most programs, created two to three processes. The biggest number of created processes was 34. This program also executed `minerd` – CPU mining service for Bitcoin and Litecoin:

```
./minerd -q -B -a scrypt -o http://p2pool.org:5643 -u \
MDFepZz9SpSbFSugUsXVE3CmrdTaKg1SWi -p pass
```

7.2 Summary

Unit tests were prepared to test software during its development. Part of these automated tests also serve as regression tests to ensure stability after implementation of new features.

Section 3.6 defined project's requirements. These requirements are fulfilled. At least three supported architectures were set as *must have* requirement. The system supports five different architectures. Although, ARM emulation support did not prove to be ideal because of the differences among ARM versions. Both unit tests and testing on real malware verified SystemTap's kernel module function. Custom YARA module – stated in *could have* requirements – was also implemented and tested.

Chapter 8

Possible Future Work

Since the thesis covers vast amount of topics, there are many possible improvements of the system. This chapter outlines interesting ideas for project extensions. The final sandbox system takes into account user plugins. Thus, any improvement mentioned in this chapter can be implemented without much knowledge of the system's implementation.

Linux Images

Section 7.1.2 describes testing on real malware samples. Some of these samples were not able to execute properly. Requirements for the proper execution can be really specific. Implemented system supports five different architectures. All of these architectures utilize glibc implementation of the standard C library. Proposed improvement could include preparation of other Linux images. These images would make possible to select different implementations like uClibc or musl. Moreover, configuration of other system libraries and their versions could be supported.

The system could be also extended to support emulation of IoT firmware. Researchers from Carnegie Mellon University and Boston University presented firmware analysis tool FIRMADYNE [6]. Its main purpose is to detect vulnerabilities in firmware images. However, FIRMADYNE can be reused for firmware extraction and system emulation.

Cuckoo Compatibility

Chapter 3 evaluates different existing solutions. The chapter outlines that the main problem of Cuckoo Sandbox is that it does not provide any Linux guest images. These must be prepared by users. Linux images that were prepared as part of this thesis can be re-used also in the Cuckoo Sandbox. This is because Cuckoo also builds on top of SystemTap kernel modules. Reusing provided images would be beneficial for teams that already integrated Cuckoo Sandbox for automated Windows malware analysis.

Machine-Learning Malware Detection

Paper [4] describes a malware detection method based on machine learning. Authors used gSpan sub-graph algorithm. The detection relies on system calls analysis. System call dependency graph is prepared and analyzed. The extension would include preparation of system call dependency graph (SCDG) and implementing similar machine learning module that would be trained on various samples of IoT malware families.

Chapter 9

Conclusion

This thesis analyzed the current state of IoT malware. It described its common strains and their typical behavior. It studied possibilities of static, dynamic, and network analysis. Considering static analysis, it mentioned various static analysis tools including recently open sourced RetDec by Avast Software and Ghidra by NSA. In dynamic analysis, it studied different approaches of system monitoring and program tracing. Considering network analysis, it analyzed application protocols that were most frequently used by IoT malware. The thesis proposed a general concept of Linux malware analysis sandbox. This concept was designed with focus on extensibility and modularity.

Main contributions of this project are, firstly, network analysis, detection of anomalies and implementation of C++ library with Python binding that in some aspects overcome libraries Scapy and dpkt. Secondly, it is preparation of SystemTap monitoring environment and its cross-compilation toolchain. Full sandbox system was implemented so that it is capable of providing static, dynamic, and network analysis. The system can be controlled via REST API or web interface. It is fully scalable to support any number of concurrent analysis workers.

The final sandbox was tested on a dataset provided by Avast Software. The testing showed some shortcomings of the final solution. These were present mostly because of diversity of architectures. Overall, system supports five different architectures.

Project was open sourced on GitHub[1] in April 2019. Parts of this thesis were also presented on students' conference Excel@FIT [29].

[1] https://github.com/danieluhricek/LiSa

Bibliography

[1] Angrishi, K.: Turning Internet of Things (IoT) into Internet of Vulnerabilities (IoV): Iot Botnets. *arXiv preprint arXiv:1702.03681*. 2017.

[2] Antonakakis, M.; April, T.; Bailey, M.; et al.: Understanding the Mirai Botnet. In *26th USENIX Security Symposium (USENIX Security 17)*. 2017. pp. 1093–1110.

[3] Bellard, F.: QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, vol. 41. 2005. page 46.

[4] Ben Said, N.; Biondi, F.; Bontchev, V.; et al.: Detection of Mirai by Syntactic and Behavioral Analysis. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. Oct 2018. ISSN 2332-6549. pp. 224–235. doi:10.1109/ISSRE.2018.00032.

[5] Bovet, D. P.; Cesati, M.: *Understanding the Linux Kernel: from I/O ports to process management*. O'Reilly Media. 2005.

[6] Chen, D. D.; Woo, M.; Brumley, D.; et al.: Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *NDSS*. 2016. pp. 1–16.

[7] Corey, V.; Peterman, C.; Shearin, S.; et al.: Network forensics analysis. *IEEE Internet Computing*. vol. 6, no. 6. 2002: pp. 60–66.

[8] Cozzi, E.; Graziano, M.; Fratantonio, Y.; et al.: Understanding Linux Malware. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018. pp. 161–175.

[9] De Donno, M.; Dragoni, N.; Giaretta, A.; et al.: Analysis of DDoS-capable IoT malwares. In *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE. 2017. pp. 807–816.

[10] Eilam, E.: *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons. 2011.

[11] Fielding, R.; Reschke, J.: *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. IETF. June 2014.

[12] Forshaw, J.: *Attacking Network Protocols: A Hacker's Guide to Capture, Analysis, and Exploitation*. No Starch Press. 2018.

[13] Garfinkel, S.: Network Forensics: Tapping the internet. *IEEE Internet Computing*. vol. 6. 2002: pp. 60–66.

[14] Greamo, C.; Ghosh, A.: Sandboxing and Virtualization: Modern Tools for Combating Malware. *IEEE Security & Privacy*. vol. 9, no. 2. 2011: pp. 79–82.

[15] Jamalpur, S.; Navya, Y. S.; Raja, P.; et al.: Dynamic Malware Analysis Using Cuckoo Sandbox. In *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*. IEEE. 2018. pp. 1056–1060.

[16] Jujjuri, V.; Van Hensbergen, E.; Liguori, A.; et al.: VirtFS – A virtualization aware file system pass-through. In *Ottawa Linux Symposium (OLS)*. 2010. pp. 109–120.

[17] Keniston, J.; Mavinakayanahalli, A.; Panchamukhi, P.; et al.: Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps. In *Proceedings of the 2007 Linux symposium*. 2007. pp. 215–224.

[18] Kolias, C.; Kambourakis, G.; Stavrou, A.; et al.: DDoS in the IoT: Mirai and Other Botnets. *Computer*. vol. 50, no. 7. 2017: pp. 80–84.

[19] Křoustek, J.; Iliushin, V.; Shirokova, A.; et al.: *Torii botnet - Not another Mirai variant*. 2018. [cit. 2019-05-10].
Retrieved from: https://blog.avast.com/new-torii-botnet-threat-research

[20] Levine, J. R.: *Linkers & loaders*. Morgan Kaufmann. 2010.

[21] Mavinakayanahalli, A.; Panchamukhi, P.; Keniston, J.; et al.: Probing the Guts of Kprobes. In *Linux Symposium*, vol. 6. 2006. page 5.

[22] Mockapetris, P.: *Domain names - Implementation and specification*. RFC 1305. IETF. November 1987.

[23] Oikarinen, J.; Reed, D.: *Internet Relay Chat Protocol*. RFC 1459. IETF. May 1993.

[24] Olowoyeye, O.: *Evaluating Open Source Malware Sandboxes with Linux Malware*. PhD. Thesis. Auckland University of Technology. 2018.

[25] Pilli, E. S.; Joshi, R.; Niyogi, R.: A Generic Framework For Network Forensics. *International Journal of Computer Applications*. vol. 1, no. 11. 2010: pp. 1–6.

[26] Postel, J.; Reynolds, J.: *Telnet Protocol Specification*. RFC 854. IETF. May 1983.

[27] Sikorski, M.; Honig, A.: *Practical Malware Analysis: The Hands-on Guide to Dissecting Malicious Software*. No Starch Press. 2012.

[28] Thönes, J.: Microservices. *IEEE Software*. vol. 32, no. 1. Jan 2015: pp. 116–116. ISSN 0740-7459. doi:10.1109/MS.2015.11.

[29] Uhříček, D.: *LiSa – Multiplatform Linux Sandbox for Analyzing IoT Malware*. Excel@FIT 2019 [cit. 2019-05-08] .
Retrieved from: http://excel.fit.vutbr.cz/submissions/2019/058/58.pdf

[30] Venkatesan, D.: *New Wave of Mirai Leverages Open-Source Project for Cross Platform Infection Technique*. 2019. [cit. 2019-04-29].
Retrieved from: https://www.symantec.com/blogs/threat-intelligence/mirai-cross-platform-infection

[31] Wallace, D. R.; Fujii, R. U.: Software verification and validation: an overview. *IEEE Software*. vol. 6, no. 3. May 1989: pp. 10–17. ISSN 0740-7459. doi:10.1109/52.28119.

Appendix A

CD Content

readme.txt
File describing content's structure.

xuhric00-thesis.pdf
PDF thesis.

xuhric00-thesis-print.pdf
Print version of PDF thesis.

thesis-src/
LaTeX source code.

cross-stap-build/
Cross-compilation environment for building SystemTap kernel modules.

disspcap/
C++ packet parser.

lisa/
Linux Sandbox – main part of the project.