Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Kernel-level tracing for detecting stegomalware and covert channels in Linux environments

Luca Caviglione [a], Wojciech Mazurczyk [b,c], Matteo Repetto [a,*], Andreas Schaffhauser [c], Marco Zuppelli [a]

[a] Institute of Applied Mathematics and Information Technologies, CNR, Italy
[b] Warsaw University of Technology, Poland
[c] FernUniversität in Hagen, Germany

ARTICLE INFO

MSC:
00-01
99-00

Keywords:
eBPF
Syscall and network tracing
Stegomalware
Covert channels
Detection

ABSTRACT

Modern malware is becoming hard to spot since attackers are increasingly adopting new techniques to elude signature- and rule-based detection mechanisms. Among the others, steganography and information hiding can be used to bypass security frameworks searching for suspicious communications between processes or exfiltration attempts through covert channels. Since the array of potential carriers is very large (e.g., information can be hidden in hardware resources, various multimedia files or network flows), detecting this class of threats is a scarcely generalizable process and gathering multiple behavioral information is time-consuming, lacks scalability, and could lead to performance degradation.

In this paper, we leverage the extended Berkeley Packet Filter (eBPF), which is a recent code augmentation feature provided by the Linux kernel, for programmatically tracing and monitoring the behavior of software processes in a very efficient way. To prove the flexibility of the approach, we investigate two realistic use cases implementing different attack mechanisms, i.e., two processes colluding via the alteration of the file system and hidden network communication attempts nested within IPv6 traffic flows. Our results show that even simple eBPF programs can provide useful data for the detection of anomalies, with a minimal overhead. Furthermore, the flexibility to develop and run such programs allows to extract relevant features that could be used for the creation of datasets for feeding security frameworks exploiting AI.

1. Introduction

Detecting modern threats requires increased efforts, for instance due to the use of multi-stage loading architectures, modular design or the adoption of sophisticated Crime-as-a-Service frameworks [1]. Attackers are progressively introducing new techniques to elude signature- and rule-based detection systems. This trend culminates into malware endowed with some form of information hiding or steganography to covertly exfiltrate data towards a remote Command & Control (C&C) facility and to embed attack routines or configuration files into innocent-looking digital images [2]. To identify anomalies and spot suspicious activity, deep visibility over the behavior of software processes is required, but this often leads to unacceptable overheads, especially for virtualized services or resource-constrained devices. Moreover, the availability of ubiquitous and seamless network connectivity, the uptake of 5G with edge/fog installations, as well as the progressive integration with IoT, build a distributed and multi-domain computing continuum where new services are created and disposed in a rapid manner. Unfortunately, security paradigms have not evolved at the same pace and legacy security perimeter models cannot effectively address new vulnerabilities and threats [3]. Thus, detecting sophisticated attacks and steganographic malware (defined in the following as *stegomalware*) is an emerging challenge that should properly balance the depth of inspection with resource consumption [1,2].

Spotting attacks targeting communication and computing infrastructures has been largely discussed in the literature. For the case of networks, many works focus on anomaly detection (see, e.g., [4] for a recent survey), which aims at recognizing deviating behaviors to prevent or reveal a wide-range of attacks like DoS, traffic amplification, spoofing and scanning attempts. Another important aspect concerns the ability of detecting threats targeting hosts, network appliances and personal devices, which are increasingly mobile [5] or interconnected with a cyber–physical system [6]. However, information-hiding-capable threats and stegomalware pose new challenges, as they exploit bandwidth-scarce channels and their detection depends on the used

* Corresponding author.
E-mail address: matteo.repetto@ge.imati.cnr.it (M. Repetto).

https://doi.org/10.1016/j.comnet.2021.108010
Received 15 September 2020; Received in revised form 13 January 2021; Accepted 11 March 2021
Available online 18 March 2021

carrier, i.e., the software/hardware entity manipulated for encoding the secret [1,7,8].

Under this perspective, new technologies that give programmatic visibility over applications, infrastructures, and devices are necessary. They should be able to monitor, inspect and trace processes at run-time, hence allowing to undertake specific attacks without overwhelming the system. In this paper, we propose to take advantage of kernel-level techniques for code augmentation and investigate their usage for detecting stegomalware. Originally proposed as a monitoring and tracing mechanism for performance tasks, the extended Berkeley Packet Filter (eBPF) framework has been already adopted for security purposes, though existing applications do not fully take advantage of its capabilities. We therefore investigate suitable indicators that can be collected via eBPF programs to reveal two different steganographic threats. First, we consider two processes running on the same host and colluding to "evade" typical security controls (i.e., sandboxing). Second, we gather low-level measurements that are not available from common tools to detect hidden communication attempts nested within network traffic.

Summarizing, the contributions of this paper are: (*i*) the development of eBPF programs to collect data from the Linux kernel in a very efficient way; (*ii*) the creation of realistic use cases for colluding applications and IPv6-capable covert channels, which has been often neglected in the literature; (*iii*) the analysis of how the collected information can be used to design effective algorithms for detecting stegomalware and covert channels; (*iv*) an analysis on the use of eBPF, by taking into account performance measurements in terms of resource consumption.

This paper is an extended version of the work presented in [9] and has the following main improvements: (*i*) the use of kernel-level tracing to support security-related tasks has been refined; (*ii*) attacks leveraging covert channels built within IPv6 traffic have been also considered, i.e., the paper does not focus anymore on colluding applications only; (*iii*) deployability and scalability of the approach have been taken into account.

The remainder of the paper is structured as follows. Section 2 reviews techniques to support the detection of malware and stegomalware. Section 3 provides background information on the considered threats and the eBPF. Section 4 showcases the use of eBPF to detect covert communications via the manipulation of permissions of files, while Section 5 investigates its ability to spot network covert channels targeting IPv6 traffic. Section 6 elaborates on the use of kernel-based measurements both in terms of overheads and perspective integration with other frameworks. Finally, Section 7 concludes the paper and portraits some future research directions.

2. Related works

Being able to collect information (e.g., network traffic) or trace the execution flow of a wide array of software entities are two core tasks to support frameworks and algorithms to detect malware, prevent attacks or engineer security-by-design systems.

To face the heterogeneity of modern deployments and to provide scalability and reusability features, virtualization has been proposed to ease data gathering operations both in computing and networking scenarios. As possible examples, in [10] and [11] authors propose the adoption of an orchestrator for controlling pervasive and lightweight security hooks embedded in the virtual layers of cloud applications. Flexible monitoring can be also obtained by enhancing network hypervisors: this allows to engineer monitoring and security-oriented services in an easier manner, especially when in the presence of complex architectures based on micro services or cloud technologies [12]. For the specific case of targeting communication networks, Deep Packet Inspection (DPI) is an important component as it allows to examine many facets of a flow. Owing to virtualization, probes can be deployed as software components over commodity hardware. For instance, [13] proposes an approach for the dynamic placement of DPI-capable software agents to contain power consumptions and costs, while delivering suitable degrees of scalability and performance. A similar blueprint can be used to enforce integrity of virtual machines, isolate higher software layers, and implement adaptive network security appliances (e.g., intrusion detection systems and firewalls) encapsulated within virtual machines [14]. When dealing with complex and virtualized scenarios, an important aspect concerns the definition and the implementation of efficient orchestration policies. A possible idea exploits meta-functions to dynamically construct security services for satisfying various security requirements [15].

The use of cloud-based frameworks or virtualized platforms for detecting stegomalware or network covert channels, i.e., hidden communication attempts laying within network traffic, has been addressed in a limited manner. Apart our preliminary study in [9], other works solely focus on the normalization of network traffic (see, e.g., [7] and [8] and the references therein) but no prior work investigates how a steganographic threat or a covert channel can be detected or prevented by means of virtualization. Typically, risks arising by attacks endowed with some form of information hiding technique are only briefly discussed (see, e.g., [16]) while threats like DoS and Distributed DoS are addressed more frequently [17]. A notable exception considering virtualization, covert channels and stegomalware concerns *colluding applications*, i.e., two entities trying to communicate outside their respective (secured) execution environments. As paradigmatic examples, [18] and [19] investigate colluding containers or virtual machines trying to communicate via a covert channel to exfiltrate data, map the underlying hardware deployment or guessing if the attacker has been confined within a honeypot. Another typical scenario for a colluding application scheme concerns the use of hidden channels between virtual machines to exfiltrate private keys [20].

Due to the nature of stegomalware and other emerging threats like cryptolockers, a recent trend concerns the gathering and monitoring of some well-defined and low-level features instead of high-level yet specific metrics [1,7,9]. Even if this could sound paradoxical, being able to gather system-specific information could allow to generalize the detection phase or make it more scalable. In this vein, examples of threats that can be detected in a more effective manner via low-level tracing, include *cryptojacking*, i.e., unauthorized utilization of the host or computing infrastructure of a victim to mine crytpocurrencies, *ransomware*, i.e., threats encrypting the file-system of the victim to obtain a ransom, or the aforementioned stegomalware. Concerning cryptojacking, in [21] authors propose to detect attacks by gathering information on system features like the CPU usage, memory consumption, intensity of disk read/write operations, and activities on the network interface. In [22], authors address JavaScript threats acting within the browser. To detect attacks, they collect information on the JavaScript runtime (including the load of events), as well as usage statistics for the network and specific libraries. Such an approach can be used to automatically identify a large population of in-browser cryptojacking attacks. A complementary technique exploits signatures in the network traffic produced by the cryptojacking daemons to send results to C&C servers [23]. For the case of ransomware, its aggressive usage of resources during the attack (i.e., the encryption of a portion of the file system) typically leads to an anomalous load of operations that could be observed. This has been used in [24] to early detect ransomware by examining how the data of user changes in a time window, if the type of files is modified or many files are deleted.

Indeed, more fine-grained measurements can be performed by operating in the lower levels of the software architecture, for instance by directly developing in-kernel probes or via ad-hoc mechanisms like eBPF. For the case of eBPF, some approaches have been already proposed in the literature. In [25] authors deal with an eBPF-based tool for enriching information extracted from network packets and to efficiently gather data when inspecting communications across virtualized hosts or containerized applications. A similar idea is used in [26], but in this

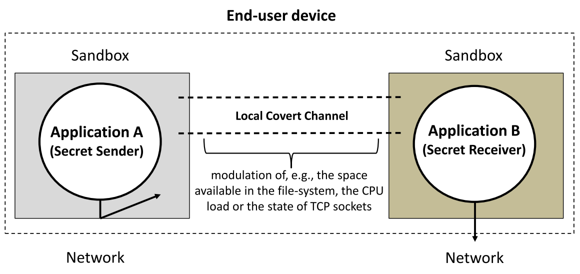

End-user device
Sandbox
Sandbox
Application A (Secret Sender)
Application B (Secret Receiver)
Local Covert Channel
modulation of, e.g., the space available in the file-system, the CPU load or the state of TCP sockets
Network
Network

Fig. 1. Reference scenario for the colluding applications technique.

case eBPF and eXpress Data Path have been primarily used for network monitoring and traffic analysis purposes, rather than for detecting attacks (see, [27] for a very recent survey on the topic). As regards approaches mixing in-kernel measurements and the eBPF engine, [28] reviews various kernel-level security mechanisms for improving container security and network function virtualization. Concerning the use of in-kernel measurements to detect malware, [29] focuses on Android OS and proposes to monitor system calls entering the kernel to detect malicious behavior potentially hidden in native code. Instead, in [30] the focus is shifted from the *syscall* to data, i.e., malware is characterized according to the properties of data objects manipulated during the attacks.

To sum up, at the best of our knowledge, there are not any prior works dealing with platform-specific or kernel-based measurements to detect stegomalware. Rather, some works abstract the detection process by computing general indicators of the presence of two applications colluding via a covert channel. For instance, in [31] authors proved that the power consumption can be used to spot such attacks. To collect information, they employed CPU usage measurements and tools interacting with the power management layer of the device. Another idea is to reveal steganographic communications by evaluating overlaps in the activation patterns of communicating processes. In this case, measurements of the CPU usage (as well as time spent in the various queues waiting for being scheduled) are needed [32].

3. Background

In this section, we provide background information on two main techniques used in stegomalware to bypass security frameworks and to exfiltrate data. Then, we introduce kernel-based data gathering.

3.1. Colluding applications

The term “colluding applications” is an umbrella for identifying a class of threats able to bypass the security policies deployed in underlying software and hardware layers, including the guest OS. Put briefly, the attack creates a local covert channel for exchanging data within the single host [8,33] to set up an “abusive” inter-process communication service between various software entities, e.g., applications and processes. Fig. 1 illustrates the reference scenario, i.e., two applications wanting to leak sensitive data outside the hosting node. Specifically, Application A can access sensible information of the victim but it is prevented by a sandbox from accessing the network layer. Instead, Application B has not access to such data, therefore is considered safer and can communicate outside the host.

A stegomalware can implement two innocent-looking applications colluding to leak sensitive information. To this aim, it should find a suitable carrier where to inject the secret data. For instance, the sender can modulate the amount of used RAM to signal a bit to the receiver inspecting the resource, e.g., 1 when allocating memory and reducing the overall availability and 0 otherwise. The carriers can be various software or hardware artifacts composing the entire architecture: approaches to create a local covert channel include modulation of the space available in the file system, the CPU load, highjacking of inter-process communication or messaging services, or the state of TCP/Unix sockets [8,33].

3.2. Network covert channels

Network covert channels enable to transmit secrets in a stealthy manner by injecting the information within a network artifact acting as the carrier. In this case, the secret sender directly embeds data in network packets or alters the temporal evolution of some features of the traffic, e.g., it modulates the inter-packet statistics or changes the packet sequence to encode information. According to the adopted approach, a *storage* or a *timing* network covert channel is created as depicted in Fig. 2. We point out that, in this work, we mainly focus on storage channels.

In general, network covert channels are used by stegomalware for data exfiltration, implementation of the C&C infrastructure, development of cloaked transfer services for retrieving additional software components, botnet orchestration, and elusion of firewall rules [7,8,18]. To inject data within a network flow, the typical mechanisms are: alteration of the volume of the produced traffic, modulation of the throughput, artificial creation of retransmissions or increased error rates, transcoding of multimedia streams for using the freed capacity to store secrets, and direct embedding of data in unused fields of headers of packets [1,7,8,18]. Recently, IPv6 is gaining its momentum and literature already proposed several techniques targeting such protocol. In particular, the seminal work in [34] introduced over twenty different covert channels exploiting various fields and internals of IPv6. Yet, the contribution is only theoretical, as no implementation or experimental evaluations have been provided. Even if real measurements downsized the capacity of IPv6-based covert channels, their stealthiness make them a suitable choice to endow malware with steganographic communication features [35].

3.3. Data gathering with eBPF

As hinted, detecting stegomalware requires temporal and spacial correlation of fine-grained system properties and actions. The Linux kernel provides quite a complete framework for giving deep visibility over the execution of applications and the kernel itself. Several data sources are available for tracing the execution of both system calls and internal functions: *kprobes*, *uprobes*, *tracepoints*, *dtrace-probes*, and *LLTng-UST* [9,36]. Even if multiple tools are available to collect information from tracing hooks (e.g., *ftrace*, *perf*, *sysdig*, *SystemTap*, and *LTTng*), the eBPF is still the most powerful framework for gathering data in the perspective of detecting stegomalware, covering both code tracing and packet inspection.

Originally used to monitor and inspect network packets [37], eBPF is basically a virtual machine within the kernel able to execute bytecode compiled from C sources in a just-in-time fashion. To avoid harming the system, eBPF programs go through an in-kernel verifier checking their control flow graph to ensure termination and that the memory and registers accessed during the execution are always in a valid state. Even if fast and flexible, eBPF has some limitations: (1) programs cannot have loops as to enforce that they will finish within a bounded execution time; (2) interactions with the user space happen via “maps”, which are key–value structures stored in a shared-memory area; (3) it is not Turing complete.

Compared with similar solutions, eBPF does not require additional kernel modules. This improves the portability of eBPF programs across different installations. In addition, programs can be chained, using a low-overhead linking primitive called *tail call*, which allows the creation of complex applications running inside the kernel. The eBPF exploits an event-driven architecture and a program is hooked to a particular type of event: each occurrence of the event will trigger its execution and, based on the type, the program might be able to alter

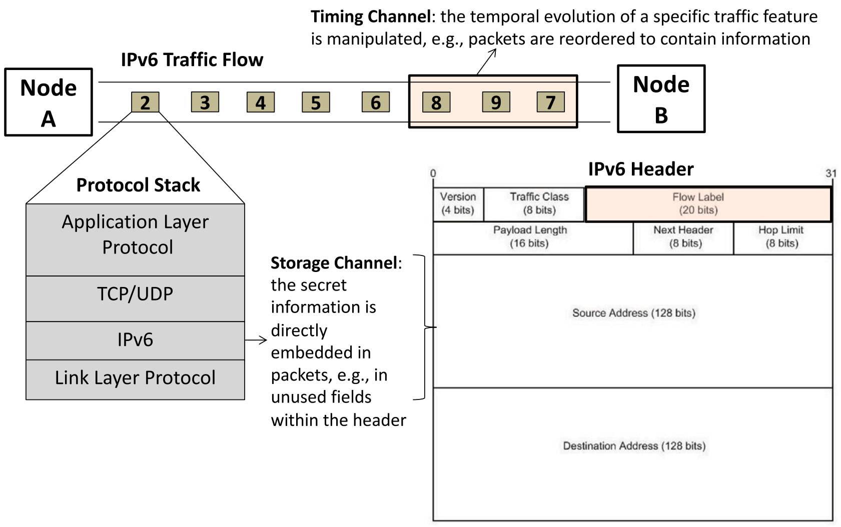

Timing Channel: the temporal evolution of a specific traffic feature is manipulated, e.g., packets are reordered to contain information
IPv6 Traffic Flow
Node A
Node B
2
3
4
5
6
8
9
7
Protocol Stack
Application Layer Protocol
TCP/UDP
IPv6
Link Layer Protocol
Storage Channel: the secret information is directly embedded in packets, e.g., in unused fields within the header
IPv6 Header
0
31
Version (4 bits)
Traffic Class (8 bits)
Flow Label (20 bits)
Payload Length (16 bits)
Next Header (8 bits)
Hop Limit (8 bits)
Source Address (128 bits)
Destination Address (128 bits)

Fig. 2. Reference scenario for network covert channels.

the event context (e.g., check the parameters of a function call or parse a network packet). This model allows to execute programs only when needed, making always-on solutions inexpensive. Programs can be attached to multiple hooks. Possible examples are: network cards via the eXpress Data Path, egress/ingress queues within the network stack, kprobes, uprobes, Userland Statically Defined Tracing probes, Java Virtual Machine, tracepoints, and seccomp/landlock security things. We point out that, in this paper, our investigation leverages both kprobe and `tc`[1] programs.

The typical use of eBPF for gathering information requires the development of both an eBPF program and a user-space utility that loads the program and interacts with the kernel counterpart via a map mainly to configure parameters and collect monitored data. To this aim, a valuable resource is the BPF Compiler Collection[2] (BCC), which is a toolkit for creating efficient kernel tracing and manipulation programs. It includes several tools for tracing the most common features, allows writing eBPF programs in C, and also offers frontends in Python and Lua. Even if originally conceived for performance analyses, the BCC can be also used "out of the box" to detect some well-known covert channels, as we will show in Section 4.1. For possible usages of the BCC for security purpose, see [9] and the references therein.

4. Data gathering for colluding applications

In this section, we investigate eBPF tracing for the detection of a stegomalware implementing a colluding applications scheme. To this aim, we implement an attack via the `chmod-stego` technique discussed in Section 4.1. Obtained numerical results are presented in Section 4.2.

4.1. Chmod-based stegomalware and its detection

To model a malware implementing a colluding applications attack (see, Fig. 1), we used chmod-stego.[3] The chmod-stego is a Python application made of two peers, i.e., the secret sender and the secret receiver. To communicate, the sender encodes the secret message within access permission numbers associated in Linux to files stored in a directory accessible by both endpoints. It can also set the *channel capacity* by imposing a delay between two consecutive invocations of the needed `chmod`(s). As a first step, the sender saves the initial "state" for all the files (by using the `stat` system call in the OS module). Moreover, it splits the secret message to be sent into chunks of a fixed size. Every character within a chunk is converted to an integer value directly mapped into the permissions of the targeted set of files. This operation is repeated until a special EOF character is found. To achieve some form of synchronization, the sender signals the encoding of a character using a ticking mechanism, i.e., each time that a permission is changed, it toggles the owner read bit of the first file in the directory. Accordingly, the receiver remains listening on the owner read bit of the file (i.e., the tick bit) in order to understand whether permissions encode new information. If yes, the receiver acquires the access permission for all the files and deciphers the character by using the ASCII encoding. The process is iterated until the secret message is transmitted in its entirety. To avoid that the communication is easily spotted due to inconsistencies in the file system, at the end of the transmission, the secret sender restores the file permissions to the original state.

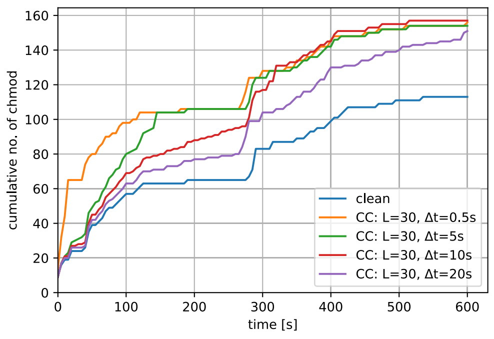
Since the chmod-stego technique is based on the manipulation of the file system, the most straightforward way to design a detection strategy is by tracing the `__x64_sys_chmod` kernel function, which provides better indications than generic I/O activity (e.g., read/write operations through `__x64_sys_read` and `__x64_sys_write`). For this purpose, we used the `trace` utility from BCC,[4] which periodically reports the number of times a given kernel function is invoked.

4.2. Numerical results

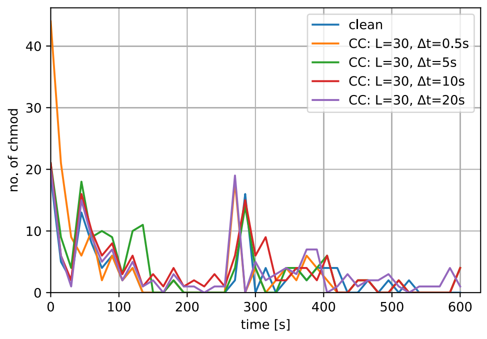
To assess if kernel-level tracing can be used to detect stegomalware, we created an experimental setup composed of a Virtual Machine running Debian GNU/Linux 10 (buster) with Linux kernel 4.20.9 and the aforementioned chmod-stego application. To create some sort of "background noise", a kernel compilation was run, which entails many I/O system calls and can be easily replicated for comparison. To gather data, a simple eBPF filter was injected to trace invocations of the `__x64_sys_chmod` kernel function and to report its relevant parameters, i.e., file and permissions, the Process ID, and the Thread ID.

We performed two different sets of experiments. The first aimed at evaluating the tradeoff between the steganographic bandwidth of

[1] `tc` (acronym for Traffic Control) is the user-space utility in Linux for managing the kernel packet scheduler, which includes classification, queuing, policing, shaping, and scheduling operations.
[2] https://github.com/iovisor/bcc.
[3] https://github.com/operatorequals/chmod-stego.
[4] bcc/trace, URL: https://github.com/iovisor/bcc/blob/master/tools/trace.py.

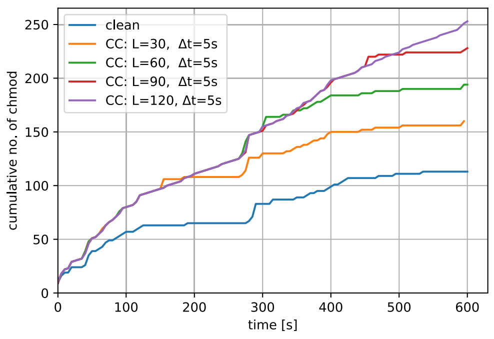


(a) Cumulative time evolution

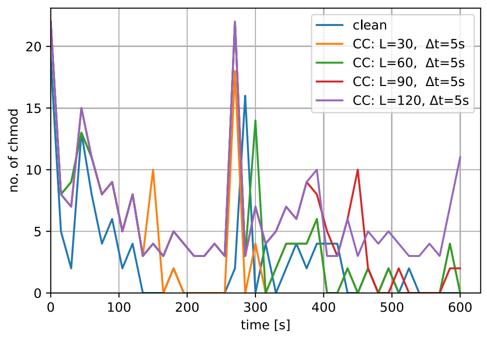


(b) Instantaneous time evolution

Fig. 3. Detected invocation of the `__x64_sys_chmod` kernel function with $L = 30$ and $\Delta t = 0.5, 5, 10, 20$ s.



(a) Cumulative time evolution



(b) Instantaneous time evolution

Fig. 4. Detected invocation of the `__x64_sys_chmod` kernel function with $\Delta t = 5$ s and $L = 30, 60, 90, 120$.

the covert channel and its detectability. To this aim, we fixed the length L of the secret message to be transmitted and we varied the time between the transmission of two consecutive characters, denoted as Δt. Specifically, we conducted trials with $L = 30$ characters and $\Delta t = 0.5, 5, 10, 20$ s. In the second round of tests, we investigated the influence of the size of the data exchanged between the two colluding applications. Hence, we set $\Delta t = 5$ s and we performed trials with $L = 30, 60, 90, 120$ characters, which may be representative of the exfiltration of a PIN, a cryptographic key or the information of a credit card. In both experiments, the "clean" configuration has been considered the one characterized by the load of traced kernel functions due to the compilation of the Linux kernel 5.5.5. All the trials lasted 10 min and the hidden communication started at the begin of the experiment. We point out that such parameters allowed to consider a wide range of threats (e.g., slow and long communications characterizing advanced persistent threats or malicious applications wanting to exfiltrate as quick as possible sensitive information) while guaranteeing the adequate statistical relevance.

Fig. 3 depicts the results of the first round of tests. As shown, the presence of an exchange of information through a covert channel (denoted as CC in the figure) affects both the number and the distribution of the `__x64_sys_chmod` kernel functions. Specifically, the presence of an anomaly can be detected by the larger number of cumulative invocations of the kernel function with respect to a known baseline (see Fig. 3(a)). We note that the higher the steganographic bandwidth (i.e., Δt decreases) the higher is the load of `__x64_sys_chmod` kernel functions at the begin. In fact, higher transmission rates reduce the time needed to transmit the secret message. This can be viewed in Fig. 3(b), where the instantaneous time evolution is shown. The cumulative number of `__x64_sys_chmod` invocations converges to a common value at the end, since the size of the message is the same in this scenario. Similar results have been observed for the second set of experiments, which are showcased in Fig. 4. In this case, the steganographic bandwidth is fixed and the length of the message is the unique factor that makes the transmission more or less detectable. The difference between cumulative counters at the end of the experiments comes from different message sizes.

For what concerns detection, in general, channels with a higher steganographic bandwidth and longer messages are easier to detect. Indeed, they imply either sudden peaks or larger volumes of `__x64_sys_chmod` kernel functions. Clearly, on-line detection is not straightforward, because of the difficulty to find an effective decision rule able to discriminate between legitimate usage and the presence of hidden transmissions for different use cases. For the case of chmod-stego technique, a possible signature is given by a sudden change in the volume of `__x64_sys_chmod` kernel functions at the end of the trials. This is due to the sender that restores the original file permissions, as to avoid the detection by common file system monitoring tools. Unfortunately, there may be false positives, as the peak in the middle of the kernel compilation. Yet, taking into account additional parameters available from tracing (e.g., the file names) can be used to further improve the likelihood of the detection.

5. Data gathering for covert channels

In this section, we investigate the use of eBPF for gathering information on a threat exfiltrating data through a network covert channel nested in IPv6 traffic. The implementation of the attack is discussed in Section 5.1, while numerical results are presented in Section 5.2.

5.1. IPv6 covert channels and their detection

To implement a realistic IPv6 network covert channel, we developed a prototypal application. Our software is written in Python 3 and makes use of Scapy and NetfilterQueue libraries for retrieving and manipulating network packets, respectively. We point out that our scope is not the weaponization of steganographic threats, so the execution footprints of the secret sender and the secret receiver are not important in our implementation.

Recalling Fig. 2, we aim at exploiting part of the IPv6 protocol data unit to inject the secret information. Even if the literature proposes several mechanisms, we directly store the covert data in the `Flow Label` field of the IPv6 header. This allows to have an adequate steganographic capacity (i.e., 20 bit per packet) and to model an attack effective in realistic scenarios [35,38]. Moreover, recent analyses highlight the fragility of the algorithms used to randomly generate `Flow Label` values in many OSes, thus understanding other potential security flaws is a prime research goal [39].

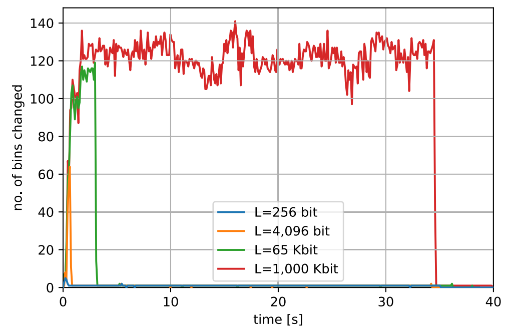
To have the needed volume of carriers, the secret endpoints "high-jack" an overt and licit IPv6 traffic flow in a Man-in-the-Middle fashion. To perform the injection and extraction of secrets, both endpoints use `ip6tables` rules combined with `NFQUEUE` programmatically driven. The overt traffic is routed in the `NFQUEUE` of the sender and triggers the injection function, which retrieves the IPv6 datagram by using the `get_payload()` call. If a packet is candidate for containing a secret (e.g., the sender needs to transmit a character), the original `Flow Label` is altered via the invocation of the `set_payload()` routine. The IPv6 packet is then released and sent through the network. To collect the secret information, the receiver inspects the incoming flow via an `ip6tables` rule as well. If the datagram contains a secret, the `Flow Label` is read and the bit of information is stored. In our testbed, steganographic packets are recognized via a counting scheme: developing more sophisticated and robust signatures is part of our ongoing research.

A detector for this kind of network covert channel should be able to inspect the `Flow Label` field and analyze its usage. In this vein, we developed an ad-hoc eBPF program for inspecting the header of IPv6 packets and attached it to the `tc` queuing subsystem. To guarantee scalability and to optimize performances, our program splits the label space (i.e., the 20-bit space of the values of the various `Flow Label` observed) into a number of equal *bins*, and counts the number of occurrences of the values in each bin. As a companion, we also developed a user-space utility that periodically collects data from the shared map allowing the percolation of information from the kernel to the user space.

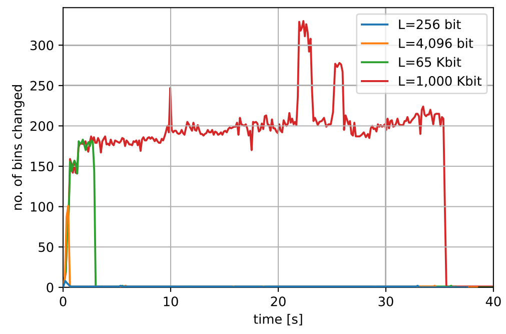
5.2. Numerical results

To evaluate the effectiveness of using eBPF to support the detection of covert network communications, we prepared an experimental testbed. A secret sender and a secret receiver exchange data through the aforementioned IPv6 covert channel running on two Virtual Machines with Debian GNU/Linux 10 (buster) with kernel 4.20.9. The overt traffic used by the two secret endpoints to embed data is an `scp`-based transfer over a native IPv6 network (i.e., no tunneling or additional 4to6 or 6to4 mechanisms were present). We underline that our tests aim at investigating how changes in the `Flow Label` affect the "histogram" measured by our eBPF filter and to understand features that should be considered in the design of effective detection algorithms. For this reason, background traffic has not been considered and more complex investigations are left as future work. A third Virtual Machine with Debian GNU/Linux 10 (buster) with kernel 4.20.9 has been set up to act as an intermediate router running the eBPF program and the user space utility. The eBPF filter was used to parse all the packet headers, extract the `Flow Label` and increase the proper bin, according to the observed value.

To precisely assess the performances of eBPF to support the detection of malware endowed with steganographic communication features, we performed different trials. The first aimed at evaluating the impact of the volume of information to be exfiltrated on the detectability of the covert channel. Hence, we varied the length of the secret message L. Specifically, we considered $L = 256$ bit (e.g., an encryption key



(a) $B = 2^8$, $\Delta s = 100$ ms, $k = 0$ (no interleaving)



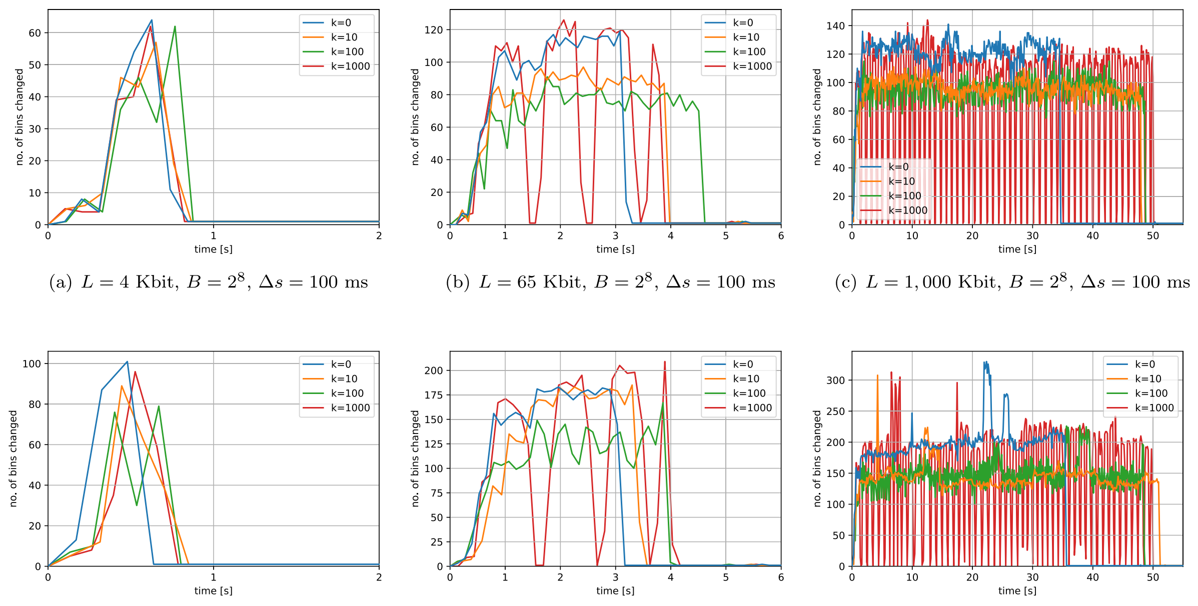
(b) $B = 2^{12}$, $\Delta s = 100$ ms, $k = 0$ (no interleaving)

Fig. 5. Numbers of changing bins with $\Delta s = 100$ ms and $k = 0$ for covert messages with various lengths L and number of bins B.

or a PIN), $L = 4096$ bit and $L = 65$ kb (e.g., multiple address book entries or sensitive data in a textual form), and $L = 1000$ kb (e.g., a highly-compressed image containing an industrial secret). For the second round, we investigated the impact of possible countermeasures deployed by the attacker. We considered a malware using covert channels implementing different "interleaving" policies to make the burst of packets containing secret data stealthier through some form of decorrelation. Thus, we performed trials with steganographic packets interleaved with k "clean" packets, i.e., with the original value of the `Flow Label` used by the legitimate endpoints. In this case, we considered $k = 0, 10, 100$ and 1000 packets, with $k = 0$ denoting a flow without interleaving. The third round of tests addressed the performances of eBPF. We repeated the aforementioned trials by varying the time between two adjacent reads of the traffic measurements via the user space tool, defined in the following as Δs. Specifically, we made trials with $\Delta s = 0.1, 1, 10$ s. We also investigated the "granularity" of the eBPF-capable gathering framework by considering different numbers of bins, denoted as B. Owing to security requirements, eBPF enforce writing programs with precisely-bounded memory usages, thus we were not able to map the `Flow Label` with a resolution greater than 2^{16} bins, i.e., we considered $B = 2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}$.

According to preliminary investigations, we found that $\Delta s = 100$ ms was the best resolution for detecting attacks. In fact, slower "sampling times" can be effective only in the presence of bandwidth-scarce environments or long-lasting communications (e.g., as it happens in advanced persistent threats). Therefore, in the rest of this section, we omit results for $\Delta s = 1$ s and 10 s since the investigation of eBPF in challenging settings is part of our ongoing research. Moreover, both for the sake of clarity and compactness, we will show trials for selected combinations of the parameters.

Fig. 5 shows the number of bins that change between two consecutive sampling intervals. In the case of a legitimate behavior, we expect



(a) $L = 4$ Kbit, $B = 2^8$, $\Delta s = 100$ ms (b) $L = 65$ Kbit, $B = 2^8$, $\Delta s = 100$ ms (c) $L = 1,000$ Kbit, $B = 2^8$, $\Delta s = 100$ ms

(d) $L = 4$ Kbit, $B = 2^{12}$, $\Delta s = 100$ ms (e) $L = 65$ Kbit, $B = 2^{12}$, $\Delta s = 100$ ms (f) $L = 1,000$ Kbit, $B = 2^{12}$, $\Delta s = 100$ ms

Fig. 6. Impact of the message length L and the interleaving scheme k on the number of changing bins with various B and $\Delta s = 100$ ms.

that each active flow uses the same `Flow Label` for its whole duration. Hence, for each sampling interval, the number of changing bins will be limited to 1 if new packets have been observed, otherwise to 0 (i.e., no traffic was present). To avoid burdening the graph, we do not show the curve representing the legitimate behavior, since it would be overlapped to the x-axis. Instead, when the IPv6 traffic is used to embed a covert channel, the different values of the `Flow Label` will spread across multiple bins. Here, the number of bins B plays a role. In fact, low values of B lead to "larger" bins, thus the likelihood that different (but "close") labels will be counted in the same bin increases. This suggests that the detection will be more accurate with a finer-grained partition of the label space (i.e., B increases). Besides, the detection is also affected by the duration of the transmission, which is proportional to the length of the secret message. As shown, very short messages (i.e., for $L = 256$ bit) are very hard to be detected especially with additional background traffic.

A similar investigation is presented in Fig. 6. In this case, we consider a stegomalware using a more sophisticated transmission scheme, i.e., data is exfiltrated in bursts interleaved with trails of k non-steganographic packets. As shown, the bursty mechanism accounts for visible oscillations in the number of changing bins. This is more clear for longer messages (i.e., when L increases), whereas for shorter covert communications the information is completely transmitted in the first burst. To correctly detect the presence of a covert communication, average values of multiple consecutive samples should be considered, or a more fine sampling Δs should be used.

Lastly, Fig. 7 reports how the number of bins B impacts the "visibility" of the covert channel. Specifically, the higher the number of bins, the larger the changes in the distribution of various values of the `Flow Label`. This lets to easily spot a covert communication even in the presence of background traffic. In fact, since almost every IPv6 conversation is characterized by a life-long value for the `Flow Label`, larger discrepancies in how new values are generated (or their statistical distribution, see, e.g., [35,39]) can be used to reveal the presence of the covert channel within the bulk of traffic. Unfortunately, larger granularities require a higher consumption of resources in the node running the eBPF filter. This is supported by the delay and lower number of measurements in the case of 2^{16} bins: despite the sampling time is set to 100 ms, the user-space program takes more than 1 s to acquire the map and to write the output to a file. As a consequence, the related trend is always "late" with respect to the other cases, especially for shorter message lengths. Delays and performance issue will be further discussed in Section 6.

6. Deployability and additional results

Kernel-level tracing can be considered an effective enabler for detecting steganographic attacks that targets both end nodes and network traffic. In general, the technique should be properly integrated in a more complex security framework. For instance, eBPF programs can be used to provide data to specific toolkits aimed at detecting stegomalware or emergent threats (as proposed in Project SIMARGL[5] - Secure Intelligent Methods for Advanced Recognition of Malware and Stegomalware) and they can be dynamically orchestrated at run-time to support multiple detection techniques (as proposed in Project ASTRID[6] - AddreSsing ThReats for virtualIzeD services). We then consider additional aspects related to efficiency and resource consumption, outline possible usage for advanced detection techniques, and point out open issues.

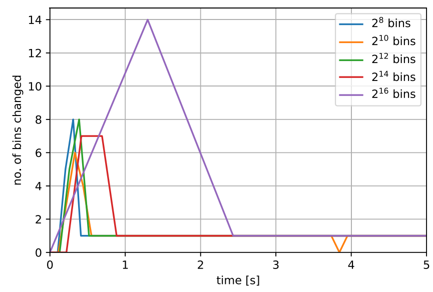
6.1. Resource usage

We took into consideration the impact of the proposed approach on resource consumption. The most relevant use case is still the IPv6 covert channel, where a larger amount of data is collected. As said, eBPF is conceived as a lightweight framework, thus its stack size is limited to 512 bytes and there is no `kmalloc`-style dynamic allocation inside `bpf` programs either. Memory maps for communicating with the user space are allowed to take more space.

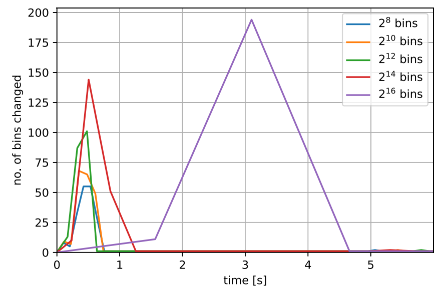
Our eBPF program is only 96 instructions long. Table 1 reports relevant statistics about memory usage when different number of bins are used (the value 0 denotes the baseline scenario when no monitoring is performed). The first section summarizes data reported by the eBPF utilities (`bpftool`, in our case). The middle section shows selected

[5] simargl.eu.

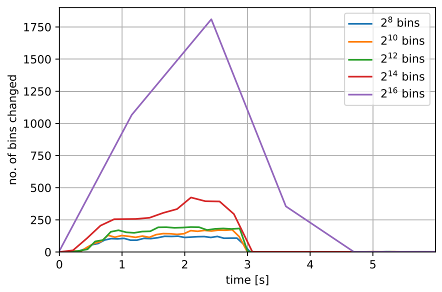
[6] www.astrid-project.eu.



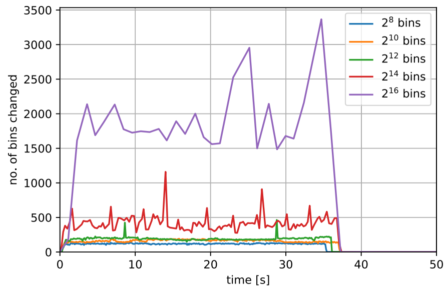
(a) $L = 256$ bit, $\Delta s = 100$ ms, $k = 0$



(b) $L = 4$ Kbit, $\Delta s = 100$ ms, $k = 0$

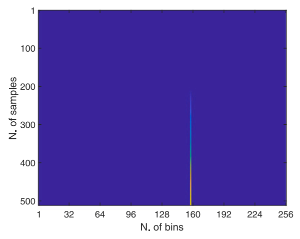


(c) $L = 65$ Kbit, $\Delta s = 100$ ms, $k = 0$

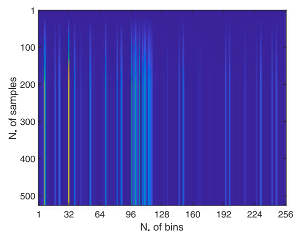


(d) $L = 1,000$ Kbit, $\Delta s = 100$ ms, $k = 0$

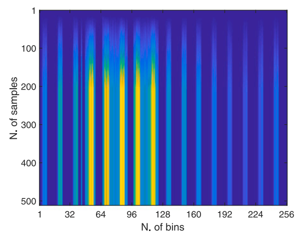
Fig. 7. Impact of the granularity of B on the number of changing bins for different message lengths L with $\Delta s = 100$ ms and $k = 0$ (no interleaving).



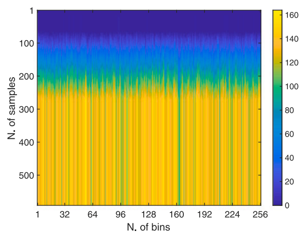
(a) Clean, $\Delta s = 100$ ms



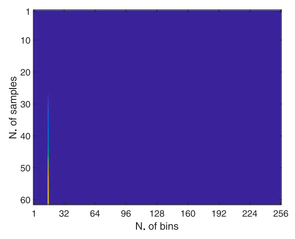
(b) Text, $\Delta s = 100$ ms



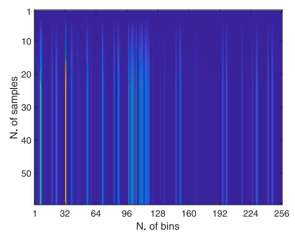
(c) JPG, $\Delta s = 100$ ms



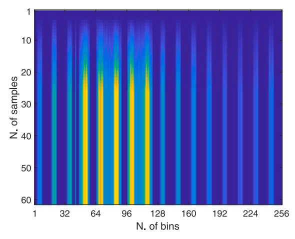
(d) Random, $\Delta s = 100$ ms



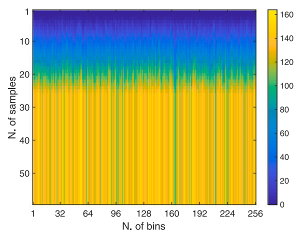
(e) Clean, $\Delta s = 1$ s



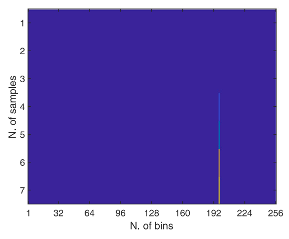
(f) Text, $\Delta s = 1$ s



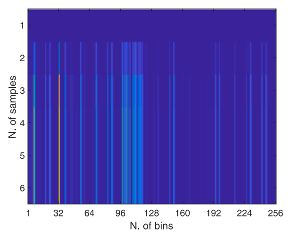
(g) JPG, $\Delta s = 1$ s



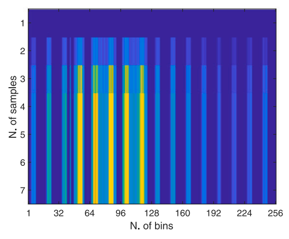
(h) Random, $\Delta s = 1$ s



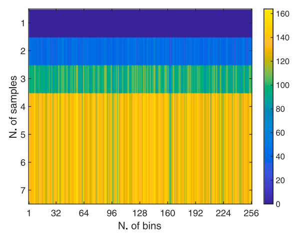
(i) Clean, $\Delta s = 10$ s



(j) Text, $\Delta s = 10$ s



(k) JPG, $\Delta s = 10$ s



(l) Random, $\Delta s = 10$ s

Fig. 8. Heatmaps for various covert transmissions, with $B = 2^8$, and different granularities Δs used for populating bins via eBPF.

Table 1
Memory usage with different number of bins B.

	0	2^8	2^{10}	2^{12}	2^{14}	2^{16}
Max entries	–	256	1024	4096	16,384	65,536
Mem size [kB]	–	4.10	12.30	36.96	135.17	528.39
N. of active slabs	5777	5899	5928	5938	5990	6004
Active slab size [MB]	48.49	49.85	49.94	50.14	50.47	50.53
Total slab size [MB]	50.01	51.35	51.47	51.67	52.01	52.06
Free memory [MB]	1715.27	1620.13	1620.81	1619.62	1619.00	1618.26
Mapped memory [MB]	81,644.00	81,728.00	81,720.00	81,740.00	81,772.00	81,808.00

Table 2
Real sampling time experienced by the user-space program.

N. of bins	2^8	2^{10}	2^{12}	2^{14}	2^{16}
$\Delta s'$ [ms]	109.40	120.40	233.48	351.13	1391.91

relevant measures of the used cache (number of active slabs[7] and their size as reported by `slabtop`). The last section reports a subset of information available from `/proc/meminfo`. Even if the memory required may increase by few kilobytes when the number of bins grows, the impact on the kernel cache is rather limited. The relative impact on the overall memory is even more limited.

Another potential performance bottleneck concerns the time needed to run an eBPF program, since this overhead is present for each packet. We took simple measurements from the Virtual Machine where the eBPF program ran in the IPv6 testbed. This machine has 4 virtual cores and 2 GB of RAM assigned and it was deployed over an Intel server with two Xeon E5-2660 v4 processors running at 2.00 GHz. Our measurements point out that such time is very small, 104.48 ns on the average. In our trials, the maximum and minimum execution times observed were $19,715$ ns and 49 ns, respectively. To be more precise, we also evaluated the cumulative execution time taken when our eBPF program is run for each packet composing a 1.2 GB file transfer (accounting for a total of more than 75,000 packets). The obtained total value is $\sim$7 ms, which leads to an almost negligible overhead.

Unfortunately, the user-space application suffers performance issues. To give a simple yet meaningful example, we set the desired sampling time $\Delta s = 100$ ms and tested our framework with different values of B. Table 2 reports the real sampling times obtained, denoted as $\Delta s'$. As shown, the user-space counterpart of our eBPF program is able to follow the expected working frequency only for the lowest number of bins. This can be mainly ascribed to the need of saving data on the file system, which is slower than the RAM. To deploy such a solution in production-quality environments, a more realistic implementation should not save all measurements on a file, so this problem could be largely mitigated.

6.2. Envisioned applications

Information gathered by kernel-level tracing and inspection can feed detection algorithms and analytics engines. Obtained insights can be directly delivered to streaming analytics in a (quasi) real-time fashion or can be used to create large collections of datasets, which is a key challenge to address for developing effective detectors based on Machine Learning or Artificial Intelligence techniques [40]. In fact, AI-capable frameworks require heterogeneous and rich information to provide satisfactory statistical performances or to discover relations “invisible” to methodologies based on the common sense. The identification of a proper set of “features” that contain relevant information for training the machine is probably the most challenging aspect for application of ML, that usually require to implement multiple kinds of measurements. In this perspective, code augmentation provides an effective and flexible mechanism for building the required feature set, especially when a measurement campaign cannot be planned *a priori*. Thus, a trial and error approach can be easily implemented by stacking multiple filters for having a suitable volume of information for feature engineering purposes or to compute efficient high-level indicators allowing to decouple the detection pipeline from the specific steganographic approach.

To understand whether our set of measurements could be used to feed a ML detector, we searched for potential correlation patterns of the `Flow Label` distribution and content of the hidden message. Therefore, we performed additional tests keeping both the same testbed and parameters as in Section 5, but varying the type of secret message exfiltrated by the malware via the IPv6 covert channel. Specifically, we considered the transmission of a text file encoded in ASCII, a JPG and a randomly generated string. Obtained value-to-bin mappings of the `Flow Label` have been depicted via heatmaps. Fig. 8 showcases selected results with different sampling times of the user-space program. Figs. 8(a), 8(e), and 8(i) depict maps of a clean IPv6 conversation, i.e., only the bin corresponding to the original value of the `Flow Label` populates depending on the produced traffic (i.e., the volume of packets).

According to the figure, correlations characterizing each content can be easily spotted via kernel-level tracing. For instance, for the case of $\Delta s = 100$ ms, the data depicted in Figs. 8(b)–8(d) can be used to train some AI for identifying the exfiltrated content. Hence, the proposed eBPF-based approach can be used in a sort of 2-tier blueprint: a first layer for detecting the presence of stegomalware by inspecting the number of changing bins for the IPv6 traffic load, and a second layer for guessing whether the covert channel is used to send commands, orchestrate a botnet or which type of information is being exfiltrated, e.g., textual or multimedia.

Based on our results, we believe that AI methodologies can be “overlaid” on top of standard tools to improve the performance of the detection, for instance to identify the severity of exfiltration. This may also be useful to support the decision process, namely to decide which type of countermeasure should be deployed. For instance, recognizing a covert channel transporting parameters for configuring a backdoor may trigger an update in the rules of a firewall or in an intrusion detection system.

As a final remark, we argue that the additional code injected in the kernel should not introduce bottlenecks, mainly to avoid degradations in the performances experienced by end users. Concerning the considered channels, filters and eBPF programs should be then able to efficiently map the “space” generated by the various values of the `Flow Label`. We already investigated this aspect in the previous Section; here, we consider how scalability might affect the detection accuracy. Our results show that the correlation is visible even in the presence of coarse-grained measurements. Despite being $B = 256$ bins, Figs. 8(j)–8(l) computed from data gathered by the eBPF program with $\Delta s = 10$ s still offer informative insights.

[7] Slabs are small portions of Linux caches and they build the “slab layer”. Each cache stores a different type of temporary objects, such as task or device structures and inodes. The slab layer improves performance by addressing efficient allocation/deallocation of frequent data structures, memory fragmentation, intelligent allocation decisions, symmetric multi-processors, etc.

6.3. Open points and limits of the approach

In order to deploy eBPF for detecting stegomalware in production-quality scenarios, some open research points have to be addressed. First, although we gave some insights about how our measurements could be used for attack detection, the design of concrete detectors falls outside the scope of this paper. We elaborated on the usage of data gathering techniques jointly with some form of machine learning or statistical tool, but the benefit of this approach against de-facto standard mechanisms like rule-based ones has to be quantified. In fact, many existing works highlighted that the need of manual labeling, the lack of scalability and the composite nature of datasets are prime obstacles for successfully mixing AI and cybersecurity [41,42]. For instance, the detection of changes in the file permissions usually falls under the scope of continuous integrity verification; hence, when two colluding applications try to communicate by altering the file-system, an efficient detection scheme may exploit inconsistencies or anomalous patterns in the access permissions (see, e.g., [43] for the case of NTFS). Similarly, the detection of covert channels leveraging the `Flow Label` field in the IPv6 header is rather straightforward if state information is kept for each flow, but this is computationally expensive and does not scale well with the number of active flows.

Second, samples of stegomalware (including threats implementing a colluding applications scheme) and IPv6 covert channels collected in-the-wild are very limited. Thus, as it often happens in the literature, our investigation is based on non-weaponized colluding applications and covert channel attacks [1,2,7,8,18,35].

Third, detecting this type of threats requires some *a priori* knowledge of the steganographic method used by the attacker (e.g., where the secret is embedded). To this extent, the flexibility of the eBPF is surely a plus, as it allows to develop in a quite simple manner several filters to differentiate the collection of data, which can be extended to consider different carriers or scenarios. Moreover, eBPF and kernel-based data gathering should be also evaluated as tools for obtaining high-level and threat-independent indicators. In addition, also the possibility to automatically generate and run new programs is really interesting yet very challenging, but requires deep usage of AI techniques and it mostly represents a long-term objective.

Lastly, the impact of software layers for gathering data and detect stegomalware should be better understood. For instance, tracing tools running on mobile devices could deplete the battery or be undeployable in resource-constrained devices. Besides, network traffic could experience additional delays and jitter impacting on the Quality of Experience of users. Thus the tradeoffs of resorting to this type of analysis should be precisely evaluated.

A limit of the proposed approach concerns the tight dependence on the Linux kernel. Even if many network devices and appliances run Linux, this OS has not the same penetration on end nodes (with the exception of Android OS). Thus, revealing steganographic attacks like colluding applications could require platform-dependent approaches or to shift the detection in the network or in some edge/cloud components. Again, this requires methods to efficiently collect various type of data, thus kernel-level measurements still deserve deeper investigation.

7. Conclusions

In this paper, we showcased how eBPF, a code augmentation framework offered by the Linux kernel, can be used for programmatically tracing and monitoring the behavior of software processes and network traffic with the aim of detecting stegomalware. To prove the effectiveness of the idea, we evaluated the use of eBPF to gather data in two different use cases. In the first, we showed how it can be used to trace specific system calls when an attack based on the colluding applications scheme is ongoing. In the second, we developed an eBPF program to evaluate the behavior of the `Flow Label` field when used to implement a covert channel within bulk of the IPv6 traffic. In both cases, results indicated that in-kernel measurement via code augmentation can be used to gather data to feed toolkits for detecting stegomalware. In addition, eBPF demonstrated to be flexible enough to provide information for more sophisticated detection frameworks, e.g., to feed detection models or create datasets for machine-learning-capable techniques.

Future work aims at refining the proposed approach. In particular, the main objective is the definition of a more programmatic process to progressively narrow down the scope from generic indicators to fine-grained tracing of execution patterns. This can be also applied to network covert channels, e.g., to shift the detection from traffic analysis to code inspection. In this respect, ongoing research deals with the development of threat-independent signatures such as energy consumption, RAM usage patterns, and the time statistics of running processes. Another relevant portion of our ongoing research extends and generalizes the approach and using eBPF to detect a wider array of threats such as cryptolockers and advanced persistent threats. In this vein, future developments also aim at using some form of artificial intelligence or machine learning in order to make the detection more effective and scalable, possibly by also identifying the class of the exfiltrated information (e.g., an encryption key, a command of a botnet or a document).

CRediT authorship contribution statement

Luca Caviglione: Conceptualization, Methodology, Writing - original draft, Supervision, Funding acquisition. **Wojciech Mazurczyk:** Conceptualization, Methodology, Writing - original draft, Funding acquisition. **Matteo Repetto:** Conceptualization, Methodology, Writing - original draft, Software, Validation, Funding acquisition. **Andreas Schaffhauser:** Software, Validation, Investigation. **Marco Zuppelli:** Conceptualization, Visualization, Investigation, Software, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been supported by the EU Project ASTRID - AddreSsing ThReats for virtualIzeD services, Grant Agreement No 786922, and by the EU Project SIMARGL - Secure Intelligent Methods for Advanced Recognition of Malware and Stegomalware, Grant Agreement No 833042.

References

[1] W. Mazurczyk, L. Caviglione, Information hiding as a challenge for malware detection, IEEE Secur. Priv. 13 (2) (2015) 89–93.
[2] K. Cabaj, L. Caviglione, W. Mazurczyk, S. Wendzel, A. Woodward, S. Zander, The new threats of information hiding: The road ahead, IT Prof. 20 (3) (2018) 31–39.
[3] R. Rapuzzi, M. Repetto, Building situational awareness for network threats in fog/edge computing: Emerging paradigms beyond the security perimeter model, Future Gener. Comput. Syst. 85 (2018) 235–249.
[4] M. Ahmed, A.N. Mahmood, J. Hu, A survey of network anomaly detection techniques, J. Netw. Comput. Appl. 60 (2016) 19–31.
[5] P. Yan, Z. Yan, A survey on dynamic mobile malware detection, Softw. Qual. J. 26 (3) (2018) 891–919.
[6] D. Ding, Q.-L. Han, Y. Xiang, X. Ge, X.-M. Zhang, A survey on security control and attack detection for industrial cyber-physical systems, Neurocomputing 275 (2018) 1674–1683.
[7] S. Zander, G. Armitage, P. Branch, A survey of covert channels and countermeasures in computer network protocols, IEEE Commun. Surv. Tutor. 9 (3) (2007) 44–57.
[8] W. Mazurczyk, L. Caviglione, Steganography in modern smartphones and mitigation techniques, IEEE Commu. Surv. Tutor. 17 (1) (2014) 334–357.

[9] A. Carrega, L. Caviglione, M. Repetto, M. Zuppelli, Programmable data gathering for detecting stegomalware, in: Proceedings of the 2nd International Workshop on Cyber-Security Threats, Trust and Privacy Management in Software-Defined and Virtualized Infrastructures (SecSoft), IEEE, 2020.
[10] M. Repetto, A. Carrega, G. Lamanna, An architecture to manage security services for cloud applications, in: 4th IEEE International Conference on Computing, Communication & Security (ICCCS-2019), Rome, Italy, 2019.
[11] S. Covaci, R. Rapuzzi, M. Repetto, F. Risso, A new paradigm to address threats for virtualized services, in: IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Tokyo, Japan, 2018, pp. 689–694.
[12] Y. Sun, S. Nanda, T. Jaeger, Security-as-a-service for microservices-based cloud applications, in: 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2015, pp. 50–57.
[13] M. Bouet, J. Leguay, V. Conan, Cost-based placement of virtualized deep packet inspection functions in sdn, in: MILCOM 2013-2013 IEEE Military Communications Conference, IEEE, 2013, pp. 992–997.
[14] J. Li, B. Li, T. Wo, C. Hu, J. Huai, L. Liu, K. Lam, Cyberguarder: A virtualization security assurance architecture for green cloud computing, Future Gener. Comput. Syst. 28 (2) (2012) 379–390.
[15] Z. Lin, D. Tao, Z. Wang, Dynamic construction scheme for virtualization security service in software-defined networks, Sensors 17 (4) (2017) 920.
[16] H.-Y. Tsai, M. Siebenhaar, A. Miede, Y. Huang, R. Steinmetz, Threat as a service?: Virtualization's impact on cloud security, IT Prof. 14 (1) (2011) 32–37.
[17] X. Luo, L. Yang, L. Ma, S. Chu, H. Dai, Virtualization security risks and solutions of cloud computing via divide-conquer strategy, in: 2011 Third International Conference on Multimedia Information Networking and Security, IEEE, 2011, pp. 637–641.
[18] S. Wendzel, S. Zander, B. Fechner, C. Herdin, Pattern-based survey and categorization of network covert channel techniques, ACM Comput. Surv. 47 (3) (2015) 1–26.
[19] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, H. Wang, Containerleaks: Emerging security threats of information leakages in container clouds, in: 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, 2017, pp. 237–248.
[20] Y. Zhang, A. Juels, M.K. Reiter, T. Ristenpart, Cross-vm side channels and their use to extract private keys, in: Proceedings of the 2012 ACM Conference on Computer and Communications Security, 2012, pp. 305–316.
[21] R. Ning, C. Wang, C. Xin, J. Li, L. Zhu, H. Wu, Capjack: Capture in-browser crypto-jacking by deep capsule network through behavioral analysis, in: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, 2019, pp. 1873–1881.
[22] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, M. Bailey, Outguard: Detecting in-browser covert cryptocurrency mining in the wild, in: The World Wide Web Conference, WWW '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 840–852.
[23] M. Caprolu, S. Raponi, G. Oligeri, R.D. Pietro, Cryptomining makes noise: a machine learning approach for cryptojacking detection, 2019, arXiv:1910.09272.
[24] N. Scaife, H. Carter, P. Traynor, K.R. Butler, Cryptolock (and drop it): stopping ransomware attacks on user data, in: Proceedings of the 36th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2016, pp. 303–312.
[25] L. Deri, S. Sabella, S. Mainardi, Combining system visibility and security using eBPF, in: P. Degano, R. Zunino (Eds.), Proceedings of the Third Italian Conference on Cyber Security, Pisa, Italy, February 13-15, 2019, in: CEUR Workshop Proceedings, vol. 2315, CEUR-WS.org, 2019.
[26] Ł. Makowski, P. Grosso, Evaluation of virtualization and traffic filtering methods for container networks, Future Gener. Comput. Syst. 93 (2019) 345–357.
[27] M.A. Vieira, M.S. Castanho, R.D. Pacífico, E.R. Santos, E.P.C. Júnior, L.F. Vieira, Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications, ACM Comput. Surv. 53 (1) (2020) 1–36.
[28] M. Bélair, S. Laniepce, J.-M. Menaud, Leveraging kernel security mechanisms to improve container security: A survey, in: Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES '19, Association for Computing Machinery, New York, NY, USA, 2019.
[29] M. Dimjašević, S. Atzeni, I. Ugrina, Z. Rakamaric, Evaluation of android malware detection based on system calls, in: Proceedings of the 2016 ACM on International Workshop on Security and Privacy Analytics, IWSPA '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 1–8.
[30] J. Rhee, R. Riley, Z. Lin, X. Jiang, D. Xu, Data-centric os kernel malware characterization, IEEE Trans. Inf. Forensics Secur. 9 (1) (2014) 72–87.
[31] L. Caviglione, M. Gaggero, J.-F. Lalande, W. Mazurczyk, M. Urbański, Seeing the unseen: revealing mobile malware hidden communications via energy consumption and artificial intelligence, IEEE Trans. Inf. Forensics Secur. 11 (4) (2015) 799–810.
[32] M. Urbanski, W. Mazurczyk, J.-F. Lalande, L. Caviglione, Detecting local covert channels using process activity correlation on android smartphones, Int. J. Comput. Syst. Sci. Eng. 32 (2) (2017) 71–80.
[33] C. Marforio, H. Ritzdorf, A. Francillon, S. Capkun, Analysis of the communication between colluding applications on modern smartphones, in: Proceedings of the 28th Annual Computer Security Applications Conference, 2012, pp. 51–60.
[34] N. Lucena, G. Lewandowski, S. Chapin, Covert channels in ipv6, in: Int. Workshop on Privacy Enhancing Technologies, Springer, 2005, pp. 147–166.
[35] W. Mazurczyk, K. Powójski, L. Caviglione, IPv6 covert channels in the wild, in: Proceedings of the 3rd Central European Cybersecurity Conference, 2019, pp. 1–6.
[36] B.M. Cantrill, M.W. Shapiro, A.H. Leventhal, Dynamic instrumentation of production systems, in: Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC04).
[37] S. McCanne, V. Jacobson, The BSD packet filter: A new architecture for user-level packet capture, in: 1993 Winter USENIX conference, San Diego, CA, USA, 1993.
[38] W. Mazurczyk, P. Szary, S. Wendzel, L. Caviglione, Towards reversible storage network covert channels, in: Proceedings of the 14th International Conference on Availability, Reliability and Security, 2019, pp. 1–8.
[39] J. Berger, A. Klein, B. Pinkas, Flaw label: Exploiting IPv6 flow label, in: Proceedings of the 2020 IEEE Symposium on Security and Privacy, IEEE, 2020, pp. 1594–1611.
[40] A.L. Buczak, E. Guven, A survey of data mining and machine learning methods for cyber security intrusion detection, IEEE Commun. Surv. Tutor. 18 (2) (2015) 1153–1176.
[41] D.S. Berman, A.L. Buczak, J.S. Chavis, C.L. Corbett, A survey of deep learning methods for cyber security, Information 10 (4) (2019) 122.
[42] S. Mahdavifar, A.A. Ghorbani, Application of deep learning to cybersecurity: A survey, Neurocomputing 347 (2019) 149–176.
[43] S. Parkinson, V. Somaraki, R. Ward, Auditing file system permissions using association rule mining, Expert Syst. Appl. 55 (2016) 274–283.

Luca Caviglione is a Senior Research Scientist at the Institute for Applied Mathematics and Information Technologies of the National Research Council of Italy. He holds a Ph.D. in Electronic and Computer Engineering from the University of Genoa, Italy. His research interests include optimization of large-scale computing frameworks, traffic analysis, and network security. He is an author or co-author of more than 150 academic publications, and several patents in the field of p2p and energy-aware computing. He has been involved in Research Projects and Network of Excellences funded by the ESA, the EU and the MIUR. Currently he is the Principal Investigator for IMATI of the project "Secure Intelligent Methods for Advanced Recognition of Malware and Stegomalware" (SIMARGL – call H2020-SU-ICT-2018, grant no. 833042). He is a Work Group Leader of the Italian IPv6 Task Force, a contract Professor in the field of networking/security and a Professional Engineer. From 2016 he is an Associate Editor of the International Journal of Computers and Applications, Taylor&Francis. He is the head of the IMATI Research Unit of the National Inter-University Consortium for Telecommunications and part of the Steering Committee of the Criminal Use of Information Hiding initiative.

Wojciech Mazurczyk received the B.Sc., M.Sc., Ph.D. (Hons.), and D.Sc. (habilitation) degrees in telecommunications from the Warsaw University of Technology (WUT), Warsaw, Poland, in 2003, 2004, 2009, and 2014, respectively. He is currently a Professor with the Institute of Computer Science at WUT and a head of the Computer Systems Security Group. He also works as a Researcher at the Parallelism and VLSI Group at Faculty of Mathematics and Computer Science at FernUniversitaet, Germany. His research interests include bio-inspired cybersecurity and networking, information hiding, and network security. He is involved in the technical program committee of many international conferences and also serves as a reviewer for major international magazines and journals. From 2016 he is Editor-in-Chief of an open access Journal of Cyber Security and Mobility, and from 2018 he is serving as an Associate Editor of the IEEE Transactions on Information Forensics and Security and as Associate Technical Editor for the IEEE Communications Magazine. He is also a Senior Member of IEEE.

Matteo Repetto received the Ph.D. degree in Electronics and Informatics in 2004 from the University of Genoa. From 2004 to 2009 he was a postdoc at University of Genoa. From 2010 to 2019 he was Research Associate at CNIT. Since 2019 he is a Research Scientist at the Institute for Applied Mathematics and Information Technologies of the National Research Council of Italy. He has been involved in many different national and European research projects in the networking area funded by the EC, the Italian MIUR and private organizations, both as researchers and principal investigator. Currently, he is the Scientific and Technical Coordinator of the projects "AddreSsing Threats for viRtualIzeD services" (ASTRID – call H2020-DS-2016-2017, grant no. 786922) and "A cybersecurity framework to GUArantee Reliability and trust for Digital service chains" (GUARD – call H2020-SU-ICT-2018, grant no. 833456). His research interests include mobility in data networks, virtualization and cloud computing, network functions virtualization and service functions chaining, network and service security. He has co-authored over 50 scientific publications in international journals and conference proceedings.

Andreas Schaffhauser is a Ph.D. student at FernUniversitaet in Hagen and a lecturer for C/C++ and Java at the University of Applied Sciences in Saarbrücken, Germany. Within the Project SIMARGL his research interests are the development of countermeasures against network covert channels and detection methods against LSB Steganography. His other research interests also include symmetric cryptography and IT security in general.

Marco Zuppelli is a Ph.D. student at University of Genova and a research fellow at the Institute for Applied Mathematics and Information Technologies of the National Research Council of Italy. Within the framework of the Project SIMARGL — Secure Intelligent Methods for Advanced Recognition of Malware and Stegomalware, funded by the European Union within the framework of H2020. His research interests include detection methods for steganographic malware and development of new countermeasures against information-hiding-capable threats (including network IPv6-based covert channel and eBPF to detect malicious activities).