THESE DE DOCTORAT DE
SORBONNE UNIVERSITE
préparée à EURECOM

École doctorale EDITE de Paris n° ED130
Spécialité: «Informatique, Télécommunications et Électronique»

Sujet de la thèse:

Binary Analysis for Linux and IoT Malware

Thèse présentée et soutenue à Biot, le 14/12/2020, par

Emanuele Cozzi

Président	Prof. Aurélien Francillon	EURECOM
Rapporteurs	Prof. Christian Rossow	CISPA Helmholtz Center for Information Security
	Prof. Lorenzo Cavallaro	King's College London
Examinateurs	Prof. Martina Lindorfer	TU Wien
	Dr. Mariano Graziano	Cisco Systems Inc.
	Prof. Aurélien Francillon	EURECOM
Directeur de thèse	Prof. Davide Balzarotti	EURECOM

Preface

Time flies.
I can still perfectly remember the day I decided to join the S3 group like it was yesterday. Now I'm here, completing my Ph.D. and ready to climb another flight of stairs.
If I look back I smile. I'm happy to have met great people, I feel lucky to have learned from great researchers, I grew up.

First and foremost I deeply want to thank Davide, my supervisor, for leading me along this journey and showing me how a good researcher is supposed to be.

I'm extremely grateful to the entire S3 group. They have always been like a family, from the first person to the last, from the beautiful days spent together to the moments when I was rather looking for support. I'm not going to write down a list of names, there would be too many, and I'm sure I could miss someone.

You, my friend from S3 or office 370, reading this preface, thank you for your support, the thousand smiles you gave me, the discussions and exchange of ideas, all the knowledge you shared with me.

I want to thank my parents, Giusi and Maurizio, and my sisters, Serena and Laura, for their infinite support. Close to me even if far away, always encouraging me to reach the top, always proud of my achievements.

Last but not least, a special thought to Fabiana, my companion on the journey of life. You grabbed me by the hair more than ten years ago, you taught me a lot, we grew up together, you joined me in this adventure. I will never have enough words to thank you.

Résumé

Au cours des deux dernières décennies, la communauté de la sécurité a lutté contre les programmes malveillants pour les systèmes d'exploitation basés sur Windows. Cependant, le nombre croissant de dispositifs embarqués interconnectés et la révolution de l'IoT modifient rapidement le paysage des logiciels malveillants. Les acteurs malveillants ne sont pas restés les bras croisés, mais ont rapidement réagi pour créer des "logiciels malveillants Linux".

Par cette thèse, nous naviguons dans le monde des logiciels malveillants basés sur Linux et mettons en évidence les problèmes que nous devons surmonter pour leur analyse correcte. Après une exploration systématique des défis liés à l'analyse des logiciels malveillants sous Linux, nous présentons la conception et la mise en œuvre du premier pipeline d'analyse des logiciels malveillants, spécialement conçu pour étudier ce phénomène émergent. Nous utilisons notre plateforme pour analyser plus de 100 000 échantillons et recueillir des statistiques et des informations détaillées qui peuvent aider à orienter les travaux futurs.

Nous appliquons ensuite des techniques de similarité de code binaire pour reconstruire systématiquement la lignée des familles de logiciels malveillants de l'IoT, et suivre leurs relations, leur évolution et leurs variantes. Nous appliquons notre approche à un ensemble de données recueillies sur une période de 3,5 ans, et nous montrons comment la libre disponibilité du code source a entraîné une grand nombre de variantes, ce qui a souvent un impact sur la classification des systèmes antivirus.

Enfin et surtout, nous abordons un problème majeur que nous avons rencontré dans l'analyse des exécutables liés statiquement. En particulier, nous présentons une nouvelle approche pour identifier la frontière entre le code utilisateur et les bibliothèques tierces, de sorte que la charge des bibliothèques puisse être supprimée en toute sécurité des tâches d'analyse binaire.

Abstract

For the past two decades, the security community has been fighting malicious programs for Windows-based operating systems. However, the increasing number of interconnected embedded devices and the IoT revolution are rapidly changing the malware landscape. Malicious actors did not stand by and watch, but quickly reacted to create "Linux malware", showing an increasing interest in Linux-based operating systems and platforms running architectures different from the typical Intel CPU. As a result, researchers must react accordingly, in order to adapt the techniques and toolchains that they initially designed to analyze Windows malware. While Linux malware can reuse well-known patterns and behaviors, the binary analysis of Linux and IoT binaries requires to tackle specific new challenges.

Through this thesis, we navigate the world of Linux-based malicious software and highlight the problems we need to overcome for their correct analysis. After a systematic exploration of the challenges involved in the analysis of Linux malware, we present the design and implementation of the first malware analysis pipeline, specifically tailored to study this emerging phenomenon. We use our platform to analyze over 100K samples and collect detailed statistics and insights that can help to direct future works.

We then apply binary code similarity techniques to systematically reconstruct the lineage of IoT malware families, and track their relationships, evolution, and variants. We apply our approach on a dataset collected over a period of 3.5 years, and we show how the free availability of source code resulted in a very large number of variants, often impacting the classification of antivirus systems.

Last but not least, we address a major problem we encountered in the analysis of statically linked executables. In particular, we present a new approach to identify the boundary between user code and third-party libraries, such that the burden of libraries can be safely removed from binary analysis tasks.

Contents

List of Figures

List of Tables

List of Acronyms

ABI Application Binary Interface

API Application Programming Interface

APT Advanced Persistent Threat

AV Antivirus

C&C Command & Control

CFG Control Flow Graph

CGI Common Gateway Interface

CPU Central Processing Unit

CRT C Runtime

CVE Common Vulnerabilities and Exposures

DB Database

DDoS Distributed Denial of Service

DNS Domain Name System

ELF Executable and Linkable Format

GOT Global Offset Table

HNSW Hierarchical Navigable Small World graphs

IOC Indicator of Compromise

IOT Internet of Things

MST Minimum Spanning Tree

OS Operating System

PCI Peripheral Component Interface

PE Portable Executable

PID Process ID

RAT Remote Access Trojan

UPX Ultimate Packer for eXecutables

VT VirusTotal

Chapter 1

Introduction

The security community has been fighting malware for over two decades. However, despite the significant effort dedicated to this problem by both academia and industry, the automated analysis and detection of malicious software remains an open problem. Historically, the vast majority of malware was designed to target almost exclusively personal computers running the Microsoft Windows operating system, mainly because of its very large market share (currently estimated at 83% [Sta] for desktop computers). Correspondingly, the security community had been initially focusing its effort on Windows-based malware—resulting in several hundreds of papers and a vast knowledge base on how to detect, analyze, and defend from different classes of malicious programs.

However, the recent exponential growth in the popularity of embedded devices is causing the malware landscape to rapidly change. Embedded devices were once manufactured with the logic entirely integrated into the hardware until they moved to more friendly environments built on top of microcontrollers and microprocessors. Today, they can either run a tailor-made software, namely a *firmware*, or a high-level software layer such as a fully equipped operating system. Embedded devices have been in use in industrial environments, mission-critical appliances, and the automotive industry for many years, but it is only recently that they started to permeate every aspect of our society, mainly (but not only) driven by the so-called "Internet of Things" (IoT) revolution. While the analysts' early projections for the number of connected devices by 2020 are controversial [Nor16], they agree that the number of IoT devices is continuously growing and it should have already surpassed the 20 billion mark.

Companies producing these devices are in a constant race to increase their market share, thus focusing mainly on a short time-to-market combined with a set of innovative features to attract new users. Too often, this results in postponing (if not simply ignoring) any security and privacy concerns. With these premises, it does not come as a surprise that the vast majority of these newly interconnected devices are routinely found vulnerable to critical security issues, ranging from Internet-facing insecure logins (e.g., easy-to-guess hardcoded passwords, exposed telnet services, or accessible debug interfaces), to unsafe default configurations and unpatched software containing well-known security vulnerabilities.

Embedded devices are profoundly different from traditional personal computers. For example, while personal computers run predominantly on x86 architectures, embedded devices are built upon a variety of other CPU architectures—and often on hardware with limited resources. To support these new systems, developers often adopt Unix-like operating sys-

tems, with different flavors of Linux quickly gaining popularity in this sector.

Not surprisingly, the astonishing number of poorly secured devices that are now connected to the Internet has recently attracted the attention of malware authors. The first Linux file infectors already appeared in the late '90s, either as public threats [VLA96] or, more often, as harmless examples realized by researches [Silc, Ale]. The details of these handicrafts were published on personal websites and technical zines. With the exception of few anecdotal proof-of-concept examples, the antivirus industry had largely ignored malicious Linux programs, and it is only by the end of 2014 that VirusTotal recognized this as a growing concern for the security community [ZDn]. Academia was even slower to react to this change, and to date it has not given much attention to this emerging threat. In the meantime, available resources are often limited to blog posts (such as the excellent *Malware Must Die* [mala]) that present the, often manually performed, analysis of specific samples. One of the few systematic works in this area is a recent study by Antonakakis et al. [Ant17] that focuses on the network behavior of a specific malware family (the Mirai botnet). However, no comprehensive study has been conducted to characterize, analyze, and understand the characteristics of Linux-based and IoT malware.

1.1 Problem statement

To date, the analysis of Linux malware samples was largely performed through many hours of manual reverse engineering work. Both companies and private researchers have studied specific malware strains and shared their knowledge through blog posts or through a set of Indicators of Compromise (IOC) [Tal18, Bit18, Ble19, Pal19]. These IOCs are then used to build static signatures, which in turn can be used to detect the presence of the malicious samples and to flag their corresponding network activity. However, while the insights gained from these reports are invaluable, they only provide a very scattered view of a much bigger picture. Little information is available on how current static and dynamic analysis techniques apply in the context of Linux and IoT malware and what the right methodology is to handle these samples. Therefore, we still lack a good understanding of how sophisticated Linux malware is as well as of how effectively we can tackle this raising threat.

In fact, researchers who wants to work on Linux malware must cope with an incredible diversity of target machines. Thus, we must ask ourselves what is the right way to work with an executable compiled for an ARM drone or a

MIPS router. We must consider that the run-time environment the malware expects may change from device to device. Are we sure that a single sandbox can satisfy all these very diverse needs? How we explain in this thesis, the analysis of Linux malware requires to fine-tune many small details, which are often overlooked. For instance, how does a sandbox successfully run a dynamically linked sample if a library is missing? And what if the sample is statically linked but our system calls table does not match the one of the binary? Malware may also adopt tricks to hinder static analysis, just as it happens constantly on Windows.

The goal of this thesis is to understand the behavior, techniques, and properties that characterize Linux and IoT malware. These includes, for instance, the packing technologies employed by malicious software, their ability to evade the analysis environment, or the persistence mechanisms adopted to survive a system reboot. More importantly, we want to investigate whether it is possible to measure all these aspects in a fully automated way, as our final goal is to scale our study to to cover the entire ecosystem.

Another characteristic of IoT malware is that the source code of some of its most famous families has been publicly available for years, leaked or published on underground forums. These releases have paved the way for myriads of variants and a tangled relationship among malware samples. The second objective of this thesis is to study the impact of code reuse by tracking variants of the same family and the appearance of new families that originated from multiple codebases. In fact, the dynamics behind the emergence of new malware strains are still unclear—resulting in the fact that AV labels are often very coarse-grained, and therefore unable to capture the continuous evolution that characterizes Linux malware. While the traditional approach for this purpose is based on static and dynamic feature-based clustering, the unique characteristics of Linux malware allow for a tailored solution based on code-based similarity.

Finally, unlike in the Windows world, static linking is much more common on Linux, as typical developers often opt for a statically linked version of their applications for portability reasons. However, this bloats the executables and it introduces new challenges that the malware analysis community has never faced before. It is possible to automatically isolate and distinguish the portion of code written by the malware author from that of the embedded libraries? This would help to reduce the complexity of static analysis and it is a required step to perform a fine-grained code comparison of multiple malware samples.

All these high level questions also introduce technical problems related to the analysis of Linux executables, problems that this thesis wants to

emphasize and wish to address.

1.2 Contributions

This thesis is a journey characterized by low-level binary challenges (both for static and dynamic analysis), with intermediate stopovers to capture the global picture by performing large-scale studies on thousands of malware samples. Overall, this manuscript sheds light on a topic that, to the best of our knowledge, has not yet received the focus it deserves.

We tackle the challenges of binary analysis for Linux and IoT malware with a bottom-up approach. First of all, we understand how to properly process malicious ELF programs, and discuss the challenges involved when dealing with this particular type of malicious files. We design and implement *padawan* [coz18], a multi-architecture pipeline specifically designed to support the analysis of Linux malware. We use padawan to conduct the first large-scale empirical study on thousands of Linux malware samples and uncover and discuss several low-level Linux-specific techniques employed by real-world malware. Our results show that Linux malware is already a multi-faced problem. This study allowed us to identify many interesting behaviors—including the ability of certain samples to properly run in multiple Unix-based operating systems, or the presence of specific malicious activities performed only when a sample is executed with certain user privileges.

Despite the efforts to explore Linux malware at scale, their growth continues to be relentless, with a plethora of new variants appearing on VirusTotal day by day. This continuous activity, made worse by the public availability of malicious source code, demands to consider Linux malware as a whole and not only as a set of individual samples. We fulfill this need by proposing a systematic way to compare IoT malware samples and display their evolution in a set of easy-to-understand lineage graphs. We rely on code-based similarity to identify various variants of each family and to capture the intra-family relationships. While our approach allows us to track the continuous evolution of malware, we discovered that also AV products are struggling to keep up with this ever-changing environment. Our code-based similarity system proved that the simple form of the current IoT malware facilitates static code analysis, which largely outperforms other feature-based solutions (e.g., based on dynamic sandbox reports.)

While performing the lineage study, we encountered the problem of deal-

ing with statically-linked (and often stripped) ELF binaries. These samples introduced limitations in our analysis, and often required special cases to circumvent and proceed with our research work. Therefore, our last contribution directly addresses the problem of isolating third-party libraries included at link time. Statically linked binaries are not only difficult to reverse engineer, but introduce noise in our code similarity pipeline. To overcome this issue, we present a method to identify the boundary between user code and library code in statically linked ELF files. Our system leverages the spatial layout of a program defined at link time to extract user code, and provide the analyst only a restricted subset of functions to look at. The advantage of this solution is invaluable, as it saves considerable analysis time and increase the precision of automated code comparison approaches.

1.3 Thesis outline

This thesis is organized into 6 chapters. We start from the basics, with Chapter 2, by giving the reader the necessary background to understand the *ELF* file format—standard for Linux executables and vitally important to bootstrap all the thesis contributions. We also examine the current progress and state-of-the-art of binary analysis for Linux and IoT malware and provide a background of previous works on Linux malware. This chapter also covers malware clustering (as applied so far on Windows malware) and recent works dealing with statically linked and stripped programs.

Chapter 3 presents the challenges to overcome when performing binary analysis on Linux malware. Moreover, in this chapter we present an analysis pipeline specifically tailored for Linux malware, which we designed and implemented to face the challenges of this new environment. By using this pipeline, we conduct the first large-scale measurement study and uncover tricks and techniques used by real-world malware authors.

Chapter 3 is based on the publication *Understanding Linux Malware, IEEE Symposium on Security & Privacy (S&P) 2018* [CGFB18].

In Chapter 4 we enlarge the focus of our investigation to capture the larger picture of IoT malware. After successfully analyzing and understanding individual samples, we move our focus to study families and variants. In particular, we present the workflow of our code-based clustering solution to systematically reconstruct the genealogy and tangled relationships of malware families. Finally, we describe their evolution, their fragmentation, and how AV companies perform in their recognition.

Chapter 4 is based on the publication *The Tangled Genealogy of IoT Malware, Annual Computer Security Applications Conference (ACSAC 2020)* [CVD+20].

With Chapter 5 we want to meditate on one of the binary analysis questions still left unsolved by our pipeline, and that we had to face to analyze Linux malware. This chapter tackled the challenges of functions and libraries' recognition in statically linked ELF files. We present our system to detect user-defined functions and "cut" benign and malicious binaries accordingly. This is helpful to remove the noise of library code from binary and malware analysis jobs.

Chapter 5 is based on an ongoing project under finalization at the time of writing.

We conclude the thesis with Chapter 6 where we treasure the experience acquired throughout this journey to suggest ideas for future works. Finally, we briefly outline how we tackled the challenges of binary analysis for Linux and IoT malware.

Chapter 2

Background

This chapter discusses the necessary background to contextualize the thesis. In particular, we give a brief introduction to the ELF file format since it is the standard for Linux executables and will follow us through the remainder of the thesis. Afterward, we will focus on prior works on Linux malware and the progress of binary analysis in this research field.

2.1 ELF file format

The Executable and Linkable Format (ELF) is the standard for many Unix-flavor operating systems. Linux, Android, and BSD use it for executables, libraries, object files, and core dumps. In order to analyze Linux malware, we first need to understand the ELF structure, its internal details, and all the information we can extract at our advantage. The importance of this step is twofold. On the one hand, it is necessary to design a set of tools and techniques to aid the analysis processes. The dissection of an ELF file depends on the CPU architecture and requires to operate on 32-bit or 64-bit structures and little- or big-endian data. On the other hand, we must be prepared to face all kinds of quirks that are introduced into malicious ELFs by malware authors to hinder the analysis. For example, pieces of code overlapping the ELF header, or fields with unusual values that can break most of the current analysis tools.

The ELF file format and its inner workings are complicated, and it is difficult to really master them at all degrees. The many aspects of this topic cannot fit within a single chapter in this thesis and we defer the interested reader to the ELF Standards [Fou] for a complete cover of the data structures. We will focus here only on the description of the aspects that are required to understand the next chapters.

A typical ELF file consists of five key elements: (i) an ELF header, (ii) a program header table, (iii) a section header table, (iv) a chunk of executable code, and (v) some data. The ELF header is always located at the beginning of the file and it contains a map that describes its internal organization. As described in Table 2.1, the header contains fixed features like the ELF type, the architecture, the entry point to start the execution, and the offsets of the program header and section header tables. Despite the ELF format being a standard, parsers and ELF-compatible operating systems do not adhere to a unique specification. Moreover, the tools developed over the years to handle ELF files were not designed to cope with the adversarial nature of malicious files. As flipping a single bit in the header could be sufficient to break common analysis routines, in this thesis we had to resort to a custom

Table 2.1: ELF Header structure

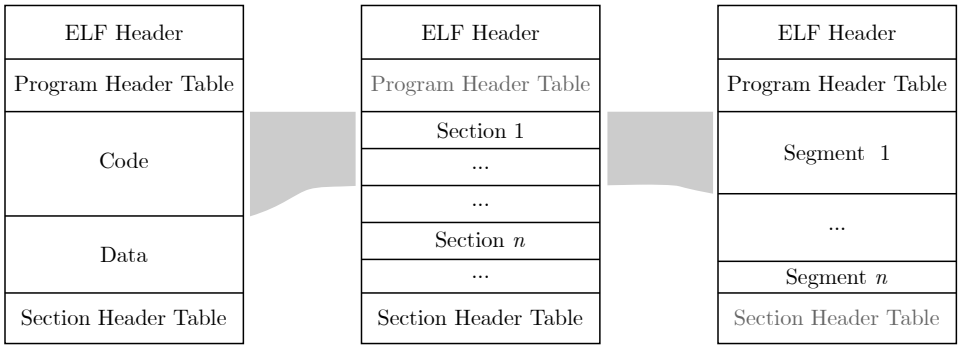
Field	Description
e_ident	Machine-independent data to decode and interpret the files's contents
e_type	Object file type
e_machine	Required architecture
e_version	Object file version
e_entry	Virtual address the system transfers control to
e_phoff	Offset of the program header table
e_shoff	Offset of the section header table
e_flags	Processor-specific flags
e_ehsize	ELF header's size
e_phentsize	Size of one entry in the program header table
e_phnum	Number of entries in the program header table
e_shentsize	Size of one entry in the section header table
e_shnum	Number of entries in the section header table
e_shstrndx	Section header table index associated with the section name string table

ELF parser specifically conceived for Linux malware.

Figure 2.1(a) illustrates the basic layout of an ELF binary. The section header table provides information describing the program's sections used for linking and debugging purposes (Figure 2.1(b).) The program header table describes the segments within the binary, as in Figure 2.1(c). The loader and the kernel setup the process image and the execution environment by using this information.

The difference between sections and segments often leads to confusion, with people wrongly mentioning one but referring to the other. Segments are mandatory to create a process, while sections can be omitted and the program will continue to work just fine. Therefore, Linux malware samples can choose to remove the section header table but they are not allowed to obfuscate the segments. We will see how samples exploit this dual view to produce errors in debugging tools like *GDB*, the de-facto standard debugger for Linux.

The ELF format accounts for more than ten types of segments, some of them mostly being placeholders for information required by the dynamic linker and the program interpreter. The Linux kernel itself normally deals with only three types of segments. The first one is the type PT_LOAD, which describes areas of the binary that need to be loaded and mapped into memory. This segment will generally contains the executable code, data (e.g. global variables), and dynamic linking information. The second type is PT_INTERP, which points to a string that declares the program



(a) Object File **(b) Linking View** **(c) Execution View**

Figure 2.1: Structure of an ELF file

interpreter. The last is `PT_GNU_STACK` that, if present, tells the kernel whether the process stack memory should be executable or not. It is worth to mention also the segment type `PT_DYNAMIC` as we will make an extensive use of it in our analysis. This segment is present if the object file requires dynamic linking and it contains information such as the list of shared libraries that must be liked at runtime, the relocation information, and the pointers to the dynamic string and symbol tables.

As we already mentioned above, sections are not a required component for the ELF execution. However, they offer better granularity for the overall inspection of a program. Sections can easily point reverse engineering tools to the code area (`.text`), the global offset table (`.got`), or the read-only variables (`.rodata`). Analysts can extract parts of the ELF content either through segments or sections. With the latter being more fragile in malicious contexts, it is always a safer choice to base binary analysis on segments whenever possible.

2.2 Linux malware

In the past two decades, the security community has focused almost exclusively on fighting malware targeting Microsoft Windows or, more recently, Android devices. As a result, hundreds of papers have described techniques to analyze PE binaries [Wic09, Fer, CJS+05, KRVV], detect ongoing threats [CJS+05, SWL05, DQG+04], and prevent possible infection attempts [MKN05, HGS01, LWKS05] on Windows operating systems. The community also developed many analysis tools for dissecting threats related to the Windows environment, ranging from dynamic analysis so-

lutions [malb, cws, anu, vts] to dissectors for file formats used as attack vectors [oleb, pee, olea].

With the exception of mobile malware, non-Windows malicious software did not receive the same level of attention. While the hacking community developed—almost two decades ago—interesting techniques to implement malicious ELF files [Silc, Silb, Sila, Zom, Ale], rootkits [dar, sd], and tools to dissect them [May, elfc, elfa], none of them has seen vast adoption. In fact, the security industry has only recently started looking at ELF files—mainly driven by newsworthy cases like the Mirai botnet [Nic] and Shellshock [Dav]. Many blog posts and papers were published for the analysis and dissection of specific families [Cat, MMDc, MMDb, MMDa, WLL+, CKVD10], but these investigations were mainly conducted by manual reverse engineering. Recent research by Shazhad *et al.* [SF12] and by Bai *et al.* [BYMM13] discuss the extraction of static features from ELF binaries to train a classifier for malware detection. Unfortunately, these works are not comprehensive, do not take into account different architectures, or are easily evaded by stripping a binary or by using packing.

Researchers have also started to explore dynamic analysis for Linux malware only very recently. The few solutions that are available at the moment support a limited number of platforms or provide very limited analysis capabilities. For example, Limon [Mon15] is an analysis sandbox based on `strace` (and thus easily detectable), and it only supports the analysis of x86 binaries. Sysdig [sysa] and PayloadSecurity [Pay] are affected by similar issues and they also only work for x86 binaries. Detux [det] supports instead four different architectures (i.e., x86, x86-64, ARM, and MIPS). However, it only performs a very basic analysis by running `readelf` and collecting network traces. Cuckoo sandbox [cuc] is another available tool that supports the analysis of Linux samples. However, the Cuckoo project only provides the external orchestration analysis framework, while the preparation of the various sandbox images is left to the user. Last, in November 2017 VirusTotal announced the integration of the Tencent HABO sandbox solution, which reportedly is able to analyze also Linux-based malware [vir]. Unfortunately, there is no public report on how the system works and it currently works only for x86 binaries.

One of the first systematic studies of IoT malware was performed by Pa *et al.* [MSY+15]. In the paper, the authors present a Telnet honeypot to measure the current attack trends as well as the first sandbox environment (IoTBOX) based on Qemu and OpenWRT for the analysis of IoT malware. The authors also discussed the issue of IoT devices exposing Telnet online and they collected few families actively targeting this service. Similarly,

Antonakakis *et al.* [Ant17] studied in detail a specific Linux malware family, the Mirai botnet. They measure systematically the evolution and growth of the botnet, mainly from a network point of view.

These works are invaluable to the community, but only look at limited aspects of the entire picture: the samples network behavior. We believe that the work presented in this thesis can complement these efforts and provide a clearer overview of how Linux malware actually *works*. Moreover, the datasets used in these previous studies are not representative of the overall Linux malware ecosystem, since they were collected via telnet-based honeypots.

The ELF file format is a standard also for Linux-based mobile operating systems e.g., Android. Android malware has received a lot of attention over the years with studies focusing on behaviors [TKFC15], characterization [ZJ12] and evolution [TFA+17]. More recently, the research around Android malware focused on the study of high-level user interactions [YLC+19] and the analysis of native components [WLO+18]. However, we believe that Android and Linux malware must be considered as distinct entities since the first relies more on the Android Framework (including Java and JNI) than Linux itself.

After we published our first study on Linux malware (described in Chapter 3) many follow-up studies have been published on the topic. For instance, Costin *et al.* [CZ18] provided a detailed survey of IoT malware samples extracting meaningful statistics from infection events and reporting on major vulnerabilities exploited by some families. Other works on the same line of research analyzed IoT infections leveraging low- and high-interaction honeypots [VS18] or endpoints reachable from the internet [CAA+19]. Dos Santos *et al.* [dSDC20] instead have shown how Linux and IoT malware can threaten building automation systems (BAS) powering smart buildings.

Recent research also focused on the defensive side. Mudgerikar *et al.* [MSB19] worked on an anomaly-based system-level IDS to profile IoT malware behaviors, while Coltellese *et al.* [CMM+19] proposed a triage system using C&C commands to model known attacks and identify new malware variants. Ding *et al.* [DLL+20] moved at a lower level using side-channels over power signals to identify malicious activities on IoT devices.

Meanwhile, the security community is also proposing studies and advances on the tools to support Linux malware analysis. For example, Darki *et al.* [DFAG+19] raised a question on the effectivity of IDA Pro (the most used disassembler in the field) for the analysis of stripped binaries. On the other hand, You *et al.* [YZK+20] compared our sandbox implementing

the analysis pipeline for Linux malware (that we present in Chapter 3.3) with their technique for malware dynamic analysis based on forced execution. Finally, PANDAcap [SBG20] aims at improving dynamic analysis by extending the PANDA framework [DGHH+15] to have automated and selective recording, and used it to implement an SSH honeypot.

2.3 Malware clustering and lineage

Malware clustering has been extensively studied in order to cope with the increasing sophistication and the rapid increase in the number of observed samples. As a result, there is a long list of works (of which we summarize what we believe to be the most relevant ones) that have looked at malware clustering and typically differ in the features or malware traits they extract and the clustering algorithm they use. Finally, we will discuss works that have used clustering to look into malware lineage in an effort to study the genealogy of malware strains.

Behavior-based malware clustering. Bailey *et al.* [BOA+07] treated user-visible system state changes (e.g., files written, processes created) to create a fingerprint of the malware's behavior, and leveraged single-linkage hierarchical clustering algorithm to automatically classify and analyze approximately 3.7k malware samples. Bayer *et al.* [BCH+09] transformed the augmented malware execution traces of 14K samples into behavioral profiles and applied a single-linkage hierarchical clustering algorithm to produce 87 clusters.

Perdisci *et al.* [PLF10] proposed a network-level behavioral malware clustering based on the analysis of structural similarities among malicious HTTP traffic traces generated by 25k malware samples. The goal was to produce high-quality malware network signatures.

Kirat *et al.* [KV15] proposed MalGene, a system to automatically identify system call events from evasive malware execution traces to build concise evasion signatures. The authors employed a complete-linkage hierarchical clustering algorithm to group 3.1k malware samples based upon pairwise Jaccard similarity of their evasion signatures.

Static analysis-based malware clustering. Hu *et al.* [HSBG13] proposed MutantX-S to exploit a hashing trick to reduce static feature dimension and leverage a prototype-based clustering algorithm to resolve the scalability issues faced by previous malware clustering approaches. On a similar note, Jiang *et al.* [JBV11] proposed BitShred to use feature

hashing to reduce the high-dimensional feature spaces that are common in malware analysis.

Evaluation of malware clustering results. Li *et al.* [LLGR10] discussed the challenges in evaluating malware clustering. The authors cautioned the security practitioners that a biased cluster-size distribution from the ground truth may lead to spurious high accuracy. Perdisci *et al.* [PU12] proposed the VAMO system to provide an automated quantitative analysis of the validity of malware clustering results. It constructed an AV label graph based upon the overlapping labels among malware verdicts, then built a reference clustering of the malware samples from this learned AV Label Graph to assess the quality of third party clustering results.

Malware lineage. The first notable piece of research looking at the malware lineage dates back to 1998 from Goldberg *et al.* [GGPS98]. Inspired by the study of the evolution of biological species, they transposed this concept into the malware area and introduced malware phylogenetic trees. Karim *et al.*in [KWLP05] presented a code fragment permutation-based technique to reconstruct malware family evolution trees. In 2011, Dumitras *et al.* [DN11] presented some guidance on malware studies and an experimental approach to study malware lineage. The authors advocated for the use of a combination of static and dynamic features, such as code fragments and dynamic control flow graphs, enriched with contextual information on the provenance of the studied samples. Lindorfer *et al.* [LDFM+12] developed Beagle, a system designed to track the evolution of a malware family. The authors relied on dynamic analysis to extract the different functionalities – in terms of API calls – exhibited by a piece of malware. They then tried to map these functionalities back to disassembled code so they could identify and characterize mutations within a malware family. In 2018, Calleja *et al.* [CTC18] – extending their previous work [CTC16] – studied the evolution of 456 Windows malware samples observed over 40+ years and identified code reuse between different families as well as with benign software. The types of code reuse they observed include essentially anti-analysis routines, shellcode, data such as credentials for brute-forcing attacks, and utility functions.

Huang *et al.* [HYD17] presented BinSequence, a tool to compare the similarity between functions extracted from binaries. The tool first uses fuzzy matching to reduce the number of pairwise comparisons. The authors then computed the similarity of functions at the instruction, basic block, and CFG levels. They applied their technique to different scenarios, including the identification of code reuse in two Windows malware families. They

also claim a function matching accuracy higher than 90%, above state-of-the-art approaches such as BinDiff or Diaphora. Jang *et al.* [JWB13] proposed iLine, a graph-based lineage recovery tool based on a combination of low-level binary features, code-level basic blocks, and binary execution traces. The technique was evaluated on a small dataset of 84 Windows malware, with an accuracy of 72%. Ming *et al.* [MXW15] also proposed an optimization for the iBinHunt binary diffing tool, which computes similarity between binaries from their execution traces. The authors further applied their optimized tool on a dataset of 145 Windows malware samples from 12 different families.

More recently, Haq *et al.* [HC19] reviewed 61 approaches from the literature on binary code similarity – some of which are used for malware lineage inference – published over the last 20 years. While they purposely focus on academic contributions rather than binary diffing tools, the authors highlight the diversity, strengths and weakness of all these techniques. They also identify several open problems, some of which were faced as well in our work, such as the scalability and the lack of support of multiple CPU architectures. We believe that binary-level or basic block-based malware slicing is likely to be prone to over-specific code reuse identification. Similarly, execution traces are likely to be too coarse-grained for variant identification.

Since the packing schemes currently used by IoT malware are simple and easy to unpack, we propose the use of function-level binary diffing to identify relevant code similarities between and within IoT malware families.

2.4 Function and library identification

Library and function identification is an open problem in the field of binary and malware analysis. It plays an important role in many security applications, such as library function detection, vulnerability re-discovery, code reuse analysis, malware detection, and authorship attribution. As a result, even if this problem dates back to the early 90s [VE93], the research community is still actively developing new solutions.

Function boundaries recognition is a core component of disassemblers and binary analysis frameworks [BJAS11, SWH+15, rad, Eag11]. While Kruegel *et al.* [KRVV04] used typical function prologs to detect the function start addresses, Rosenblum *et al.* [RZMH08] introduced machine learning as a possible solution for function identification. Shin *et al.* [SSM15] followed a similar approach but working with recurrent neural networks. In their experiments they show a drastic speedup in training time and better ac-

curacy than previous methods. Andriesse *et al.* [ASB17] preferred to avoid the intense training phases required by machine-learning approaches, and proposed a compiler-agnostic function detection algorithm at the CFG-level.

Once each function has been identified in the binary, the next problem is to recognize whether they are similar to functions in other binaries or libraries. Several works focused on the generation and matching of function signatures. Shirani *et al.* [SWD17] worked on BinShape to identify library functions using signatures based on CFG features, instruction-level properties, and statistical characteristics. Similarly, BinSign [NRM+17] uses *tracelets* extracted from the CFG and feature *min-hashing* to generate function fingerprints. Qiu *et al.* [QSM15a] also worked on CFGs for function identification, and introduced the concept of execution dependence graphs to describe the behavioral characteristics of binary code. Their approach makes it possible to recognize full library functions but also inline functions. Alrabaee *et al.* [AWD16]—with BinGold—extended CFGs with data-flow analysis components into a unique representation called *semantic flow graph*. The same authors worked on FOSSIL [ASWD18] to address the problem of function identification oriented to malware binaries. Specifically, the authors wanted to recognize—by using Bayesian networks—functions from open-source libraries compiled into malicious software. Their model relies on syntactical features of the binary, functions semantics, and the *z-score* to extract the behavior of instructions.

On the industry side, IDA FLIRT [Gui] adopts a solution based on a simple pattern-based recognition algorithm, operating on the instruction bytes. A limitation of function detection through signatures is that they involve the generation of a signatures database. This kind of recognition does not scale with the number of binaries to analyze and the many compilation flags that can be customized in modern compilers.

Library identification for statically linked binaries can be seen as a higher-level problem. Even if it can be carried out through multiple function recognition steps, it is often simplified by using code-based similarity approaches. Instead of generating a signature database for an entire library, researchers can look at the code similarity of the (malicious) binary with a pre-compiled package. One of the most famous tools for this purpose is BinDiff [Zyn]. BinDiff relies on a structural-based approach and functions isomorphism [DR05, Fla04] to find differences and similarities in disassembled code. Diaphora [urla] is a very similar but open-source alternative built around heuristics on graph theory, instructions, and function features. Other works instead focus directly on binary clone search. Farhadi *et*

al. [FFCD14] developed BinClone to detect code clones in malware. Differently from BinDiff, this solution compares regions of assembly instructions to find exact and inexact matches of code fragments. Ding *et al.*made two contributions to the same topic. The first one is Kam1n0 [DFC16], which proposes a variant of local-sensitive hashing to implement an assembly clone search engine. The second is Asm2Vec [DFC19], which relies on semantic relationships and the vector representation of assembly functions to match clones even in the presence of obfuscation techniques. Huang *et al.* [HYD17] published BinSequence, a code reuse detector that operates with a fuzzy matching approach on the instruction, basic block, and control flow levels.

Both function fingerprinting and code similarity can be used to restore the library identities in stripped binaries. However, most of these solutions are sensitive to variations in the compilation toolchain and to new library versions. Jacobson *et al.* [JRM11] used UNSTRIP to create semantic descriptors and provide high-level representations of library functions. However, UNSTRIP only works on wrapper functions containing system call invocations, thus mostly on C libraries. Debin [HIT+18] overcomes these limitations using probabilistic models to predict symbol names, types, and locations in stripped ELF files. Punstrip [PECK20] combines probabilistic fingerprint of binary code with a probabilistic graphical model to infer symbol information.

In summary, we are still unable today to recognize libraries without using signatures or binary similarity tools. In particular, we lack solutions to automatically detect the boundary addresses of code modules (e.g., to separate user code and libraries) in statically linked software. The most recent contribution in this direction is CodeCut [evm]. CodeCut introduces the concept of *local function affinity*, a call directionality metric, to detect the boundaries of object files of embedded operating systems (often compiled in a single linked program). However, our preliminary tests on Linux binaries did not produce accurate results as explained in Chapter 5.1, in which we also present our solution to the problem.

Chapter 3

Understanding Linux Malware

3.1 Introduction

This chapter presents the first large-scale empirical study conducted to characterize and understand Linux-based malware (for both embedded devices and traditional personal computers). We first systematically enumerate the challenges that arise when collecting and analyzing Linux samples. For example, we show how supporting malware analysis for "common" architectures such as x86 and ARM is often insufficient, and we explore several challenges including the analysis of statically linked binaries, the preparation of a suitable execution environment, and the differential analysis of samples run with different privileges. We also detail Linux-specific techniques that are used to implement different aspects traditionally associated with malicious software, such as anti-analysis tricks, packing and polymorphism, evasion, and attempts to gain persistence on the infected machine. These insights were uncovered thanks to an analysis pipeline we specifically designed to analyze Linux-based malware and the experiments we conducted with over 10K malicious samples. Our results show that Linux malware is already a multi-faced problem. While still not as complex as its Windows counterpart, we were able to identify many interesting behaviors—including the ability of certain samples to properly run in multiple operating systems, the use of privilege escalation exploits, or the custom modification of the UPX packer adopted to protect their code. We also found that a considerable fraction of Linux malware interacts with other shell utilities and, despite the lack of available malware analysis sandboxes, that some samples already implement a wide range of VM-detections approaches. Finally, we also performed a differential analysis to study how the malware behavior changes when the same sample is executed with or without *root* privileges.

In summary, this chapter brings the following contributions:

- We document the design and implementation of several tools we designed to support the analysis of Linux malware and we discuss the challenges involved when dealing with this particular type of malicious files.

- We present the first large-scale empirical study conducted on 10,548 Linux malware samples obtained over a period of one year.

- We uncover and discuss a number of low-level Linux-specific techniques employed by real-world malware and we provide detailed statistics on the current usage.

We make the raw results of all our analyzed samples available to the research community and we provide our entire infrastructure as a free service to other researchers.

3.2 Challenges

The analysis of generic (and potentially malicious) Linux programs requires tackling a number of specific challenges. This section presents a systematic exploration of the main problems we encountered in our study.

3.2.1 Target Diversity

The first problem relates to the broad diversity of the possible target environments. The general belief is that the main challenge is about supporting different architectures (e.g., ARM or MIPS), but this is in fact only one aspect of a much more complex problem. Malware analysis systems for Windows, MacOS, or Android executables can rely on detailed information about the underlying execution environment. Linux-based malware can instead target a very diverse set of targets, such as Internet routers, printers, surveillance cameras, smart TVs, or medical devices. This greatly complicates their analysis. In fact, without the proper information about the target (unfortunately, program binaries do not specify where they were supposed to run) it is very hard to properly configure the right execution environment.

Computer Architectures. Linux is known to support tens of different architectures. This requires analysts to prepare different analysis sandboxes and port the different architecture-specific analysis components to support each of them. In a recent work covering the Mirai botnet [Ant17], the authors supported three architectures: MIPS 32-bit, ARM 32-bit, and x86 32-bit. However, this covers a small fraction of the overall malware landscape for Linux. For instance, these three architectures together only cover about 32% of our dataset. Moreover, some families (such as ARM) are particularly challenging to support because of the large number of different CPU architectures they contain.

Loaders and Libraries. The ELF file format allows a Linux program to specify an arbitrary *loader*, which is responsible to load and prepare the executable in memory. Unfortunately, a copy of the requested loader may not be present in the analysis environment, thus preventing the sample from starting its execution. Moreover, dynamically linked binaries expect their required libraries to be available in the target system: once again, it is

enough for a single library to be missing to prevent the successful execution of the program. Contrary to what one would expect, in the context of this work these aspects affect a significant portion of our dataset. A common example are Linux programs that are dynamically linked with *uClibc* or *musl*, smaller and more performant alternatives to the traditional *glibc*. Not only does an analysis environment need to have these alternatives installed, but their corresponding loaders are also required.

Operating System. This work focuses on Linux binaries. However, and quite unexpectedly, it can be challenging to discern ELF programs compiled for Linux from other ELF-compatible operating systems, such as FreeBSD or Android. The ELF headers include an "OS/ABI" field that, in principle, should specify which operating system is required for the program to run. In practice, this is rarely informative. For example, ELF binaries for both Linux and Android specify a generic "System V" OS/ABI. Moreover, current Linux kernels seem to ignore this field, and it is possible for a binary that specifies "FreeBSD" as its OS/ABI to be a valid Linux program, a trick that was abused by one of the malware sample we encountered in our experiments. Finally, while a binary compiled for FreeBSD can be properly loaded and executed under Linux, this is only the case for dynamically linked programs. In fact, the syscalls numbers and arguments for Linux and FreeBSD do not generally match, and therefore statically linked programs usually crash when they encounter such a difference. These differences may also exist between different versions of the Linux kernel, and custom modifications are not too rare in the world of embedded devices. This has two important consequences for our work: On the one hand, it makes it hard to compile a dataset of Linux-based malware. On the other hand, this also results in the fact that even well-formed Linux binaries may not be guaranteed to run correctly in a generic Linux system.

3.2.2 Static Linking

When a binary is statically linked, all its library dependencies are included in the resulting binary as part of the compilation process. Static linking can offer several advantages, including making the resulting binary more portable (as it is going to execute correctly even when its dependencies are not installed in the target environment) and making it harder to reverse engineer (as it is difficult to identify which library functions are used by the binary).

Static linking introduces also another, much less obvious challenge for malware analysis. In fact, since these binaries include all their libraries,

the resulting application does not rely on any external wrapper to execute system calls. Normal programs do not call system calls directly, but invoke instead higher level API functions (typically part of the `libc`) that in turn wrap the communication with the kernel. Statically linked binaries are more portable from a library dependency point of view, but less portable as they may crash at runtime if the kernel ABI is different from what they expected (and what was provided by the—unfortunately unknown—target system).

3.2.3 Analysis Environment

An ideal analysis sandbox should emulate as closely as possible the system in which the sample under analysis was supposed to run. So far we have discussed challenges related to setting up an environment with the correct architecture, libraries, and operating system, but these only cover part of the environment setup. Another important aspect is the privileges the program should run with. Typically, malware analysis sandboxes execute samples as a normal, unprivileged user. Administration privileges would give the malware the ability to tamper with the sandbox itself and would make the instrumentation and observation of the program behavior much more complex. Moreover, it is very uncommon for a Windows sample to expect super-user privileges to work.

Unfortunately, Linux malware is often written with the assumption (true for some classes of embedded targets) that its code would run with root privileges. However, since these details are rarely available to the analyst, it is difficult to identify these samples in advance. We will discuss how we deal with this problem by performing a differential analysis in Chapter 3.3.

3.2.4 Lack of Previous Studies

To the best of our knowledge, this is the first work that attempts to perform a comprehensive analysis of the Linux malware landscape. This mere fact introduces several additional challenges. First, it is not clear how to design and implement an analysis pipeline specifically tailored for Linux malware. In fact, analysis tools are tailored to the characteristics of the existing malware samples. Unfortunately, the lack of information on how Linux-based malware works complicated the design of our pipeline. Which aspects should we focus on? Which architectures do we need to support? A second problem in this domain is the lack of a comprehensive dataset. One of the few works looking at Linux-based malware focused only on botnets, thus using honeypots to build a representative dataset. Unfortunately,

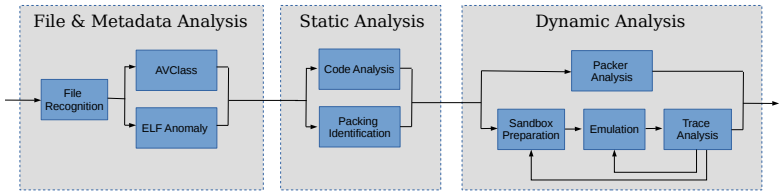


Figure 3.1: Overview of the analysis pipeline for Linux malware.

this approach would bias our study towards those samples that propagate themselves on random targets.

3.3 Analysis Infrastructure

The task of designing and implementing an analysis infrastructure for Linux-based malware was complicated by the fact that when we started our experiments we still knew very little about how Linux malware worked and of which techniques and components we would have needed to study its behavior. For instance, we did not know a priori any of the challenges we discussed in the previous section and we often had wrong expectations about the prevalence of certain characteristics (such as static linking or malformed file headers) or their impact on our analysis strategy.

Despite our extensive experience in analyzing malicious files for Windows and Android, we only had an anecdotal knowledge of Linux-based malware that we obtained by reading online reports describing manual analysis of specific families. Therefore, the design and implementation of an analysis pipeline became a trial-and-error process that we tackled by following an incremental approach. Each analysis task was implemented as an independent component, which was integrated in an interactive framework responsible to distribute the jobs execution among multiple parallel workers and to provide a rich interface for human analysts to inspect and visualize the data. As more samples were added to our analysis environment every day, the system identified and reported any anomaly in the results or any problem that was encountered in the execution of existing modules (such as new and unsupported architectures, errors that prevented a sample from being correctly executed in our sandboxes, or unexpected crashes in the adopted tools). Whenever a certain issue became widespread enough to impact the successful analysis of a considerable number of samples, we introduced new analysis modules and designed new techniques to address the problem. Our framework was also designed to keep track of which version of each module

was responsible for the extraction of any given piece of information, thus allowing us to dynamically update and improve each analysis routine without the need to re-start each time the experiments from scratch.

Our final analysis pipeline included a collection of existing state-of-the-art solutions (such as AVClass [SRKC16a], IDA Pro, radare2 [rad], and Nucleus [ASB17]) as well as completely new tools we explicitly designed for this work. Due to space limitations we cannot present each component in details. Instead, in the rest of this section we briefly summarize some of the techniques we used in our experiments, organized in three different groups: *File and Metadata Analysis*, *Static Analysis*, and *Dynamic Analysis* components.

3.3.1 Data Collection

To retrieve data for our study we used the VirusTotal intelligence API to fetch the reports of *every* ELF file submitted between November 2016 and November 2017. Based on the content of the reports, we downloaded 200 candidate samples per day. Our selection criteria were designed to minimize non-Linux binaries and to select at least one sample for each family observed during the day. We also split our selection in two groups: 140 samples taken from those with more than five AV positive matches, and 60 samples with an AV score between one and five.

3.3.2 File & Metadata Analysis

The first phase of our analysis focuses on the file itself. Certain fields contained in the ELF file format are required at runtime by the operating system, and therefore need to provide reliable information about the architecture on which the application is supposed to run and the type of code (e.g., executable or shared object) contained in the file. We implemented our custom parser for the ELF format because the existing ones (as explained in Chapter 3.5.1) were often unable to cope with malformed fields, unexpected values, or missing information.

We use the data extracted from each file for two purposes. First, to filter out files that were not relevant for our analysis. For instance, shared libraries, core dumps, corrupted files, or executables designed for other operating systems (e.g., when a sample imported an Android library). Second, we use the information to identify any anomalous file structure that, while not preventing the sample to run, could still be used as anti-analysis routine and prevent existing tools to correctly process the file (see Chapter 3.5.1 for more details about our findings).

Finally, as part of this first phase of our pipeline, we also extract from the VirusTotal reports the AV labels for each sample and fed them to the AVClass tool to obtain a normalized name for the malware family. AVClass, recently proposed by Sebastián et al. [SRKC16a], implements a state-of-the-art technique to normalize, remove generic tokens, and detect aliases among a set of AV labels assigned to a malware sample. Therefore, whenever it is able to output a name, it means that there was a general consensus among different antivirus on the class (family) the malware belongs to.

3.3.3 Static Analysis

Our static analysis phase includes two tasks: binary code analysis and packing detection. The first task relied on a number of custom IDA Pro scripts to extract several code metrics—including the number of functions, their size and cyclomatic complexity, their overall coverage (i.e., the fractions of the `.text` section and `PT_LOAD` segments covered by the recognized functions), the presence of overlapping instructions and other assembly tricks, the direct invocation of system calls, and the number of direct/indirect branch instructions. In this phase we also computed aggregated metrics, such as the distribution of opcodes, or a rolling entropy of the different code and data sections. This information is used for statistical purposes, but also integrated in other analysis components, for instance to identify anti-analysis behaviors or packed samples.

The second task of the static analysis phase consists of combining the information extracted so far from the ELF headers and the binary code analysis to identify likely packed applications (see Chapter 3.5.5 for more details). Binaries that could be statically unpacked (e.g., in the common case of UPX) were processed at this stage and the result fed back to be statically analyzed again. Samples that we could not unpack statically were marked in the database for a subsequent more fine-grained dynamic attempt.

3.3.4 Dynamic Analysis

We performed two types of dynamic analysis in our study: a five-minute execution inside an instrumented emulator, and a custom packing analysis and unpacking attempt. For the emulation, we implemented two types of dynamic sandboxes: a KVM-based virtualized sandbox with hardware support for x86 and x86-64 architectures, and a set of QEMU-based emulated sandboxes for ARM 32-bit little-endian, MIPS 32-bit big-endian, and PowerPC 32-bit. These five sandboxes were nested inside an outer VM

dedicated to dispatch each sample depending on its architecture. Our system also maintained several snapshots of all VMs, each corresponding to a different configurations to choose from (e.g., execution under user or root accounts and *glibc* or *uClibc* setup). All VMs were equipped with additional libraries, the list of which was collected during the static analysis phase, as well as popular loaders (such as the *uClibc* commonly used in embedded systems).

For the instrumentation we relied on SystemTap [sysb] to implement kernel probes (*kprobes*) and user probes (*uprobes*). While, according to its documentation, SystemTap should be supported on a variety of different architectures (such as x86, x86-64, ARM, aarch64, MIPS, and PowerPC), in practice we needed to patch its code to support ARM and MIPS with o32 ABI. Our patches include fixes on syscall numbers, CPU registers naming and offsets, and the routines required to extract the syscall arguments from the stack. We designed our SystemTap probes to collect every system call, along with its arguments and return value, and the instruction pointer from which the syscall was invoked. We also recompiled the *glibc* to add *uprobes* designed to collect, when possible, additional information on string and memory manipulation functions.

At the end of the execution, each sandbox returns a text file containing the full trace of system calls and userspace functions. This trace is then immediately parsed to identify useful feedback information for the sandbox. For example, this preliminary analysis can identify missing components (such as libraries and loaders) or detect if a sample tested its user permissions or attempted to perform an action that failed because of insufficient permissions. In this case, our system would immediately repeat the execution of the sample, this time with `root` privileges. As explained in Session 3.5.4, we later compare the two traces collected with different users as part of our differential analysis to identify how the sample behavior was affected by the privilege level. Finally, the preliminary trace analysis can also report to the analyst any error that prevented the sample to run in our system. As an example of these warnings, we encountered a number of ARM samples that crashed because of a four-byte misalignment between the physical and virtual address of their `LOAD` segments. These samples were probably designed to infect an ARM-based system whose kernel would memory map segments by considering their physical address, something that does not happen in common desktop Linux distributions. We extended our system with a component designed to identify these cases by looking at the ELF headers and fix the data alignment before passing them to the dynamic analysis stage.

To avoid hindering the execution and miss important code paths, we gave samples partial network access, while monitoring the traffic for signs of abuse. Although not an ideal solution, a similar approach has been previously adopted in other behavioral analysis experiments [vir, Ant17] as it is the only way to observe the full behavior of a sample.

Our system also record PCAP files of the network traffic, due to space limitations we will not discuss their analysis as this is the only aspect of Linux-based malware that was already partially studied in previous works [Ant17]. Finally, to dynamically unpack unknown UPX variants we developed a tool based on Unicorn [Ngu]. The system emulates instructions on multiple architectures and behaves like a tiny kernel that exports the limited set of system calls used by UPX during unpacking (supporting a combination of different system call tables and system call ABIs). As we explain in Chapter 3.5.5, this approach allowed us to automatically unpack all but three malware samples in our dataset.

3.4 Dataset

Our final dataset, after the filtering stage, consisted of 10,548 ELF executables, covering more than ten different architectures (see Table 3.1 for a breakdown of the collected samples). Note again how the distribution differs from other datasets collected only by using honeypots: x86, ARM 32-bit, and MIPS 32-bit covered 75% of the data used by Antonakakis et al. [Ant17] on the Mirai botnet, but only account for 32% of our samples.

Here we just want to focus on their broad variety and on the large differences that exist among all features we extracted in our database. For example, our set of Linux-based malware vary considerably in size, from a minimum of 134 bytes (a simple backdoor) to a maximum of 14.8 megabytes (a botnet coded in Go). IDA Pro was able to recognize (in dynamically linked binaries) from a minimum of zero (in two samples) to a maximum of 5685 unique functions. Moreover, we extracted from the ELF header of dynamically linked malware the symbols imported from external libraries—which can give an idea of the most commonly used functionalities. Most samples import between 10 and 100 symbols. Interestingly, there are more than 10% of the samples that use `malloc` but never use `free`. And while `socket` is one of the most common functions (confirming the importance that interconnected devices have nowadays) less than 50% of the binaries requests file-based routines (such as `fopen`). Finally, entropy plays an important role to identify potential packers or encrypted binary blobs. The vast majority of the binaries in our dataset has entropy around six, a com-

Table 3.1: Distribution of the 10,548 downloaded samples across architectures

Architecture	Samples	Percentage
X86-64	3018	28.61%
MIPS I	2120	20.10%
PowerPC	1569	14.87%
Motorola 68000	1216	11.53%
Sparc	1170	11.09%
Intel 80386	720	6.83%
ARM 32-bit	555	5.26%
Hitachi SH	130	1.23%
AArch64 (ARM 64-bit)	47	0.45%
others	3	0.03%

mon value for compiled but not packed code. However, one sample had entropy of only 0.98, due to large blocks of null bytes inserted in the data segment.

3.4.1 Malware Families

The AVClass tool was able to associate a family (108 in total) to 83% of the samples in our dataset. As expected, botnets, often dedicated to run DDoS attacks, dominate the Linux-based malware landscape—accounting for 69% of our samples spread over more than 25 families. One of the reasons for this prevalence is that attackers often harvest poorly protected IoT devices to join large remotely controlled botnets. This task is greatly simplified by the availability of online services like Shodan [sho] or scanning tools like ZMap [DWH13] that can be used to quickly locate possible targets. Moreover, while it may be difficult to monetize the compromise of small embedded devices that do not contain any relevant data, it is still easy to combine their limited power to run large-scale denial of service attacks. Another possible explanation for the large number of botnet samples in our dataset is that the source code of some of these malware family is publicly available—resulting in a large number of variations and copycat software.

Despite their extreme popularity, botnets are not the only form of Linux-based malware. In fact, our dataset contains also thousands of samples belonging to other categories, including backdoors, ransomware, cryptocurrency miners, bankers, traditional file infectors, privilege escalation tools,

Table 3.2: ELF Header Manipulation

Technique	Samples	Percentage
Segment header table pointing beyond file data	1	0.01%
Overlapping ELF header/segment	2	0.02%
Wrong string table index (`e_shstrndx`)	60	0.57%
Section header table pointing beyond file data	178	1.69%
Total Corrupted	211	2.00%

rootkits, mailers, worms, RAT programs used in APT campaigns, and even CGI-based binary webshells. While these malware dominates the number of families in our dataset, many of them exist in a single variant, thus resulting in a lower number of samples.

While we may discuss particular families when we present our analysis results, in the rest of the chapter we prefer to aggregate figures by counting individual samples. This is because even though samples in the same family may share a common goal and overall structure, they can be very diverse in the individual low-level techniques and tricks they employ (e.g., to achieve persistence or obfuscate the program code). We will return to this aspect of Linux malware and discuss its implications in Chapter 3.6.

3.5 Under the Hood

In this section we present a detailed overview of a number of interesting behaviors we have identified in Linux malware and, when possible, we provide detailed statistics about the prevalence of each of these aspects. Our goal is not to differentiate between different classes of malware or different malware families (i.e., to distinguish botnets from backdoors from ransomware samples), but instead to focus on the tricks and techniques commonly used by malware authors—such as packing, obfuscation, process injection, persistence, and evasion attempts. To date, this is the most comprehensive discussion on the topic, and we hope that the insights we offer will help to better understand *how* Linux-based malware works and will serve as a reference for future research focused on improving the analysis of this type of malware.

Table 3.3: ELF samples that cannot be properly parsed by known tools

Program	Errors on Malformed Samples
readelf 2.26.1	166 / 211
GDB 7.11.1	157 / 211
pyelftools 0.24	107 / 211
IDA Pro 7	- / 211

3.5.1 ELF headers Manipulation

The Executable and Linkable Format (ELF) is the standard format used to store (among others) all Linux executables. The format has a complex internal layout, and tampering with some of its fields and structures provides attackers a first line of defense against analysis tools.

Some fields, such as `e_ident` (which identifies the type of file), `e_type` (which specifies the object type), or `e_machine` (which contains the machine architecture), are needed by the kernel even before the ELF file is loaded in memory. Sections and segments are instead strictly dependent on the source code and the compilation process, and are needed respectively for linking and relocation purposes and to tell the kernel how the binary must be loaded in memory for program execution.

Our data shows that malware developers often tamper with the ELF headers to fool the analyst or crash common analysis tools. In particular, we identified two classes of modifications: those that resulted in *anomalous* files (but that still follow the ELF specifications), and those that produced *invalid* files—which however can still be properly executed by the operating system.

Anomalous ELF. The most common example in the first category (5% of samples in our dataset) consists in removing all information about the ELF sections. This is valid according to the specifications (as sessions information are not used at runtime), but it is an uncommon case that is never generated by traditional compilers. Another example of this category consists of reporting false information about the executable. For example, a Linux program can report a different operating system ABI (e.g., FreeBSD) and still be executed correctly by the kernel. Samples of the *Mumblehard* family report in the header the fact that they require FreeBSD, but then test the system call table at runtime to detect the actual operating system and execute correctly under both FreeBSD and Linux.

For this reason, in our experiments we did not trust such information

and we always tried to execute a binary despite the values contained in its identification field. If the required ABI was indeed different, the program would crash at runtime trying to execute invalid system calls—a case that was recognized by our system to filter out non-Linux programs.

Invalid ELF. This category includes instead those samples with malformed or corrupted sections information (2% of samples in our dataset), typically the result of an invalid `e_shoff` (offset of the section header table), `e_shnum` (number of entries in the section header table), or `e_shentsize` (size of section entries) fields in the ELF header. We also found evidence of samples exploiting the ELF header file format to create overlapping segments header. For instance, three samples belonging to the *Mumblehard* family declared a single segment starting from the 44^{th} byte of the ELF header itself and zeroed out any field unused at runtime. Table 3.2 summarizes the most common ELF manipulation tricks we observed in our dataset.

Impact on Userspace Tools. To measure the consequences of the previously discussed transformations, in Table 3.3 we report how popular tools (used to work with ELF files) react to unusual or malformed files. This includes *readelf* (part of GNU Binutils), *pyelftools* (a convenient Python library to parse and analyze ELF files), *GDB* (the de-facto standard debugger on Linux and many UNIX-like systems), and *IDA Pro 7* (the latest version, at the time of writing, of the most popular commercial disassembler, decompiler, and reverse engineering tool).

Our results show that all tools are able to properly process anomalous files, but unfortunately often result in errors when dealing with invalid fields. For example, *readelf* complained for the absence of a valid table on hundreds of sample, but was able to complete the parsing of the remaining fields in the ELF header. On the other side, *pyelftools* denies further analysis if the section header table is corrupted, while it can instead parse ELF files if the table is declared as empty. Because of this poor management of erroneous conditions, for our experiments we decided to write our own custom ELF parser, which was specifically designed to work in presence of unusual settings, inconsistencies, invalid values, or malformed header information.

Despite its widespread use in the *nix world, GDB showed a severe lack of resilience in dealing with corrupted information coming from a malformed section header table. The presence of an invalid value results in GDB not being able to recognize the ELF binary and in its inability to start the program.

Finally, IDA Pro 7 was the only tool we used in our analysis pipeline that was able to handle correctly the presence of any corrupted session

Table 3.4: ELF binaries adopting persistence strategies

Path	Samples	
	w/o root	w/ root
/etc/rc.d/rc.local	-	1393
/etc/rc.conf	-	1236
/etc/init.d/	-	210
/etc/rcX.d/	-	212
/etc/rc.local	-	11
systemd service	-	2
~/.bashrc	19	8
~/.bash_profile	18	8
X desktop autostart	3	1
/etc/cron.hourly/	-	70
/etc/crontab	-	70
/etc/cron.daily/	-	26
crontab utility	6	6
File replacement	-	110
File infection	5	26
Total	1644 (21.10%)	

information or other fields that would not affect the program execution.

3.5.2 Persistence

Persistence involves a configuration change of the infected system such that the malicious executable will be able to run regardless of possible reboot and power-off operations performed on the underlying machine. This, along with the ability to remain hidden, is one of the first objectives of malicious code.

A broad and well-documented set of techniques exists for malware authors to achieve persistence on Microsoft Windows platforms. The vast majority of these techniques relies on the modification of Registry keys to run software at boot, when a user logs in, when certain events occurs, or to schedule particular services. Linux-based malware needs to rely on different strategies, which are so far more limited both in number and in nature. We group the techniques that we observed in our dataset in four categories, described next.

Subsystems Initialization. This appears to be the most common ap-

proach adopted by malware authors and takes advantage of the well known Linux *init* system. Table 3.4 shows that more than 1000 samples attempted to modify the system *rc* script (executed at the end of each run-level). Instead, 210 samples added themselves under the `/etc/init.d/` folder and then created soft-links to directories holding run-level configurations. Overall, we found 212 binaries displacing links from `/etc/rc1.d` to `/etc.rc5.d`, with 16 of them using the less common run-levels dedicated to machine halt and reboot operations. Note how malicious programs still largely rely on the System-V init system and only two samples in our dataset supported more recent initialization technologies (e.g., *systemd*). More important, this type of persistence only works if the running process has privileged permissions. If the user executing the ELF is not root or a user under privileged policies, it is usually impossible to modify services and initialization configurations.

Time-based Execution. This technique is the second choice commonly used by malware and relies on the presence of *cron*, the time-based job scheduler for Unix systems. Malicious ELF files try to modify, with success when running under adequate higher privileges, *cron* configuration files to get scheduled execution at a fixed time interval. As for subsystem initialization, time-based persistence will not work if the malware is launched by unprivileged users unless the sample invokes the system utility *crontab* (a SUID program specifically designed to modify configuration files stored under `/var/spool/cron/`).

File Infection and Replacement. Another approach for malware to maintain a foothold in the system is by replacing (or infecting) applications that already exist in the target. This includes both a traditional *virus*-like behavior (where the malware locates and infect other ELF files without a strategy) as well as more targeted approaches that subvert the original functionalities of specific system tools.

Our dynamically analysis reports allow us to observe infection and replacement of system and user files. Examples in this category are samples in the family *EbolaChan*, which inject their code at the beginning of the `ls` tool and append the original code after the malicious data. Another example are samples of the *RST*, *Sickabs* and *Diesel* families, which still use a 20 years old ELF infections techniques [Silc]. The first group limits the infection to other ELF files located in the current working directory, while the second adopts a system-wide infection that also targets binaries in the `/bin/` folder. Interestingly, samples of this family were first observed in 2001, according to a Sophos report they were still widespread in 2008 [Sop], and our study shows that they are still surprisingly active today. A different approach is taken by samples in the *Gates* family, which fully replace system tools in

Table 3.5: ELF programs renaming the process

Process name	Samples	Percentage
sshd	406	5.21%
telnetd	33	0.42%
cron	31	0.40%
sh	14	0.18%
busybox	11	0.14%
other tools	22	0.28%
empty	2034	26.11%
other *	973	12.49%
random	618	7.93%
Total	4091	52.50%

* Names not representing system utilities

`/bin/` or `/usr/bin/` folders (e.g., `ps` and `netstat`) after creating a backup copy of the original code in `/usr/bin/dpkgd/`.

User Files Alteration. As shown in the middle part of Table 3.4, very few samples modify configuration files in the user home directory such as shell configurations. Malware writers adopting this method can ensure persistence at user level, but other Linux users, beside the infected one, will not be affected by this persistence mechanism. While the most common, changes to the shell configuration are not the only form of per-user persistency. Few samples (such as those in the *Handofthief* family) that target desktop Linux installations, modified instead the `.desktop` startup files used by the windows manager.

Table 3.4 reports a summary of the amount of samples using each technique. Surprisingly, only 21% of our ELF files implemented at least one persistence strategy. However, samples that do try to be persistent often try multiple techniques in a row to reach their objective. As an example, in our experiments we noticed that user files alteration was a common fallback mechanism when the sample failed to achieve system-wide persistency.

3.5.3 Deception

Stealthy malware may try to hide their nature by assuming names that look genuine and innocuous at a first glance, with the objective of tricking

Table 3.6: ELF samples getting privileges errors or probing identities

Motivation	Samples	Percentage
EPERM error	986	12.65%
EACCES error	716	9.19%
Query user identity *	1609	20.65%
Query group identity *	877	11.26%
Total	2637	33.84%

* Also include checks on effective and real identity

the user to open an apparently benign program, or avoid showing unusual names in the list of running processes.

Overall, we noted that this behavior, already common on Windows operating systems, is also widespread on Linux-based malware. Table 3.5 shows that over 50% of the samples assumed different names once in memory, and also reports the top benign application that are impersonated. In total we counted more than 4K samples invoking the system call `prctl` with request `PR_SET_NAME`, or simply modifying the first command line argument of the program (the program name). Out of those, 11% adopted names taken from common utilities. For example, samples belonging to the *Gafgyt* family often disguise as *sshd* or *telnetd*. It is also interesting to discuss the difference between the two renaming techniques. The first (based on the `prctl` call) results in a different process name listed in `/proc/<PID>/status` (and used by tools like `pstree`), while the second modifies the information reported in `/proc/<PID>/cmdline` (used by `ps`). Quite strangely, none of the malware in our dataset combined the two techniques (and therefore could all be easily detected by looking for name inconsistencies).

The remaining 88% of the samples either adopted an empty name, a name of a fictitious (but not existing) file, or a random-looking name often seeded by a combination of the current time and the process PID. This last behavior, implemented by some of the *Mirai* samples, results in the fact that the malicious process assumes a different name at every execution.

3.5.4 Required Privileges

Our tests show that the distinction between administrator (`root`) and normal user is very important for Linux-based malware. First, malicious samples can perform different actions and show a different behavior when they

Table 3.7: Behavioral differences between user/root analysis

Different behavior	Samples	Percentage
Execute privileged shell command	579	21.96%
Drop a file into a protected directory	426	16.15%
Achieve system-wide persistence	259	9.82%
Tamper with Sandbox	61	2.31%
Delete a protected file	47	1.78%
Run *ptrace* request on another process	10	0.38%

are executed with super-user privileges. Second, especially when targeting low-end embedded systems or IoT devices, malware may even be designed to run as `root`—and thus fail to execute if analyzed with more limited privileges.

Therefore, we first executed every sample with normal user privileges. If, during the execution, we detected any attempt to retrieve the user or group identities (which could be used by the program to decide the malware's next actions) or to access any resource that returned a EPERM or EACCES errors, we repeated the analysis by running the sample with `root` privileges. This was the case for 2637 samples (25% of the dataset) and in 89% of them we detected differences in the sample behavior extracted from the two execution traces.

Table 3.7 presents a list of behaviors that were executed when running as root but were not observed when running as a normal user. Among these, privileged shell commands and operations on files are predominant, with malware using elevated privileges to create or delete files in protected folders. For instance, samples of the *Flooder* and *IoTReaper* families hide their traces by deleting all log files in `/var/log`, while samples of the *Gafgyt* family only delete last login and logout information (`/var/log/wtmp`). Moreover, in few cases malware running as root were able to tamper with the sandboxed execution: we found binaries that, upon detection of the emulated execution environment, would kill the SSH daemon or even delete the entire file system.

We now look in more details at two specific actions that are determined by the execution privileges: privileges escalation exploits and interaction with the OS kernel.

Privileges Escalation. On the one hand, one of the advantages of using kernel probes for dynamic analysis is its ability to trace functions in the OS kernel—making possible for us to detect signs of successful exploita-

tions. For example, by monitoring `commit_creds` we can detect when a new set of credentials has been installed on a running task. On the other hand, the sandboxes built to host the execution of each sample were deployed with up-to-date and fully-patched Linux operating systems—which prevented binaries from exploiting old vulnerabilities.

According to our trace analysis, there was no evidence of samples that successfully elevated their privileges inside our machines, or that had been able to perform privileged actions under user credentials. Regarding older (and therefore unsuccessful) exploits, we developed custom signatures to identify the ten most common escalation attacks based on known vulnerabilities in the Linux kernel[1], for which an exploitation proof-of-concept is available to the public. Our tests revealed that CVE-2016-5195 was the most frequently used vulnerability, with a total of 52 ELF programs that tried to exploit it in our sandbox. We also detected five attempts to exploit CVE-2015-1328, while the remaining eight checks did not return any positive match.

Kernel Modules. System calls tracing allows our system to track attempts to load or unload a kernel module, especially when samples are executed with `root` privileges. Interestingly, among the 2,637 malware samples we re-executed with `root` privileges, only 15 successfully loaded a kernel module and none of them performed an unload procedure. All these cases involved the standard `ip_tables.ko`, necessary to setup IP packet filter rules. We also identified 119 samples, belonging to the *Gates* or *Elknot* families that *attempted* to load a custom kernel module but failed as the corresponding `.ko` file was not present during the analysis.[2]

3.5.5 Packing & Polymorphism

Runtime packing is at the same time one of the most common and one of the most sophisticated obfuscation techniques adopted by malware writers. If properly implemented, it completely prevents any attempt to statically analyze the malware code and it also considerably slows down an eventual manual reverse engineering effort. While hundreds of commercial, free, and underground packers exist for Microsoft Windows, things are different in the Linux world: only a handful of ELF packers have been proposed

[1] CVE-2017-7308, CVE-2017-6074, CVE-2017-5123, CVE-2017-1000112, CVE-2016-9793, CVE-2016-8655, CVE-2016-5195, CVE-2016-0728, CVE-2015-1328, CVE-2014-4699.

[2] This is a well-known problem affecting dynamic malware analysis systems, as samples are collected and submitted in isolation and can thus miss external components that were part of the same attack.

Table 3.8: ELF packers

Process name	Samples	Percentage
Vanilla UPX	189	1.79%
Custom UPX Variant	188	1.78%
- Different Magic	129	
- Modified UPX strings	55	
- Inserted junk bytes	126	
- All of the previous	16	
Mumblehard Packer	3	0.03%

so far [Tea, elfb, gru], and the vast majority of them are proof-of-concept projects. The only exception is UPX, a popular open source compression packer introduced in 1998 to reduce the size of benign executables, which is freely available for many operating systems.

Automatic recognition and analysis of packers is a subtle problem, and it has been the focus on many academic and industrial studies [LH07, UPBSB16, CX10, PLL08, STMF]. For our experiment, we relied on a set of heuristics based on the file segments entropy and on the results of the static analysis phase (i.e., number of imported symbols, percentage of code section correctly disassembled, and total number of functions identified) to flag samples that were likely packed. Moreover, since UPX-like variants seem to dominate the scene, we decided to add to our pipeline a set of custom analysis routines to identify possible UPX variants and a generic multi-architecture unpacker that can retrieve the original code of samples packed with these techniques.

UPX Variations. Vanilla UPX and its variants are by far the most prevalent form of packing in our dataset. As shown in Table 3.8, out of 380 packed binaries only three did not belong to this category. The table also highlights the modifications made to the UPX format with the goal of breaking the standard UPX unpacking tool. This includes a modification to the magic number (so that the file does not appear to be packed with UPX anymore), the modification of UPX strings, and the insertion of junk bytes (to break the UPX utility). However, all these samples share the same underlying packing structure and the same compression algorithm—showing that malware writers simply applied "cosmetic" variations to the open source UPX code.

Custom packers. Linux does not count on a large variety of publicly available packers and UPX is usually the main choice. However, we detected

Table 3.9: Top ten common shell commands executed

Shell command	Samples	Percentage
sh	400	5.13%
sed	243	3.12%
cp	223	2.86%
rm	216	2.77%
grep	214	2.75%
ps	131	1.68%
insmod	124	1.59%
chmod	113	1.45%
cat	93	1.19%
iptables	84	1.08%

three samples (all belonging to the *Mumblehard* family) that implemented some form of custom packing, where a single unpacking routine is executed before transferring the control to the unpacked program [M.L]. In one case, the malware started a separate process running a *perl* interpreter and then used the main process to decrypt instructions and feed them into the interpreter.

3.5.6 Process Interaction

This section covers the techniques used by Linux malware to interact with child processes or other binaries already installed or running in the system.

Multiple Processes. 25% of our samples consists of a single process, 9% spawn a new process, 43% involves three processes in total (largely due to the "double-fork" pattern used to daemonize a program), while the remaining 23% created a higher number of separate processes (up to 1684).

Among the samples that spawn multiple processes we find many popular botnets such as *Gafgyt*, *Tsunami*, *Mirai*, and *XorDDos*. For instance, *Gafgyt* creates a new process for every attempt to connect to its command and control (C&C) server. *XorDDos*, instead, creates parallel DDos attack processes.

Shell Commands. 13% of the samples we analyzed inside our sandbox executed at least one external shell command. In total, we registered the execution of 93 unique command-line tools—the most prevalent of which are summarized in Table 3.9. Commands such as *sed*, *cp*, and *chmod* are

often executed to achieve persistence on the target system, while *rm* is used to unlink the sample itself or to delete the *bash* history file. Several malware families also try to kill previous infections of the same malware. *Hijami*, the counter-malware to "vaccinate" *Mirai*, uses *iptables* to close and open network ports, while *Mirai* tries to close vulnerable ports already used to infect the system.

Process Injection An attacker may want to inject new code into a running process to change its behavior, make the sample more difficult to debug, or to hook interesting functions in order to steal information.

Our system monitors three different techniques a process can use to write to the memory of another program: 1) a `ptrace` syscall that requests the `PTRACE_POKETEXT`, `PTRACE_POKEDATA`, or `PTRACE_POKEUSER` functionalities; 2) a `PTRACE_ATTACH` request followed by read/write operations to `/proc/<TARGET_PID>/mem`; and 3) an invocation to the `process_vm_writev` system call.

It is important to mention that the Linux kernel has been hardened against `ptrace` calls in 2010. Since then it is not possible to use `ptrace` on processes that are not direct descendant of the tracer process, unless the unprivileged user is granted the `CAP_SYS_PTRACE` capability. The same capability is required to execute the `process_vm_writev` call, a new system call introduced in 2012 with kernel 3.2 to directly transfer data between the address spaces of two processes.

We found a sample performing injection by using the first technique mentioned above. It injects a dynamic library in every active process that uses *libc* (but excludes `gnome-session`, `dbus` and `pulseaudio`). In the injected payload the malware uses the *libc* function `___libc_dlopen_mode` to load dynamic objects at run-time. This function is similar to the well-known `dlopen`, which is less preferable because implemented in *libdl*, not already included in the *libc*. After the new code is mapped in memory, the malware issues `ptrace` requests to backup the registers values of the victim process, hijack the control flow to execute its malicious behavior, and restore the original execution context.

3.5.7 Information Gathering

Information gathering is an important step of malware execution as the collected information can be used to detect the presence of a sandbox, or to control the execution of the sample. Data stored on the system can also be exfiltrated to a remote location, as it often happens with programs controlled by a C&C server. In this section we look at which portions of

Table 3.10: Top ten Proc file system accesses by malicious samples

Path	Samples	Percentage
/proc/net/route	3368	43.22%
/proc/filesystems	649	8.33%
/proc/stat	515	6.61%
/proc/net/tcp	498	6.39%
/proc/meminfo	354	4.54%
/proc/net/dev	346	4.44%
/proc/<PID>/stat	320	4.11%
/proc/cmdline	278	3.57%
/proc/<PID>/cmdline	259	3.32%
/proc/cpuinfo	226	2.90%

Table 3.11: Top ten Sysfs file system accesses by malicious samples

Path	Samples	Percentage
/sys/devices/system/cpu/online	338	4.34%
/sys/devices/system/node/node0/meminfo	26	0.33%
/sys/module/x_tables/initstate	22	0.28%
/sys/module/ip_tables/initstate	22	0.28%
/sys/class/dmi/id/sys_vendor	18	0.23%
/sys/class/dmi/id/product_name	18	0.23%
/sys/class/net/<interface>tx_queue_len	9	0.12%
/sys/firmware/efi/systab	3	0.04%
/sys/devices/pci0000:00/<device>	3	0.04%
/sys/bus/usb/devices/<device>	2	0.03%

Table 3.12: Top ten accesses on `/etc/` by malicious samples

Path	Samples	Percentage
/etc/rc.d/rc.local	1393	17.88%
/etc/rc.conf	1236	15.86%
/etc/resolv.conf	641	8.23%
/etc/nsswitch.conf	453	5.81%
/etc/hosts	423	5.43%
/etc/passwd	244	3.13%
/etc/host.conf	201	2.58%
/etc/rc.local	170	2.18%
/etc/localtime	165	2.12%
/etc/cron.deny	101	1.30%

the file system are inspected by malware and discuss security-relevant paths analysts should monitor when inspecting new malware strains.

***Proc* and *Sysfs* File Systems.** The *proc* and *sysfs* virtual file systems contain, respectively, runtime system information on processes, system and hardware configurations, and information on the kernel subsystems, hardware devices, and kernel drivers. We divide the type of information collected by malware samples in three macro categories: system configuration, processes information, and network configuration. The network category is the most common in our dataset with more than 3000 samples, as shown in Table 3.10, which accessed `/proc/net/route` (system routing table) to get the list of active network interfaces with their relative configuration. Additional information is extracted from `/proc/net/tcp` (active TCP sockets) and `/proc/net/dev` (sent and received packets). Moreover, 111 samples in our dataset read `/proc/net/arp` to retrieve the system ARP table. For the `sysfs` counterpart, reported in Table 3.11, we found accesses to `/sys/class/net/` to get the transmission queue length, a relevant information for DDoS attacks.

The system configuration category is the second most common, with hundreds of samples that extracted the amount of installed memory, the number of available CPU cores, and other CPU characteristics. The files used for sandbox detection and evasion also fall into this category (see Chapter 3.5.8) as well as the lists of USB and PCI connected devices. This category also includes accesses to `/proc/cmdline` to retrieve the name of the running kernel image.

Table 3.13: ELF programs showing evasive features

Type of evasion	Samples	Percentage
Sandbox detection	19	0.24%
Processes enumeration *	259	3.32%
Anti-debugging	63	0.81%
Anti-execution	3	0.04%
Stalling code	0	-

* Not used for evasion but candidate behavior

Another common type of information gathering focuses on processes enumeration. This is used to prevent multiple executions of the same malware (e.g., by the *Mirai* family), or to identify other relevant programs running on the target machine. As reported in Table 3.9, we found 131 samples executing the shell command `ps`, used as a fast interface to get the list of running processes. For example, 67 samples of the *BitcoinMiner* family invoke `ps` and then try to kill other crypto-miner processes that may interfere with their malicious activity.

Configuration Files. System configuration files are contained in the `/etc/` folder. As reported in Table 3.12, configuration files required to achieve persistence are the ones accessed more often. Network-related configuration files also appear to be popular, with `/etc/resolv.conf` (the DNS resolver) or `/etc/hosts` (the mapping between hosts and IP addresses). Among the top entries we also find `/etc/passwd` (list of registered accounts). For instance, *Flooder* samples use it to check for the presence of a backdoor account on the system. If not found, they add a new user by directly writing to `/etc/passwd` and `/etc/shadow`.

3.5.8 Evasion

The purpose of evasion is to hide the malicious behavior and remain undetected as long as possible. This typically requires the sample to detect the presence of analysis tools, or to distinguish whether it is running within an analysis environment or on a real target device. We now present more details about the different evasion techniques, whose prevalence in our dataset is summarized in Table 3.13.

Sandbox Detection. Our string comparison instrumentation detected a number of programs that attempted to detect the presence of a sandbox by comparing different pieces of information extracted from the system with

Table 3.14: File system paths leading to sandbox detection

Path	Detected Environments	#
/sys/class/dmi/id/product_name	VMware/VirtualBox	18
/sys/class/dmi/id/sys_vendor	QEMU	18
/proc/cpuinfo	CPU model/hypervisor flag	1
/proc/sysinfo	KVM	1
/proc/scsi/scsi	VMware/VirtualBox	1
/proc/vz and /proc/bc	OpenVZ container	1
/proc/xen/capabilities	XEN hypervisor	1
/proc/<PID>/mountinfo	chroot jail	1

strings such as "VMware" or "QEMU." Table 3.14 reports the files where the information was collected. Ten samples who tested the `sys_vendor` file were able to detect our analysis environment when executed with root privileges (as we restricted the permissions to files exposing the motherboard DMI zone information reported by the kernel). We also identified samples attempting to detect `chroot()`-based jails (by comparing `/proc/1/mountinfo` with `/proc/<malware PID>/mountinfo`), OpenVZ containers [ope], and even one binary (from the *Handofthief* family) trying to evade IBM mainframes and IBM's virtualization technology. It is also interesting to note how some samples simply decide to exit when they detect they are running in a virtual environment, while other adopt a more aggressive (but less stealthy) approach, such as trying to delete the entire file system.

Processes Enumeration. It is common in Windows to evade analysis by verifying the presence of a particular set of processes, or inspecting the goodness and authenticity of companion processes that live on the system. We investigated whether Linux malware samples already employ similar techniques and found 259 samples that perform a full scan of the `/proc/<PID>` directories. However, none of the samples appeared to perform these scans for evasive purposes but instead to test if the machine was already infected or to identify target processes to kill (as we explain in Chapter 3.5.6).

Anti-Debugging. The most common anti-debugging technique is based on the `ptrace` system call that provides to debuggers the ability to "attach" to a target process to programmatically inspect and interact with it. As a given process can only have at most *one* debugger attached to it, one common evasion technique used by malware consists of invoking the `ptrace` system call with flags `PTRACE_TRACEME` or `PTRACE_ATTACH` on themselves to detect if another debugger is already attached or prevent it

to do so while the sample is running. We found 63 samples employing this mechanism. We also identified one sample checking the presence of the LD_PRELOAD environment variable, which is often used to override functions in dynamically loaded libraries (with the goal of dynamically instrumenting their execution).

It is important to note that the tracing system we use in our sandbox is based on kernel probes (as described in Chapter 3.3.4), and it cannot be detected or tampered with by using anti-debugging techniques.

Anti-Execution. Our experiments detected samples belonging to the *DnsAmp* malware family that did not manifest any behavior, except from comparing their own file name with a hardcoded string. A closer look at these samples showed that the malware authors used this trick as an evasive solution, as many malware collection infrastructures and analysis sandboxes often rename the files before their analysis.

Stalling Code. Windows malware is known to often employ *stalling code* that, as the name suggests, is a technique used to delay the execution of the malicious behavior – assuming an analysis sandbox would only run each sample for few minutes. We investigated whether Linux malware is already using simple variants of this technique by scanning our execution traces for samples using time- or sleep-related functions. We found that 64% of the binaries we analyzed make use of the `nanosleep` system call, with values ranging from less than a second to higher than three hours. However, none of them appear to use these delays to stall their execution (in fact, our traces contained clear signs of their behavior), but rather to coordinate child processes or network communications.

3.5.9 Libraries

There are two main ways an executable can make use of libraries. In the first (and more common) case, the executable is *dynamically linked* and external libraries are loaded at run-time, permitting code reuse and localized upgrades. Conversely, an executable that is *statically linked* includes the object files of its libraries as part of its executable file—removing any external dependency of the application and thus making it more portable.

More than 80% of the samples we analyzed are statically linked. Nevertheless, we note that only 24% of these samples have been stripped from their symbols, with the remaining ones often including even functions and variables names used by developers. Similarly for dynamically linked samples in our dataset, only 33% of them are stripped. We find this trend very interesting as apparently malware developers lack motivation to obfuscate

Table 3.15: Top 20 libraries included by dynamically linked executables

Library	Percentage	Library	Percentage
glibc	74.21%	libscotch	1.23%
uclibc	24.24%	libtinfo	0.75%
libgcc	9.74%	libgmp	0.75%
libstdc++	7.12%	libmicrohttpd	0.64%
libz	5.24%	libkrb5	0.64%
libcurl	3.64%	libcomerr	0.64%
libssl	2.35%	libperl	0.59%
libxml2	1.44%	libhwloc	0.59%
libjansson	1.39%	libedit	0.54%
libncurses	1.28%	libopencl	0.54%

their code against manual analysis—which is in sharp contrast with the complexity of evasive Windows malware.

Common Libraries. Table 3.15 lists the dynamic libraries that are most often imported by malware samples in our dataset. This lists shows two important aspects. First, that while the GNU C library (*glibc*) is (expectedly) the most requested library, we found that 24% of samples link against smaller implementations like *uClibc*, often used in embedded systems. It is also interesting to see how almost 10% of the dataset links against *libgcc*, a library used by the GCC compiler to handle arithmetic operations that the target processor cannot perform directly (e.g., floating-point and fixed-point operations). This library is rarely used in the context of desktop environments, but it is often used in embedded devices with architectures that do not support floating point operations. The second interesting aspect is that, while in total we identified more than 200 different libraries, the distribution has a very long tail and it drops very steeply. For instance, the tenth most popular library is only used by 1% of the samples.

3.6 Intra-family variety

In the previous section we described several characteristics of Linux-based malware. For each of them, we presented the number of samples instead of the count of families that exhibited a given trait. This is because we noted that samples belonging to the same family often had very different characteristics, probably due to the availability of the source codes for several classes of Linux malware.

As an example of this variety, we want to discuss the case of a popular malware family, *Tsunami*, for which we have 743 samples in our dataset. Those samples are compiled for nine different architectures, the most common being *x86-64*, and the rarest being *Hitachi SuperH*. In total, 86% of them are statically linked and 13% are stripped. Dynamically linked *Tsunami* samples rely on different loaders, and their entropy varies from 1.85 to 7.99. Out of the 19 samples with higher entropy, one is packed with vanilla UPX while the other 18 use modified versions of the same algorithm.

This variability is not limited to static features. For instance, looking at our dynamic traces we noted the use of different persistence techniques with some samples only relying on user-level approached and other using run-level scripts or *cron* jobs for system-wide persistence. Concerning unprivileged and privileged execution, only 15% of the *Tsunami* samples we analyzed in our sandboxes tested the user privileges or got privileges-related errors. Differences arise even in terms of evasion: 17 samples contain code to evade the sandbox while all the others did not include evasive functionalities.

3.7 Conclusions

This chapter presents the first comprehensive study of Linux-based malware. We document the design and implementation of the first analysis pipeline specifically tailored for Linux malware, and we discuss the results of the first large-scale empirical study on *how* Linux malware implements its malicious behavior. While the complexity of current Linux malware is not very high, we have identified a number of samples already adopting techniques borrowed from their Windows counterparts. We believe these insights can be the foundation for more systematic future works in the area, which is, unfortunately, bound to have an ever-increasing importance.

Chapter 4

The Tangled Genealogy of IoT Malware

4.1 Introduction

Little is still known about the dynamics behind the emergence of new malware strains and today IoT malware is still classified based on the labels assigned by AV vendors. Unfortunately, these labels are often very coarse-grained, and therefore unable to capture the continuous evolution and code sharing that characterize IoT malware. For instance, it is still unclear how many variants of the *Mirai* botnet have been observed in the wild, and what makes each group different from the others. We also have a poor understanding of the inner relationships that link together popular families, such as the *Mirai* and *Gafgyt* botnets and the infamous *VPNFilter* malware.

This chapter aims at filling this gap by proposing a systematic way to compare IoT malware samples and display their evolution in a set of easy-to-understand lineage graphs. While there exists a large corpus of works that focused on the clustering of traditional malware [BOA+07, BCH+09, PLF10, KV15, JBV11] and exploring their lineage [JWB13, MXW15, KWLP05, DN11, LDFM+12, HYD17] proving the complexity of these problems, in this chapter we show that the peculiarities of IoT malware require the adoption of customized techniques. On the other hand, to our advantage, the current number of samples and the general lack of code obfuscation make possible, for the first time, to draw a complete picture that covers the entire ecosystem of IoT malware.

Our main contribution is twofold. First, we present an approach to reconstruct the lineage of IoT malware families and track their evolution. This technique allows identifying various variants of each family and also the intra-family relationships that occur due to the code-reuse among them. Second, we report on the insights gained by applying our approach on the largest dataset of IoT malware ever assembled to date, which include all malicious samples collected by VirusTotal between January 2015 and August 2018[1].

Our lineage graphs enabled us to quickly discover over a hundred mislabeled samples and to assign the proper name to those for which AV products did not reach a consensus. Overall, we identified and validated over 200 variants in the top families alone, we show the speed at which new variants were released, and we measured for how long new samples belonging to each variant appeared on VirusTotal. By looking at changes in the functions, we also identify a constant evolution of thousands of small variations within each malware variant. Finally, our experiments also emphasize how

[1] As explained in Chapter 4.2, we included in our analysis only samples detected as malicious by at least five AV systems.

the frequent code reuse and the tangled relationship among all IoT families complicate the problems of assigning a name to a given sample, and to clearly separate the end of a family and the beginning of another.

We make the full dataset and the raw results available to researchers [2]. We also share the high resolution figures of the lineage graphs made by architecture for ease of exploration.

4.1.1 Why this Study Matters

IoT malware is an important emerging phenomenon [PSY+15], not just because of its recent development but also because IoT devices might not be able to run anti-malware solutions comparable to those we use today to protect desktop computers. However, to be able to design new solutions, it is important for the security community to precisely understand the characteristics of the current threat landscape. This need prompted researchers to conduct several measurement studies, focused for instance on the impact of the *Mirai* botnet [AAB+17] or on the techniques used by Linux-based malicious samples [CGFB18].

This work follows the same direction, but it is over one order of magnitude larger than previous studies and includes *all* malicious samples submitted to VirusTotal over a period of 3.5 years. A consequence of the scale of the measurement is that the manual analysis used in previous studies had to be replaced with fully automated comparison and clustering techniques.

Our findings are not just curiosities, but carry important consequences for future research in this field. For example, static analysis was the preferred choice for program analysis, until researchers showed that the widespread use of packing and obfuscation made it unsuitable in the malware domain [MKK07]. Our work shows that this is not yet the case in the IoT space, and that today static code analysis provides more accurate results than looking at dynamic sandbox reports or static features. The fragmentation of IoT families also casts some doubts on the ability of AV labels to characterize the complex and tangled evolution of IoT samples.

Finally, while not our main contribution, our work also reports on the largest clustering experiments conducted to date on dynamic features extracted from malicious samples [BOA+07, BCH+09, JBV11].

[2] Dataset and figures: *https://github.com/eurecom-s3/tangled_iot/*

4.2 Dataset

To study the genealogy of IoT malware, our first goal was to collect a large and representative dataset of malware samples. For this purpose, we downloaded all ELF binaries that have been submitted to VirusTotal [urlb] over a period of almost four years (from January 2015 to August 2018) and that had been flagged as malicious by at least five anti-virus (AV) vendors. Since our goal is to analyze malware that targets IoT devices, we purposely discarded all Android applications and shared libraries. Furthermore, we also removed samples compiled for the Intel and AMD architectures because it is very difficult to distinguish the binaries for embedded devices from the binaries for Linux desktop systems. This selection criteria resulted in a dataset of 10,548 samples, one order of magnitude larger than any other study conducted on Linux-based malware. As a comparison, the largest measurement study to date was performed on 10,548 Linux binaries [CGFB18], of which a considerable fraction (64.56%) were malware targeting x86 desktop computers. Moreover the purpose of this dataset was to study the general behavior of modern Linux malware and not the tangled relationships between them.

We could have easily extended our dataset to Linux malware for desktops and servers. On the other hand, we preferred to focus specifically on IoT malware, given their high infection rate on real devices and the variety of the underlying hardware architectures. This possibly requires platform customizations implemented as ad-hoc malware variants. Moreover, less known architectures are more likely to show those small bits which tend to be ignored on more comfortable and extensively studied counterparts e.g., x86.

Figure 4.1 shows the volume of samples in our dataset submitted to VirusTotal over the data collection period and the dramatic increase in the number of IoT malware samples after the outbreak of the infamous *Mirai* botnet in October 2016. Before that, the number of malicious IoT binaries was very low. For instance, only 363 of our 93K samples were observed in that period. This number progressively increased to reach an average of 7.8k new malicious binaries per month in 2018. This trend can be attributed to several factors, including the evolving IoT threat landscape [Sym18, Sym19, VS18, JMM18], the source code availability of several popular families [JMM18], and the proliferation of IoT honeypots that allowed researchers to rapidly collect a large number of samples spreading in the wild [VS18].

Table 4.1 reports the compilation details of the samples in our dataset.

Table 4.1: Breakdown of samples per architecture.

CPU Architecture	Samples No. (%)	Dynamically Linked			Statically Linked		
		Stripped	Unstripped	Total	Stripped	Unstripped	Total
ARM 32-bit	36,574 (39.05)	3,012	645	3,657	16,049	16,868	32,917
MIPS I	25,201 (26.91)	325	345	670	12,714	11,817	24,531
PowerPC 32-bit	10,916 (11.66)	100	258	358	5,180	5,378	10,558
SPARC	8,412 (8.98)	100	119	219	3,489	4,704	8,193
Hitachi SH	6,477 (6.92)	63	107	170	2,190	4,117	6,307
Motorola 68000	5,982 (6.39)	52	82	134	2,130	3,718	5,848
Tilera TILE-Gx	27 (0.03)	0	1	1	26	0	26
ARC International ARCompact	27 (0.03)	16	2	18	7	2	9
Interim Value tba	9 (0.01)	2	7	9	0	0	0
SPARC Version 9	8 (0.01)	1	6	7	1	0	1
PowerPC 64-bit	6 (0.01)	1	4	5	1	0	1
Others	13 (0.01)	4	8	12	1	0	1
Total	93,652	3,676	1,584	5,260	41,788	46,604	88,392

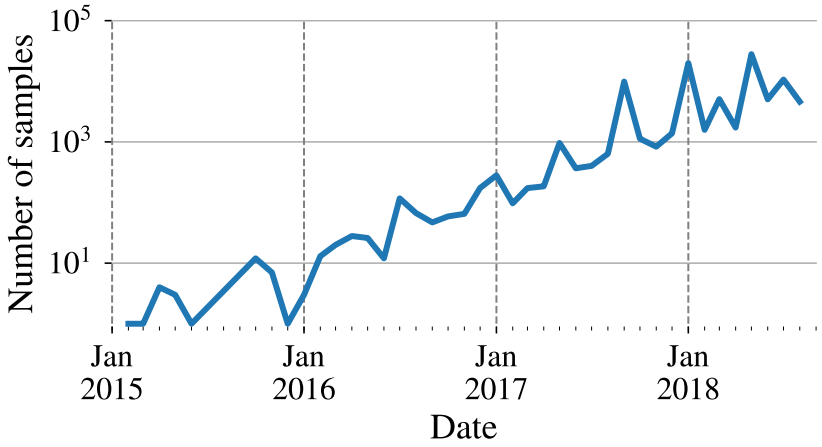


Figure 4.1: Number of samples in our dataset submitted to VirusTotal over time.

The first two architectures, *ARM 32-bit* and *MIPS I*, account together for two thirds of all samples. This can be explained by the large popularity of these processor architectures for popular consumer IoT devices commonly targeted by these malware, such as home routers, IP cameras, printers, and NAS devices. Another interesting aspect is the fact that almost 95% of the ELF files in our dataset were statically linked. Additionally, as already noted by Cozzi et al. [CGFB18], a large fraction of them (roughly 50% in our dataset) have not been stripped from their symbols.

In addition to downloading the binaries, we also retrieved the VirusTotal reports. We then processed them with AVClass [SRKC16b], a state-of-the-art technique that relies on the consensus among the AV vendors to determine the most likely family name attributed to malware samples. Table 4.2 lists the top ten AVClass labels, with *Gafgyt* and *Mirai* largely dominating the dataset. However, there is a long tail of families (90 in total) that contain only a small number of samples. Finally, it is interesting to note that AVClass was unable to find a consensus for a common family name for only 3.7K samples. While this might seem very small (especially compared with figures obtained on Windows malware), if we remove *Mirai* and *Gafgyt*, a common label was not found for one third of the remaining samples.

Table 4.2: Breakdown of the top 10 IoT malware families in our dataset.

Rank	Label (AVClass)	Samples No. (%)
1	Gafgyt	46,844 (50.02)
2	Mirai	33,480 (35.75)
3	Tsunami	3,364 (3.97)
4	Dnsamp	2,235 (3.59)
5	Hajime	1,685 (2.39)
6	Ddostf	840 (0.90)
7	Lightaidra	360 (0.38)
8	Pnscan	212 (0.23)
9	Skeeyah	178 (0.19)
10	VPNFilter	135 (0.14)
	Total	89,935 (96.03)
	Unlabelled	3,717 (3.97)

4.3 Features-based Clustering

The field that studies the evolution of malware families and the way malware authors reuse code between families as well as between variants of the same family is known as *malware lineage*. Deriving an accurate lineage is a difficult task, which in previous studies has often been performed with help from manual analysis and over a small limited number of samples [LDFM+12, HYD17]. However, given the scale of our dataset, we need to rely on a fully automated solution. The traditional approach for this purpose is to perform malware clustering based on static and dynamic features [DN11, KWLP05, JBV11, LDFM+12]. When also the time dimension is combined in the analysis, clustering can help to derive a complete timeline of malware evolution, also known as *phylogenetic tree* of the malware families. However, studying malware lineage based on AV labels is prone to errors (due to AV errors or inconsistencies among labels) and it can only provide a very coarse-grained clustering as AV labels tend to only identify macro-families (e.g., all Mirai samples). We, on the other hand, are interested in a more fine-grained classification that would allow us to study the differences among different sub-families and the overall intra-family evolution and relationship. Therefore, by building upon previous work in this area, in our first attempt we decided to cluster samples based on a broad set of both static and dynamic features that capture different aspects of malware samples.

Despite our multiple attempts, while this approach provided interest-

ing results, it also always resulted in a large number of noisy clusters – therefore requiring a substantial amount of manual adjustments and validation. While this could be done as a one-time effort, our goal was to obtain an automated solution that could be continuously run as new samples get collected. Thus, as explained in Chapter 4.4, we eventually adopted a different solution to perform our clustering process. However, as features-based clustering is often used in malware studies, we believe there is a value in reporting the results of this attempt and discuss the reasons behind its failure.

4.3.1 Feature Extraction

To analyze each sample, we leverage a free ELF binary analysis service[3] based on a recent work [CGFB18]. The service relies on a combination of static and dynamic analyses to comprehensively evaluate ELF binaries. It provides runtime behavioral reports via its multi-architecture sandboxing environment, from which we extract 146 features that belong to five groups. We refer the reader to Table 4.4 for the complete list of extracted features.

1. **ELF and byte-level** features capture low-level characteristics of the binary, such as its architecture, whether it is statically or dynamically linked, stripped or unstripped, the number of ELF sections, its file size, the entropy of each section and its most common bytes, etc.

2. **Binary disassembly** features report numerical statistics extracted with IDA Pro, such as the number of functions, their complexity, the number of instructions, etc.

3. **Strings** includes printable strings extracted from the binary, grouped into IP addresses, URLs, and UNIX paths.

4. **Runtime behavior** covers the information extracted from the execution of the binary in a sandbox, including whether the sample was executed correctly, the list of issued system calls, the different files opened, modified or deleted, whether the binary has attempted to achieve persistence on the system, etc.

5. **Network traffic** features provide a detailed breakdown of all network connections observed while the binary was running, as extracted by the Zeek (formally Bro) IDS, including contacted IP addresses, files transferred, domain name resolved, etc.

[3]Padawan: https://padawan.s3.eurecom.fr

4.3.2 Clustering

Our dataset is large and very complex, containing 93K samples and 146 features, several of them categorical. We converted categorical features to numeric ones with the standard *one-hot encoding* technique, whereby each categorical feature becomes a set of n boolean representing whether each item belongs to each of the n categories for that feature. For categorical features, we ended up with a sparse matrix having tens of thousands of columns: such a large dimensionality is generally very problematic in terms of scalability for generic clustering algorithms. To deal with it, we use FISHDBC [Del19], a density-based clustering algorithm designed for scalability for complex datasets and arbitrary/non-metric distance functions. This algorithm outputs hierarchical clustering results in a top-down approach—from the most coarse-grained to the most fine-grained—and allows to identify the level that yields the best classification.

We consider numeric and categorical features for each group separately; for categorical features we pre-process the dataset using *tf-idf* and the *Cosine distance*, while we use the *Euclidean distance* for numerical features. To empirically assess the impact of feature groups, we performed 25 rounds of clustering including different combinations of feature groups, i.e., by including or discarding some of the five categories.

To get a rough estimation of the quality of the clustering we use AV labels as a provisional ground truth. In fact, even if some errors in the label may exist, we still expect to find samples in the same cluster to largely come from the same family. By using the output of AVClass, we flag each cluster as one of four categories: (i) *Pure* if it contains all samples with the same AV label, (ii) *Single* if it contains a combination of samples with the same AV label and unlabelled samples, (iii) *Majority* if more than 90% of samples in the cluster have the same AV label, and (iv) *Mixed* if it does not fit any of the previous categories. Table 4.3 provides a summary of the results of the 25 rounds of clustering. For the sake of conciseness, we only provide the best results obtained per combination of feature groups across all tested weights. Note that the clustering on the IDA Pro features could only be performed on a restricted set of the 4,960 samples dynamically linked samples, to avoid introducing noise in the IDA Pro features due to the large amount of embedded library code. Moreover, the table does not contain results for the network features alone because network features were too sparse and could not be used by themselves to build our hierarchical clusters.

Table 4.3 shows that individual sets successfully identify several groups of samples belonging to the same family (i.e., *pure* clusters), but then also

cluster together many samples that have little or nothing in common (e.g., *mixed* clusters). The results do not improve much by combining all features, as the limitation of each group tends to increase the noise in the overall classification. Out of all combinations we tried in our experiments, the ELF and bytes features alone produced the best clustering results with a total of 44,491 samples in *pure* clusters and only 14,204 samples in *mixed* clusters. However, even in this case roughly one third of our dataset was placed in *majority* clusters which erroneously contained samples of different families.

We then performed an investigation on the resulting clusters produced by the different feature group combinations. Here we wanted to understand whether these clusters could be directly used to group together samples that belong to the same variant or sub-family and, if the answer is affirmative, what exactly was changed between one version and the other. We first looked at the *pure clusters*. We noticed that *all* medium-to-large size malware families were broken down by our system in many *pure clusters*. If we consider the combination that produced the best clustering results, i.e., the ELF and bytes combination, 20,027 *Gafgyt* samples were clustered in 1,071 different pure clusters. Also, as many as 13,391 *Mirai* samples populated a total of 654 pure clusters. Initially, this would make them good candidates for our sub-family investigation. As expected, indeed different clusters often captured different common features of the samples. For example, they separated dynamically vs statically linked binaries, or those samples that successfully executed in our VM from those that did not (and therefore resulted in an empty dynamic behavior profile). However, our goal was not to distinguish *Mirai* samples that were dynamically or statically linked, but rather identify its evolution over time. Unfortunately, the resulting clusters did not capture our need to isolate sub-families but rather samples that produced similar features (e.g., two samples that immediately terminate with an error message are not necessarily similar, despite the common behavior). During the manual investigation of the clustering results, we also noticed that the captured runtime and network behavior of different variants of the same family, when not missing, were often identical or so similar that the clustering algorithm would hardly differentiate them. For example, most variants of Mirai would follow the same high-level process after the device is compromised: (i) reach out to the C&C server, (ii) retrieve some target IP addresses to scan for worm-like replication, (iii) launch scanning, (iv) receive DDoS attack target(s), and (v) launch DDoS attack(s). This hinders the identification of variants from such a trace. Additionally, considering finer-grained features is likely to introduce overly specific clusters.

We also manually investigated those clusters that contained samples

Table 4.3: Clustering results: static and dynamic features.

Feature groups					Clusters (# samples)			
ELF	IDA Pro	strings	behaviour	network	*pure*	*single*	*majority*	*mixed*
✓					**44,491**	**4,657**	31,649	14,204
	✓				3,677	45	316	1,082
		✓			18,141	3,120	23,412	50,328
			✓		27,889	1,097	5,726	60,289
✓	✓	✓	✓	✓	34,313	2,337	12,741	45,610
✓	✓	✓	✓		38,825	3,062	24,234	27,531
✓	✓	✓			39,904	2,495	17,667	33,586
✓	✓				42,427	2,587	**34,118**	14,520
		✓	✓	✓	20,822	983	12,964	58,883

with different AV labels. In particular, we looked at those that had a predominant number of samples with a consistent AV label, and a small number of samples with a different one (*majority clusters*), e.g., `(gafgyt:33)`, `(aidra:2)`. While intuitively this could have been the result of errors in AV classifications, after dozens of manual investigations we could not find a single mis-labeled sample. Please remember that this does not mean there were no errors in individual AV labels (we did find several of those), but that by applying the majority voting provided by AVclass the result (when a consensus was reached) was always correct. Errors in the majority voting also existed, as explained in more details in Chapter 4.5.2, but we needed a more precise clustering to successfully isolate them from the noise.

4.3.3 Lessons Learned

After several weeks of experimentation, by adding and removing individual features and manually investigating why different samples always ended up clustered together (sometimes even by pure chance, like when samples had the same file size, same sections entropy, and many strings in common due to some persistence mechanisms), we had to conclude that traditional clustering based on static and dynamic features did not satisfy our needs. In particular, when applied to a large dataset, the number of errors largely exceeded the ability to manually investigate and correct the results.

It was also interesting to note that features related to runtime behavior and network traffic tended to introduce more noise in the clustering and failed to accurately classify samples even into coarse-grained malware fam-

ilies. On the other hand, we observed that ELF and bytes features would produce very compact micro clusters sensitive to very fine-grained changes in the binary representation of malware samples. While this was more successful to group together samples belonging to the same family, such oversensitive classification turned out to be inappropriate for the identification of malware variants. Given that the notion of a variant varies between one family and another, it is difficult to transfer such a fuzzy concept into a clustering algorithm. We therefore concluded that if we wanted to reliably identify variants of malware families we had to perform a deeper investigation by looking at their code.

Table 4.4: List of features used for static and dynamic clustering.

Feature name: **Description**
bytes.common_bytes: List of the three most common bytes (with counter)
bytes.entropy: The entropy of the binary
bytes.header: First 16 bytes of the file
bytes.footer: Last 16 bytes of the file
bytes.longest_sequence.length: Longest sequence of the same byte (byte, offset, length)
bytes.min_entropy: Lowest entropy among 16K bytes blocks
bytes.max_entropy: Highest entropy among 16K bytes blocks
bytes.null_bytes: Number of null (0) bytes
bytes.printable: Number of printable bytes
bytes.rarest_bytes: List of the three rarest bytes (with counter)
bytes.unique_bytes: Number of unique bytes (0-255)
bytes.white_spaces: Number of white-spaces (0x32,\n,\r,\t) bytes
elf.anomalies.ehph_diff: Difference between segment virtual address and file offset
elf.anomalies.entrypoint.permission: Anomalous entrypoint: Permission
elf.anomalies.entrypoint.section: Anomalous entrypoint: Section
elf.anomalies.entrypoint.segment: Anomalous entrypoint: Segment
elf.anomalies.sections.cpp_prelink: Anomalous sections: C++ prelink section
elf.anomalies.sections.grub_module: Anomalous sections: Grub module
elf.anomalies.sections.headers: Anomalous sections: Wrong number of section headers
elf.anomalies.sections.high_entropy: Anomalous sections: High entropy
elf.anomalies.sections.kernel_object: Anomalous sections: Kernel object
elf.anomalies.sections.section_header_null: Anomalous sections: Null section headers
elf.anomalies.sections.shentsize_empty: Size of section header table's entry null

elf.anomalies.sections.shnum_empty: Anomalous sections: Number of section headers empty
elf.anomalies.sections.shnum_pastfile: Anomalous sections: Section header table beyond file
elf.anomalies.sections.shoff_empty: Anomalous sections: Section header table offset empty
elf.anomalies.sections.shoff_pastfile: Anom. sec.: Section header table offset beyond file
elf.anomalies.sections.uncommon: Anomalous sections: Uncommon sections
elf.anomalies.sections.wrong_shstrndx: Anom. sec.: Wrong section name string table index
elf.anomalies.segments.error: Error in segments table
elf.anomalies.segments.headers: Anomalous segments: Wrong number of program headers
elf.anomalies.segments.high_entropy: Anomalous segments: High entropy
elf.anomalies.segments.high_mem: Segment memory size much bigger than physical size
elf.anomalies.segments.wx: Anomalous segments: `W&X` permission
elf.class: ELF file's class
elf.comment: .comment section of the ELF, if present
elf.data: Data encoding of the-specific data
elf.debug: If the binary contains debug information (compiled with -g)
elf.dynfuncs: Dynamic symbols being used, of type FUNC in particular
elf.entrypoint: Binary entrypoint
elf.e_phentsize: Size in bytes of one entry in the program header table
elf.e_phnum: Number of entries in the program header table
elf.e_phoff: Program header table's file offset in bytes
elf.e_shentsize: Size in bytes of one entry in the section header table
elf.e_shnum: Number of entries in the section header table
elf.e_shoff: Section header table's file offset in bytes
elf.e_shstrndx: Index of section header table containing section names
elf.gdb: Error raised by gdb
elf.interpreter: ELF's declared interpreter
elf.link: Statically or dynamically linked
elf.machine: Required architecture for the file
elf.malformed.entrypoint: Malformed ELF: Wrong entrypoint
elf.malformed.pastload: Malformed ELF: Beyond LOAD segment
elf.malformed.pastphnum: Malformed ELF: Beyond program header table
elf.malformed.pastsegment: Malformed ELF: Beyond segment
elf.needed: DT_NEEDED entries for dynamic ELF files
elf.note: .note.* sections of the ELF, if present
elf.nsections: Number of sections
elf.nsegments: Number of segments

elf.osabi: Operating system/ABI identification
elf.pyelftools: Exception raised by pyelftools, if any
elf.readelf: Error raised by readelf
elf.soname: PT_SONAME entry for dynamic ELF files
elf.stripped: Whether the binary has been stripped or not
elf.stripped_sections: Whether the sections table of the binary has been stripped or not
elf.type: Object file type

---

strings.ip: Potential IPs (v4 and v6) found in the binary
strings.path: Potential UNIX paths found in the binary
strings.url: Potential URLs found in the binary

---

idapro.average_bytes_func: Average size in bytes of a function
idapro.avg_basic_blocks: Average number of basic blocks respect to functions
idapro.avg_cyclomatic_complexity: Average cyclomatic complexity respect to functions
idapro.avg_loc: Average lines of code respect to functions
idapro.branch_instr: Number of branch instructions
idapro.bytes_func: Total size in bytes of the functions
idapro.call_instr: Number of call instructions
idapro.func_loc: Percentage of instructions belonging to functions
idapro.indirect_branch_instr: Number of indirect branch instructions
idapro.loc: Explored lines of code
idapro.max_basic_blocks: Max basic blocks
idapro.max_cyclomatic_complexity: Max cyclomatic complexity
idapro.nfuncs: Number of functions detected
idapro.percent_load_covered: Percentage of covered load segment
idapro.percent_text_covered: Percentage of covered text section
idapro.syscall_instr: Number of syscall instructions

---

behavior.user.argv0_rename: Procs renaming argv0
behavior.user.askroot: Wheter the execution got permission related errors
behavior.user.checkgid: If gid is checked
behavior.user.checkuid: If uid is checked
behavior.user.cmds: System cmds
behavior.user.compare: `strcmp` or `memcmp` comparison
behavior.user.cve: Possible CVEs exploited
behavior.user.dropped.create: Dropped files: Create
behavior.user.dropped.link: Dropped files: Link
behavior.user.dropped.linkfrom: Dropped files: Link from
behavior.user.dropped.modify: Dropped files: Modify
behavior.user.empty: Empty or no trace
behavior.user.errors.enosys: Errors from execution: Syscall not implemented
behavior.user.errors.execfault: Errors from execution: Execution fault

behavior.user.errors.illegal: Errors from execution: Illegal instruction
behavior.user.errors.missinglibs: Errors from execution: Missing library
behavior.user.errors.segfault: Errors from execution: Segmentation fault
behavior.user.errors.sigbus: Errors from execution: Bus error
behavior.user.errors.wronginterp: Errors from execution: Wrong interpreter
behavior.user.ioctl.fail: Ioctls: Fail
behavior.user.ioctl.success: Ioctls: Success
behavior.user.ioctl.total_no: Ioctls: Total number
behavior.user.libccalls.total_no: Libc calls from execution: Total number
behavior.user.libccalls.unique: Libc calls from execution: Unique
behavior.user.libccalls.unique_no: Libc calls from execution: Unique number
behavior.user.lineslost: Amount of trace lines not correctly parsed
behavior.user.persistence.create: Sample persistence: Create
behavior.user.persistence.link: Sample persistence: Link
behavior.user.persistence.linkfrom: Sample persistence: Link from
behavior.user.persistence.modify: Sample persistence: Modify
behavior.user.proc_rename: Procs renaming
behavior.user.procs: Number of processes spawned
behavior.user.ptrace_request: Ptrace requests
behavior.user.read_only: Files being read
behavior.user.rooterr.EACCES: EACCES type of permission related error
behavior.user.rooterr.EPERM: EPERM type of permission related error
behavior.user.sleep_max: Max sleep
behavior.user.syscalls.total_no: Syscalls from execution: Total number
behavior.user.syscalls.unique: Syscalls from execution: Unique
behavior.user.syscalls.unique_no: Syscalls from execution: Unique number
behavior.user.unlink: Unlink files
behavior.user.unlink_itself: Unlink itself

---

dynamic.error: Errors encountered during sandboxing
dynamic.stderr: Standard output during analysis
dynamic.stdout: Standard error during analysis

---

nettraffic.conn.avg_duration: Average duration of connections
nettraffic.conn.bytes: Number of bytes exchanged
nettraffic.conn.conns: Number of connections
nettraffic.conn.ips: List of unique IP addresses contacted
nettraffic.conn.pkts: Number of packets exchanged
nettraffic.conn.ports: List of unique destination ports
nettraffic.dns.qry_resp: List of unique DNS queries and their responses
nettraffic.dns.queried_domains: List of unique domains resolved through DNS
nettraffic.files.dropped_files_hash: List of unique hashes (SHA-256) of dropped files

nettraffic.files.dropped_files_mimetype: List of unique MIME types of dropped files
nettraffic.files.dropped_files_source_ips: List of unique IP addresses from which dropped files have been downloaded
nettraffic.files.dropping_protos: List of unique protocols used to drop files
nettraffic.ssl.ssl_domains: List of unique domains contacted over SSL/TLS

4.4 Malware Lineage Graph Extraction

The field that studies the evolution of malware families and the way malware authors reuse code between families as well as between variants of the same family is known as *malware lineage*. Deriving an accurate lineage is a difficult task, which in previous studies has often been performed with help from manual analysis and over a small limited number of samples [LDFM+12, HYD17]. However, given the scale of our dataset, we need to rely on a fully automated solution. The traditional approach for this purpose is to perform malware clustering based on static and dynamic features [DN11, KWLP05, JBV11, LDFM+12]. When also the time dimension is combined in the analysis, clustering can help derive a complete timeline of malware evolution, also known as *phylogenetic tree* of the malware families.

A common and simple other way to do that would be to rely on AV labels, more oriented to only identify macro-families.

We, on the other hand, work towards a finer-grained classification that would enable us to study differences among sub-families and the overall intra-family evolution and relationships.

In our first attempt we decided to cluster samples based on a broad set of both static and dynamic features. This approach not only required a substantial amount of manual adjustments and validation, it also always resulted in noisy clusters.

4.4.1 Code-based Clustering

We decided to resort to a more complex and time consuming solution based on code-level analysis and function similarity. The advantage is that code does not lie, and therefore can be used to precisely track both the evolution over time of a given family as well as the code reuse and functionalities borrowed among different families.

The main drawback of clustering based on code similarity is that the distance among two binaries is difficult to compute. Binary code similarity is still a very active research area [HC19], but tools that can scale to our

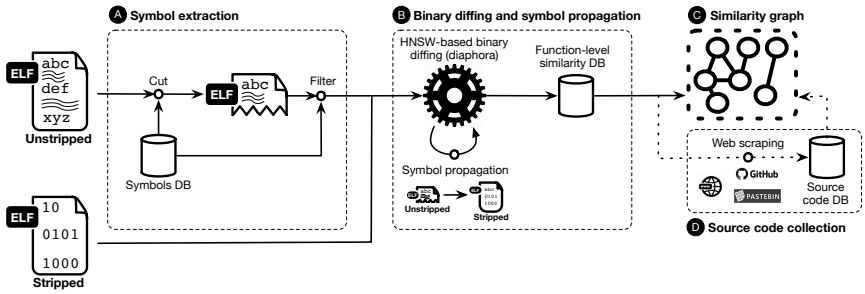


Figure 4.2: The workflow of our system.

dataset size are scarce and often in a prototype form. Moreover, to be able to compare binaries, three important conditions must be satisfied: 1) each sample needs to be first properly unpacked, 2) it must be possible to correctly disassemble its code, and 3) it must be possible to separate the code of the application from the code of its libraries. The first two constitute major problems that had hindered similar experiments on Windows malware. However, IoT malware samples are still largely un-obfuscated and packers are the exception instead of the norm [CGFB18]. While this is a promising start, the third condition turns out to be a difficult issue (ironically this is the only one not causing problems for traditional Windows malware).

Figure 4.2 shows the workflow of our code-based clustering. The process is divided in three macro phases. (A) First we process unstripped binaries and we analyze the symbols to locate library code in statically linked files. (B) Then we perform an incremental clustering based on the code-level similarity, while propagating symbols to each new sample. (C) Finally, we build the family graphs (one for each CPU architecture) and (D) and we use available symbols to pin samples and clusters to code snippets we were able to scrape from online code repositories to obtain more detailed understanding about the evolution of malware families.

Recall that our goal is not to provide a future-proof IoT malware analysis technique. We rather seek to identify a scalable approach that enables us to reconstruct the lineage for the 93K samples in our 3.5 year-long dataset so we can report on their genealogy. We thus take advantage of the current sophistication of IoT malware, which is currently rudimentary enough to enable code-based analysis, aware that malware authors could easily employ tricks to hinder such analysis in the future.

4.4.2 Symbols Extraction

IoT malware is often shipped statically linked. The fact that 88,392 samples out of 10,548 (94.3%) in our dataset are statically linked tend to confirm this assumption. This is most likely due to an effort to ensure the samples can run on devices with various system configurations. However, performing code similarity on statically linked binaries is useless, as two samples would be erroneously considered very similar simply because they might include the same *libc* library. Therefore, to be able to identify the relevant functions in such binaries, we first need to distinguish the user-defined code from the library code embedded in them. Unfortunately, when dealing with stripped binaries, this is still an open problem and the techniques proposed to date have large margins of errors, which are not suitable for our large-scale, unsupervised experiments.

We thus start our analysis by extracting symbols from unstripped binaries and leveraging them to add semantics to the disassembled code. Luckily, as depicted in Table 4.1, 53% of statically-linked and 30% of dynamically linked samples contain symbol information. We used IDA Pro to recognize functions and extract their names. We then use a simple heuristic to cut the binary in two. The idea is to locate some library code, and then simply consider everything that comes after library code as well. While it is possible for the linker to interleave application and library objects in the final executable, this would simply result in discarding part of the malware code from our analysis. However, this is not a common behavior, and lacking any better solution to handle this problem, this is a small price to pay to be able to compute binary similarity on our dataset.

We therefore built a database of symbols (symbols DB in Figure 4.2) extracted from different versions of *Glibc* and *uClibc* and use the database to find a "cut" that separates user from library code. After extracting the function symbols from unstripped ELF samples, we start scanning them linearly with respect to their offsets. We move a sliding window starting from the entry point function `_start` and define a cutting point as soon as all of the function names within that window have a positive match in the symbols DB. Using a window instead of a single function match avoids erroneous cases where a user function name may be wrongly interpreted as a library function. We experimentally set this window size to 2 and verified the reliability of this heuristic by manually analyzing 100 cases. Once the cutting point is identified, all symbols before this point are kept and the remaining ones are discarded.

We chose to operate only on *libc* variants for two reasons. First, because *libc* is always included by default by compilers into the final executable when

producing statically linked files. Moreover, we observed that less than 2% of the dynamically linked samples in our dataset require other libraries on top of *libc*.

Finally, after removing the library code, we further filter out other special symbols, including `___start`, `__start` and architecture-dependent helpers like the `___aeabi_*` functions for ARM processors.

4.4.3 Binary Diffing and Symbol Propagation

Binary diffing constitutes the core of our approach as it enables us to assess the similarity between binaries at the code level. However, given the intrinsic differences at the (assembly) code level between binaries of different architectures, we decided to diff together only binaries compiled for the same architecture – therefore producing a different clustering graph, and a different malware lineage, for each architecture. While this choice largely reduces the number of possible comparisons, our datasets still contains up to 36,574 files per architecture (ARM 32-bit), making the computation of a full similarity matrix unfeasible.

To mitigate this problem we adopt Hierarchical Navigable Small World graphs (HNSW) [MY18], an efficient data structure for approximate nearest neighbor discovery in non-metric spaces, to overcome the time complexity and discover similarities in our dataset. The core idea that accelerates this and similar approaches [DML11, FXWC19] is that items only get compared to neighbors of previously-discovered neighbors, drastically limiting the number of comparisons while still maintaining high accuracy. While adding files to the HNSW, our distance function will be called on a limited number of file pairs (on average, adding an element to the HNSW requires only 244 comparisons in our case) while still being able to link it to its most similar neighbors. We configured the HNSW algorithm to take advantage of parallelism and provide high-quality results as suggested by existing guidelines in the clustering literature [Del19].

We use *Diaphora* [urla] to define our dissimilarity function for HNSW. This function is non-metric as the triangle inequality rule does not necessarily hold. However, in the following we will call it *distance function* without implying it is a proper metric. This has not consequences on the precision of our clustering, as the HNSW algorithm is explicitly designed for non-metric spaces. One of the advantages of using Diaphora is that the tool works with all the architectures supported by IDA Pro, which covers 11 processors and 99.9% of the samples in our dataset, while other binary code similarity solutions recently proposed in academia handle only few architectures and do not provide publicly available implementations [HC19]. When two bina-

ries are compared, Diaphora outputs a per-function similarity score ranging from 0 (no match) to 1 (perfect match). To aggregate individual function scores in a single distance function we experimented with different solutions based on the average, maximum, normalized average, and sum of the scores. We finally decided to count the number of functions with similarity greater than 0.5, which is the threshold suggested by Diaphora's authors to discard *unreliable* results. This has the advantage of providing a symmetric score (e.g., if the similarity of A to B is 4 then the similarity of B to A is also 4) that constantly increase as more and more matching functions are found among two binaries. For HNSW we then report the inverse of this count to translate the value into a distance (where higher values mean two samples are further apart and lower values mean they share more code).

Before running HNSW to perform pairwise comparison on the whole dataset, we unpacked 6,752 packed samples. Since they were all based on variations of UPX, we were able to easily do that by using a simple generic unpacker. We then add each sample to HNSW one by one, in two rounds, sorted by their first seen timestamp on VirusTotal (to simulate the way an analyst would proceed when collecting new samples over time).

In the first round we added all dynamically linked or unstripped samples, which account for 55% of the entire dataset. By relying on the symbols extracted in the previous phase, we only perform the binary diffing on the user-defined portion of the code, and omit comparisons on library code. In the second round we then added the statically linked stripped samples. Being without symbols, there is no direct way to distinguish user functions from library code. Attempts to recover debugging information from stripped binaries, such as with *Debin* [HIT+18], only target a limited set of CPU architectures.

We tackle this problem by leveraging the binary diffing itself to iteratively "propagate" symbols. When a function in a stripped sample has perfect similarity with an unstripped one, we label it with the same symbol. This methodology enables us to perform similarity analysis also for stripped samples, which would otherwise be discarded. However, this step comes with some limitations. While we are able to discard library functions we also potentially discard user functions that didn't match any function already in the graph. For instance, if two stripped statically linked samples share a function that is never observed in unstripped or dynamically linked binaries, this similarity would not be detected by our solution. We add the stripped samples to HNSW only after the unstripped ones have all been added to contain this problem as much as possible, but the probabilistic nature of HNSW can decrease this benefit as not all comparisons are

computed for each sample.

This means that our graph is an under-approximation of the perfect similarity graph (we can miss some edges that would link together different samples, but not create false connections) with over 18.7M one-to-one binary comparisons and 595,039 function symbols propagated from unstripped to stripped binaries.

4.4.4 Source Code Collection

The symbols extracted from unstripped malware and propagated in the similarity phase also helps us locate and collect snippets of source codes from online sources. In fact, the source code for many Linux-based IoT malware families has been leaked on open repositories hosting malicious packages ready to be compiled and deployed. This has resulted in a very active community of developers that cloned, reused, adapted, and often re-shared variations of existing code.

We took advantage of this to recognize open source and closed source families, split our dataset accordingly and, more importantly, to assign labels to groups of nodes in the similarity graph. While we also use AV labels for this purpose, those labels often correspond to generic family names, while online sources can help disambiguate specific variants within the same bigger family.

To locate examples of source code, we queried search engines with the list of user-defined function symbols extracted in the previous phase. We were able to find several matches on public services as GitHub or Pastebin, both for entire code bases (e.g., on GitHub) and for single source files (e.g., on Pastebin). Interestingly, on GitHub we found tens of repositories forked thousands of times (not necessarily for malicious purposes, as often security researchers also forked those repositories). Moreover, we found a Russian website hosting a repository regularly populated with several malware projects, exploits, and cross-compilation resources. From this source alone we were able to retrieve the code of 76 variants of *Gafgyt*, 50 variants of *Mirai*, 19 projects generically referred as "CnC Botnet" and "IRC Sources" (which resemble *Tsunami* variants) and a number of exploits for widely deployed router brands. Some variants contained changelog information that made us believe these projects had been collected from leaks and underground forums.

4.4.5 Phylogenetic Tree of IoT Malware

As a preamble to the function level similarity analysis of IoT malware we post-processed the sparse similarity graph G obtained by running HNSW and using the distance function as weight. Since we store in a database the detailed comparisons, the actual weight on the similarity graph can be tuned depending on the purpose of the analysis.

For instance, the analyst can use only best matches if the goal is to highlight perfect similarity (e.g., code reused as is) between two binaries, or a combination of best and partial matches if we want to capture more generic dependencies between two binaries, including minor variations and "evolutions" of the code.

Another problem with the similarity graph is that it contains a large number of edges, with many samples being variations (or simple recompilation) of the same family. Therefore, to make the output more readable and better emphasize the evolution lines, in our graphs we visualize the Minimum Spanning Tree (*MST*) G' of G that shows the path of minimum binary difference among all samples. This approach to cluster binaries is inspired by the works in clustering literature that are based on the minimum spanning tree (MST) of the pairwise distance matrix between elements [ABKY88, CMS13].

Furthermore, we observed that MSTs—which are in general used as an intermediate representation of the clustering structure—faithfully convey information about the relationships between items in our dataset which is not always preserved when converting the MST to a set of clusters. For this reason, we base our analysis on minimum spanning trees.

The tree can be further colored according to AV labels (to get an overview of the relationships among different families and spot erroneous labels assigned by AV engines) or to the closest source file we downloaded using the symbol names (thus leading to a more clear picture of the genealogy of a single malware family). In the next sections we will explore these two views and present a number of examples of the main findings.

4.5 Results

We used the workflow for code-based clustering presented in the previous section to plot phylogenetic trees for the six top architectures in our dataset. We found that the current IoT malware scene is mainly invaded by three families tightly connected to each other: *Gafgyt*, *Mirai* and *Tsunami*. They contain hundreds of variants grouped under the same AV label and are the

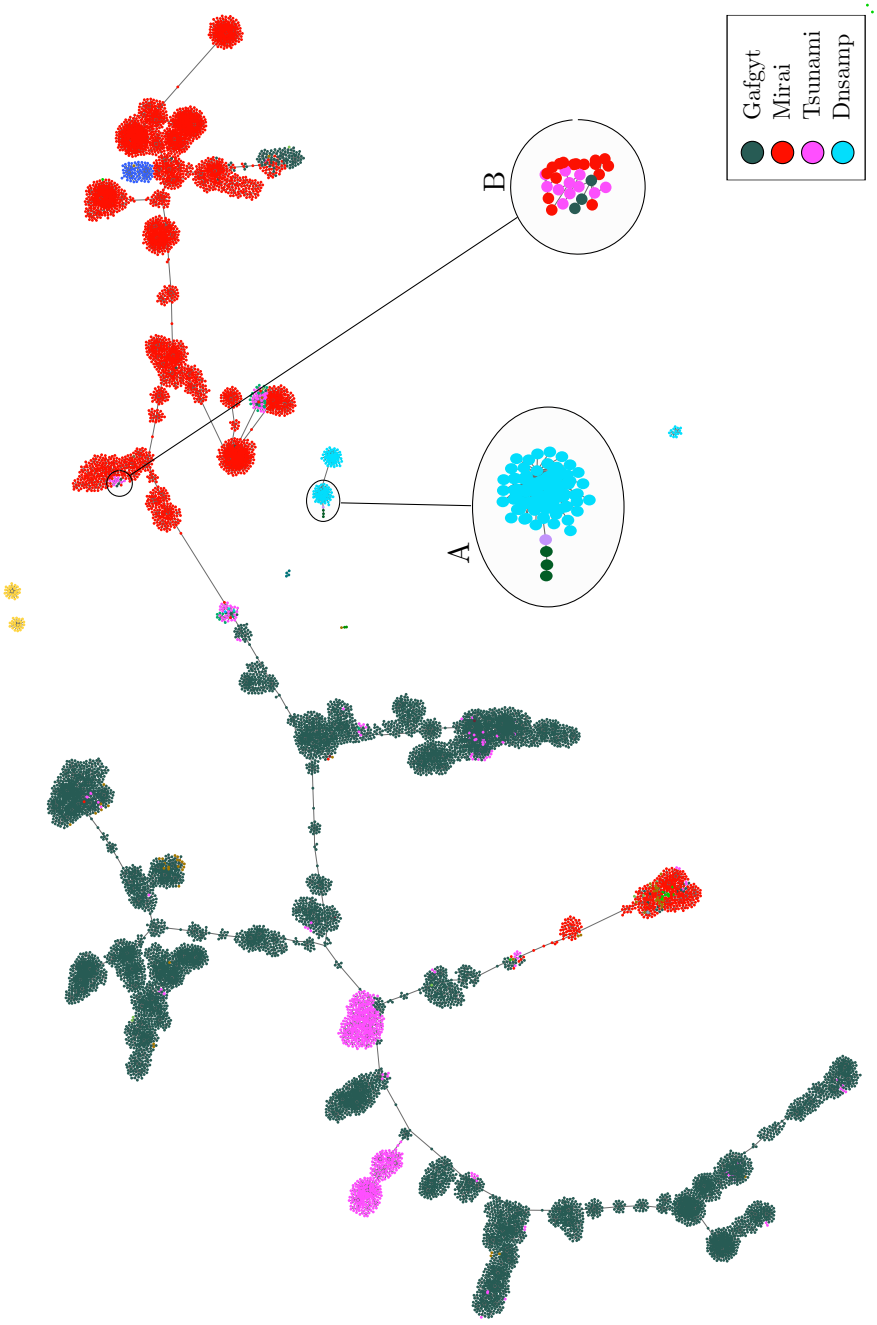


Figure 4.3: Lineage graph of MIPS samples colored by family.

Table 4.5: Common functions across top10 malware families.

VS	Gafgyt	Mirai	Tsunami	Dnsamp	Hajime	Ddostf	Lightaidra	Pnscan	Skeeyah	VPNFilter
Gafgyt		115	189	3	1	2	18	-	-	-
Mirai			63	1	1	-	2	-	-	-
Tsunami				4	-	3	1	-	-	-
Dnsamp					-	65	-	-	-	-
Hajime						-	-	-	-	-
Ddostf							-	-	-	-
Lightaidra								-	-	-
Pnscan									-	-
Skeeyayh										-
VPNFilter										

ones with longer persistence on VirusTotal. All three started to present fused traits over time and they still hit on VirusTotal. On the other hand, more specialized IoT malware targets specific CPU architectures and have a much shorter appearance. Today IoT malware code is not as complex as the one found in Windows malware, yet AVs may lose robustness when it comes to identifying widely reused functions and packed samples.

As described in Chapter 4.4.5, the distance function we used for the HNSW algorithm is based on the number of functions with binary similarity ≥ 0.5 (as suggested by Diaphora). The analyst can then adjust this threshold when plotting the graphs to either display even uncertain similarities among families (at 0.5 threshold) or highlight only the perfect matches of exact code reuse (at 1.0 threshold).

4.5.1 Code Reuse

Figure 4.3 shows the lineage graph for MIPS samples plotted at similarity $\geq$ 0.9 and with node colored according to their AVClass labels. For all other architectures, miniaturized graphs are reported in Figures 4.7 - 4.12 at the end of the chapter.

Overall, MIPS samples include 39 different labels. However, the graph is dominated by few large families: *Gafgyt*, *Tsunami* and *Mirai*. These three families cover 87% of the MIPS samples and they are also the ones that served as inspiration for different groups of malware developers, most likely

Table 4.6: Outlier samples and AVClass labels

Architecture	Number of samples		
	Wrong label	Without label	Total
ARM 32-bit	19	9	28
MIPS I	25	41	66
PowerPC	1	4	5
SPARC	2	0	2
Hitachi SH	7	0	7
Motorola 6800	8	2	10
Total	62	56	118

because of the fact their source code can be found online. It is interesting to note how this tangled dependency is reflected in the fact that the most of the *Tsunami* variants are located on the left side of the picture close to *Gafgyt*, but some of them appear also on the right side due to an increased number of routines borrowed from the Mirai code.

Besides these three main players, the graph also shows samples without any label or belonging to minor families. For example, the zoom region [A] contains a small connected component of 283 *Dnsamp* samples with a tail of 4 samples: 1 with label *Ganiw* and 3 with label *Kluh*. All together are linked to *ChinaZ*, a group known for developing DDoS ELF malware. The very high similarity between *Ganiw* and *Kluh* seems to be more interesting, since *Kluh* could be seen as an evolution of the first (and appeared 3 months after on VirusTotal), yet AVs assign them different labels.

Table 4.5 reports the number of shared functions (at 0.9 similarity) across the top 10 families in our dataset and takes into account the full picture of the six main architectures. The code sharing for *Mirai*, *Gafgyt* and *Tsunami* is once again confirmed to play a fundamental role in IoT malware with hundreds of functions shared across the three. However, we can see their incidence in minor families like *Dnsamp*, which borrows functions for random numbers generation and checksum computations, or *Lightaidra*, reusing 18 functions from *Gafgyt*. Less widespread families such as *Dnsamp* and *Ddostf* also show high similarity with a total of 65 shared functions. Instead, targeted campaigns like *VPNFilter* do not overlap with main components of the famous families.

4.5.2 Outliers and AV Errors

One of the analysis we can perform on the phylogenetic trees is the detection of *anomalous* labels, by looking for *outlier* nodes. We define as outlier a (set of) nodes of one color which is part of a cluster that contains only nodes of a different color. Outliers can correspond to samples that are misclassified by the majority of AV scanners or to variants of a given family that have a considerable amount of code in common with another family (and for which, therefore, it is difficult to decide which label is more appropriate). But outlier can also be used to assign a label, based on its neighbors, to samples for which AVClass did not return one.

Although the number of mislabelled samples is not significant in our dataset, we can use our automated pipeline to promptly detect suspicious cases in newly collected data. The outliers discussed in this section also show that a very high ratio of code similarity can often confuse several AV signatures.

Based on a manual inspection of each group of outliers, Table 4.6 reports a lower bound estimation of the mislabelling cases broken down by architecture. Overall we found 118 cases with 62 samples we believe to have a wrong AVClass label and 56 for which AVClass was not able to agree on the AV labels. ARM and MIPS (which cover 66% of our dataset) are responsible for over 80% of the errors, with MIPS samples being apparently the most problematic to classify. The pattern is reversed for less popular architectures, like Hitachi SH and Motorola 68000 (13.3% of the dataset) that account for 17 mislabelled samples, while PowerPC and SPARC (20.6% of the dataset) had only 7 cases.

Looking closer, all cases of wrong labels seemed to be due to a high portion of code reuse between two or more families. The zoom region [B] in Figure 4.3 is an example of this type of errors. A Tsunami variant that borrows a number of utility functions from Mirai resulted in few of its samples being misclassified as Gafgyt by many AV vendors.

Another example, this time related to a smaller family, is a set of 12 *Remaiten* samples that AVClass reported as *Gafgyt* (*Remaiten* is a botnet discovered by ESET that reuse both *Tsunami* and *Gafgyt* code, that extend with a set of new features). We also observed that in some cases AVs assign different labels for samples with an almost full code overlap. For example, under PowerPC, a binary is assigned the label *Pilkah*, thus giving birth to a new family, even if it is only a very minor variation of *Lightaidra*.

Finally, we found examples of how an extremely simple and well known packer like UPX can still cause troubles to AV software. For instance, 29 packed samples for MIPS did not get an AVClass label even if their code

Table 4.7: Number of variants recognized for top 10 families in our dataset. Malware families with - contained *only* stripped samples which prevented any accurate variant identification.

Family	Candidate Variants	Validated Variants (Source code)	Number of samples			Persistence (days)	
			Min.	Max.	Avg.	Max.	Avg.
Gafgyt	1428	140	1	4499	285.59	1210	283.21
Mirai	386	57	1	776	39.05	661	103.35
Tsunami	210	27	1	544	93.59	1261	421.63
Dnsamp	48	4	3	1394	362.75	1444	691.25
Hajime	1	1	1	1	1	1	1.00
Ddostf	11	3	2	755	260.00	483	308.33
Lightaidra	7	7	1	4	1.43	299	43.57
Pnscan	1	1	2	2	2.00	1	1.00
Skeeyah	-	-	-	-	-	-	-
VPNFilter	-	-	-	-	-	-	-
Total	240	2091					

was very close to *Gafgyt*.

4.5.3 Variants

The phylogenetic trees produced by our method can also be used to identify fine-grained modifications and relationships among variants within the same malware family.

In order to bootstrap the identification of variants we decided to take advantage of the binary similarity-based symbol propagation described in Chapter 4.4.3. As a first step we identify candidate variants by grouping all malware samples based on their set of unique symbols. These symbols were either present in the binary (in case of an original unstripped binary) or were propagated from other unstripped binaries (in case of an original stripped binary). While this symbol-based variant identification technique is subject to errors – noise from symbol extraction, incomplete symbol propagation – it gives a first estimate of the number of variants by capturing fine-grained differences such as added, removed or renamed functions. Table 4.7 provides the number of identified variants for the top 10 largest malware families in our dataset. We can see that *Gafgyt*, and to a less extent *Mirai* and *Tsunami* appear to have spurred more than 2,000 variants all together. This phenomenon is supposedly fueled by the availability of the source code online for these three major malware families. It is important to note that

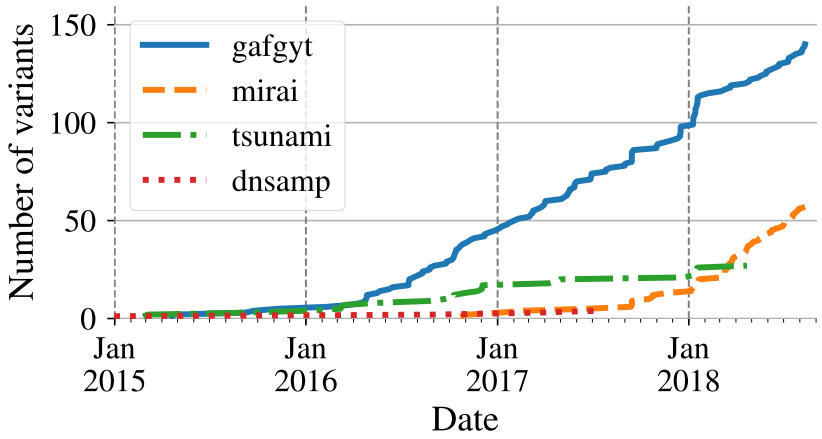


Figure 4.4: Appearance of new variants over time.

given that this step relies on symbols it excludes all stripped samples for which symbols could not be propagated, e.g., all samples of the *VPNFilter* malware were stripped hindering the identification of variants.

As a second step we rely on the leaked source code collected from online repositories, as described in Chapter 4.4.4 to validate previously identified variants. By matching symbols found or propagated in the binaries with functions found in the source code we were able to validate more than 200 variants. It is interesting to see that as much as 50.3% of the samples had at least a partial match to our collected source code – but only 740 samples resulted in a perfect match of the entire code. This suggests that many malware authors take inspiration from leaked source code, yet they introduce new modifications, thus creating new independent variants. The surprisingly high number of variants having their source code online is a great opportunity for us to validate and better study them. Validating the others unfortunately require time-consuming manual analysis. From Table 4.7 we can see that the collected source code enabled us to validate 240 variants with *Gafgyt* taking a slice equal to 58% of the total, followed by *Mirai* with a lower share of almost 24%. The source code we collected matched also minor families. For example *Hajime*, known to come with stripped symbols, was found to have one sample referring to the *Gafgyt* and *Mirai* variant *Okane*, actually suggesting the *Hajime* sample was misclassified by AVs. In a similar way, two samples of *Pnscan* partially matched with a port scanner tool named like the family and available on GitHub. However, the authors of these samples introduced new functionalities to the original code. While

the availability of IoT malware source code online facilitates the development of variants, it can also be leveraged to identify and validate them. Finally, in order to evaluate the accuracy of the source code matching we took an extra step and manually verified and confirmed some of the variants that matched source code.

Another important aspect to understand the genealogy of IoT malware is the combination of binary data with timing information. By measuring the first and last time associated to each variant we can get a temporal window in which the samples of each variants appeared in the wild (shown in the last two columns of Table 4.7). Here we can notice how quickly-evolving families like *Gafgyt* and *Mirai* tend to result in short-lived variants. For instance, *Gafgyt* variants appeared in VirusTotal for an average of 10 months, and *Mirai* variants for four. Instead, *Tsunami* and *Dnsamp* variants persisted for longer periods: respectively one year and two months the first and almost two years for the second. Figure 4.4 shows, in a cumulative graph, the number of new variants that appeared over time for the three main families. It is interesting to observe the almost constant new release of *Gafgyt* variations over time, the slower increase of *Tsunami* variants, and the rapid proliferation of *Mirai*-based malware in 2018.

4.6 Case Studies

After showing our automated approach for systematic identification of code reuse in Chapter 4.4 and presenting an overview of the phylogenetic tree in Chapter 4.5, we now discuss in more details two case studies. We use these examples as an opportunity to provide a closer look at two individual families and discuss their evolution and the multitude of internal variants.

It is important to note that the exact time at which each sample was developed is particularly difficult to identify as malware could remain undetected for long periods of time. Since ELF files do not contain a timestamp of when they were compiled, we can only rely on public discussions and on the VirusTotal first submission time as source for our labeling. Some families are only discussed in blog posts by authors that did not submit their samples to VirusTotal. Previous research also found that for APT campaigns the initial VirusTotal collection time often pre-dates the time in which the samples are "discovered" and analyzed by human experts by months or even years [GCB+15]. Therefore, in our analysis we simply report the earliest date among the ones we found in online sources and among all samples submitted for the same variant to VirusTotal. However, this effort is only performed for presentation purpose, as we believe that detecting the

similarities and changes among samples (the goal of our analysis) is more important than determining which ones came first.

Example I – Tsunami (medium-sized family)

Tsunami is a popular IRC botnet with DDoS capabilities whose samples represent almost 4% of our dataset. Its code is available online and gives birth to a continuous proliferation of new variants, sometimes with minimal differences, other times with major improvements (i.e., new exploits and new functionalities). Tsunami's main goal is to compromise as many devices as possible to build large DDoS botnets. Therefore, we obtained samples compiled also for less common architectures such as Motorola 68K or SuperH. Overall, 76% of its samples are statically linked but with the original symbols in place. When constructing the genealogy graph of *Tsunami*, we not only took advantage of the extracted symbols from the binaries but we also cross-correlated them with available source code of multiple variants we scraped from online forums, as explained in Chapter 4.4.4. This way we were able to color the graph and assign a name to different variants.

The top part of Figure 4.5 shows the mini-graph for six different architectures. The main part of the figure further zooms in on the evolution of a group of 748 ARM 32-bit binaries. These samples all share the main functionality of *Tsunami* and therefore the functions for DDoSing and contacting the CnC remained the same across all of them.

On the most right of Figure 4.5, there is a visible section in which the vast majority of samples are labeled as *Kstd* according to the AV labels. With only two flooders, *Kstd* represents one of the oldest and most famous sub-family which acted as a skeleton and inspiration for newer malware strains. By moving left on the graph, we meet a fairly high dynamic area with binaries very similar to each other but with new features such as frequent updates and new flooders. The first samples in this group correspond to the *Capsaicin* sub-family, for which we performed a manual investigation to identify the new functionalities. *Capsaicin* includes 16 flooders based on TCP, UDP and amplification attacks. It uses *gcc* directly on the infected device, taking its presence for granted. Some *Tsunami* variants are also examples of inter-family code reuse, with code borrowed from both *Mirai* and *Gafgyt*. For example, *Capsaicin* borrows from Mirai the code for the random generation of IP addresses that is used to locate candidate victims to infect. Some *Tsunami* samples also perform horizontal movement reusing *Mirai*'s Telnet scanner or SSH scanners also found in *Gafgyt*, while others use open source code as inspiration (e.g., the *Uzi* scanner).

Moving left we then encounter the *Weebsquad* and *Uzi* variants. The

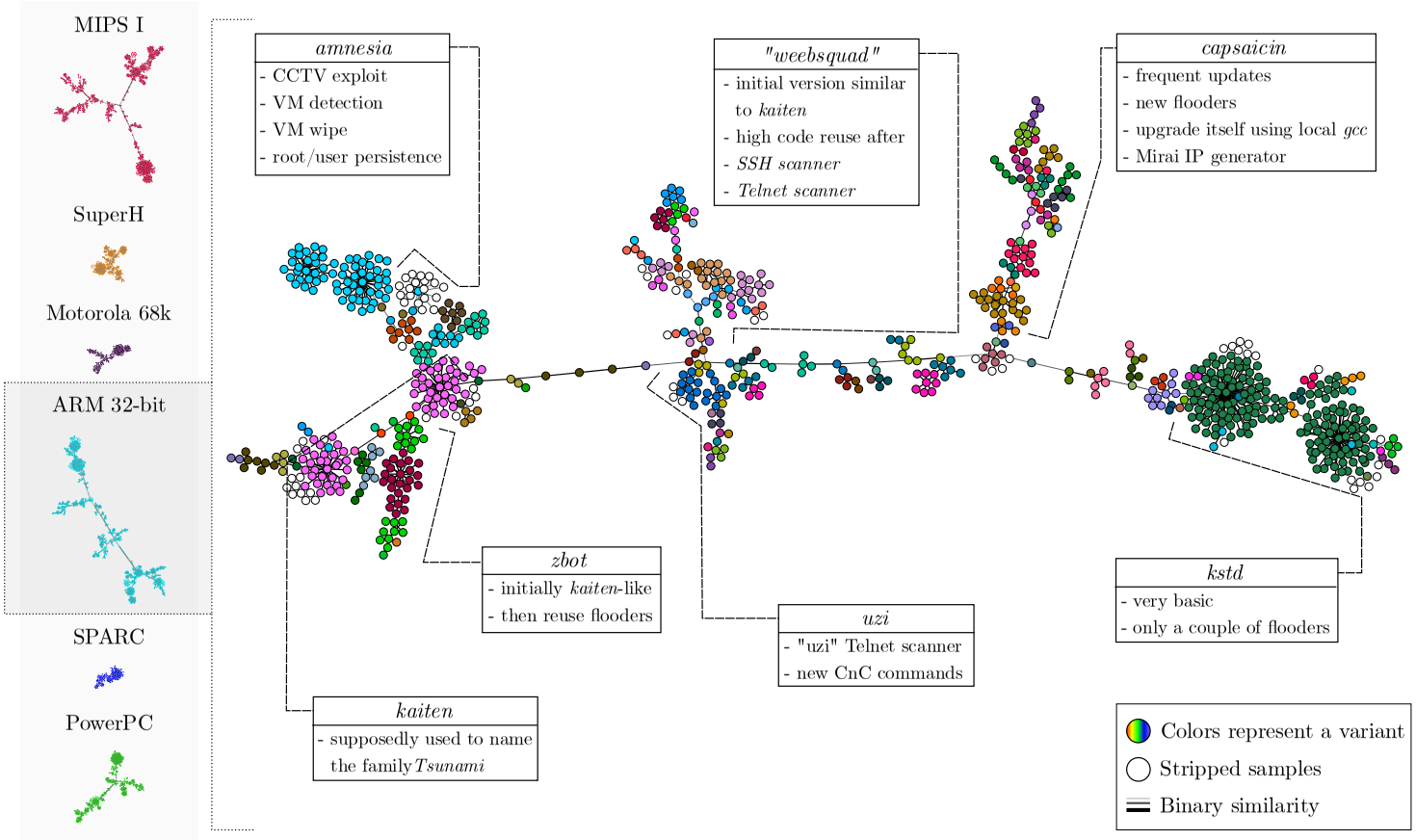


Figure 4.5: Lineage graph of *Tsunami* samples for *ARM 32-bit*.

first is a branch spreading over Telnet and SSH, for which we could not find any online source code that matched our samples. We named these variants based on the fact that they all included their name in the binaries. Interestingly some AVs on VirusTotal mislabeled these samples as *Gafgyt*, possibly because of the code-reuse between *Tsunami* and *Gafgyt* we mentioned earlier.

In the left side of Figure 4.5 we encounter *Kaiten*, another popular variant from which many malware writers forked their code to create their own projects. For instance, *Zbot* (bottom-left on the graph) is a *Kaiten* fork available on GitHub, in which the authors added two additional flooder components.

Our similarity analysis also recognizes *Amnesia*, a variant which was discovered by M. Malík of ESET in January 2017. This sub-family includes exploits for CCTV systems and it is one of the rare Linux malware adopting Virtual Machine (VM) detection techniques. Unlike most of the samples in the graph, *Amnesia* is stripped and dynamically linked. However, our system detected very high code similarity with another unstripped sample which uses the same CCTV scanner and persistence techniques, but without VM detection capabilities. Thanks to our symbol propagation technique we also managed to connect the *Amnesia* samples to the rest of the family graph.

Example II – Gafgyt (large family)

Gafgyt is the most active IoT botnet to date. It is comprised of hundreds of variants and is the biggest family in our dataset with 50% of the samples. It targets home routers and other classes of vulnerable devices, including gaming services [Hao].

We visualize the code-similarity analysis of samples for ARM 32-bit in Figure 4.6. Our system identified more than 100 individual variants. Like the Tsunami case study, we were often able to leverage available source code snippets to validate the identified variants.

The graph is clearly split into two main areas, with binaries compiled with *THUMB mode* support on the left, and with *ARM mode* only on the right. Since the two halves are specular we label variants separately on one or the other side of the graph to improve readability. *Bashlite* is believed to be one of the first variants of *Gafgyt*. Its samples are often mistaken for the *Qbot* variant (the two are frequently presented as a synonym) but their code presents significant differences. For example, *Qbot* uses two additional attack techniques (e.g., DDoS using the GNU utility `wget`). Our method

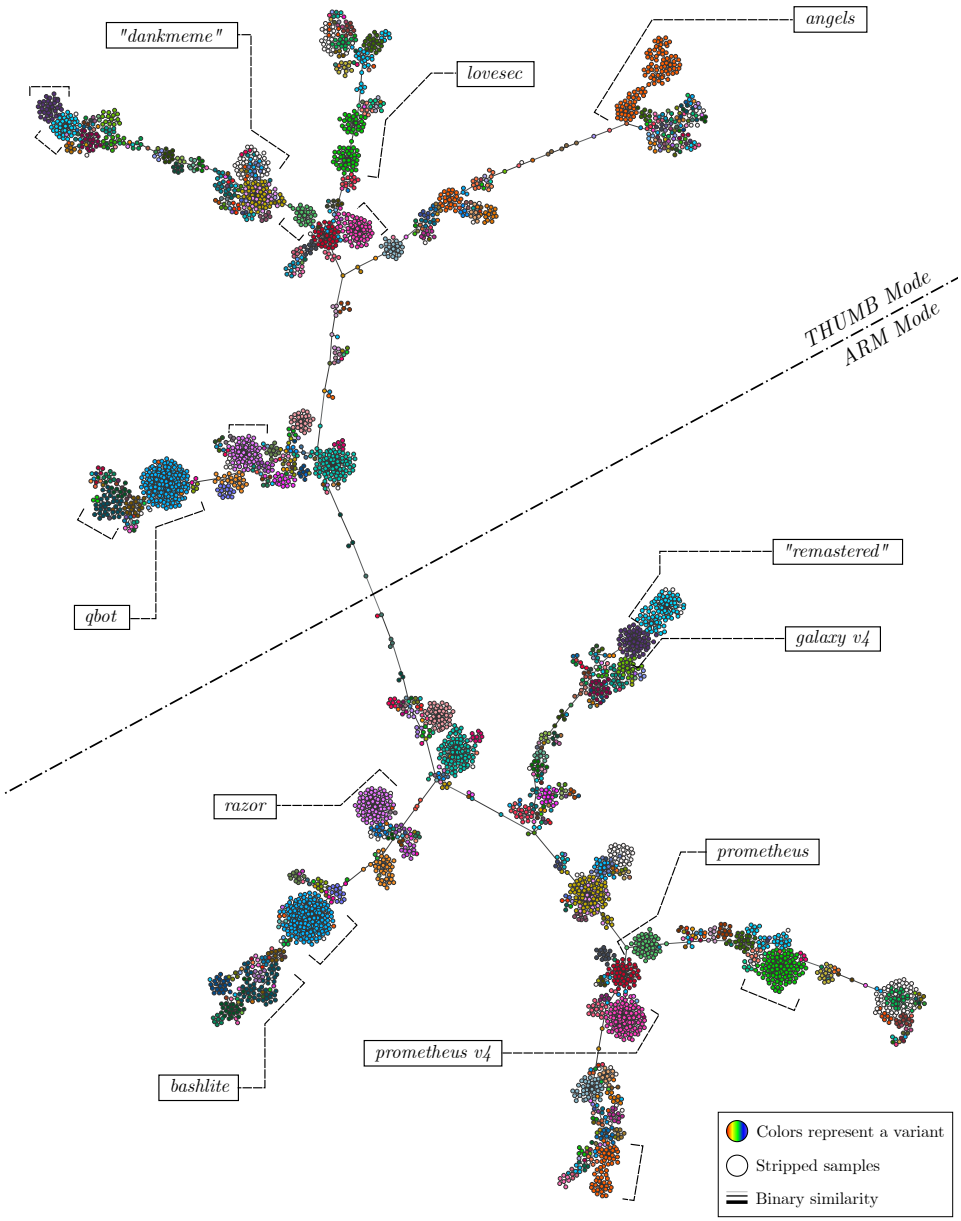


Figure 4.6: Lineage graph of *Gafgyt* samples for *ARM 32-bit*.

rightfully recognizes them as belonging to the same family but as distinct variants.

The next cluster in our genealogy refers to *Razor*, which fully reuses the previous source code but adds an additional CnC command to clear log files, delete the shell history, and disable `iptables`. *Prometheus*, for which we crawled two bundles, is an example of malware versioning. Among the features of *Prometheus*, we see self upgrade capabilities and usage of *Python* scripts (served by the CnC) for scanning. Its maintainer added a Netis[4] scanner in *V4* to reinforce self propagation through exploitation. Self propagation and infection is further enhanced in *Galaxy* with a scanner dubbed *BCM* and one called *Phone* suggesting it targets real phones. Next to *Galaxy* we find an almost one-to-one fork we call *Remastered* which does a less intrusive cleanup procedure, cleaning temporary directories and history but without stopping `iptables` and `firewalld`.

Finally, in the top left-hand corner of Figure 4.6 we uncover *Angels*, reusing some of *Mirai*'s code for random IP generation (like other variants) and targeting specific subnets hard-coded in the binaries.

4.7 Conclusions

We have presented the largest study known to date over a dataset consisting of 93K malicious samples. We use binary similarity-based techniques to uncover more than 1500 malware variants and validate more than 200 of them thanks to their source code leaked online. AV signatures appear to be not robust enough against small modifications inside binaries. As such rewriting a specific function or borrowing it from another family can be enough to derail AVs often leading to mislabeling or missed detections.

[4]Netis (a.k.a. Netcore in China) is a brand of routers found to contain an RCE vulnerability in 2014 [Yeh].

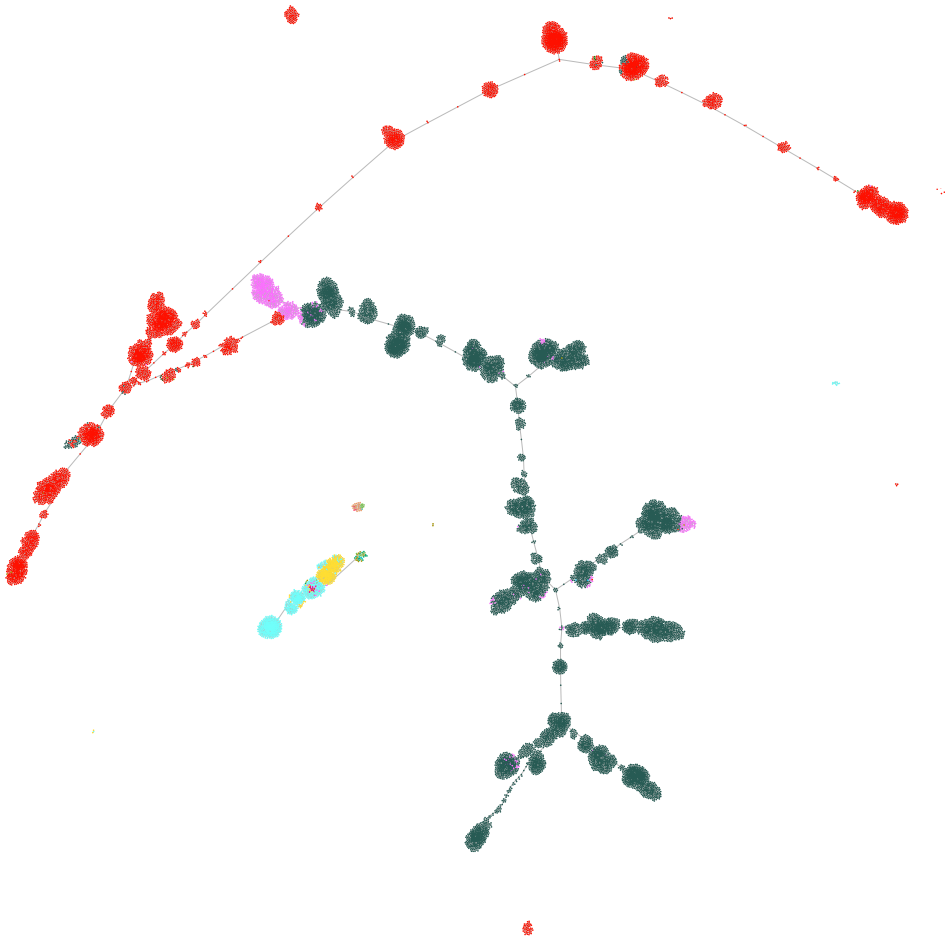


Figure 4.7: Lineage graph for *ARM 32-bit* architecture colored by family.

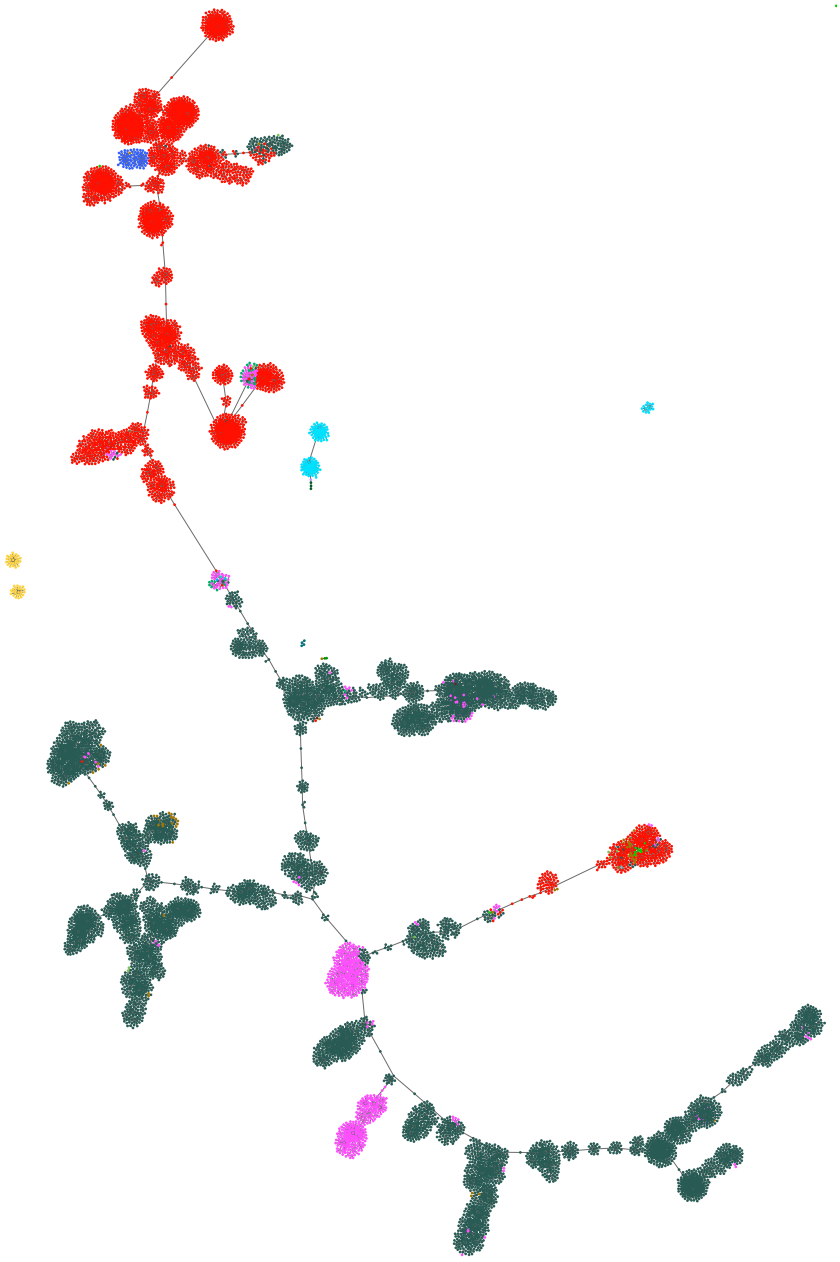


Figure 4.8: Lineage graph for *MIPS I* architecture colored by family.

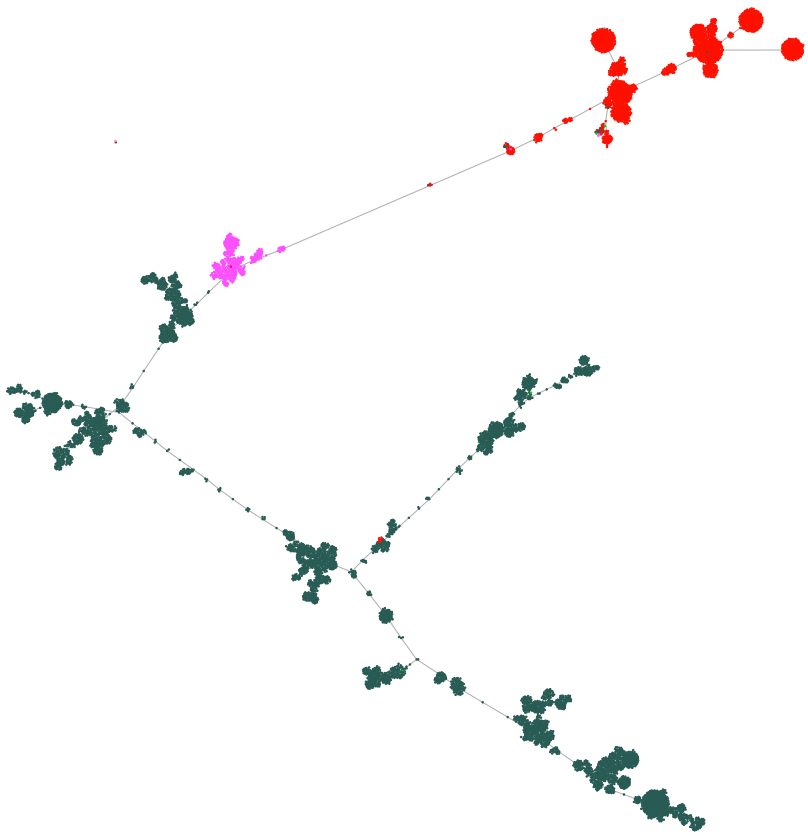


Figure 4.9: Lineage graph for *PowerPC* architecture colored by family.

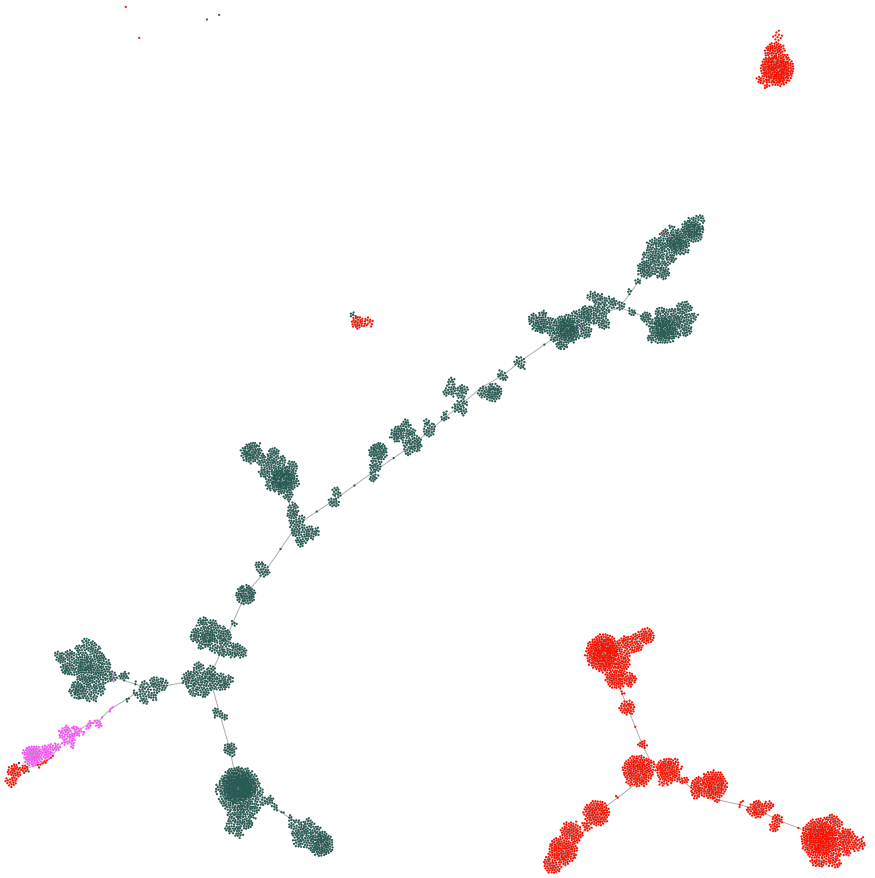


Figure 4.10: Lineage graph for *SPARC* architecture colored by family.

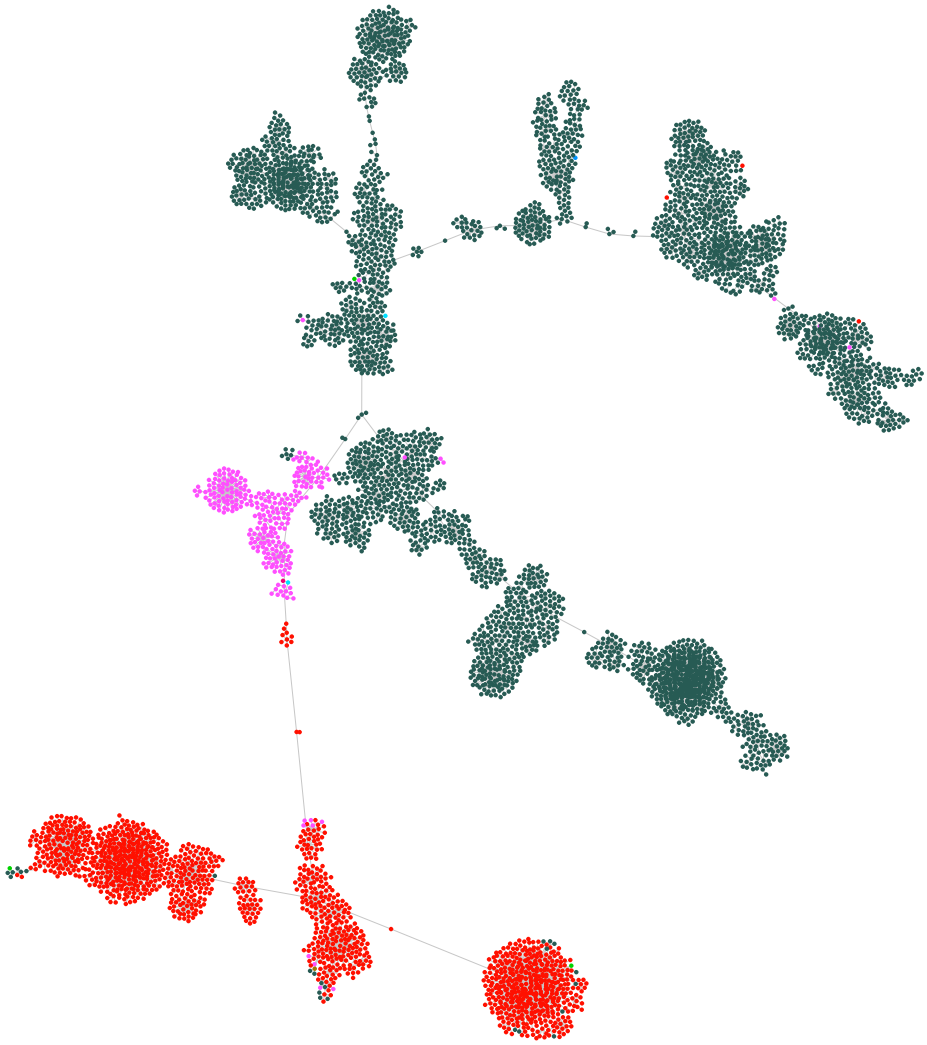


Figure 4.11: Lineage graph for *Hitachi SH* architecture colored by family.

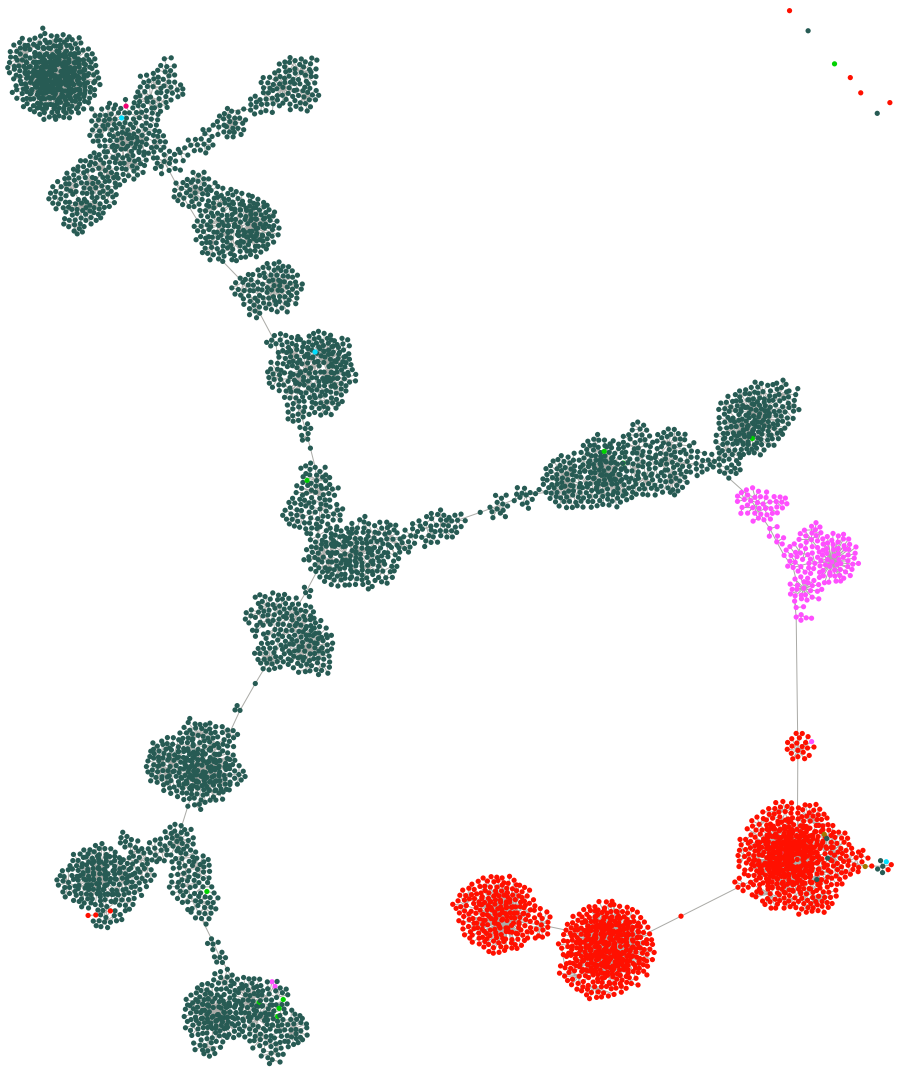


Figure 4.12: Lineage graph for *Motorola 68000* architecture colored by family.

Chapter 5

User Code Identification in Statically Linked Binaries

5.1 Introduction

Reverse engineering for binary and malware analysis is a tedious and time-consuming task often burdened by the lack of crucial information. The availability of symbols and debug information in the executable image can significantly help reverse engineering. When binaries do not provide this advantage, they must be tackled in their *stripped* version. In this case, the analyst cannot rely on function names to understand the semantic of a routine, there is no knowledge of data types, and variables do not have the same name assigned by the original developer. Moreover, while statically linked programs are self-contained to promise good portability across different machines, they pose an additional challenge: there is no explicit boundary between user-defined code and third-party library code. As a consequence, the analyst is overwhelmed by a pre-analysis phase to discern library routines from user-defined functions. Libraries can provide utility functions or control the interaction with the operating system (e.g., the C library). On the other hand, reverse engineering user-defined code gives the analyst a high-level understanding of the overall execution flow.

Knowing the boundary between user and library code has direct consequences also in code similarity tasks. In fact, the binary comparison of two statically linked files can produce a large number of spurious matches. If binaries are stripped from their debug information, there is no way to preserve the distinction of individual code objects (e.g., a user function erroneously compared with a library function). The same problem holds when unstripped and stripped binaries are compared for *resymbolication* purposes, thus an unknown function can be assigned the wrong name. This is extremely unfavorable in the field of malware analysis, especially on Linux-based operating systems, where static linking is much more common than on Windows. Recent studies on Linux and IoT malware reported that usually more than 8 samples out of 10 are statically linked [CGFB18, CVD+20].

Both academy and industry research made a considerable effort to identify functions and libraries in statically linked and stripped files. Many works have explored this problem, mostly focusing on pattern-based function signatures [Gui], control flow analysis [SWD17, NRM+17, QSM15b], and re-labeling of stripped binaries [JRM11, HIT+18, PECK20]. On the code similarity side, we have seen contributions for pure similarity comparisons [Zyn], or involving code clone detectors [FFCD14, DFC19, HYD17]. However, some of these works rely on databases of function fingerprints which are difficult to generate in real use-cases (and in particular on embedded systems, due to the diversity of libraries and compiler toolchains).

Others approaches require ground-truth information such as an unstripped library to compare with a target binary.

The most recent contribution with a similar goal is CodeCut [evm], a tool measuring the call directionality within code modules to detect the boundaries of object files. In particular, while functions at the beginning of a module should call higher addresses, the directionality of function calls switches in the opposite direction towards the end of the module. CodeCut has been designed to aid the analysis of embedded operating systems linked into a single executable. However, we tested CodeCut on statically linked ELF binaries and noticed that its assumption on the call directionality tends to produce more boundaries than the actual ones, often resulting in many virtual object files of few functions each.

To the best of our knowledge, we still lack a technique that can be used to automatically recognize user-defined code in arbitrary statically linked programs. To cover this gap, in this chapter we present a novel approach to identify the boundary between user-defined code and library code in statically linked ELF files. We implemented our approach in a tool named *BinCut*, which works on the functions call graph to extract code module dependencies from the spatial layout of a binary. It then employs a search algorithm to find all the possible boundaries (*cut points*) and then employ an extendable set of heuristics to select only the most suitable cuts. Finally, we build a classification model to pre-select the heuristic that provides the best result on each target binary. We are aware that our approach does not yet provide the perfect solution to the problem of function and library identification, but it rather gives insights based on the internal characteristics of a statically linked file.

In this chapter we make the following contributions:

- We present binary layout analysis to explore (in)dependencies across code modules.
- We present BinCut, our system to identify the boundary between user and libraries code in statically linked programs.
- We provide an evaluation of BinCut on a dataset made of 222 open-source binaries for *x64* (from generic tools to complex packages) and a collection of 11,471 IoT malware for *MIPS*.
- We present a case study of a real BinCut use-case involving code similarity of malicious statically linked samples.

BinCut can find the boundary function between user and library code with an average deviation error of less than 2% from the real function.

Finally, we will talk about some known limitations of the technique and present our concluding remarks.

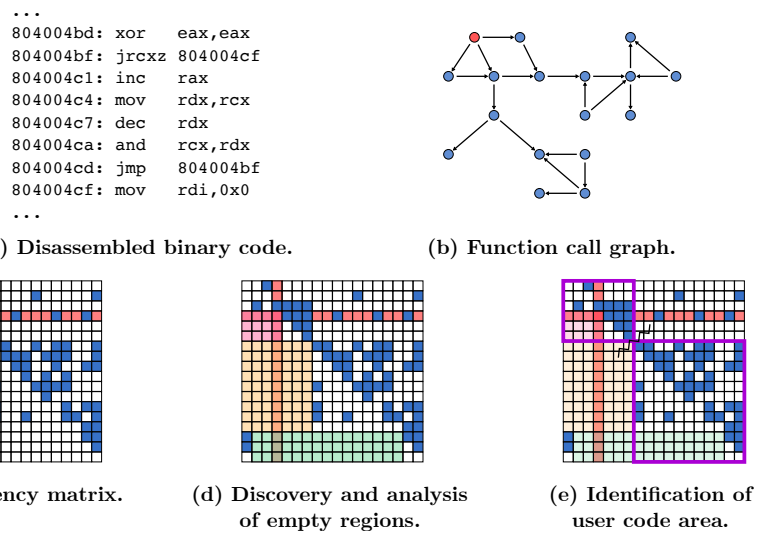
5.2 Overview

In this section we present our code boundaries recognition system, with an overview of the main components involved. Our method is illustrated over five main steps in Figure 5.1. We start by disassembling the binary as shown in Figure 5.1(a) to identify its functions. The detection of the function entry points is a necessary building block, since the functions are used to reconstruct the call graph represented in Figure 5.1(b). At the same time, we capture the address of the `main` function (the red elements in the figures) and use it as a hint later in the analysis, since `main` most likely belongs to user code. Later on, in Figure 5.1(c) we represent the call graph as an adjacency matrix. The core idea is that functions belonging to the same object file (and library) will usually call more neighboring functions than code from other modules. As a consequence of spatial locality, if we look at the adjacency matrix, each code module should be naturally clustered in a limited area of the program address space. We detect localized regions of code using a reverse approach. We scan the adjacency matrix to discover the empty largest regions or, in other words, we mark the sets of function not doing any localized calls. This step is represented in Figure 5.1(d). Finally, we use a set of heuristics to pinpoint candidate *cut* points and choose the most relevant among them through a classification process. The *cut* point given as output will be the boundary between user-defined code and library code, as illustrated in Figure 5.1(e).

5.2.1 Static Analysis

Given a statically linked program, it is important to detect as many functions start addresses as possible to minimize the loss of precision in the following analysis steps. Our system relies on the function detection algorithm of IDA Pro, even though other disassemblers may perfectly fit into our system. We use IDA Pro to extract three types of information: the address of `main`, the call graph, and the list of data references.

When a binary is both stripped and statically linked—the worst case for binary analysis—the analyst can only rely on a limited set of information. For example, the entry point (commonly named as `_start`) is the initial trampoline to user code for the binary execution. Developers are free to define their own customized start routine, but the one provided by *libc*



(a) Disassembled binary code.

(b) Function call graph.

(c) Adjacency matrix.

(d) Discovery and analysis of empty regions.

(e) Identification of user code area.

Figure 5.1: An overview of the steps to identify user code in statically linked binaries.

has long been the de-facto standard for the compilation of Linux software. In particular, the C Runtime objects (CRT) provide bootstrap code that is executed before jumping to `main`. We follow the disassembly of `_start` until we reach `___libc_start_main` and extract the address of `main` from its arguments. Other flavors of C libraries work similarly e.g., *uClibc*, mostly used in embedded devices, thus we are not limited to *Glibc* only. While the bootstrap code is normally linked at lower addresses, the location of `main` (thus the user-defined code) depends on the order in which the object files have been linked statically. It is important to note that we cannot assume that the user code is always linked before the libraries, since their actual order in the context of the linking process can be controlled in a fine-grained manner.

IDA Pro is also used to construct the call graph of the functions detected in the binary. When the sections' information of the ELF is available, we consider only the functions belonging to *.text* and discard the executable code possibly discovered elsewhere (e.g. *.plt*, *.init*, *.fini* or data.) On the other hand, if the binary is fully stripped—thus there is no knowledge of sections—we extract all the functions of the segment having the execute permission bit set. We then scan the cross-references of every function to build the final call graph of the program under analysis. The graph is stored as an adjacency matrix indexed by caller (rows) and callee (columns) with 1s where there is an edge between the two and 0s where such edge does not

exist in the code.

Finally, we use static analysis to collect specific data references. We consider only the data references matching the following three conditions:

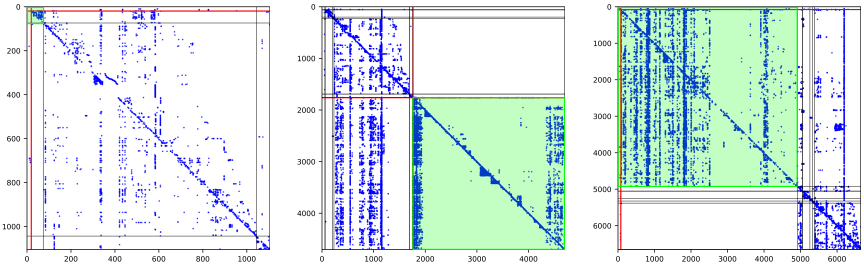
- the data address is referenced by at least one function in our graph;
- the address is not flagged as code by IDA Pro;
- the address is part of a segment with write permissions.

In other words, we do not consider the variables containing function pointers and those which are in read-only areas. We give more details about the limitations of the latter and how we use the data references in Chapter 5.2.3.

5.2.2 Binary Layout Analysis

The intuition behind the method we use to isolate user-defined code from statically linked libraries can be visualized graphically. Figure 5.2 contains the plots of the adjacency matrix computed on the famous Linux malware Mirai, busybox, and Python. All the binaries are compiled and statically linked respectively with *gcc 7.5.0* and *ld 2.30*. In each figure, we draw a red line to show the position of `main`, and black lines representing the boundaries of the libraries linked into the final executable. The green squares highlight the region that we want to recognize and extract with our tool, as they correspond to the areas of user-defined code. The linker can place specific object files even before the user-defined code e.g., *libc* initialization routines. This case can be recognized by the presence of sparse edges below the green squares. The three binaries in Figure 5.2 differ in size, ranging from about 1,000 functions to more than 6,000, and in their linking properties. Mirai and Python have the user code at lower addresses, but the first covers less than 5% of the all functions in the binary, while the second covers more than 50% of the binary. In contrast, the user code of busybox has been linked at higher addresses—*after* the embedded libraries.

The geometric layout of the call graph represented as adjacency matrix, made us refer to this analysis phase as binary layout analysis. We can recognize from the plots that functions within the same module tend to be naturally clustered together in square shapes arranged on a staircase pattern. However, these regions do not have perfectly well-defined and dense square shapes, because it is never the case that all the functions in a module call each other. A statically linked program can indeed have many pairs of functions without any relationship. For this reason, we consider the opposite problem: The layout analysis step is based on the research



(a) Mirai (b) Busybox 1.31.1 (c) Python 3.8.5

Figure 5.2: Examples of adjacency matrices for three binaries.

of sets of contiguous functions that are **not** calling any other contiguous portion of code. We scan the adjacency matrix by row and columns to look for the biggest *empty* regions with a corner on the diagonal. By *empty* we mean that there are no edges between the functions on the rows and columns. We define the corner found on the diagonal as a *cut* point. This condition allows us to divide the adjacency matrix into four sub-matrices: the region we already marked as blank, the same area flipped across the diagonal, and the two complementary regions which cover one or more code modules each. The first two regions group together inter-module function calls, whereas the other two also include intra-module function calls. In addition to this, the empty region can be either on the lower triangular matrix or on the upper part, as a result of the linking process. If most of the function calls are towards higher addresses the lower triangular matrix will be less dense. Instead, the upper triangular matrix will be less dense if most of the calls are towards lower addresses.

We continue the research of empty regions recursively over the two sub-matrices defined along the diagonal. The analysis continues until no more new cut points are found. A cut point (and the relative blank area) is valid only if it respects the following restrictions:

- the lower or upper triangular matrix (depending on the linking directionality) must contain at least one edge;
- the area of the empty region mirrored across the diagonal must contain more edges (to satisfy and respect the directionality of function calls).
- the empty region must expand for at least 1/2 of the maximum area that can be covered from the cut point (to deal with code modules making intensive use of both backward and forward calls);

When a cut point is invalid, we continue the research on the same recursion

level. We will talk more about these heuristics and how they affect our results in Chapter 5.3.

We store all the cut points in a binary tree where the root node contains the point related to the biggest blank area in the adjacency matrix (the first ever found), while the leaves represent the points for the smallest regions.

5.2.3 Global Variables

The linker usually places data and global variables after the code section and grouped by library, unless the user provides specific customizations. The way code sections access data (in particular global variables) can suggest whether two pieces of code belongs to the same code unit or not. Variables are generally tied to the module using them and not shared across libraries and user-defined code. But while most of the data references happen internally to each module, there are some exceptions. The linker can reduce multiple declarations of the same read-only variables to unique instances. In other cases, constant values can be exported to library end-user or special variables may control particular actions of a program, e.g., the I/O streams in *libc*. However, a variable with write permissions should not be shared unless it is protected for concurrent access from external sources.

We traverse all the data references collected by IDA Pro and for each variable, we first extract its first and last access over the binary address space, and then draw a line between each couple. Finally, we sum all the lines to obtain a cumulative distribution in terms of possible accesses to global variables. Figure 5.3 is an example of the cumulative distribution of global variables for the compression utility *tar*. The black lines on the function axis represent the boundaries across the user and all the libraries linked into the executable. The references to global variables reach the highest values for the initial 500 functions approximately (the user code) to then quickly decrease as we reach `libtar`. The distribution has strong variations also across libraries except between `libtar` and `libgnu`, meaning that the first does not introduce new references to global variables. This analysis step is one of the heuristics we use for the user code boundary analysis. Curves like the one in Figure 5.3 may assist the boundary identification process depending if the user code makes higher (or lower) usage of global variables respect to libraries.

5.2.4 User Code Boundary Analysis

The challenging nature of statically linked binaries does not give any insights on the semantic of the libraries being used, their size, and the order

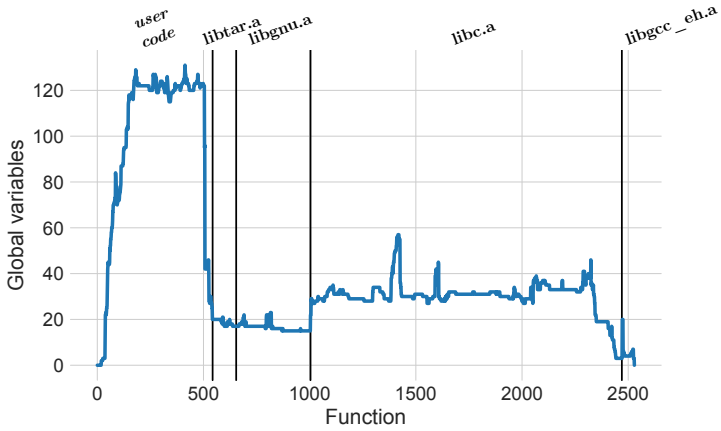


Figure 5.3: Cumulative distribution of global variables accesses for *tar 1.32*

in which they are located in the final executable. Therefore, our system needs to work without any knowledge of the binary under analysis. To overcome this limitation, we rely on four heuristics to select candidate cut points out of all the ones identified by the binary layout analysis. We call this task *user code boundary analysis* since it gives as output the functions that most likely represent the boundary between user and library code. We discuss below the implementation details of the heuristics and then report about their accuracy in Chapter 5.3.2.

Root cut point. The larger the empty region we found during the binary layout analysis is, the more likely its cut point falls between user and library code or between two libraries. While small regions can be less accurate and eventually fall within the same code module (since our search algorithm is recursive), our experiments show that the largest empty region found with our algorithm, normally represents a true separation between independent modules. This heuristic performs the best when the size of the user code is comparable with the total size of the libraries.

High-impact cut point. Similar to the previous heuristic, here we consider the regions found by the first two levels of recursion, and select as candidate cut point the one that is closer to `main`. This heuristic suffers when the user portion of code is negligible compared to the overall number of functions in the binary. We address this case with the following heuristic.

Main-closest cut point. As we said, sometimes the user-defined code

only covers a small percentage of the entire executable. For example, the SSH daemon *sshd* from the *openssh* suite counts 6,571 compiled functions. However, the user functions represent roughly 5% of the total. We found that in these situations the candidate cut point would be the one that is closer to the function `main`. Selecting the `main`-closest cut point does not involve any considerations on the size of the annexed empty region. Thus, we prefer to discard all the regions with an area below the median of all the areas found in the binary layout analysis. As already explained in the previous heuristics, regions that are too small correspond mostly to false-positive cut points.

Data-based cut point. This heuristic is based on the global variables analysis we described in Chapter 5.2.3, and we proceed as follows. First, we apply a *Savitzky–Golay* filter [SG64] on the curve of the cumulative distribution computed on the number of references to global variables (i.e., Figure 5.3). While the purpose of this filter is to smooth the curve, we also want to make sure that the most important variations on its trend are preserved. After that, we extract the subcurve delimited by the cut point of the first empty region (found by the binary layout analysis module), and consider only the portion where `main` is included. We do this since the user code boundary will fall in this restricted area. At this point, we compute the median of the subcurve and find the function that is between the longest sequences above and below the median. Finally, we consider the empty region closest to this function and declare it the respective candidate cut point. This heuristic is designed to cope with binaries where the user code makes an intensive use of global variables.

5.2.5 Boundaries Classification and Selection

The analysis modules described above give strong feedback about where the binary is to be cut to discard the libraries. An analyst can look at the plot of the binary layout to understand where is the user code, and she can get more rational suggestions by using the user code boundary analysis module. This makes sense in the context of reverse engineering and manual inspection of unknown binaries. However, we believe there is value in having an easy-to-use tool that takes a binary and provides a fully automated answer that does not involve any manual expertise.

For this reason, the last module of our system implements a data classification model to output only a single cut point. The classification process is based on numerical features which are static characteristics of the binary (e.g., number of functions and number of backward and forward calls), or

features collected upon the first three levels of recursive search of cut points. We report a detailed list of features in Table 5.1.

For example, we use as features the overall number of cut points, metrics on the empty regions tied to them, or the depth of the cut points in the tree which stores them. In total, we extract 43 features, which we then use to train a Random Forest classifier to predict the most-likely cut point.

The choice of the classifier is based on three considerations:

- The domain of our features is the set of real numbers $\mathbb{R}$, and we need a classifier that is robust to high numerical variations (e.g., a binary can have 100 or 10,000 functions).
- Feature normalization can be an issue on volatile numbers.
- The training phase on large datasets should be parallelizable.

We evaluate the model using K-fold cross-validation and report on both its performances and results in the following section.

Table 5.1: List of features used for classification.

Feature name: **Description**
num_functions: Number of functions recognized
back_forw_calls: If the binary has mostly backward for forward calls
tree_nodes: Nodes found during the recursive search
tree_levels: Maximum level of recursion
fp: Number of empty areas recognized as false positive
min_area: Area of the smallest empty region
max_area: Area of the biggest empty region
mean_area: Average surface area of empty regions
med_area: Median surface area of empty regions
std_area: Standard deviation of empty regions
var_area: Variance of empty regions
node_cut*: Cut point of the region
node_area*: Area of the region
node_global*: Number of globals crossing the region
node_level*: Level of recursion
*Features computed for each node on the first three recursion levels (total 32)

5.3 Results

In this section we present an overview of the results obtained with our system. We tested BinCut on over 200 statically linked ELF files containing hundreds to many thousands of functions, some of them with more complex compilation customizations. Overall, our tool was able to identify the exact cut point between code and libraries for 13 binaries, and it made an average error below 2% in the function boundaries in the remaining cases.

We also performed an additional experiment on more than 10k malicious samples from the dataset described in Chapter 4.2. In this second experiment, BinCut achieved a mean error below 1% and found the perfect cut for over 35% of the samples in the dataset. Finally, we used the malicious dataset to evaluate the automated usage of BinCut through a multi-class classifier.

5.3.1 Dataset

BinCut supports all executable ELF files regardless of the underlying architecture. We collected some of the most used software packages for Linux-based distributions in order to keep our experiments as much realistic as possible. We further enlarged the dataset with complex binaries and tested BinCut under circumstances which are normally more difficult to handle. Our final dataset consists of 222 ELF programs for x64 architecture statically compiled with their default building scripts. Moreover, we produced the linker map file for each binary and used it as ground truth to measure how far our tool diverges from the real boundary between user code and libraries.

Our dataset needed to be compiled manually, thus limiting the number of programs we could use in this first experiment. In fact, we needed to satisfy all the static dependencies and edit the makefiles to add support for the generation of the map files. This operation was often time-consuming, especially for packages that did not offer direct support for static linking.

Table 5.2 reports the programs included in the dataset and gives an overview of their characteristics (e.g., number of functions, number of libraries, and the percentage of user functions). The binaries range from 588 functions for *mirai* to almost 21k for *ffmpeg* and are spread over six libraries on average. Another feature that may affect the behavior of our system—independently from the number of functions and libraries—is the percentage of user code in a binary. For this reason, we compiled binaries like *vim* where the portion of user code covers up to 76.6% of the entire binary.

Table 5.2: Dataset of open-source packages.

Package	Bins No.	Functions		Libraries		User Code(%)	
		Min	Max	Min	Max	Min	Max
Mirai	2	588	589		4	12.9%	12.93%
coreutils 8.32	107	865	1530	4	7	3.4%	10.6%
sysvinit 2.97	14	861	1096	4	5	4.02%	7.12%
gzip 1.10	1		1082		5		10.72%
diffutils 3.7	4	966	1372		5	4.87%	9.84%
debianutils 4.11.1	3	1060	1221		3	2.77%	3.19%
findutils 4.6.0	6	982	1736	5	7	3.87%	17.51%
sed 4.8	1		1269		5		9.69%
tar 1.32	2	1167	2530	4	5	3.17%	21.46%
dropbear 2020.80	4	1821	2419	6	9	13.95%	21.24%
binutils 2.34	18	860	3464	4	10	1.92%	21.91%
busybox 1.31.1	1		4703		7		37.3%
openssh 7.6p1	8	1388	6538	7	12	0.89%	8.05%
vim 8.2.0736	1		6503		7		76.63%
Python 3.8.5	1		6653		9		74.01%
qemu 5.0.0	36	3869	14741	9	11	28.2%	69.66%
radare2 1.6.0-git	11	2355	17313		8	0.29%	5.35%
ffmpeg 4.2.2	2	20536	20575		14	8.95%	9.05%
Total	222						

We also evaluated BinCut on malicious samples and tested its usability in the field of malware analysis. The second dataset counts 11,471 unstripped IoT malware compiled for MIPS. Given that the unavailability of the source code makes it impossible to generate the map files, this time we identified the true user boundary using the debug symbols, as already described in Chapter 4.4.2.

5.3.2 User Code Identification

Our approach uses a search algorithm to inspect the geometric layout of a binary and four heuristics that allow the analyst to only focus on the most probable cut points. To evaluate the performances of BinCut, we measured the error of each heuristic taken separately. We define as error the distance between the cut point observed with a heuristic and the true user-libraries boundary, thus we count the number of functions between the two.

Table 5.3: Heuristics performances grouped by software package. Functions and heuristics are the average over Bins No.

Package	Functions	D-cut (%)	M-cut (%)	H-cut (%)	R-cut (%)
Mirai	588	41 (6.97 %)	**6**(**1.02**%)	**6**(**1.02**%)	**6**(**1.02**%)
sysvinit 2.97	993	348 (35.11%)	**6**(**0.65**%)	241 (24.27%)	818 (82.69%)
coreutils 8.32	1059	44 (4.09 %)	**16**(**1.45**%)	38 (3.38 %)	123 (11.53%)
gzip 1.10	1082	**24**(**2.22**%)	77 (7.12 %)	77 (7.12 %)	47 (4.34 %)
diffutils 3.7	1092	35 (3.16 %)	**24**(**1.85**%)	40 (3.85 %)	118 (10.02%)
debianutils 4.11.1	1121	157 (13.84%)	**9**(**0.77**%)	356 (31.28%)	432 (37.99%)
findutils 4.6.0	1176	32 (2.85 %)	56 (3.73 %)	**24**(**2.26**%)	117 (9.42 %)
sed 4.8	1269	**1**(**0.08**%)	74 (5.83 %)	48 (3.78 %)	232 (18.28%)
tar 1.32	1848	30 (1.64 %)	254 (10.25%)	**26**(**1.22**%)	268 (12.3 %)
dropbear 2020.80	2106	**46**(**2.41**%)	341 (15.72%)	90 (4.39 %)	480 (22.8 %)
binutils 2.34	2566	611 (23.85%)	**133**(**4.61**%)	666 (25.57%)	1248 (48.12%)
busybox 1.31.1	4703	1671 (35.53%)	29 (0.62 %)	**0**(**0.0**%)	**0**(**0.0**%)
openssh 7.6p1	4833	1620 (27.99%)	**62**(**1.51**%)	524 (10.27%)	3478 (60.71%)
vim 8.2.0736	6503	1287 (19.79%)	4893 (75.24%)	528 (8.12 %)	**0**(**0.0**%)
Python 3.8.5	6653	1938 (29.13%)	4827 (72.55%)	1053 (15.83%)	**0**(**0.0**%)
qemu 5.0.0	8684	1880 (18.44%)	3783 (41.76%)	**563**(**5.03**%)	1478 (17.93%)
radare2 1.6.0-git	15943	2823 (17.92%)	**21**(**0.22**%)	7509 (45.0 %)	11583 (70.3 %)
ffmpeg 4.2.2	20555	1028 (5.0 %)	**106**(**0.52**%)	2370 (11.53%)	4917 (23.92%)

Table 5.4: Cut error between user and libraries code. Cut error is the number of functions between the cut point and the real boundary.

Dataset	Bins No.	Perfect Cut	Cut error(%)		
			Mean	Median	Std
Benign	222	13 (5.86%)	71.54 (1.43%)	9 (0.84%)	205.93 (1.74%)
Malware	11471	4241 (36.97%)	0.94 (0.29%)	1 (0.39%)	5.52 (0.49%)

Table 5.3 shows the results obtained on the dataset of open-source programs. We group the results by software package and report the numbers as the average over the number of binaries. From the table we can notice that there is no strict relationship between the results found by BinCut and features like the number of functions or the libraries linked into the executable image. For example, M-cut (main-closest cut point) is among the best heuristics for *mirai* with a boundary error of 6 functions, but it is also the one achieving the smallest error on *ffmpeg*. However, while the latter has 20k more functions, they both have the user code extending for roughly 10% of the binary. On the other hand, R-cut (root cut point) appears to be the best method when either the binary layout is really simple (e.g., *mirai*), or the user code takes a considerable space of the binary. R-cut found a perfect boundary match on binaries like *vim* or *python* with the user code extension of about 75%. D-cut (data-based cut point) instead improves the cut point when the usage of global data is unbalanced between user code and libraries, like in *sed* and *gzip*. In these two cases, the heuristic scoring as second is H-cut (high-impact cut point), which would otherwise return a cut point including the user code and a library perfectly aligned with it.

Overall, by considering the best cut point heuristic, our system proposes a boundary with an average distance from the real cut point of about 71 functions and 1.43% of the total number of functions in the binary. BinCut found the real cut point on 13 binaries out of 222. In table 5.4 we report also the median of the errors and the standard deviation to account for packages like *coreutils*—representing the 50% of the dataset with 107 binaries.

Finally, we repeated the same measurements over the malware dataset (Table 5.4). BinCut was able to find the correct cut point for 4,241 samples (37% of the dataset) and a total mean error of less than 1%. Moreover, the standard deviation of the error went down to 5 functions, and shows how the behavior of BinCut is more stable despite the increase in the number

Table 5.5: Classification report for the malware dataset.

Heuristic	Precision	Recall	F_1	Support
D-cut	0.93	0.95	0.94	243
M-cut	0.99	0.99	0.99	3109
R-cut	1.00	0.94	0.97	86
H-cut	1.00	0.67	0.80	3

of ELF files. We give credit for this higher success rate to two factors. First of all, most of the IoT malware is more rudimentary than the binaries we compiled from open-source bundles. For instance, families like *mirai*, *tsunami* or *gafgyt* are only statically linked with *libc* and *libgcc*. Second, the malware dataset is dominated (in terms of number of samples) by few big families, thus reducing the diversity of compiler options and toolchains in the group.

5.3.3 Classification

The experiments conducted on the malware dataset give insights on the efficiency of BinCut over a real use-case. In this situation, our tool must be able to work autonomously without the analyst's intervention.

We split the malware dataset into a training set (70%) and a test set (30%) to build a classification model based on a Random Forests classifier. Moreover, we observed that the dataset contains minority classes e.g., heuristics rarely considered as the best option. We cannot know in advance the heuristic that will perform better for a specific sample, thus we address this unbalance by using class weighting. Specifically, we place a heavier penalty on misclassifying the minority classes.

Table 5.5 shows the classification report using 10-fold cross-validation. From a detailed review of the results, the binary layout analysis of BinCut can correctly characterize the malware samples with an accuracy of 0.99. The class with the fewer misclassification errors is M-cut, suggesting that this heuristic is well suited for IoT malware. However, being M-cut also the class with the largest support in the test set, we believe that the accuracy of the model points out how BinCut performs on this particular dataset, rather than a unique way to represent its usability.

5.4 Case Study

BinCut can have many applications, one of them being to aid in code similarity tasks of stripped binaries. Malware analysis for variants recognition falls into this category. The problem with stripped malware is that we cannot prevent the binary comparisons of third-party libraries, normally without any special value to the analyst.

In Chapter 4.4 we used debug symbols to filter unwanted functions, rather than a solution to cut the user portion of code from the malware. This required to use code similarity to propagate symbols from the unstripped to stripped samples. However, this means that we could only compare stripped binaries when the symbol propagation succeeded. Moreover, this approach was able to discard the libraries but also all the user-based functions that remained unnamed.

We now show that BinCut can help to overcome these limitations, effectively enabling us to discover new malware relationships that would remain otherwise undetected. Our experiment is based on the methodology described in Chapter 4.4, that we extend by taking advantage of BinCut to automatically recognize the user code also in the stripped malware.

We selected from the malware dataset all the samples labeled as *tsunami*, counting for a total of 952 binaries. After that, we generated the phylogenetic tree based on their code similarity. Figure 5.4 displays the lineage graph of *tsunami* samples for the MIPS architecture. We color the graph according to whether the samples are unstripped (in pink) or stripped (in green). From a first overview of the figure, the stripped samples immediately show similarities not appearing among them in previous lineage graphs (see Figure 5.5)—since stripped/stripped similarity was only possible in the presence of propagated symbols.

Thanks to BinCut the lineage graph of *tsunami* now reveals new clusters of stripped samples, and can correctly show relationships among binaries that were packed with UPX (but unpacked for the analysis). In particular, the regions marked as [A] and [B] in Figure 5.4 contain stripped samples of the *muhstik* family. Muhstik is a Linux botnet known to infect GPON routers, which recently started to attack also web services. It is made of two components: a scanner to attack vulnerable devices (variant [A]) and an implant to execute malicious actions on the infected machine (variant [B]). After a detailed manual analysis of these samples, we can confirm that the lineage graph can precisely differentiate between the scanner and the implant components, recognizes different time-frames of development, and

spots new functionalities introduced over time.

We also believe that current Antivirus Softwares classify these malware samples with the wrong label. Most of them use *tsunami* probably because *muhstik* implants ([B]) are known to reuse its code. Indeed these samples are connected to our lineage graph because the implant shares common code with *tsunami*. Other AVs instead use the label *PNScan*, especially for the scanner component ([A]). We believe this is a consequence of the fact that some *muhstik* samples integrate the open-source network scanner tool called PNScan. However, the scanner has a weak connection to the lineage graph for a BinCut inaccuracy, and not because it shares code with *tsunami*. We found only 3 functions with a similarity match but all of them belonging to *uClibc* code. On the contrary, we can correctly separate the two modules and aggregate similar samples.

5.5 Limitations

Our solution to identify the boundary between user and library code combines information extracted from function dependencies with the functions spatial layout in the binary object. Since a stripped statically linked binary does not offer any information about the number and location of library code, we had to resort to a set of heuristics to aid the user code boundary analysis.

In particular, our binary layout analysis step tries to detect independent code modules by making assumptions based on the number of backward and forward calls in the binary. In fact, in our experiments we saw that the call directionality depends on the linking order. While our approach cannot work in the extreme case of symmetric directionality, we believe that a binary completely based on circular dependencies would be impractical. Similarly, our approach would not handle correctly cases in which the user code objects are manually linked as both head and tail of a third-party library (in this situation the result provided by our tool for user code would include also the library in between). Even though such configurations are technically possible, we never encountered them in any of our experiments.

Another fundamental part of our analysis is based on the intra-module density of function calls. Functions that are defined but remain unreferenced decrease the density, thus lowering the precision of the boundary point. We incur a similar issue when the user portion of the code is very small compared to the size of the linked libraries. One example could be a statically linked executable with only a unique user routine.

Automated binary analysis is a key element when working on several

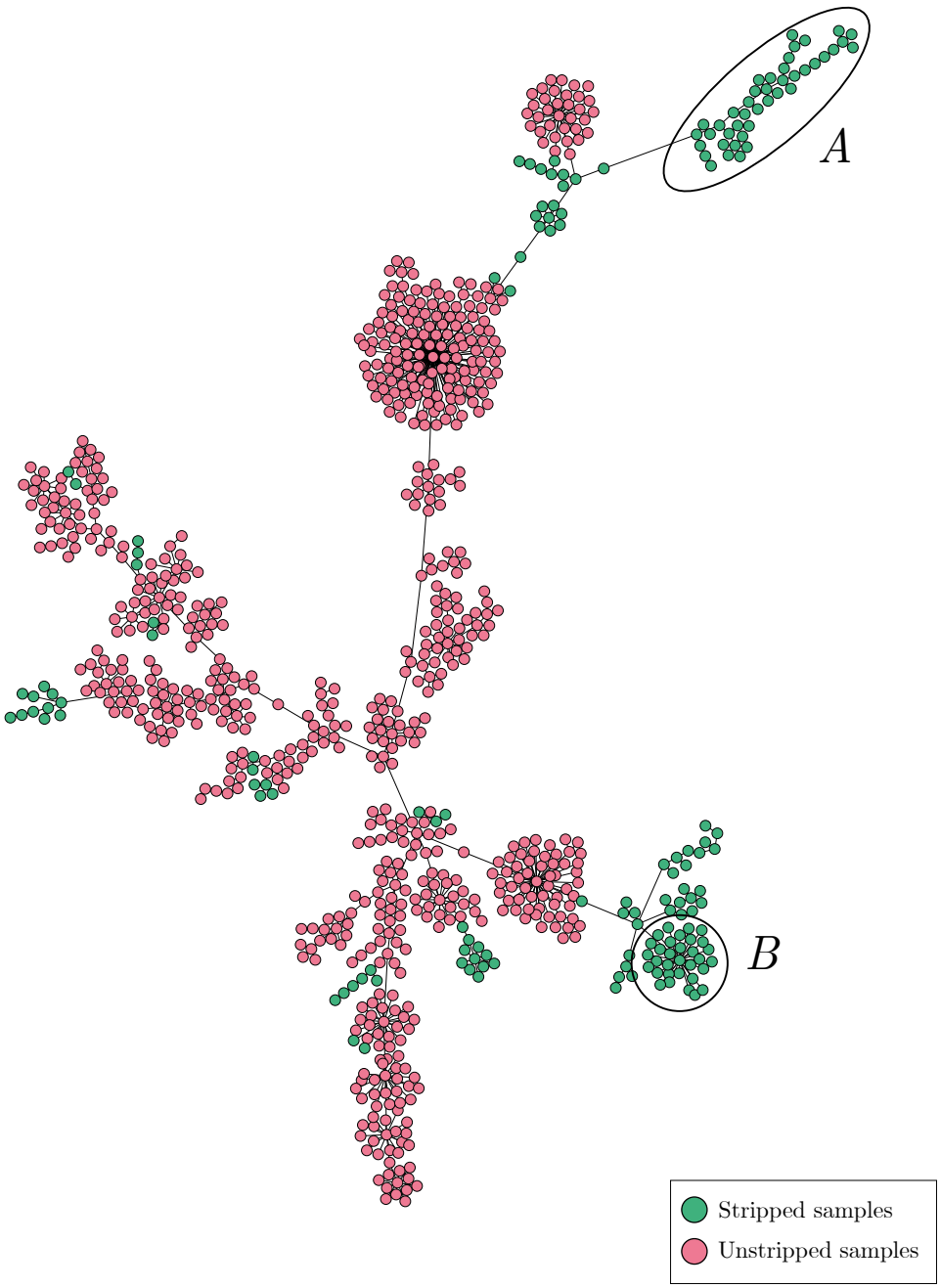


Figure 5.4: Lineage graph of *Tsunami* samples for *MIPS with* BinCut extension.

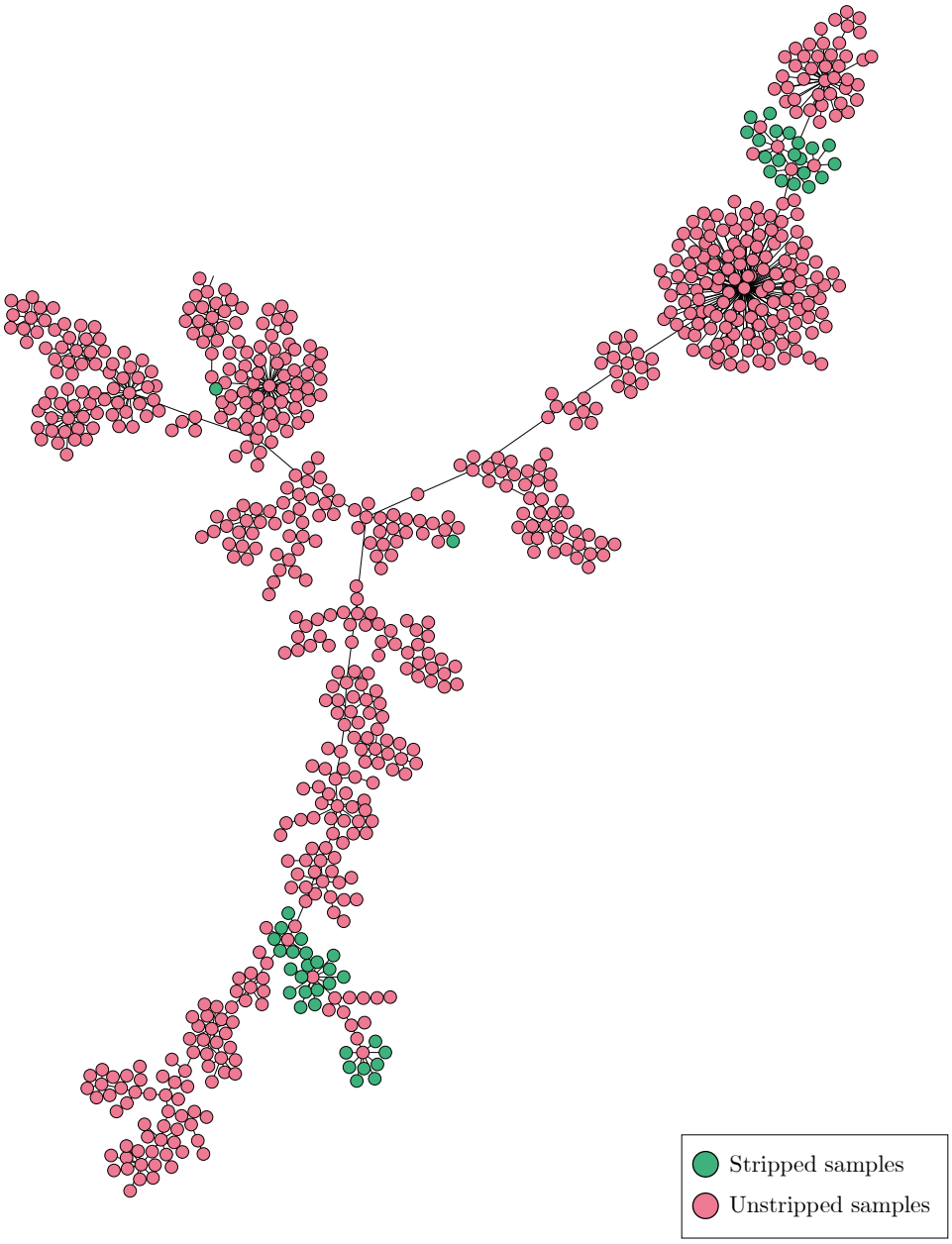


Figure 5.5: Lineage graph of *Tsunami* samples for *MIPS without* BinCut extension.

binaries. We used BinCut with a multi-class classifier so that the analyst is given only a unique cut point out of all the possibilities. The classification model that we trained on the malware dataset gives promising results. However, these binaries cannot be considered complex pieces of software nor representative of an average statically linked program. Even though we think that a more comprehensive dataset could allow building a generic classification model, unfortunately the nature of our classes (the heuristics for boundary identification) makes this a difficult task. Having balanced classes in the train set would require the analyst to compile several programs and manually inspect all the cut points returned by BinCut.

5.6 Conclusions

In this chapter we presented a novel approach to identify the boundary between user-defined and library code statically linked into the same executable image. We implemented our technique in a tool named BinCut that we believe can provide a useful support for both reverse engineering in general and malware analysis in particular. BinCut operates at the call graph level in order to recognize code modules dependencies by making considerations on their spatial locality. At the core of our approach, we analyze the geometric layout of function calls to find a set of cut points, and employ an extensible set of heuristics to reduce the false positives. Finally, we use a Random Forests classifier to select the most promising result among those returned by the different heuristics.

We evaluated BinCut on 222 statically linked ELF executables and over 10K malware samples (for which we had ground-truth information) and discuss both the results and the limitations. We then show how the use of Bincut can improve on the results of our previous lineage analysis study, by replacing the symbol propagation phase described in Chapter 4.4.

Chapter 6

Conclusion and Future Work

6.1 Future work

Linux and IoT malware are continuously evolving. It is undeniable that their current footprint is by no means comparable to the threat landscape of ten years ago.

If we combine the fact that interconnected devices will not be turned off any time soon with the recent interest from malware authors, we believe that the analysis of Linux-based malware will soon become an established research area, alongside the more traditional studies focusing on the Windows and Android ecosystems. Nowadays, Linux malware has also attracted the focus of nation-states and private intelligence agencies, often related to the development of debatable surveillance suites. For instance, in 2020 Amnesty International published a full report on *FinSpy* [amn20], a German-made spyware targeting Egyptian human rights defenders and supporting Windows, macOS, and Linux operating systems. This malware uses some of the persistence techniques we already covered in Chapter 3.

This thesis does not offer any final solution but wants instead to serve as a starting point for future research works in both academia and industry. Our experiments show that the analysis of Linux and IoT malware is a challenging task, even on today's rudimentary samples. Unfortunately, the ELF file format tells us the required CPU architecture but it remains agnostic about any other detail of the underlying system. To make things worse, when an analyst downloads samples from VirusTotal, she has no way of knowing their origin or intended targets and the binary themselves are not telling whether the analyst is dealing with desktop programs or with binaries designed to run on a smart speaker.

This problem remains open and in our study we had to resort to a number of best effort heuristics to filter unwanted programs like Android executables and detect the platform needed to run the remaining samples. Furthermore, there is still plenty of room to improve our dynamic analysis pipeline. While it is hard to properly configure the right execution environment of unknown devices, we can follow an iterative approach to create ad-hoc sandboxes. Our analysis pipeline already goes in this direction but only collects the dynamic libraries required to execute the sample. We can think of a malware which absolutely needs *systemd* for persistence, or a particular hardware peripheral attached to the emulator. If any of the two is not available, then the malware could fail to execute properly, and thus remain unexplored.

Throughout our work, we observed the rise of script-based malware, usually under the form of a *bash*, *python*, or *perl* scripts. These scripts

initially served as droppers for multi-stage executions, until they started to implement malicious behaviors directly. To the best of our knowledge, currently there are no works designed to comprehensively analyze script-based malware. Dynamically tracing the script interpreters would offer a quick and straightforward solution, but this approach would also introduce a significant amount of 'noise' to the results. Instead, we think it is possible to design systems that provide a single unified behavioral reports which only tracks the malicious behavior, as we did with our analysis pipeline for ELF malware.

The work presented in this thesis on the genealogy of IoT malware raised questions on the current effectiveness of AV products for Linux. In particular, we observed how a high ratio of code similarity can induce errors in AV signatures. The same holds for packed malware. While it is true that we did not see complex packers at the moment, an extremely simple compressor based on UPX can still cause troubles for automated systems. Unfortunately, the high volume of ELF malware is prohibitive to manually inspect potentially mislabelled or unclassified samples. Following this line of research, future contributions may want to look into systematic approaches to evaluate the precision of Linux-based AV software.

Moving to the code similarity, we had to tackle specific challenges to succeed in the analysis of statically linked and stripped ELF files. In the beginning we completely relied on symbol-based heuristics to exclude library code, and then proposed a new system to automatically identify user-code boundaries. Our goal was to give insights to the analyst about where the user-code and the libraries are located. However, our approach looks promising for future extensions. For example, it would be incredibly helpful to recognize the boundaries between the libraries themselves. Our binary layout analysis stage already work in this direction, even if a complete solution would require to address additional challenges. One interesting idea in this direction could be to convert the adjacency matrix of a call graph into a machine-learning assisted image processing problem.

6.2 Conclusion

This thesis presents the first comprehensive study of the feasibility and challenges of binary analysis for Linux and IoT malware.

First, in Chapter 3 we propose an automated analysis pipeline specifically tailored to the analysis of this emerging threat. The pipeline served as a basis to characterize, analyze, and understand the behavior of Linux malware samples. While the average complexity is still low, we already found samples that adopt more complex techniques borrowed from their Windows counterpart.

After studying how Linux malware implements malicious behaviors, in Chapter 4 we investigated their evolution over time and the myriads of variants that are created at a very fast pace. In particular, we systematically reconstruct the lineage of IoT malware families by using binary code similarity to show their tangled relationships and the difficulty for AVs to track them down correctly.

Last but not least, in Chapter 5 we focus our attention to the static analysis of Linux programs and we propose a method to extract and recognize user-defined code in statically linked binaries. Our experiments show that discarding third library functions highly improves both binary code similarity and reverse engineering efforts of malware.

I hope that my thesis will bring better visibility into malware infecting Linux-based operating systems and IoT devices, and will assist researchers and industries to further counteract this rapidly-increasing threat.

Appendices

Appendix A

French Summary

A.1 Introduction

Les systèmes d'exploitation Linux ont cessé d'être la plate-forme de niche comme le penserait l'utilisateur moyen d'un ordinateur. Le monde dans lequel nous vivons est également soutenu par les noyaux Linux, malgré la façon dont Windows est naturellement répandu sur la grande majorité des machines. Notre ère de l'"Internet des objets" pousse à l'adoption de machines à la sauce Linux, ce qui représente un grand pas en avant. Les dispositifs embarqués, dont la logique était autrefois entièrement intégrée au matériel, peuvent maintenant bénéficier d'une plus grande flexibilité et d'une réduction des coûts de production. La tendance est à l'utilisation d'une unité de traitement générique avec une couche logicielle de haut niveau - un système d'exploitation - par-dessus. D'autre part, la course à l'armement des systèmes IoT et Linux n'a pas pris la scène des logiciels malveillants au dépourvu.

Les auteurs de logiciels malveillants ont récemment manifesté leur intérêt pour une plus grande variété de systèmes d'exploitation (par exemple Linux) et pour des plateformes généralement construites sur un ensemble d'hypothèses et de propriétés qui n'existent pas dans les ordinateurs personnels (par exemple les systèmes embarqués et les dispositifs IoT.) De nos jours, les logiciels malveillants n'infectent pas seulement les ordinateurs personnels et les serveurs, mais se répandent par le biais de millions de dispositifs connectés en permanence, par exemple les routeurs ou les caméras IP. Alors que nos ordinateurs de bureau fonctionnent sur des architectures x86, les systèmes embarqués s'appuient sur des processeurs ARM, MIPS ou plus exotiques. Pire encore, il est de plus en plus fréquent de voir apparaître la même famille de logiciels malveillants sous Windows, mais aussi sous Linux et MacOS.

Les premiers virus et infecteurs Linux sont déjà apparus à la fin des années 90, sous forme de menaces publiques ou souvent réalisées dans le cadre de recherches privées. Les détails de bas niveau ont surtout vu le jour sur des sites web personnels et des zines techniques. Ce type de contenu a fasciné les responsables de la sécurité pendant des années, mais n'a jamais réussi à percer le marché et les sociétés informatiques.

Les échantillons de logiciels malveillants sont historiquement connus pour cibler le système d'exploitation Windows. C'est pourquoi nous nous sommes engagés dans la recherche sur les logiciels malveillants Windows au cours des deux dernières décennies, laissant un certain terrain libre aux personnes désireuses d'exploiter d'autres plateformes à leur avantage. La communauté des chercheurs a déployé de grands efforts dans l'analyse des

logiciels malveillants Windows : de l'analyse statique de fichiers binaires complexes aux méthodologies d'une analyse dynamique aussi exhaustive que possible. Toutefois, nous pourrions nous inspirer des connaissances acquises autour de Windows pour commencer enfin à nous attaquer au problème des logiciels malveillants de Linux. Cette transition n'est pas gratuite et nécessite de relever plusieurs défis spécifiques.

Mes recherches sont principalement basées sur l'étude d'échantillons binaires téléchargés sur VirusTotal[1] [urlb] entre 2015 et 2018, années où les logiciels malveillants de Linux ont commencé à exploser et où l'opinion publique a commencé à se comporter et à réagir en conséquence. Une fois que nous avons un flux d'échantillons de Linux, nous devons faire face à de multiples architectures et environnements d'exécution pour pouvoir les analyser. Nous devons gérer des échantillons, des chargeurs et des bibliothèques liés dynamiquement, mais aussi les binaires liés statiquement et dépouillés les plus exigeants. Nous voulons comprendre les comportements, les techniques, les propriétés qui les caractérisent, l'état actuel de l'empaquetage, de l'évasion ou de la persistance. Pouvons-nous mesurer tout cela ? Pouvons-nous l'automatiser ? De plus, le code source des familles de logiciels malveillants les plus célèbres est accessible au public depuis des années, parfois même avec des fuites. Quel est l'impact de la réutilisation du code dans les logiciels malveillants de Linux et de l'IoT sur les scanners antivirus (AV) ? Est-il possible de suivre les variantes d'une même famille ou les nouvelles familles provenant de plusieurs bases de code ? Alors que l'approche traditionnelle à cet effet est basée sur une mise en grappe statique et dynamique basée sur les caractéristiques, l'essence simpliste des logiciels malveillants Linux observée jusqu'à présent peut nécessiter des solutions sur mesure comme la mise en grappe basée sur le code. Cependant, pour réaliser la similarité du code dans un système où les programmes liés statiquement sont la norme plus qu'une exception, il faut répondre à des questions encore ouvertes. Le programme lié statiquement analysé comporte-t-il des bibliothèques de code intégrées ? Peut-on isoler la partie du code écrite par l'utilisateur (ou l'auteur du malware) du code de la bibliothèque ?

Tout cela a déclenché des problèmes en cours de route, des problèmes que cette thèse veut énoncer et souhaite aborder.
Ma thèse se veut un voyage à travers les questions soulevées ci-dessus. Un voyage qui se caractérise par des défis binaires de bas niveau (par exemple, l'analyse binaire) tout en passant par des études à grande échelle de milliers de logiciels malveillants. Les contributions rapportées dans ce manuscrit

[1] service en ligne où les utilisateurs peuvent télécharger des éléments à inspecter avec plus de 70 antivirus.

espèrent apporter un éclairage sur un sujet qui, à notre connaissance, n'a pas encore reçu l'attention qu'il mérite.

Nous commençons par les bases, avec le chapitre 2, en donnant au lecteur quelques informations sur le format de fichier *ELF*, utilisé dans les exécutables Linux, d'une importance vitale pour amorcer l'analyse. Nous prenons en compte les progrès actuels et l'état de l'art de l'analyse binaire et des logiciels malveillants sous Linux. Au chapitre 3, nous présentons un pipeline d'analyse des logiciels malveillants sous Linux. Nous utilisons ce pipeline pour mener la première étude de mesure à grande échelle et découvrir les astuces et les techniques utilisées par les logiciels malveillants du monde réel. Au chapitre 4, nous parlons de la reconstruction systématique de la généalogie des familles de logiciels malveillants de l'IoT grâce à l'utilisation de la similarité des codes binaires. Nous décrivons la fragmentation des familles IoT et les performances des étiquettes antivirus dans leur reconnaissance. Le chapitre 5 est une contribution au problème de la reconnaissance des fonctions et des bibliothèques dans les binaires dépouillés. Nous décrivons notre approche pour détecter les fonctions définies par l'utilisateur dans des échantillons liés statiquement et "découper" les binaires bénins et malveillants en conséquence, ce qui permet de supprimer le bruit du code de la bibliothèque dans les travaux d'analyse binaire et des logiciels malveillants. Enfin, nous apportons au chapitre 6 les conclusions, les leçons apprises et les suggestions pour les travaux futurs.

A.2 Comprendre les logiciels malveillants de Linux

Linux est une plate-forme majeure pour les ordinateurs de bureau et les serveurs, mais il fait l'objet d'un grand consensus, même en ce qui concerne les appareils de réseau et de nombreux petits appareils connectés. La révolution de l'IoT pousse les entreprises à adopter des solutions faciles à déployer, c'est-à-dire Linux, et le paysage des logiciels malveillants à réagir en conséquence. Le nombre étonnant de dispositifs mal sécurisés connectés à l'internet a récemment attiré l'attention des auteurs de logiciels malveillants. Des services comme VirusTotal, utilisé par la communauté pour partager et analyser des fichiers suspects, contiennent déjà des milliers et des milliers de binaires Linux potentiellement malveillants. L'industrie des antivirus a largement ignoré les programmes Linux malveillants et ce n'est qu'à la fin de 2014 que VirusTotal a reconnu qu'il s'agissait d'une préoccupation croissante pour la communauté de la sécurité [ZDn]. Pendant ce temps, les ressources disponibles sont souvent limitées aux articles de blog publiés par

des entreprises ou des chercheurs indépendants, et se limitent généralement à l'analyse manuelle d'échantillons spécifiques. Le monde universitaire a même été lent à réagir à ce changement, proposant peu de travaux systématiques comme l'étude d'Antonakakis et al. [AAB+17] qui se concentrent sur le comportement en réseau d'une seule famille, le botnet Mirai.

Nous souhaitons combler cette lacune en présentant la première étude empirique à grande échelle menée pour caractériser et comprendre les logiciels malveillants basés sur Linux, pour les appareils embarqués et les ordinateurs personnels. L'étude est basée sur un ensemble de données constitué de fichiers ELF exécutables malveillants de 10,548 pour Linux collectés auprès de VirusTotal entre novembre 2016 et novembre 2017, et couvrant plus de dix architectures différentes. Nous avons utilisé AVClass [SRKC16a] pour trouver un consensus sur les étiquettes des antivirus et pouvoir associer une famille (108 au total) à 83% des échantillons de notre ensemble de données.

L'analyse des programmes Linux nécessite de relever des défis qui sont un nouveau respect de l'analyse binaire de Windows :

- *Target diversity* - Il est généralement admis que le principal défi consiste à soutenir différentes architectures. Le fait que les logiciels malveillants basés sur Linux puissent cibler un ensemble très diversifié d'appareils complique grandement leur analyse. L'analyste doit porter différents composants spécifiques à l'architecture pour prendre en charge chaque architecture et doit tenir compte des informations intrinsèques contenues dans l'en-tête ELF. Par exemple, un programme Linux lié dynamiquement peut spécifier un *chargeur* arbitraire et s'attendra à ce que certaines bibliothèques soient disponibles dans le système cible. Une partie importante de notre ensemble de données est liée avec *uClibc* ou *musl*. Le format de fichier ELF est également utilisé dans d'autres systèmes d'exploitation compatibles ELF, tels qu'Android ou FreeBSD. En principe, nous pourrions distinguer les programmes ELF pour Linux en regardant le champ "OS-/ABI" dans leur en-tête. En pratique, cela n'aide que rarement et les noyaux Linux actuels ignorent même ce champ.

- *Static linking* - Dans un binaire lié statiquement, toutes les dépendances de la bibliothèque sont incluses dans celui-ci à la suite du processus de compilation. La liaison statique offre une portabilité car l'environnement cible n'a pas besoin d'installer les dépendances de la bibliothèque, mais rend les programmes difficiles à désosser. Il introduit également un autre défi beaucoup moins évident. Comme les appels API vers le système de niveau supérieur sont intégrés dans le

binaire, les programmes Linux peuvent planter en cours d'exécution si l'ABI du noyau est différente de ce qu'ils attendent (ou de ce qui a été fourni par le système cible).

- *Analysis environment* - Un bac à sable d'analyse idéal devrait imiter le plus fidèlement possible le système dans lequel l'échantillon analysé était censé fonctionner. La diversité des cibles ne couvre qu'une partie de la configuration de l'environnement. Un autre aspect est celui des privilèges avec lesquels le programme doit s'exécuter. Si les sandboxes typiques exécutent les échantillons en tant qu'utilisateur non privilégié, également pour empêcher un programme malveillant de modifier le sandbox, un malware IoT peut avoir besoin des privilèges root pour accéder à un périphérique particulier. En fait, les logiciels malveillants pour Linux sont souvent écrits en supposant que leur code s'exécutera avec les privilèges de l'utilisateur root.

- *L'absence d'études antérieures* - Ce domaine manque d'informations sur le fonctionnement des logiciels malveillants basés sur Linux et d'un ensemble de données complet non biaisé par rapport aux réseaux de zombies typiques capturés par les honeypots. Il n'est pas clair comment concevoir et mettre en œuvre un pipeline d'analyse spécifiquement adapté aux logiciels malveillants basés sur Linux. De plus, les outils d'analyse construits jusqu'à présent sont adaptés aux caractéristiques des échantillons de logiciels malveillants existants.

La conception et la mise en œuvre de notre pipeline d'analyse sont devenues un processus de suivi et d'erreur pour surmonter les défis décrits ci-dessus. Notre pipeline comprend un ensemble de solutions de pointe existantes (telles que AVClass, IDA Pro, radare2 et Nucleus [ASB17]) et est basé sur trois modules : analyse des fichiers et des métadonnées, analyse statique et analyse dynamique. À notre connaissance, il s'agit du premier travail d'analyse complète du paysage des logiciels malveillants sous Linux.

A.2.1 Analyse de l'infrastructure

L'analyse commence par l'inspection de l'en-tête de l'ELF. Nous mettons en œuvre notre analyseur personnalisé pour le format ELF car les solutions existantes étaient souvent incapables de traiter un en-tête malformé. Nous extrayons un ensemble d'informations de chaque fichier pour comprendre tout d'abord l'architecture matérielle. Nous excluons ensuite les fichiers non pertinents pour notre analyse comme les bibliothèques partagées, les fichiers corrompus ou les exécutables compilés pour d'autres systèmes

d'exploitation. En même temps, les structures de fichiers anormales peuvent être utilisées comme des routines d'anti-analyse ou empêcher les outils existants de traiter correctement le malware. Enfin, nous avons inséré AVClass dans le cadre de ce module. Nous avons collecté chaque rapport VirusTotal pour chaque échantillon de l'ensemble de données afin d'extraire toutes les étiquettes AV pour obtenir un nom normalisé pour la famille de logiciels malveillants.

Le module d'analyse statique comprend deux tâches : l'analyse du code binaire et l'identification de l'emballage. La première tâche s'appuie sur des scripts IDA Pro personnalisés pour extraire plusieurs mesures de code, des astuces d'assemblage et des mesures agrégées telles que l'entropie glissante des différentes sections de code et de données. Ces informations sont utilisées à des fins statistiques, mais aussi dans d'autres composantes de l'analyse, par exemple pour identifier les comportements anti-analytiques ou les échantillons emballés. La deuxième tâche de cette phase d'analyse combine toutes les informations extraites jusqu'à présent pour identifier les binaires ELF probablement emballés. Nous avons procédé à un déballage statique de tous les échantillons pour lesquels cela était possible, par exemple des fichiers compressés avec UPX. Les nouveaux fichiers obtenus à ce stade ont été renvoyés au module d'analyse statique. Les échantillons que nous n'avons pas pu décompresser ont été marqués pour une tentative dynamique plus fine.

Le module d'analyse dynamique est divisé en deux tâches principales. D'une part, nous effectuons une analyse personnalisée de l'emballage et une tentative de déballage, tandis que d'autre part, nous effectuons une exécution complète du logiciel malveillant à l'intérieur d'un émulateur instrumenté. Nous avons préparé des sandboxes virtualisés KVM pour les architectures x86 et x86-64 et des sandboxes émulés basés sur QEMU pour les programmes ARM, MIPS et PowerPC. Chaque machine dispose de plusieurs snapshots correspondant à une configuration différente au choix (par exemple, exécution par l'utilisateur ou par le super-utilisateur, configuration des chargeurs). Ces cinq sandboxes ont été imbriqués à l'intérieur d'une VM externe dédiée à l'envoi de chaque échantillon en fonction de son architecture et de l'environnement nécessaire. Toutes les sandboxes s'appuient sur SystemTap pour mettre en œuvre les sondes du noyau (*kprobes*) et les sondes de l'utilisateur (*uprobes*) et collecter chaque appel système, avec ses paramètres et son résultat, ainsi que des informations supplémentaires sur les fonctions de manipulation des chaînes et de la mémoire lorsque cela est possible.

Chaque sandbox renvoie une trace complète qui rend compte du com-

portement du malware qui est analysé pour en extraire des informations en retour. Nous pouvons identifier les composants manquants, détecter si un échantillon a testé la permission de l'utilisateur ou tenté d'effectuer une action qui a échoué en raison de permissions insuffisantes. Dans ce cas, le module dynamique répète immédiatement l'analyse en sélectionnant la machine avec la configuration racine.

Enfin, nous avons développé à partir de zéro un minuscule émulateur pour décompresser dynamiquement les variantes inconnues d'UPX. L'émulateur—multi-architecture–mimique un ensemble limité d'appels système utilisés par UPX pendant le processus de déballage. Cette approche nous a permis de décompresser automatiquement tous les échantillons de logiciels malveillants, sauf trois, dans notre ensemble de données.

A.2.2 Sous le capot

Nous fournissons des statistiques détaillées et une discussion des comportements intéressants que nous avons identifiés pour mieux comprendre comment fonctionnent les logiciels malveillants basés sur Linux. Nous espérons que les informations que nous offrons serviront de référence pour de futures recherches visant à améliorer l'analyse de ce type de logiciels malveillants.

Les développeurs de logiciels malveillants manipulent souvent les entêtes ELF pour tromper l'analyste ou faire planter les outils d'analyse courants. Nous avons identifié deux catégories de modifications : Les fichiers *anomaux* (mais qui suivent toujours les spécifications ELF), et les fichiers *invalides*–qui peuvent cependant toujours être exécutés correctement. La première classe est représentée par 5% des échantillons de notre base de données, la seconde par 2%. Si les outils d'analyse peuvent traiter des fichiers anormaux, ils entraînent souvent des erreurs lorsqu'ils traitent des fichiers non valides.

Les binaires malveillants peuvent modifier la configuration du système infecté afin de pouvoir fonctionner indépendamment des éventuelles opérations de redémarrage et de mise hors tension. Nous appelons ce comportement *persistance*, vu dans 21% de l'ensemble des données. L'approche prédominante adoptée par les auteurs de logiciels malveillants est l'initialisation du sous-système (plus de 1000 échantillons), en utilisant le système Linux *init* ou *systemd*. Le deuxième choix couramment utilisé pour la persistance est l'exécution basée sur le temps lorsqu'un malware modifie les fichiers de configuration *cron* pour obtenir une exécution programmée à un intervalle de temps fixe. Une autre approche pour maintenir une emprise sur le système consiste à remplacer les applications qui existent déjà dans la

cible. Très peu d'exemples modifient à la place les fichiers de configuration utilisateur, comme les configurations shell.

Nous appelons *deception* la technique par laquelle un malware tente de cacher sa nature en assumant des noms qui semblent authentiques au premier abord. Ce comportement, déjà courant sur les systèmes d'exploitation Windows, est également répandu sur les logiciels malveillants basés sur Linux. Plus de 50% des échantillons ont pris des noms différents lors de leur exécution. Parmi ceux-ci, 11% ont adopté des noms tirés d'utilitaires courants et les autres ont adopté soit un nom vide, soit le nom d'un fichier fictif, soit un nom d'apparence aléatoire.

Nous avons doublement exécuté les 25% des échantillons de l'ensemble de données, d'abord dans un bac à sable non privilégié, puis dans un bac à sable privilégié avec des autorisations de base. Nous avons détecté des différences dans le comportement des échantillons dans 89% d'entre eux. Les commandes et les opérations privilégiées sur les fichiers sont prédominantes, les logiciels malveillants utilisant la racine pour créer ou supprimer des fichiers dans des dossiers protégés. De plus, nous avons trouvé des binaires qui, en cas de détection de l'environnement d'exécution émulé, tueraient le démon SSH ou même supprimeraient l'ensemble du système de fichiers.

Le runtime packing est une technique d'obscurcissement courante qui rend plus difficile toute tentative d'analyse statique d'un malware. Vanilla UPX et ses variantes sont de loin la forme de conditionnement la plus répandue dans notre ensemble de données. Sur les 380 binaires de conditionnement, seuls trois ont mis en œuvre une forme de conditionnement personnalisé ne se référant pas à UPX. Près de 200 échantillons ont plutôt apporté des modifications au format UPX dans l'intention de casser l'outil de déballage UPX standard.

Une autre technique importante très courante dans les logiciels malveillants Windows est l'évasion, pour cacher le comportement malveillant et rester non détecté aussi longtemps que possible. Les logiciels malveillants sous Linux ne font pas exception à la règle. Nous regroupons quatre types de techniques d'évasion : la détection par bac à sable, l'énumération des processus, l'anti-débogage et l'anti-exécution. Nos sondes de l'espace utilisateur détectent les programmes qui tentent de s'échapper des environnements VMware et QEMU en lisant les informations de la zone DMI de la carte mère. Les logiciels malveillants peuvent également détecter les prisons basées sur *chroot*, les conteneurs OpenVZ ou l'hyperviseur XEN à partir du système de fichiers *procfs*. Notre système de traçage est basé sur des sondes de noyau, il ne peut donc pas être détecté ou altéré par l'utilisation de

techniques anti-débogage. Nous avons vu des logiciels malveillants utiliser l'appel système `ptrace` pour détecter si un autre débogueur est déjà attaché. Un échantillon vérifie même la présence de la variable d'environnement `LD_PRELOAD`, qui est souvent utilisée pour remplacer des fonctions dans des bibliothèques chargées dynamiquement.

A.3 L'enchevêtrement de la généalogie des logiciels malveillants IoT

Les botnets traditionnels et les outils DDoS cohabitent désormais avec les crypto-mines, les logiciels espions, les logiciels de rançon et les échantillons de cibles conçus pour mener le cyber-espionnage. Pour aggraver les choses, la disponibilité publique du code source associé à certaines des principales familles de logiciels malveillants de l'IoT a ouvert la voie à des myriades de variantes et d'enchevêtrements de similarités et de réutilisation du code. On sait encore peu de choses sur la dynamique qui sous-tend l'émergence de nouvelles souches de logiciels malveillants et, aujourd'hui encore, les logiciels malveillants de l'IoT sont classés en fonction des étiquettes attribuées par les éditeurs de logiciels audiovisuels. On ne sait pas exactement combien de variantes du botnet *Mirai* ont été observées ni quelles sont les relations internes qui relient les familles populaires entre elles. Nous cherchons à combler cette lacune en proposant un moyen systématique de comparer des échantillons de logiciels malveillants de l'IoT et d'afficher leur évolution dans un ensemble de graphiques de lignage faciles à comprendre.

Tout d'abord, nous présentons notre approche pour reconstituer la lignée des familles de logiciels malveillants de l'IoT et suivre leur évolution. Nous avons identifié les variantes de chaque famille ainsi que les relations intra-familiales qui se produisent en raison de la réutilisation du code. Nous rendons ensuite compte des connaissances acquises en appliquant notre approche à des milliers d'échantillons binaires. Les graphiques de lignage nous ont permis de découvrir plus d'une centaine d'échantillons mal étiquetés et d'attribuer le nom propre à ceux pour lesquels les produits AV n'ont pas fait l'objet d'un consensus.

Pour étudier la généalogie des logiciels malveillants de l'IoT, nous avons téléchargé tous les binaires ELF qui ont été soumis à VirusTotal pendant près de quatre ans (de janvier 2015 à août 2018) et signalés comme malveillants par au moins cinq antivirus. Après avoir filtré les programmes non pensés pour les appareils IoT, tels que les ordinateurs de bureau Android ou Linux, notre ensemble de données a donné 93 652 échantillons prêts à être analysés.

Déterminer un graphique de lignage précis est une tâche difficile, qui, dans les études précédentes, a souvent été réalisée à l'aide d'une analyse manuelle et sur un petit nombre limité d'échantillons [LDFM+12, HYD17]. Étant donné l'échelle de notre ensemble de données, un ordre de grandeur plus grand que les études précédentes, nous devons nous appuyer sur une solution entièrement automatisée. Nous profitons de la sophistication actuelle (rudimentaire) des logiciels malveillants de l'IoT et de l'absence générale d'obscurcissement du code pour recourir à une analyse plus précise basée sur la similarité au niveau du code et le regroupement des codes. Notre processus d'analyse peut être divisé en quatre macro-zones. Tout d'abord, nous traitons les binaires non dépouillés et nous analysons les symboles pour localiser le code de la bibliothèque dans les fichiers liés statiquement. Ensuite, nous effectuons une mise en grappes progressive basée sur la similarité au niveau du code tout en propageant les symboles à chaque nouvel échantillon. Enfin, nous construisons les graphiques de famille et nous utilisons les symboles disponibles pour épingler les échantillons et les grappes pour coder les bribes que nous avons pu extraire des sites web en ligne afin d'obtenir une compréhension plus détaillée de l'évolution des familles de logiciels malveillants.

A.3.1 Extraction du graphe de lignage des logiciels malveillants

Les logiciels malveillants IoT sont souvent expédiés en liaison statique, une hypothèse confirmée par notre jeu de données avec plus de 94% des échantillons en liaison statique. Cela est très probablement dû à un effort pour s'assurer que les échantillons peuvent fonctionner sur des appareils ayant des configurations système différentes. En même temps, il est difficile de réaliser la similarité du code sur ces binaires, car deux échantillons seraient considérés comme très similaires simplement parce qu'ils pourraient inclure une bibliothèque de code. Si les programmes sont dépouillés, l'analyse devient encore plus difficile.

Nous commençons notre analyse en extrayant des symboles de binaires non dépouillés et utilisons une heuristique simple pour couper le binaire en deux. L'idée est de localiser un code de bibliothèque, puis de considérer également tout ce qui vient après le code de bibliothèque. Nous avons construit une base de données de symboles extraits de différentes versions de *Glibc* et *uClibc*, puisque le *libc* est toujours inclus par défaut par les compilateurs et que moins de 2% des échantillons liés dynamiquement nécessitent d'autres bibliothèques tierces. Après avoir extrait les symboles de fonc-

tion des binaires ELF non dépouillés, nous commençons à les analyser de manière linéaire à l'aide d'une fenêtre coulissante. Le point de découpage est défini dès que tous les noms de fonctions dans cette fenêtre ont une correspondance positive dans la base de données des symboles.

La diffraction binaire est au cœur de notre approche car elle nous permet d'évaluer la similarité entre les binaires au niveau du code. Malheureusement, étant donné la taille de notre ensemble de données, le calcul d'une matrice de similarité complète est impossible. Nous atténuons ce problème grâce aux graphiques HNSW (Hierarchical Navigable Small World graphs) [MY18], une structure de données efficace pour la découverte approximative du plus proche voisin dans des espaces non métriques. L'idée de base qui accélère cette approche est que les objets ne sont comparés qu'aux voisins des voisins précédemment découverts, ce qui limite considérablement le nombre de comparaisons tout en maintenant une grande précision.

Nous utilisons *Diaphora* [urla] pour définir notre fonction de dissimilarité pour HNSW. De plus, Diaphora fonctionne avec toutes les architectures supportées par IDA Pro, alors que d'autres outils de dissimilitude également proposés par le monde universitaire ne gèrent que quelques architectures. Nous agrégeons les scores des fonctions individuelles en comptant le nombre de fonctions dont la similarité est supérieure à 0,5, ce qui est le seuil suggéré par les auteurs de Diaphora pour écarter les résultats *non fiable*. Pour HNSW, nous indiquons l'inverse de ce comptage pour traduire la valeur en une distance.

Les échantillons sont ajoutés à HNSW un par un, en deux tours, triés selon leur premier horodatage vu sur VirusTotal. Dans le premier cycle, nous avons ajouté tous les échantillons liés ou non dynamiquement, qui représentent 55% de l'ensemble des données. Nous nous sommes appuyés sur la phase précédente d'extraction des symboles pour effectuer la diffraction binaire sur la partie du code définie par l'utilisateur, et nous avons omis les comparaisons sur le code de la bibliothèque. Dans la deuxième phase, nous avons ensuite ajouté tous les échantillons dépouillés liés statiquement. À ce stade, nous utilisons la diffraction binaire elle-même pour propager itérativement les symboles. Lorsqu'une fonction dans un échantillon dépouillé présente une similitude parfaite avec un échantillon non dépouillé, nous l'étiquetons avec le même symbole.

La dernière étape de notre analyse de similarité implique la génération d'arbres phylogénétiques des logiciels malveillants de l'IoT. Nous avons post-traité le graphique de similarité épars obtenu en exécutant HNSW et en utilisant la fonction de distance comme poids. Comme le graphe de similarité contient un grand nombre d'arêtes, nous visualisons dans notre graphe

l'arbre à portée minimale (MST) du graphe HNSW. Cette approche nous permet de rendre le résultat plus lisible et de mieux mettre en évidence les lignes d'évolution.

L'arbre peut être coloré selon les étiquettes AV (pour avoir une vue d'ensemble des relations entre les différentes familles et repérer les étiquettes erronées) ou selon le fichier source le plus proche que nous avons téléchargé en utilisant les noms des symboles.

A.3.2 Résultats

Nous avons utilisé le flux de travail pour le regroupement basé sur le code afin de tracer des arbres phylogénétiques pour les six principales architectures de notre ensemble de données. La scène actuelle des logiciels malveillants de l'IoT est principalement envahie par trois familles étroitement liées : *Gafgyt*, *Mirai* et *Tsunami*. Leurs centaines de variantes sont regroupées sous la même étiquette AV, partageant parfois des traits provenant de deux familles ou plus. Les familles mineures empruntent également des codes aux trois principaux acteurs. Par exemple, *DnsAmp* réutilise le code pour la génération de nombres aléatoires et les calculs de checksum, ou *Lightaidra* réutilise 18 fonctions de *Gafgyt*. Les campagnes ciblées comme *VPNFilter* ne chevauchent pas, au contraire, les principaux éléments des familles célèbres.

Nous pouvons utiliser les arbres phylogénétiques pour détecter les étiquettes *anomalie*, en recherchant les nœuds aberrants dans le graphique. Les valeurs aberrantes peuvent correspondre à des échantillons mal classés par la majorité des scanners AV ou à des variantes d'une famille donnée qui ont une quantité considérable de code en commun avec une autre famille. Au total, nous avons trouvé 118 cas avec 62 échantillons dont nous pensons qu'ils ont une étiquette AVClass erronée et 56 pour lesquels AVClass n'a pas pu se mettre d'accord sur l'étiquette AV.

Les arbres phylogénétiques aident à identifier les modifications fines et les relations entre les variantes au sein d'une même famille de logiciels malveillants. Dans un premier temps, nous identifions les variantes candidates en regroupant tous les échantillons de logiciels malveillants en fonction de leur ensemble de symboles uniques. Cela a permis de suivre plus de 2000 variantes de *Gafgyt*, et dans une moindre mesure de *Mirai* et *Tsunami*. Dans un deuxième temps, nous nous appuyons sur le code source collecté à partir de dépôts en ligne pour valider plus de 200 variantes précédemment identifiées. Enfin, nous avons combiné les résultats de notre analyse avec des informations sur le calendrier pour mesurer la première et la dernière fois que chaque variante est apparue dans la nature. Alors que les familles à évolution rapide comme *Gafgyt* et *Mirai* ont tendance à donner des variantes

de courte durée, les variantes *Tsunami* et *DnsAmp* ont persisté pendant des périodes plus longues.

A.4 Identification du code utilisateur dans les binaires liés statiquement

L'ingénierie inverse des fichiers binaires est une tâche difficile qu'il n'est pas toujours possible d'automatiser. Dans le contexte des fichiers ELF et du code binaire, le degré de difficulté dépend des choix du programmeur pendant les phases de développement et de compilation. Les compilateurs optimisent le code, le transforment ou appliquent des couches d'obfuscation pour entraver l'analyse statique. Les compilateurs génèrent également des structures de données spécifiques contenant des informations de débogage (DWARF) pour aider au débogage et faciliter l'inspection du programme résultant. Les outils de sécurité comme IDA Pro utilisent les informations de débogage pour obtenir une décompilation presque parfaite du code. En revanche, les développeurs choisissent souvent de supprimer les symboles et les noms de fonctions pour réduire la taille du fichier ELF final ou pour masquer son identité. Les programmes ELF liés statiquement ajoutent un certain degré de difficulté à la rétro-ingénierie, surtout lorsqu'ils sont dépouillés. Dans ce cas, l'analyste traite souvent d'énormes morceaux de code sans en connaître les limites sous-jacentes. Tout indice permettant de reconnaître des fonctions liées à des bibliothèques externes, isolant ainsi le code défini par l'utilisateur, accélérerait certainement les travaux d'analyse binaire.

Les techniques actuelles visant à récupérer les informations ne permettent pas de résoudre directement le problème de la séparation du code de la bibliothèque et du code de l'utilisateur. Elles peuvent plutôt être utilisées indirectement à cette fin, parfois avec une faible précision ou un coût élevé en termes de complexité et d'évolutivité. Nous avons vu des efforts pour la reconnaissance de fonctions basées sur des signatures [Gui], des approches d'apprentissage machine [HIT+18], et des techniques d'analyse basées sur des graphes [SWD17, AWD16, DR05]. L'un des travaux les plus récents s'appuie plutôt sur la directivité des appels de fonction pour récupérer les limites des objets

Avec ce travail, nous présentons notre approche d'analyse binaire pour déduire un point *cut* entre le code défini par l'utilisateur et le code de la bibliothèque dans les exécutables ELF liés statiquement. Nous avons implémenté notre système dans un outil et évalué sa précision et ses performances sur un large éventail de binaires du système d'exploitation Debian et sur des

projets complexes comprenant des milliers de fonctions de plusieurs bibliothèques.

Notre technique tente de réduire les relations entre fonctions en un problème de reconnaissance de la mise en page et s'appuie sur un ensemble d'heuristiques pour définir les points de coupure les plus probables dans un binaire lié statiquement. Nous recueillons finalement les caractéristiques extraites lors de l'analyse du programme pour alimenter un algorithme de classification et obtenir les limites du code utilisateur à partir de l'heuristique qui se comporte le mieux.

A.4.1 Analyse des limites de code

Notre système de reconnaissance des limites de code est basé sur deux modules principaux. Premièrement, nous décompilons le binaire pour reconstruire son graphe d'appel et capturer l'adresse de la fonction `main`. Ensuite, nous construisons une matrice de contiguïté avec les fonctions *caller* et *callee*. Nous effectuons notre analyse des fonctions en étudiant la "géométrie" des programmes. L'idée de base est que les fonctions appartenant au même module (ou bibliothèque) appelleront plus de fonctions voisines que les fonctions des autres modules. En conséquence, chaque module de code doit être naturellement regroupé dans une zone limitée de l'espace d'adressage du programme.

Nous utilisons IDA Pro pour décompiler les fichiers ELF et détecter les adresses de début de fonction. Nous analysons ensuite les références croisées de et vers chaque fonction pour construire le graphe d'appel du binaire. Comme un programme pourrait être compilé avec du code exécutable dans les sections de données, et considérant que nous voulons analyser le binaire de manière statique, nous ne considérons que les fonctions détectées dans la section *.text*. En outre, nous récupérons l'adresse de la fonction `main` et l'utilisons comme indice sur l'endroit où se trouve le code défini par l'utilisateur.

L'analyse *cut* part de la matrice de contiguïté du graphe d'appel. Nous indexons les lignes et les colonnes de la matrice avec un numéro pour chaque fonction au lieu de leurs adresses. Les adresses apportent la connaissance de la taille des fonctions, mais nous voulons considérer de la même manière une petite zone de code utilisateur et une grande bibliothèque (ou le contraire). De plus, si nous considérons que le graphe d'appel est dirigé, nous pouvons extraire la directivité des appels de fonction de la matrice de contiguïté. Lorsque la matrice triangulaire supérieure comporte plus d'entrées que la matrice triangulaire inférieure, alors le binaire fait surtout des appels de fonction dirigés. En d'autres termes, les fonctions appellent principalement

des adresses plus élevées. Au contraire, une matrice triangulaire inférieure plus dense signifie que les appels de fonction en arrière sont plus fréquents.

L'analyse est basée sur la recherche d'ensembles de fonctions contiguës n'appelant aucune autre partie contiguë du code. Nous analysons la matrice de contiguïté par ligne et par colonne pour rechercher la plus grande région vide avec un coin sur la diagonale. Nous définissons ce coin comme un point de coupe candidat. Les fonctions appartenant à un même module ont tendance à être plus localisées, ce qui signifie que - dans une situation idéale - les modules sont représentés dans la matrice sous la forme d'un motif en escalier le long de la diagonale. Nous poursuivons l'analyse de manière récursive en effectuant les recherches suivantes sur les deux sous-matrices définies par le point de coupe candidat précédent. Une région vide (et son point de coupe candidat) n'est valable que si elle respecte certaines restrictions que nous imposons, comme la densité des sous-matrices, la zone qu'elle couvre, ou si la matrice triangulaire supérieure ou inférieure est vide. Lorsqu'une région est invalide, nous poursuivons la recherche au même niveau de récursivité.

L'algorithme stocke tous les points candidats dans un arbre binaire où le nœud racine contient le point lié à la plus grande zone vide dans la matrice de contiguïté, tandis que les feuilles représentent les points des plus petites régions.

Enfin, nous définissons quelques heuristiques pour obtenir les trois fonctions les plus probables où le binaire devrait être coupé pour diviser le code utilisateur des bibliothèques en conséquence. Par exemple, nous avons constaté que dans certains binaires ELF, le point de coupure se trouve dans la plus grande région vide après la fonction `main`. Au lieu de cela, nous sélectionnons le premier point à côté de `main` lorsque le code utilisateur est très court. Dans certains cas, nous avons sélectionné le point de coupure en fonction des accès aux variables globales : l'idée est que chaque module de code fonctionne souvent sur son espace de données restreint. Nous avons testé notre système sur plus de 200 fichiers ELF liés statiquement et contenant des centaines, voire des milliers de fonctions, dont certaines avec des personnalisations de compilation plus complexes. Dans l'ensemble, notre outil a pu trouver les limites du code utilisateur avec une distance moyenne par rapport à la fonction réelle inférieure à 2%.

A.4.2 Collection et classification des caractéristiques

En fonction de la manière dont un programme ELF a été compilé, de la quantité d'appels en avant et en arrière, et de ses caractéristiques internes, une heuristique retournera un résultat plus précis que les autres. C'est pourquoi nous utilisons l'apprentissage machine pour classer un binaire en

fonction des caractéristiques que nous avons recueillies, et nous ne produisons que le point de coupure donné par l'heuristique qui se comporte le mieux.

Nous utilisons comme caractéristiques le nombre de fonctions, des métriques sur la géométrie du binaire, et des informations dérivées d'un nombre limité de nœuds de l'arbre binaire construit pendant la recherche récursive. Nous prenons en considération le point de coupe candidat et la zone de la région vide associée, la hauteur du nœud dans l'arbre binaire et le nombre maximum de variables globales utilisées dans la zone adjacente à la région vide. Les caractéristiques alimentent un classificateur d'arbre de décision formé avec 70% des échantillons, et finalement testé avec les 30% restants. Nous évaluons le modèle en utilisant la validation croisée K-fold et rendons compte des résultats, de l'exactitude et de la précision de l'outil.

A.5 Conclusion

Dans ma thèse, j'explore les us et coutumes de Linux et des logiciels malveillants de l'IoT et je propose des moyens de les analyser de manière statique et dynamique. Tout d'abord, je propose un pipeline d'analyse automatisé spécifiquement adapté aux logiciels malveillants de Linux. Ce pipeline a servi de base pour les caractériser, les analyser et les comprendre. Après avoir étudié comment les logiciels malveillants Linux mettent en œuvre des comportements malveillants, j'ai étudié leur évolution dans le temps. Je reconstruis systématiquement la lignée des familles de logiciels malveillants de l'IoT et leurs relations enchevêtrées en utilisant la similarité du code binaire. Enfin, je me concentre sur l'analyse statique des programmes Linux et propose une méthode pour reconnaître et extraire le code défini par l'utilisateur dans des binaires liés statiquement. L'élimination des fonctions des bibliothèques système améliore considérablement la similarité du code binaire et les efforts de rétro-ingénierie des logiciels malveillants.

J'espère que ma thèse apportera une meilleure visibilité sur les logiciels malveillants qui infectent les systèmes d'exploitation basés sur Linux et les dispositifs IoT, et qu'elle aidera les chercheurs et les industries à contrer cette menace toujours croissante.

References

[AAB+17] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *USENIX Security*, 2017.

[ABKY88] T. Asano, B. Bhattacharya, M. Keil, and F. Yao. Clustering algorithms based on minimum and maximum spanning trees. In *Proceedings of the Fourth Annual Symposium on Computational Geometry*, SCG '88, pages 252–257, New York, NY, USA, 1988. Association for Computing Machinery.

[Ale] Alexander Bartolich. The ELF Virus Writing HOWTO. http://www.linuxsecurity.com/resource_files/documentation/virus-writing-HOWTO/_html/index.html.

[amn20] German-made finspy spyware found in egypt, and mac and linux versions revealed. https://www.amnesty.org/en/latest/research/2020/09/german-made-finspy-spyware-found-in-egypt-and-mac-and-linux-versions-revealed/, 2020. Accessed: 2020-09-26.

[Ant17] Antonakakis et al. Understanding the Mirai Botnet. In *Proceedings of the USENIX Security Symposium*, 2017.

[anu] Anubis. https://anubis.iseclab.org.

[ASB17] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-Agnostic Function Detection in Binaries. In *IEEE European Symposium on Security and Privacy*, 2017.

[ASWD18] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Fossil: a resilient and efficient system for identifying foss functions in malware binaries. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):1–34, 2018.

[AWD16] Saed Alrabaee, Lingyu Wang, and Mourad Debbabi. Bingold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs). *Digital Investigation*, 18:S11–S22, 2016.

[BCH+09] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, 2009.

[Bit18] BitDefender. New Hide 'N Seek IoT Botnet using custom-built Peer-to-Peer communication spotted in the wild. https://labs.bitdefender.com/2018/01/new-hide-n-seek-iot-botnet-using-custom-built-peer-to-peer-communication-spotted-in-the-wild/, January 2018.

[BJAS11] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[Ble19] BleepingComputer. Cr1ptTor Ransomware Infects D-Link NAS Devices, Targets Embedded Systems. https://www.bleepingcomputer.com/news/security/cr1pttor-ransomware-infects-d-link-nas-devices-targets-embedded-systems/, February 2019.

[BOA+07] Michael Bailey, Jon Oberheide, Jon Andersen, Z Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *RAID*, 2007.

[BYMM13] Jinrong Bai, Yanrong Yang, Shiguang Mu, and Yu Ma. Malware detection through mining symbol table of Linux executables. *Information Technology Journal*, 2013.

[CAA+19] Jinchun Choi, Afsah Anwar, Hisham Alasmary, Jeffrey Spaulding, DaeHun Nyang, and Aziz Mohaisen. Iot malware ecosystem in the wild: a glimpse into analysis and exposures. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 413–418, 2019.

[Cat] Cathal, Mullaney and Sayali, Kulkarni. VB2014 paper: Linux-based Apache malware infections: biting the hand that serves us all. https://www.virusbulletin.com/virusbulletin/

2016/01/paper-linux-based-apache-malware-infections-biting-hand-serves-us-all/.

[CGFB18] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding Linux Malware. In *IEEE S&P*, 2018.

[CJS+05] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, SP '05, 2005.

[CKVD10] Pavel Celeda, Radek Krejci, Jan Vykopal, and Martin Drasar. Embedded malware-an analysis of the chuck norris botnet. In *Computer Network Defense (EC2ND), 2010 European Conference on*, pages 3–10. IEEE, 2010.

[CMM+19] Simone Coltellese, Fabrizio Maria Maggi, Andrea Marrella, Luca Massarelli, and Leonardo Querzoni. Triage of iot attacks through process mining. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 326–344. Springer, 2019.

[CMS13] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. Density-based clustering based on hierarchical density estimates. In *PAKDD*, 2013.

[coz18] Padawan - platform for multi-architecture elf analysis. https://padawan.s3.eurecom.fr/, 2018. Accessed: 2020-09-26.

[CTC16] Alejandro Calleja, Juan Tapiador, and Juan Caballero. A Look into 30 Years of Malware Development from a Software Metrics Perspective. volume 9854, pages 325–345, 09 2016.

[CTC18] Alejandro Calleja, Juan Tapiador, and Juan Caballero. The MalSource Dataset: Quantifying Complexity and Code Reuse in Malware Development. 11 2018.

[cuc] Cuckoo Sandbox 2.0 Release Candidate 1. https://cuckoosandbox.org/blog/cuckoo-sandbox-v2-rc1.

[CVD+20] Emanuele Cozzi, Pierre-Antoine Vervier, Matteo Dell'Amico, Yun Shen, Leyla Bilge, and Davide Balzarotti. The tangled genealogy of iot malware. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*, 2020.

[cws] CWsandbox. http://www.mwanalysis.org.

[CX10] Silvio Cesare and Yang Xiang. Classification of malware using structured control flow. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107*, pages 61–70. Australian Computer Society, Inc., 2010.

[CZ18] Andrei Costin and Jonas Zaddach. Iot malware: Comprehensive survey, analysis framework and case studies. *BlackHat USA*, 2018.

[dar] darkangel. Mood-NT. http://darkangel.antifork.org/codes/mood-nt.tgz.

[Dav] Dave Lee. Shellshock: 'Deadly serious' new vulnerability found. http://www.bbc.com/news/technology-29361794.

[Del19] Matteo Dell'Amico. FISHDBC: Flexible, incremental, scalable, hierarchical density-based clustering for arbitrary data and distance, 2019.

[det] Multiplatform Linux Sandbox. https://detux.org/.

[DFAG+19] Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh, Manu Sridharan, et al. Idapro for iot malware analysis? In *12th {USENIX} Workshop on Cyber Security Experimentation and Test ({CSET} 19)*, 2019.

[DFC16] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Kam1n0: Mapreduce-based assembly clone search for reverse engineering. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 461–470, 2016.

[DFC19] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

[DGHH+15] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, pages 1–11, 2015.

[DLL+20] Fei Ding, Hongda Li, Feng Luo, Hongxin Hu, Long Cheng, Hai Xiao, and Rong Ge. Deeppower: Non-intrusive and deep learning-based detection of iot malware using power side channels. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 33–46, 2020.

[DML11] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, pages 577–586. ACM, 2011.

[DN11] Tudor Dumitraş and Iulian Neamtiu. Experimental Challenges in Cyber Security: A Story of Provenance and Lineage for Malware. In *CEST*, 2011.

[DQG+04] David Dagon, Xinzhou Qin, Guofei Gu, Wenke Lee, Julian Grizzard, John Levine, and Henry Owen. Honeystat: Local worm detection using honeypots. In *RAID*, volume 4, pages 39–58. Springer, 2004.

[DR05] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects (english version). *Sstic*, 5(1):3, 2005.

[dSDC20] Daniel Ricardo dos Santos, Mario Dagrada, and Elisa Costante. Leveraging operational technology and the internet of things to attack smart buildings. *Journal of Computer Virology and Hacking Techniques*, pages 1–20, 2020.

[DWH13] Zakir Durumeric, Eric Wustrow, and Alex Halderman. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *Proceedings of the USENIX Security Symposium*, 2013.

[Eag11] Chris Eagle. *The IDA pro book*. no starch press, 2011.

[elfa] elfmaster. ECFS. https://github.com/elfmaster/ecfs.

[elfb] elfmaster. ELF Packer v0.3. http://www.bitlackeys.org/projects/elfpacker.tgz.

[elfc] elfmaster. ftrace. https://github.com/elfmaster/ftrace.

[evm] evm. A code pirate's cutlass - recovering software architecture from embedded binaries. https://recon.cx/2018/montreal/schedule/events/109.html. Accessed: 2020-09-16.

[Fer] Ferrie, Peter and Peter, Ször. Hunting for metamorphic. http://vxer.org/lib/apf39.html.

[FFCD14] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 78–87. IEEE, 2014.

[Fla04] Halvar Flake. Structural comparison of executable objects. In *Detection of intrusions and malware & vulnerability assessment, GI SIG SIDAR workshop, DIMVA 2004*. Gesellschaft für Informatik eV, 2004.

[Fou] Linux Foundation. Elf and abi standards. https://refspecs.linuxfoundation.org/. Accessed: 2020-11-03.

[FXWC19] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment*, 12(5):461–474, 2019.

[GCB+15] Mariano Graziano, Davide Canali, Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. Needles in a Haystack: Mining Information from Public Dynamic Analysis Sandboxes for Malware Intelligence. In *Proceedings of the 24rd USENIX Security Symposium (USENIX Security)*, August 2015.

[GGPS98] Leslie Ann Goldberg, Paul W Goldberg, Cynthia A Phillips, and Gregory B Sorkin. Constructing Computer Virus Phylogenies. *J. Algorithms*, 26(1), 1998.

[gru] grugq and scut. Armouring the ELF: Binary encryption on the UNIX platform. http://phrack.org/issues/58/5.html.

[Gui] Ilfak Guilfanov. Ida f.l.i.r.t. technology: In-depth. https://www.hex-rays.com/products/ida/tech/flirt/in_depth/. Accessed: 2020-10-07.

[Hao] M. Hao. A Look into the Gafgyt Botnet Trends from the Communication Traffic Log. https://nsfocusglobal.com/look-gafgyt-botnet-trends-communication-traffic-log/.

[HC19] Irfan Ul Haq and Juan Caballero. A Survey of Binary Code Similarity, 2019.

[HGS01] David Harley, Urs E Gattiker, and Robert Slade. *Viruses revealed*. McGraw-Hill Professional, 2001.

[HIT+18] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680. ACM, 2018.

[HSBG13] Xin Hu, Kang G Shin, Sandeep Bhatkar, and Kent Griffin. Mutantx-s: Scalable malware clustering based on static features. In *USENIX ATC*, 2013.

[HYD17] He Huang, Amr M. Youssef, and Mourad Debbabi. BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 155–166. ACM, 2017.

[JBV11] Jiyong Jang, David Brumley, and Shobha Venkataraman. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *ACM CCS*, 2011.

[JMM18] Rommel Joven, Jasper Manuel, and David Maciejack. Mirai: Beyond the Aftermath. https://www.botconf.eu/wp-content/uploads/2018/12/2018-R-Joven-Mirai-Beyond-the-Aftermath.pdf, December 2018.

[JRM11] Emily R Jacobson, Nathan Rosenblum, and Barton P Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 1–8, 2011.

[JWB13] Jiyong Jang, Maverick Woo, and David Brumley. Towards Automatic Software Lineage Inference. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 81–96, Washington, D.C., 2013. USENIX.

[KRVV] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries.

[KRVV04] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, pages 18–18, 2004.

[KV15] Dhilung Kirat and Giovanni Vigna. Malgene: Automatic extraction of malware analysis evasion signature. In *ACM CCS*, 2015.

[KWLP05] Md Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1, 11 2005.

[LDFM+12] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of Malicious Code: Insights into the Malicious Software Industry. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 349–358. ACM, 2012.

[LH07] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2), 2007.

[LLGR10] Peng Li, Limin Liu, Debin Gao, and Michael K Reiter. On challenges in evaluating malware clustering. In *RAID*, 2010.

[LWKS05] Michael E Locasto, Ke Wang, Angelos D Keromytis, and Salvatore J Stolfo. Flips: Hybrid adaptive intrusion prevention. In *RAID*, pages 82–101. Springer, 2005.

[mala] Malware Must Die! http://blog.malwaremustdie.org/.

[malb] malwr. https://www.malwr.com/.

[May] Mayhem. The Cerberus ELF Interface. http://phrack.org/issues/61/8.html.

[MKK07] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430, 2007.

[MKN05] Peter Mell, Karen Kent, and Joseph Nusbaum. *Guide to malware incident prevention and handling.* US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 2005.

[M.L] M.Léveillé, Marc-Etienne. Unboxing Linux/Mumblehard. https://www.welivesecurity.com/wp-content/uploads/2015/04/mumblehard.pdf.

[MMDa] MMD. MMD-0025-2014 - ITW Infection of ELF .IptabLex and .IptabLes China DDoS bots malware. http://blog.malwaremustdie.org/2014/06/mmd-0025-2014-itw-infection-of-elf.html.

[MMDb] MMD. MMD-0030-2015 - New ELF malware on Shellshock: the ChinaZ. http://blog.malwaremustdie.org/2015/01/mmd-0030-2015-new-elf-malware-on.html.

[MMDc] MMD. MMD-0062-2017 - Credential harvesting by SSH Direct TCP Forward attack via IoT botnet. http://blog.malwaremustdie.org/2017/02/mmd-0062-2017-ssh-direct-tcp-forward-attack.html.

[Mon15] KA Monnappa. Automating Linux Malware Analysis Using Limon Sandbox. *Black Hat Europe 2015*, 2015.

[MSB19] Anand Mudgerikar, Puneet Sharma, and Elisa Bertino. E-spion: A system-level intrusion detection system for iot devices. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 493–500, 2019.

[MSY+15] Yin Pa Minn, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, and Christian Rossow. IoTPOT: Analysing the rise of IoT compromises. In *9th USENIX Workshop on Offensive Technologies (WOOT). USENIX Association*, 2015.

[MXW15] Jiang Ming, Dongpeng Xu, and Dinghao Wu. Memoized Semantics-Based Binary Diffing with Application to Malware Lineage Inference. In *IFIP Advances in Information and Communication Technology*, volume 455, pages 416–430, 05 2015.

[MY18] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2018.

[Ngu] Nguyen Anh Quynh. Unicorn Emulator. https://github.com/unicorn-engine/unicorn.

[Nic] Nicky Woolf. DDoS attack that disrupted internet was largest of its kind in history, experts say. https:

//www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet.

[Nor16] Amy Nordrum. The internet of fewer things [news]. *IEEE Spectrum*, 53(10):12–13, 2016.

[NRM+17] Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, and Aiman Hanna. Binsign: fingerprinting binary functions to support automated analysis of code executables. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 341–355. Springer, 2017.

[olea] oledump-py. https://blog.didierstevens.com/programs/oledump-py/.

[oleb] oletools - python tools to analyze OLE and MS Office files. https://www.decalage.info/python/oletools.

[ope] OpenVZ, a container-based virtualization for Linux. https://openvz.org/Main_Page.

[Pal19] PaloAlto Networks. Home & Small Office Wireless Routers Exploited to Attack Gaming Servers. https://unit42.paloaltonetworks.com/home-small-office-wireless-routers-exploited-to-attack-gaming-servers/, November 2019.

[Pay] PayloadSecurity. VxStream Sandbox Linux. https://www.payload-security.com/products/linux.

[PECK20] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. Probabilistic naming of functions in stripped binaries. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*, 2020.

[pee] peepdf - PDF Analysis Tool. http://eternal-todo.com/tools/peepdf-pdf-analysis-tool.

[PLF10] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral Clustering of HTTP-based Malware and Signature Generation Using Malicious Network Traces. In *NSDI*, 2010.

[PLL08] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. Mcboost: Boosting scalability in malware collection and analysis using

statistical classification of executables. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 301–310. IEEE, 2008.

[PSY+15] Yin Minn Pa Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. IoTPOT: analysing the rise of IoT compromises. In *WOOT*, 2015.

[PU12] Roberto Perdisci and ManChon U. VAMO: Towards a Fully Automated Malware Clustering Validity Analysis. In *ACSAC*, 2012.

[QSM15a] Jing Qiu, Xiaohong Su, and Peijun Ma. Library functions identification in binary code by using graph isomorphism testings. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 261–270. IEEE, 2015.

[QSM15b] Jing Qiu, Xiaohong Su, and Peijun Ma. Using reduced execution flow graph to identify library functions in binary code. *IEEE Transactions on Software Engineering*, 42(2):187–202, 2015.

[rad] radare2, a portable reversing framework. http://www.radare.org/.

[RZMH08] Nathan E Rosenblum, Xiaojin Zhu, Barton P Miller, and Karen Hunt. Learning to analyze binary computer code. In *AAAI*, pages 798–804, 2008.

[SBG20] Manolis Stamatogiannakis, Herbert Bos, and Paul Groth. Pandacap: a framework for streamlining collection of full-system traces. In *Proceedings of the 13th European workshop on Systems Security*, pages 1–6, 2020.

[sd] sd and devik. Linux on-the-fly kernel patching without LKM. http://phrack.org/issues/58/7.html.

[SF12] Farrukh Shahzad and Muddassar Farooq. Elf-miner: Using structural knowledge and data mining methods to detect new (linux) malicious executables. *Knowledge and Information Systems*, 2012.

[SG64] Abraham Savitzky and Marcel JE Golay. Smoothing and differentiation of data by simplified least squares procedures. *Analytical chemistry*, 36(8):1627–1639, 1964.

[sho] Shodan, the world's first search engine for Internet-connected devices. https://www.shodan.io/.

[Sila] Silvio Cesare. Runtime kernel kmem patching. https://github.com/BuddhaLabs/PacketStorm-Exploits/blob/master/9901-exploits/runtime-kernel-kmem-patching.txt.

[Silb] Silvio Cesare. Shared Library Redirection via ELF PLT Infection. http://www.phrack.org/issues/56/7.html#article.

[Silc] Silvio Cesare. Unix ELF parasites and virus. https://web.archive.org/web/20150713122748/http://vxer.org/lib/vsc01.html.

[Sop] SophosLabs. Botnets, a free tool and 6 years of Linux/Rst-B. https://nakedsecurity.sophos.com/2008/02/13/botnets-a-free-tool-and-6-years-of-linuxrst-b.

[SRKC16a] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. AVclass: A Tool for Massive Malware Labeling. In *RAID*, 2016.

[SRKC16b] Marcos Sebastian, Richard Rivera, Platon Kotzias, and Juan Caballero. AVclass: A Tool for Massive Malware Labeling. In *RAID*, 2016.

[SSM15] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 611–626, 2015.

[Sta] StatCounter. Desktop Operating System Market Share Worldwide. http://gs.statcounter.com/os-market-share/desktop/worldwide.

[STMF] M Zubair Shafiq, S Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. Springer.

[SWD17] Paria Shirani, Lingyu Wang, and Mourad Debbabi. Binshape: Scalable and robust binary library function identification using function shape. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 301–324. Springer, 2017.

[SWH+15] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[SWL05] Salvatore J Stolfo, Ke Wang, and Wei-Jen Li. Fileprint analysis for malware detection. *ACM CCS WORM*, 2005.

[Sym18] Symantec. Symantec internet security threat report (istr). https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf, March 2018.

[Sym19] Symantec. Symantec internet security threat report (istr). https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf, February 2019.

[sysa] Sysdig. https://www.sysdig.org/.

[sysb] SystemTap. https://sourceware.org/systemtap/.

[Tal18] Talos. New VPNFilter malware targets at least 500K networking devices worldwide. https://blog.talosintelligence.com/2018/05/VPNFilter.html, May 2018.

[Tea] Team TESO. Burneye ELF encryption program. https://packetstormsecurity.com/files/30648/burneye-1.0.1-src.tar.bz2.html.

[TFA+17] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):1–41, 2017.

[TKFC15] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *Ndss*, 2015.

[UPBSB16] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. RAMBO: Run-time packer Analysis with Multiple Branch Observation. July 2016.

[urla] Diaphora, a free and open source program diffing tool. http://diaphora.re/.

[urlb] VirusTotal. https://www.virustotal.com/.

[VE93] Mike Van Emmerik. Signatures for library functions in executable files. 1993.

[vir] Malware analysis sandbox aggregation: Welcome Tencent HABO. http://blog.virustotal.com/2017/11/malware-analysis-sandbox-aggregation.html.

[VLA96] VLAD. Staog linux virus. http://www.wiw.org/~meta/vlad.php?read=ARTICLE.2_4&issue=7&desc=STAOG%20Linux%20Virus, 1996. Accessed: 2020-09-20.

[VS18] Pierre-Antoine Vervier and Yun Shen. Before Toasters Rise Up: A View into the Emerging IoT Threat Landscape. In *RAID*, 2018.

[vts] VirusTotal += Behavioural Information. http://blog.virustotal.com/2012/07/virustotal-behavioural-information.html.

[Wic09] Georg Wicherski. pehash: A novel approach to fast malware clustering. In *Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, LEET'09, 2009.

[WLL+] Aohui Wang, Ruigang Liang, Xiaokang Liu, Yingjun Zhang, Kai Chen, and Jin Li. *An Inside Look at IoT Malware.*

[WLO+18] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1137–1150, 2018.

[Yeh] T. Yeh. Netis Routers Leave Wide Open Backdoor. https://blog.trendmicro.com/trendlabs-security-intelligence/netis-routers-leave-wide-open-backdoor/.

[YLC+19] Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yong Li, and Yunhao Liu. Understanding and detecting overlay-based android malware at market scales. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 168–179, 2019.

[YZK+20] Wei You, Zhuo Zhang, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Harmon, and Xiangyu Zhang. Pmp: Cost-effective forced execution with probabilistic memory preplanning. In *2020 IEEE Symposium on Security and Privacy, SP 2020, Proceedings*, pages 18–20, 2020.

[Zom] Zombie. Injected Evil. http://zombie.daemonlab.org/infelf.html.

[ZDn] ZDnet. Google's VirusTotal puts Linux malware under the spotlight. http://www.zdnet.com/article/googles-virustotal-puts-linux-malware-under-the-spotlight/.

[ZJ12] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*, pages 95–109. IEEE, 2012.

[Zyn] Zynamics. Bindiff - a comparison tool for binary files. https://www.zynamics.com/bindiff.html. Accessed: 2020-10-20.