Understanding Linux Malware

Emanuele Cozzi
Eurecom

Mariano Graziano
Cisco Systems, Inc.

Yanick Fratantonio
Eurecom

Davide Balzarotti
Eurecom

***Abstract*—For the past two decades, the security community has been fighting malicious programs for Windows-based operating systems. However, the recent surge in adoption of embedded devices and the IoT revolution are rapidly changing the malware landscape. Embedded devices are profoundly different than traditional personal computers. In fact, while personal computers run predominantly on x86-flavored architectures, embedded systems rely on a variety of different architectures. In turn, this aspect causes a large number of these systems to run some variants of the Linux operating system, pushing malicious actors to give birth to "Linux malware."**

To the best of our knowledge, there is currently no comprehensive study attempting to characterize, analyze, and understand Linux malware. The majority of resources on the topic are available as sparse reports often published as blog posts, while the few systematic studies focused on the analysis of specific families of malware (e.g., the Mirai botnet) mainly by looking at their network-level behavior, thus leaving the main challenges of analyzing Linux malware unaddressed.

This work constitutes the first step towards filling this gap. After a systematic exploration of the challenges involved in the process, we present the design and implementation details of the first malware analysis pipeline specifically tailored for Linux malware. We then present the results of the first large-scale measurement study conducted on 10,548 malware samples (collected over a time frame of one year) documenting detailed statistics and insights that can help directing future work in the area.

I. Introduction

The security community has been fighting malware for over two decades. However, despite the significant effort dedicated to this problem by both the academic and industry communities, 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 Microsoft's Windows operating system, mainly because of its very large market share (currently estimated at 83% [1] for desktop computers). Therefore, the security community has also been 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 popularity of embedded devices is causing the malware landscape to rapidly change. Embedded devices have been in use in industrial environments 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. 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 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 hard-coded 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 systems, 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. However, 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 [2]. 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* [3]) 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. [4] 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 malware.

This work aims at filling this gap by presenting 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 paper 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.

II. 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.

A. 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 [4], 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.

B. 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).

C. 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 Section III.

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

III. 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 [5], IDA Pro, radare2 [6], and Nucleus [7]) as well as completely new tools we explicitly designed for this paper. 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.

A. Data Collection

To retrieve data for our study we used the VirusTotal intelligence API to fetch the reports of *every* ELF file submitted

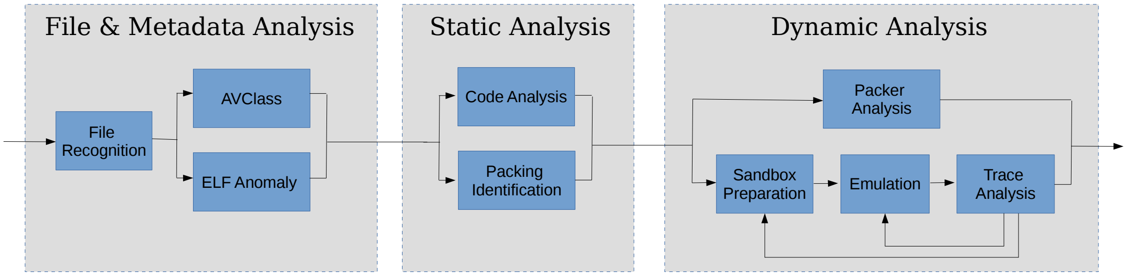

Fig. 1. Overview of our analysis pipeline.

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.

B. 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 Section V-A) 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 Section V-A 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. [5], 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.

C. 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 Section V-E 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.

D. 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 [8] 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 Section V-D, 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 [4], [9] 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 [4]. Finally, to dynamically unpack unknown UPX variants we developed a tool based on Unicorn [10]. 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 Section V-E, this approach allowed us to automatically unpack all but three malware samples in our dataset.

TABLE I
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%

IV. DATASET

Our final dataset, after the filtering stage, consisted of 10,548 ELF executables, covering more than ten different architectures (see Table I 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. [4] on the Mirai botnet, but only account for 32% of our samples.

We report detailed statistics about the distribution of samples in our dataset in Appendix. 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 common 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.

A. 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

TABLE II
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%

greatly simplified by the availability of online services like Shodan [11] or scanning tools like ZMap [12] 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, 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 paper 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 Section VI.

V. 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 III
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

A. 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 sections 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 44th byte of the ELF header itself and zeroed out any field unused at runtime. Table II 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 III 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 section information or other fields that would not affect the program execution.

B. 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.

TABLE IV
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%)	

Subsystems Initialization. This appears to be the most common approach adopted by malware authors and takes advantage of the well known Linux *init* system. Table IV 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.

TABLE V
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

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 [13]. 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 [14], 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 `/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 IV, 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 IV 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.

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

TABLE VI
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

TABLE VII
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%

Overall, we noted that this behavior, already common on Windows operating systems, is also widespread on Linux-based malware. Table V 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.

D. 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 are executed with super-user privileges. Second, especially when targeting lowend 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 VII 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 exploitations. 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]

TABLE VIII
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%

E. 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 so far [15]–[17], 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 [18]–[22]. 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 VIII, 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

[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 IX
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%

TABLE X
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%

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 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 [23]. 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.

F. 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 IX. 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 runtime. 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.

G. 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

TABLE XI
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 XII
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%

TABLE XIII
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

TABLE XIV
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

by a C&C server. In this section we look at which portions of 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 X, 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 XI, 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 Subsection V-H) 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.

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 IX, 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 XII, 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`.

H. 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 XIII.

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 strings such as "VMware" or "QEMU." Table XIV 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 [24], 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 Section V-F).

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 section III-D), 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.

TABLE XV
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%

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

Common Libraries. Table XV 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.

VI. 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.

VII. Related Work

In the past two decades the security community has focused almost exclusively on fighting malware targeting Microsoft Windows. As a result, hundreds of papers have described techniques to analyze PE binaries [25]–[28], detecting ongoing threats [27], [29], [30], and preventing possible infection attempts [31]–[33] on Windows operating systems. The community also developed many analysis tools for dissecting threats related to the Windows environment, ranging from dynamic analysis solutions [34]–[37] to dissectors for file formats used as attack vectors [38]–[40].

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 [13], [41]–[44], rootkits [45], [46], and tools to dissect them [47]–[49], 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 [50] and Shellshock [51]. Many blog posts and papers were published for the analysis and dissection of specific families [52]–[57], but these investigations were mainly conducted by manual reverse engineering. Recent research by Shazhad et al. [58] and by Bai et al. [59] extracts 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 non-Windows 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 [60] is an analysis sandbox based on `strace` (and thus easily detectable), and it only supports the analysis of x86-flavored binaries. Sysdig [61] and PayloadSecurity [62] are affected by similar issues and they also only work for x86 binaries. Detux [63], instead, supports four different architectures (i.e., x86, x86-64, ARM, and MIPS). However, it only performs a very basic analysis by running `readelf` and provides network dumps. Cuckoo sandbox [64] 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 [9]. 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 done by Pa et al. [65]. In their paper, they present a Telnet honeypot to measure the current attack trends as well as the first sandbox environment based on Qemu and OpenWRT called IoTBOX for analyzing IoT malware. They showed the issue of IoT devices exposing Telnet online and they collected few families actively targeting this service. Similarly, Antonakakis et al. [4] studied in detail a specific Linux malware family, the Mirai botnet. They measure systematically the evolution and growth of the botnet mainly from the 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 our work 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, since they were collected via telnet-based honeypots.

VIII. Conclusions

This paper 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.

References

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

[2] ZDnet, “Google’s VirusTotal puts Linux malware under the spotlight.” http://www.zdnet.com/article/googles-virustotal-puts-linux-malware-under-the-spotlight/.

[3] “Malware Must Die!.” http://blog.malwaremustdie.org/.

[4] Antonakakis et al., “Understanding the Mirai Botnet,” in *Proceedings of the USENIX Security Symposium*, 2017.

[5] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, “AVclass: A Tool for Massive Malware Labeling,” in *RAID*, 2016.

[6] “radare2, a portable reversing framework.” http://www.radare.org/.

[7] D. Andriesse, A. Slowinska, and H. Bos, “Compiler-Agnostic Function Detection in Binaries,” in *IEEE European Symposium on Security and Privacy*, 2017.

[8] “SystemTap.” https://sourceware.org/systemtap/.

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

[10] Nguyen Anh Quynh, “Unicorn Emulator.” https://github.com/unicorn-engine/unicorn.

[11] “Shodan, the world’s first search engine for Internet-connected devices.” https://www.shodan.io/.

[12] Z. Durumeric, E. Wustrow, and A. Halderman, “ZMap: Fast Internet-wide Scanning and Its Security Applications,” in *Proceedings of the USENIX Security Symposium*, 2013.

[13] Silvio Cesare, “Unix ELF parasites and virus.” http://vxer.org/lib/vsc01.html.

[14] 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.

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

[16] elfmaster, “ELF Packer v0.3.” http://www.bitlackeys.org/projects/elfpacker.tgz.

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

[18] R. Lyda and J. Hamrock, “Using entropy analysis to find encrypted and packed malware,” *IEEE Security & Privacy*, vol. 5, no. 2, 2007.

[19] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, “RAMBO: Run-time packer Analysis with Multiple Branch Observation,” July 2016.

[20] S. Cesare and Y. Xiang, “Classification of malware using structured control flow,” in *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107*, pp. 61–70, Australian Computer Society, Inc., 2010.

[21] R. Perdisci, A. Lanzi, and W. Lee, “Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables,” in *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pp. 301–310, IEEE, 2008.

[22] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq, “Pe-miner: Mining structural information to detect malicious executables in realtime.,” Springer.

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

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

[25] G. 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.

[26] Ferrie, Peter and Peter, Szr, “Hunting for metamorphic.” http://vxer.org/lib/apf39.html.

[27] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, “Semantics-aware malware detection,” in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, SP ’05, 2005.

[28] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, “Static disassembly of obfuscated binaries,”

[29] S. J. Stolfo, K. Wang, and W.-J. Li, “Fileprint analysis for malware detection,” *ACM CCS WORM*, 2005.

[30] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen, “Honeystat: Local worm detection using honeypots,” in *RAID*, vol. 4, pp. 39–58, Springer, 2004.

[31] P. Mell, K. Kent, and J. Nusbaum, *Guide to malware incident prevention and handling*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 2005.

[32] D. Harley, U. E. Gattiker, and R. Slade, *Viruses revealed*. McGraw-Hill Professional, 2001.

[33] M. E. Locasto, K. Wang, A. D. Keromytis, and S. J. Stolfo, “Flips: Hybrid adaptive intrusion prevention,” in *RAID*, pp. 82–101, Springer, 2005.

[34] “malwr.” https://www.malwr.com/.

[35] “CWsandbox.” http://www.mwanalysis.org.

[36] “Anubis.” https://anubis.iseclab.org.

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

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

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

[40] “oledump-py.” https://blog.didierstevens.com/programs/oledump-py/.

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

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

[43] Z0mbie, “Injected Evil.” http://z0mbie.daemonlab.org/infelf.html.

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

[45] darkangel, “Mood-NT.” http://darkangel.antifork.org/codes/mood-nt.tgz.

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

[47] Mayhem, “The Cerberus ELF Interface.” http://phrack.org/issues/61/8.html.

[48] elfmaster, “ftrace.” https://github.com/elfmaster/ftrace.

[49] elfmaster, “ECFS.” https://github.com/elfmaster/ecfs.

[50] 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.

[51] Dave Lee, “Shellshock: ’Deadly serious’ new vulnerability found.” http://www.bbc.com/news/technology-29361794.

[52] 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/.

[53] 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.

[54] 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.

[55] 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.

[56] A. Wang, R. Liang, X. Liu, Y. Zhang, K. Chen, and J. Li, *An Inside Look at IoT Malware*.

[57] P. Celeda, R. Krejci, J. Vykopal, and M. Drasar, “Embedded malware-an analysis of the chuck norris botnet,” in *Computer Network Defense (EC2ND), 2010 European Conference on*, pp. 3–10, IEEE, 2010.

[58] F. Shahzad and M. Farooq, “Elf-miner: Using structural knowledge and data mining methods to detect new (linux) malicious executables,” *Knowledge and Information Systems*, 2012.

[59] J. Bai, Y. Yang, S. Mu, and Y. Ma, “Malware detection through mining symbol table of Linux executables,” *Information Technology Journal*, 2013.

[60] K. Monnappa, “Automating Linux Malware Analysis Using Limon Sandbox,” *Black Hat Europe 2015*, 2015.

[61] “Sysdig.” https://www.sysdig.org/.

[62] PayloadSecurity, “VxStream Sandbox Linux.” https://www.payload-security.com/products/linux.

[63] “Multiplatform Linux Sandbox.” https://detux.org/.

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

[65] Y. P. Minn, S. Suzuki, K. Yoshioka, T. Matsumoto, and C. Rossow, “IoTPOT: Analysing the rise of IoT compromises,” in *9th USENIX*

Workshop on Offensive Technologies (WOOT). USENIX Association, 2015.

APPENDIX

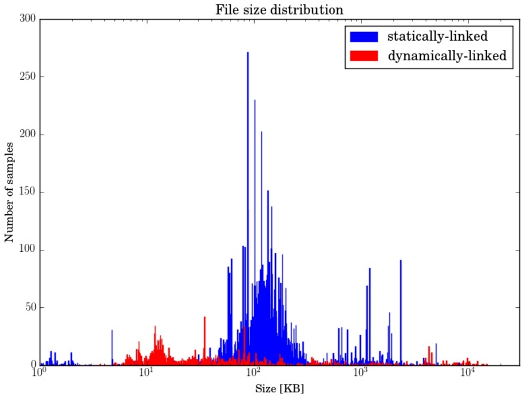

Fig. 2. File size distribution of ELF malware in the dataset

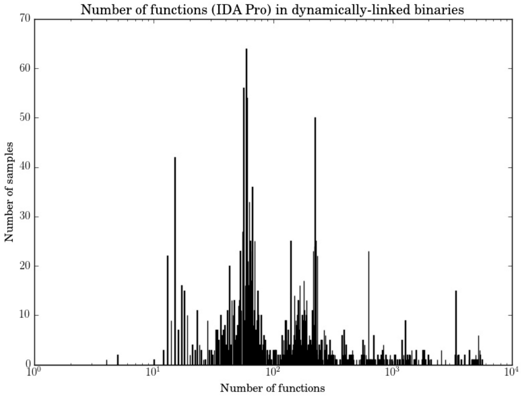

Fig. 3. Number of functions identified by IDA Pro in dynamically linked samples

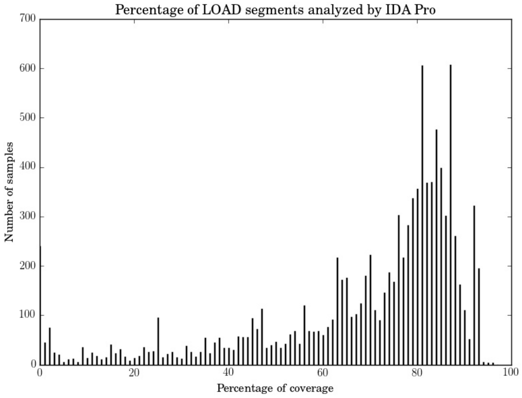

Fig. 4. Percentage of LOAD segments analyzed by IDA Pro

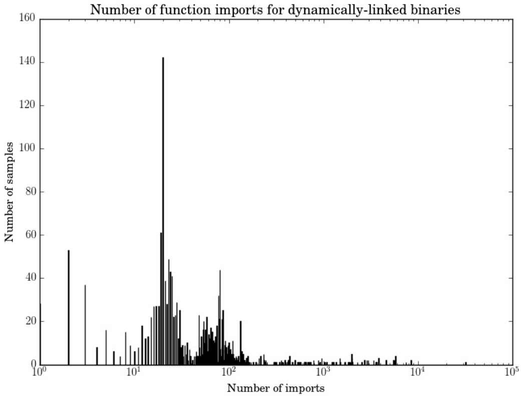

Fig. 5. Number of imported symbols in dynamically linked samples

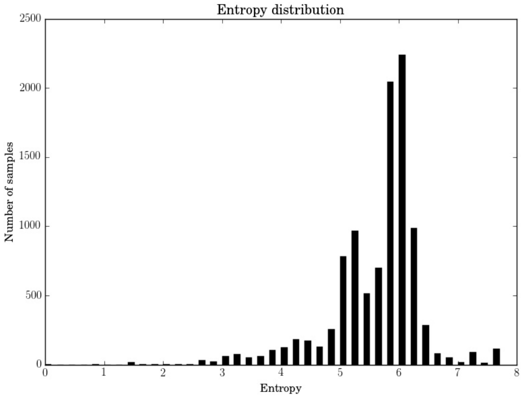

Fig. 6. Entropy distribution over the dataset

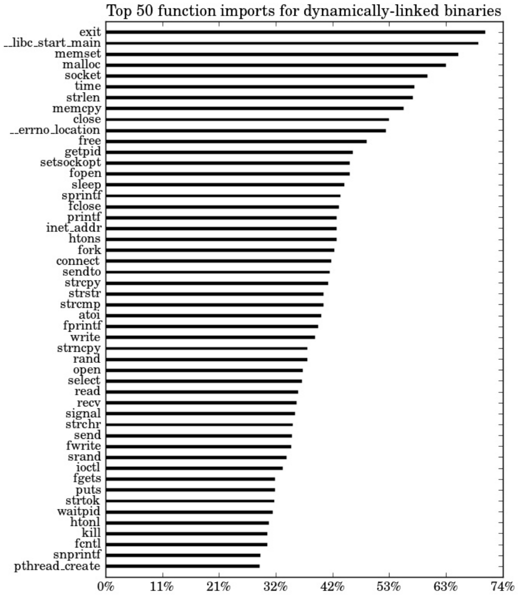

Fig. 7. Library imports for dynamically linked executables