

Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi

**Detection and Mitigation of
Steganographic Malware**

by

Marco Zuppelli

Theses Series

DIBRIS-TH-2023-XXXV

DIBRIS, Università degli Studi di Genova

Via All'Opera Pia, 13, 16145 Genova, GE, Italia

<http://www.dibris.unige.it/>

Università degli Studi di Genova

Dipartimento di Informatica, Bioingegneria,

Robotica ed Ingegneria dei Sistemi

Ph.D. Thesis in Computer Science and Systems Engineering

Secure and Reliable Systems Curriculum

**Detection and Mitigation of
Steganographic Malware**

by

Marco Zuppelli

January, 2023

Dottorato di Ricerca in Informatica ed Ingegneria dei Sistemi
Indirizzo Informatica
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università degli Studi di Genova

DIBRIS, Università degli Studi di Genova
Via Opera Pia, 13
I-16145 Genova, Italy
<http://www.dibris.unige.it/>

Ph.D. Thesis in Computer Science and Systems Engineering
Secure and Reliable Systems Curriculum
(S.S.D. INF/01)

Submitted by
Marco Zuppelli
DIBRIS, Università degli Studi di Genova

....

Date of submission: October 2022

Title: Detection and Mitigation of Steganographic Malware

Advisors: Luca Caviglione, Alessio Merlo

Istituto di Matematica Applicata e Tecnologie Informatiche
Consiglio Nazionale delle Ricerche

Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università degli Studi di Genova

...

Ext. Reviewers:

Prof. Luca Veltri - Università degli Studi di Parma - luca.veltri@unipr.it
Prof. Steffen Wendzel - Hochschule Worms - wendzel@hs-worms.de

Abstract

A new attack trend concerns the use of some form of steganography and information hiding to make malware stealthier and able to elude many standard security mechanisms. Therefore, this Thesis addresses the detection and the mitigation of this class of threats¹. In particular, it considers malware implementing covert communications within network traffic or cloaking malicious payloads within digital images.

The first research contribution of this Thesis is in the detection of network covert channels. Unfortunately, the literature on the topic lacks of real traffic traces or attack samples to perform precise tests or security assessments. Thus, a propaedeutic research activity has been devoted to develop two ad-hoc tools. The first allows to create covert channels targeting the IPv6 protocol by eavesdropping flows, whereas the second allows to embed secret data within arbitrary traffic traces that can be replayed to perform investigations in realistic conditions. This Thesis then starts with a security assessment concerning the impact of hidden network communications in production-quality scenarios. Results have been obtained by considering channels cloaking data in the most popular protocols (e.g., TLS, IPv4/v6, and ICMPv4/v6) and showcased that de-facto standard intrusion detection systems and firewalls (i.e., Snort, Suricata, and Zeek) are unable to spot this class of hazards. Since malware can conceal information (e.g., commands and configuration files) in almost every protocol, traffic feature or network element, configuring or adapting pre-existent security solutions could be not straightforward. Moreover, inspecting multiple protocols, fields or conversations at the same time could lead to performance issues. Thus, a major effort has been devoted to develop a suite based on the extended Berkeley Packet Filter (eBPF) to gain visibility over different network protocols/components and to efficiently collect various performance indicators or statistics by using a unique technology. This part of research allowed to spot the presence of network covert channels targeting the header of the IPv6 protocol or the inter-packet time of generic network conversations. In addition, the approach

¹The research presented in this Thesis has been partially funded by the Horizon 2020 Program through the Project Secure Intelligent Methods for Advanced RecoGnition of malware and stegomalware - SIMARGL, H2020-SU-ICT-01-2018, Grant Agreement No. 830929.

based on eBPF turned out to be very flexible and also allowed to reveal hidden data transfers between two processes co-located within the same host. Another important contribution of this part of the Thesis concerns the deployment of the suite in realistic scenarios and its comparison with other similar tools. Specifically, a thorough performance evaluation demonstrated that eBPF can be used to inspect traffic and reveal the presence of covert communications also when in the presence of high loads, e.g., it can sustain rates up to 3 Gbit/s with commodity hardware. To further address the problem of revealing network covert channels in realistic environments, this Thesis also investigates malware targeting traffic generated by Internet of Things devices. In this case, an incremental ensemble of autoencoders has been considered to face the “unknown” location of the hidden data generated by a threat covertly exchanging commands towards a remote attacker.

The second research contribution of this Thesis is in the detection of malicious payloads hidden within digital images. In fact, the majority of real-world malware exploits hiding methods based on Least Significant Bit steganography and some of its variants, such as the Invoke-PSImage mechanism. Therefore, a relevant amount of research has been done to detect the presence of hidden data and classify the payload (e.g., malicious PowerShell scripts or PHP fragments). To this aim, mechanisms leveraging Deep Neural Networks (DNNs) proved to be flexible and effective since they can learn by combining raw low-level data and can be updated or retrained to consider unseen payloads or images with different features. To take into account realistic threat models, this Thesis studies malware targeting different types of images (i.e., favicons and icons) and various payloads (e.g., URLs and Ethereum addresses, as well as webshells). Obtained results showcased that DNNs can be considered a valid tool for spotting the presence of hidden contents since their detection accuracy is always above $\sim 90\%$ also when facing “elusion” mechanisms such as basic obfuscation techniques or alternative encoding schemes. Lastly, when detection or classification are not possible (e.g., due to resource constraints), approaches enforcing “sanitization” can be applied. Thus, this Thesis also considers autoencoders able to disrupt hidden malicious contents without degrading the quality of the image.

Acknowledgements

I would like to thank my supervisors Luca and Alessio for giving me this opportunity.

I would also like to thank Matteo, Massimo, Nunzio, Giuseppe, Wojciech, and all the people I have had the pleasure and honor of collaborating with during these years.

I would like to thank all the friends and colleagues from IMATI for being close to me, for the support, but above all, for the “yogurtino” breaks.

Finally, I would like to express my gratitude to my family for their understanding, encouragement, and support all through my studies.

This work has been partially supported and funded by the Horizon 2020 Program through the Project Secure Intelligent Methods for Advanced RecoGnition of malware and stegomalware - SIMARGL, H2020-SU-ICT-01-2018, Grant Agreement No. 830929.

Table of Contents

I Preliminaries	6
Chapter 1 Introduction	7
1.1 Information Hiding	7
1.2 Covert Channels	8
1.2.1 Local Covert Channels	11
1.2.2 Network Covert Channels	12
1.2.3 Detection of Network Covert Channels	15
1.3 Digital Media Steganography	18
1.3.1 Detection of Malware Hidden in Digital Images	20
1.4 Real Attacks	21
1.5 Research Contributions and Thesis Outline	23
II Detection of Covert Channels	25
Chapter 2 Testing Tools and Security Assessment	26
2.1 Offline Creation of Covert Channels	27
2.1.1 Software Architecture	29
2.1.2 Performance of the Tool	31
2.1.2.1 Data Injection and Replaying	31
2.1.2.2 Generation of Metrics	33

2.1.2.3	Resource Usage	35
2.2	Online Creation of Covert Channels	38
2.2.1	Covert Channels in IPv6	38
2.2.2	Covert Channels in TLS	39
2.3	Security Assessment	40
2.3.1	Experimental Setup and Results	41
2.4	Conclusions and Future Works	44
Chapter 3 Data Gathering for Covert Channels		45
3.1	Collecting Statistics From Protocol Headers	45
3.2	The <code>bccstego</code> Framework	48
3.2.1	The eBPF Technology	48
3.2.2	The <code>ipstats.py</code> Tool	50
3.3	Numerical Results	52
3.3.1	Towards the Detection of Covert Channels	55
3.4	Other Tracing Technologies	57
3.5	Conclusions and Future Works	58
Chapter 4 Detection of Covert Channels via Kernel-level Tracing		59
4.1	Data Gathering for Colluding Applications	60
4.1.1	<code>chmod</code> -based Stegomalware and its Detection	60
4.1.2	Numerical Results	61
4.2	Data Gathering for Network Covert Channels	64
4.2.1	IPv6 Covert Channels and Their Detection	64
4.2.2	Numerical Results	64
4.3	Deployability and Additional Results	68
4.3.1	Resource Usage	68
4.3.2	Envisioned Applications	69

4.3.3	Open Points and Limits of the Approach	71
4.4	Conclusions and Future Works	72
Chapter 5	Detection of Network Covert Channels via Code Layering	74
5.1	Reference Architecture	75
5.2	Threat Model	77
5.3	Experimental Setup	78
5.4	Detection of Storage Covert Channels	80
5.4.1	Detection of Channels Targeting the Flow Label	81
5.4.2	Sensitivity Analysis	82
5.4.3	Channels Targeting Other IPv6 Fields	85
5.5	Detection of Timing Covert Channels	86
5.5.1	Numerical Results	87
5.6	Performance Comparison	88
5.6.1	Impact on Packet Transmission	89
5.6.2	CPU and Memory Usage	90
5.7	eBPF for Security Tasks	92
5.8	Conclusions and Future Works	94
Chapter 6	Towards a Real-World Deployment	95
6.1	Pros and Cons of Monitoring and Inspection Processes	95
6.2	Monitoring Technologies	96
6.2.1	Extending Zeek to Handle Network Covert Channels	97
6.2.2	Implementing the bin-based Approach with libpcap	98
6.3	Complexity Analysis	98
6.4	Performance Evaluation	99
6.4.1	Impact on Packet Transmission	100
6.4.2	CPU Usage	103

6.4.3	Memory Allocation	104
6.5	Conclusions and Future Works	108
Chapter 7	Detection of Network Covert Channels via AI	109
7.1	Attack Model and Design of the Covert Channel	110
7.2	Deep Ensemble Learning Scheme	112
7.2.1	Detection Through a Single Autoencoder	113
7.2.2	Learning and Combining Different Detectors	115
7.3	Performance Evaluation	116
7.3.1	Dataset Preparation	116
7.3.2	Preprocessing, Parameters and Evaluation Metrics	117
7.3.3	Numerical Results	118
7.4	Conclusions and Future Works	121
III	Detection of Malware Hidden in Digital Images	122
Chapter 8	Revealing Threats in Favicons via AI	123
8.1	Attack Model and Solution Approach	124
8.2	Detection via Deep Learning Models	126
8.3	Performance Evaluation	128
8.3.1	Dataset Preparation	128
8.3.2	Parameters, Evaluation Metrics, and Testbed	129
8.3.3	Numerical Results	130
8.4	Conclusions and Future Works	132
Chapter 9	Revealing Threats in Icons via AI	134
9.1	Attack Model	135
9.2	Detection Approach	136

9.2.1	Architectural Blueprint	137
9.2.2	Hidden Content Detection Approach	138
9.2.3	Neural Detector Architecture	139
9.3	Performance Evaluation	141
9.3.1	Design of Attacks and Dataset Preparation	142
9.3.2	Parameters, Evaluation Metrics, and Testbed	144
9.3.3	Numerical Results	144
9.4	Conclusions and Future Works	148
Chapter 10	Sanitization of Images Containing Stegomalware via AI	149
10.1	Background	150
10.1.1	Stegomalware and Attack Model	150
10.1.2	Machine Learning for Image Processing	151
10.2	Sanitization Through Machine Learning	152
10.2.1	Architectural Blueprint	152
10.2.2	Methodology	153
10.3	Performance Evaluation	154
10.4	Conclusions and Future Works	158
IV	Conclusions and Appendices	159
Chapter 11	Conclusions and Future Works	160
Bibliography		164
Appendix A	Software and Datasets	185
Appendix B	Publications Used in This Thesis	187
Appendix C	Publications Made During the PhD	192

Part I

Preliminaries

Chapter 1

Introduction

This Thesis deals with the detection of malware endowed with information hiding and steganographic techniques, which can be used to avoid detection, bypass security perimeters or exfiltrate sensitive information without being noticed. Accordingly, this class of threats is often identified as *stegomalware*, i.e., steganographic malware. As it will be detailed later, the majority of attacks observed in real scenarios mainly cloaks information in network traffic and digital images. Therefore, this Thesis faces the problems related to the detection and mitigation of stegomalware hiding arbitrary data in network conversations and concealing additional payloads in various types of images.

1.1 Information Hiding

The umbrella term *information hiding* includes the broad range of disciplines used to conceal messages, secret information, or arbitrary data within various digital objects defined as *carriers*. Such techniques are typically used for different purposes, for instance, they allow to protect intellectual properties or enforce copyright by avoiding duplication, illegal distribution or tampering of software components and digital objects. The latter can be also uniquely identified via a “hidden” fingerprint, e.g., a serial number, which prevents unauthorized usages. In this case, the hidden data is commonly referred as *watermark*. Such mechanisms are now commonly applied to network traffic for a wide-array of tasks, for instance, to identify flows and trace them across the Internet or to implement traffic engineering policies [IE16]. However, the investigation of watermarking techniques is outside the scope of this Thesis.

Despite licit usages, information hiding techniques are increasingly adopted by attackers to make malware stealthier, more sophisticated and capable of eluding some security frameworks [MC15]. For example, mechanisms to cloak data can be used to conceal malicious payloads or

attack routines into innocent-looking images, escape file system analysis, covertly exfiltrate sensitive data, remotely orchestrate botnets, implement multi-stage loading architectures, or retrieve additional configuration files or commands [CCC⁺20]. As regards the first notable example of a long-lasting attack campaign leveraging information hiding mechanisms, we mention the Operation Shady RAT started in mid 2006. In more detail, the attack campaign leverages a phishing email to decoy the victim. When the attachment is executed, an exploit downloads an additional malware component to open a backdoor for communicating towards a Command & Control (C&C) server. To avoid detection, attackers cloak malicious instructions in the HTML source, i.e., the commands are hidden in HTML comments and obfuscated by using Base64 encoding. When retrieved by the backdoor, the additional commands enable the attackers to access the targeted machine and rapidly escalate privileges [Alp11].

Nowadays, the adoption of hiding techniques to empower different stages of the cyber kill chain is becoming widespread, so much that researchers named this class of threats *stegomalware* [CM22]. This term was firstly introduced in [STTPL14] to showcase how mobile applications containing hidden malicious code remained undetected within the Google Play Store for several months. Although the literature and real-world attacks take advantage of a great variety of different information hiding techniques [WCM⁺22], especially in terms of carriers and embedding mechanisms, in this Thesis we address two major domains: the creation of covert channels (especially for the network case) and the use of digital media steganography for cloaking malicious assets.

1.2 Covert Channels

Among the various techniques that can be used to make malware stealthier or to implement sophisticated attack schemes, *covert channels* are of prime importance. Originally introduced by Lampson in 1973, covert channels are “[channels] not intended for information transfer at all” [Lam73]. Figure 1.1 depicts the hidden communication model for implementing a covert channel. Such a model has been firstly formalized by Simmons in 1983 [Sim84] and it considers two prisoners, defined as the *covert sender* and the *covert receiver*, wanting to escape. A third-party entity, defined as the *warden*, monitors their communications. For this reason, the prisoners want to deceive the warden by finding a way to secretly communicate and arrange an escape plan. To do this, they must agree on a pre-shared hiding mechanism, select a carrier to “transport” the secret information and then, set up a covert channel.

To not appear suspicious, the covert endpoints should choose popular carriers in order to pass unnoticed amongst the bulk of data and to increase the overall stealthiness. Moreover, the carrier containing the secret should preserve its original functionalities and should not appear anomalous or suspicious to the warden. In other words, the “Carrier + Secret” element depicted in the figure should be as much as possible similar to the original “Carrier”.

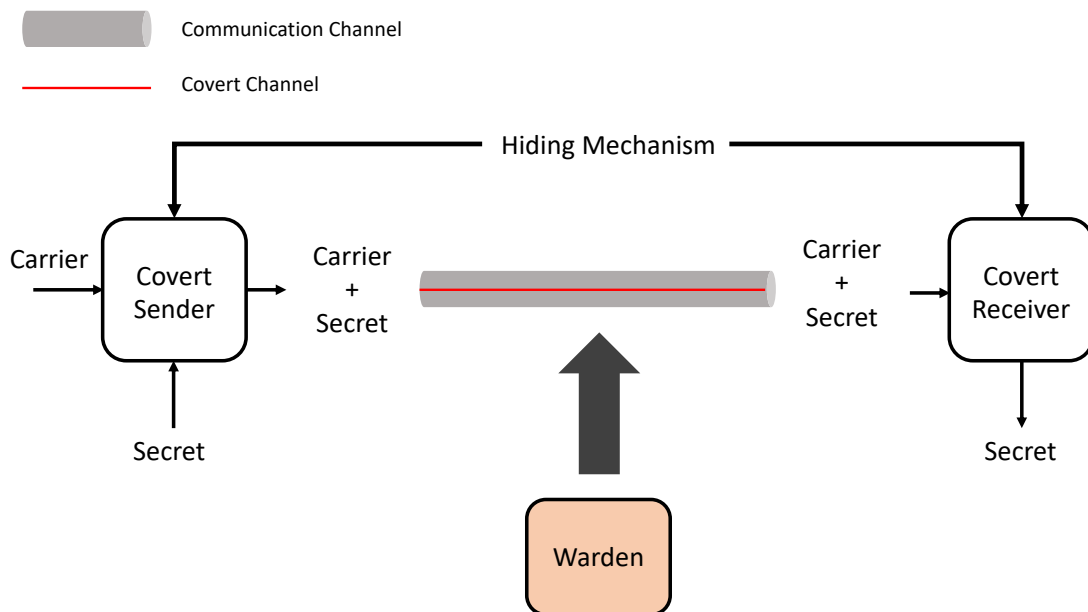


Figure 1.1: Model for attack schemes implementing covert channels.

Even if during the years mechanisms for implementing the reference model of Figure 1.1 proliferated, two major classes of covert channels exist: *storage* and *timing* covert channels. The first group consists of channels created by directly writing the secret information in the carrier. For instance, a payload can be hidden within the header of a network protocol. The second group is composed of those channels created by encoding the secret information through the temporal evolution of a certain event. Figure 1.2 depicts two different approaches for implementing timing channels. In particular, Figure 1.2(a) depicts temporal techniques, which are based on the frequency of events. Considering the example of a covert communication within a mobile device, a covert sender can use the status of the screen as the carrier. Hence, it can manipulate its behavior by alternating between ON and OFF states. Then, the covert receiver monitors the screen: if the time elapsed between two activation events is greater than a certain threshold, then the bit 1 is transmitted [SZZ⁺11, LW13]. Instead, Figure 1.2(b) depicts volume techniques, which are based on the modification of the magnitude of an event. For example, the covert sender can encode the bits composing a secret message in the amount of threads generated in a certain time window: if the amount of threads is higher than an agreed value, then the bit 1 is transmitted [MRFC12].

In general, the performance of a covert channel is described via different but interdependent metrics, often denoted as the “magic triangle” [FPK07, MC14]. Specifically, the properties are: the *steganographic bandwidth*, i.e., how much secret information can be sent per time unit, the *undetectability*, i.e., how the channel is difficult to spot, and the *robustness*, i.e., how the channel can resist against delays, errors and deliberate manipulations from a security tool acting as a warden. These metrics are tightly coupled. In fact, it is not possible to maximize a metric

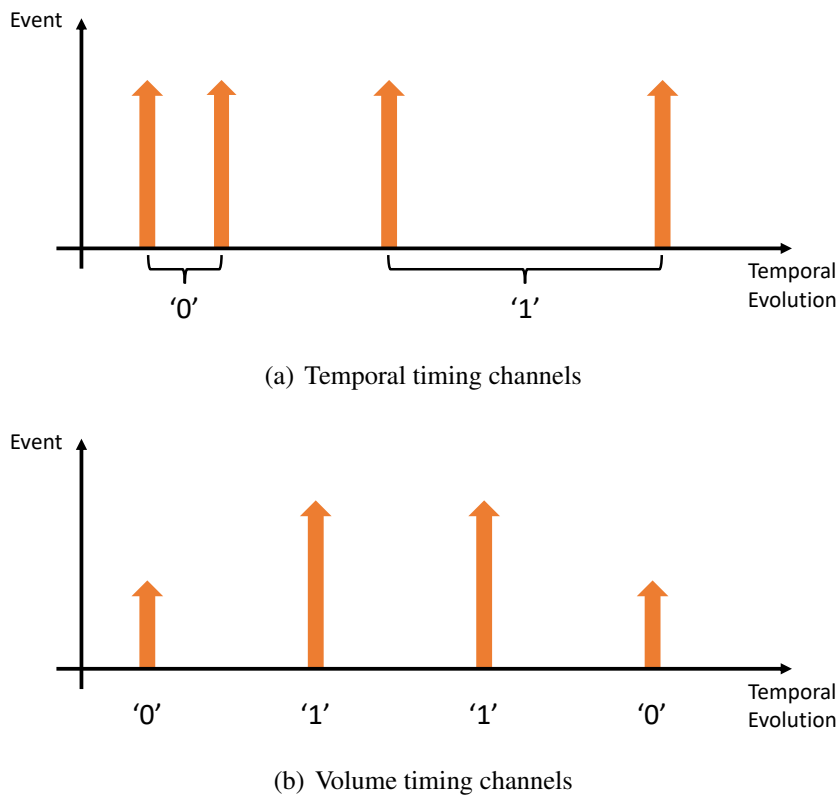


Figure 1.2: Reference models for timing covert channels.

belonging to the magic triangle without degrading the others. As a paradigmatic example, the steganographic bandwidth cannot be increased without influencing the undetectability of the channel: the more the carrier is manipulated to contain secrets, the higher the chance of artifacts that can reveal the presence of hidden information. Similarly, the robustness of the channel may require to deploy an error correction algorithm causing an erosion of the available space within the carrier or accounting for computational overheads that generate lags in the device of the victim [UMLC17].

Some works also propose an additional metric called the *steganographic cost*, which allows to evaluate the degradation experienced by the carrier due to the presence of the secret. As an example, for a covert channel nested in a VoIP conversation, the steganographic cost can represent the alteration of the audio quality, e.g., additional delays, reduced signal to noise ratio performances of the codec, or audible artifacts [Maz13].

From the viewpoint of empowering a malware, there are two main types of channels: *local covert channels* and *network covert channels*.

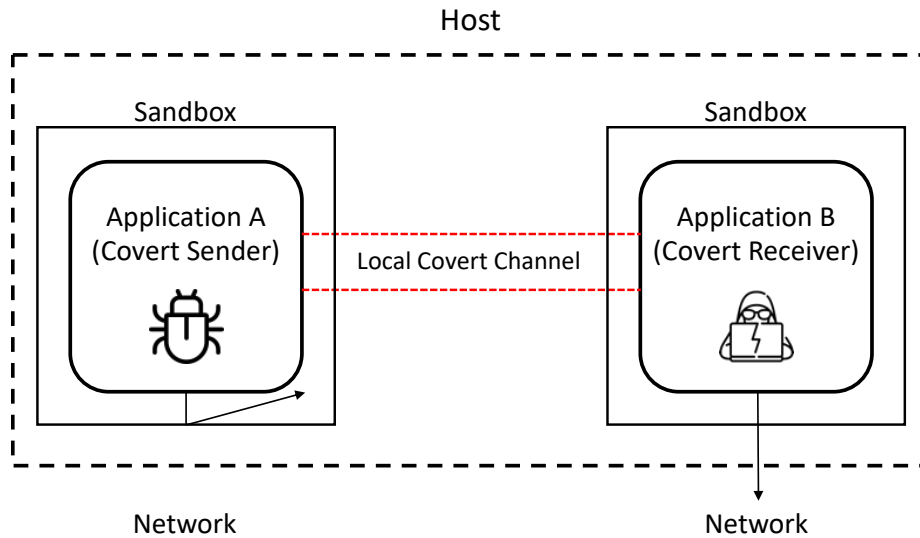


Figure 1.3: Reference model for the colluding applications scheme.

1.2.1 Local Covert Channels

Many attacks exploiting covert channels are created among entities wanting to communicate on the same device [MC14]. In this Thesis, we refer to this kind of hidden communications as local covert channels. Example of such techniques include the modulation of resources such as the load of the CPU or the memory consumption, the manipulation of the file system, the high-jacking of inter-process communications or messaging services, or the state of TCP/Unix sockets [MRFC12, MC14, UMLC17]. For the case of developing a malware or different advanced threats, local covert channels are often used due to their ability of creating a *colluding applications* attack scheme. Therefore, in this Thesis we will consider their utilization mainly in the view of such an attack template. In more detail, with colluding applications we refer to an umbrella for identifying a class of threats able to bypass the security policies deployed in underlying software and hardware layers, including the guest Operating System (OS). Put briefly, the attacker sets up an “abusive” inter-process communication service between various software entities, e.g., applications or processes, for exchanging data within the single host [MRFC12, MC14].

Figure 1.3 illustrates the reference scenario, i.e., two applications wanting to leak sensitive data outside the hosting node. Both entities are sandboxed by the OS or by some third-party software layers deployed to enforce security proprieties. The applications are then restricted to access only some resources (e.g., volume settings or shared files) or functionalities (e.g., network services). In particular, the Application A, i.e., the covert sender, has access to sensitive information, therefore it is prevented by the sandbox from accessing the network layer. The Application B, i.e., the covert receiver, has not access to such data, thus it is considered safer and can then communicate

with a node outside the host. In order to leak the sensitive data, the covert sender implements a local channel to transmit the information and bypass its sandbox. To this aim, it should find a suitable carrier where to inject the secret information. For instance, the sender can modulate the amount of used RAM to signal bits, e.g., 1 when allocating memory and reducing the overall availability on the host and 0 otherwise. The covert receiver monitors the utilization of the RAM, reconstructs the hidden message and sends the information outside the host by taking advantage of its network privileges.

Several examples of local covert channels have been proposed to exfiltrate sensitive data from personal devices. In particular, the work [SZZ⁺11] showcases how two colluding applications can exchange data through the vibration settings of an Android device, i.e., by exploiting the notifications sent in broadcast when changing the configuration. Similarly, the volume can be altered and then synchronously checked. Another example is based on taking locks on shared files, i.e., signaling 1 by requesting the lock on a file shared between the covert sender and the covert receiver. The work [MRFC12] extends this idea and introduces channels communicating via threads enumeration, i.e., the sender generates a certain number of threads to indicate a 0 or a 1 and the receiver monitors this value by checking the `proc` directory. The work also showcases a colluding applications attack scheme achieved by writing and deleting files on the storage unit, hence modulating the free space on the device. Other examples are provided in [WZFH15] and [GGK⁺17], which investigate colluding containers or virtual machines trying to communicate via a local covert channel to exfiltrate data, map the underlying hardware or guess 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 [ZJRR12].

1.2.2 Network Covert Channels

One of the most popular and effective advanced offensive mechanism based on information hiding concerns the creation of a network covert channel, i.e., a hidden communication path laying within an overt traffic flow acting as the carrier. The exploitation of network covert channels has become popular in recent years, especially since traffic flows are often “boundless” and it is not uncommon to have network infrastructures with flows that last several hours and can be exploited by Advanced Persistent Threats (APTs) [MC15, CCC⁺20].

Typically, 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 [ZAB07, MC14, WZFH15]. A network covert channel is created between two covert endpoints (defined again as the covert sender and the covert receiver, respectively) wanting to remotely communicate in a hidden manner. To this end, a legitimate, overt network flow is used as the carrier to contain the secret information. Figure 1.4 shows the reference scenario and the considered attack model.

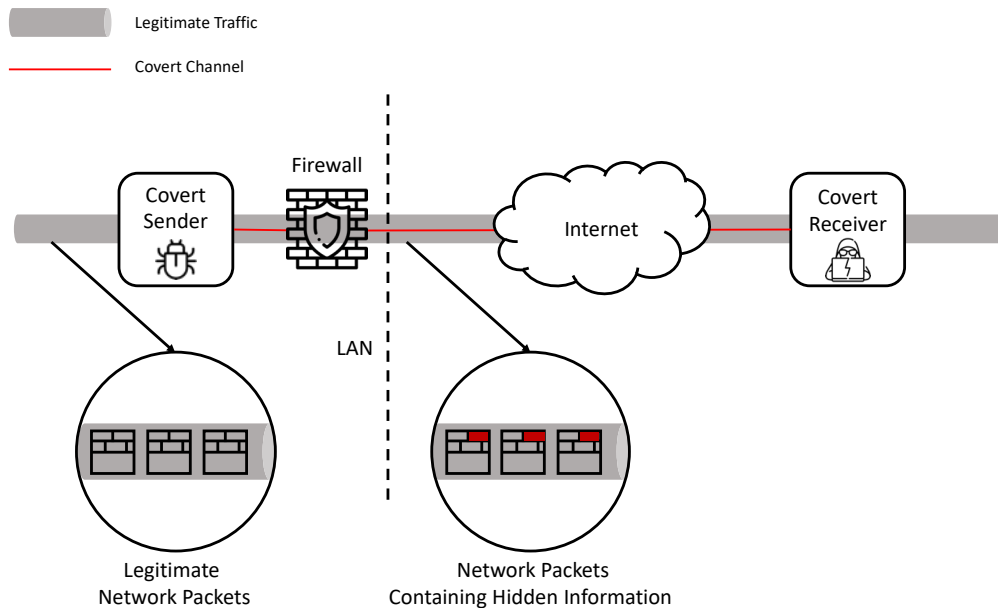


Figure 1.4: Reference model for network covert channels.

Specifically, the figure depicts the covert sender, e.g., a compromised node belonging to a LAN, wanting to secretly communicate with the covert receiver, e.g., the attacker or the C&C facility. As it usually happens, a firewall, an Intrusion Detection System (IDS) or a monitoring tool is placed at the border of the network to protect end nodes. To cloak data within a network flow, different portions of the protocol stack or behaviors of the traffic can be used. Possible examples are the alteration of the volume of the produced traffic, the modulation of the throughput, the artificial creation of retransmissions or increased error rates, the transcoding of multimedia streams for using the freed capacity to store secrets, and the direct embedding of data in unused fields of headers of packets [ZAB07, MC14, MC15, WZFH15]. As a result, these hiding mechanisms enable the covert sender and the receiver to bypass the firewall or the network security tool.

The literature offers several examples of network traffic features and protocols that can be used to implement a covert channel (see, e.g., [ZAB07, WZFH15] for two comprehensive surveys on the topic). Due to its ubiquitous availability, many works explore how to inject secret data in the TCP/IP protocol suite. For instance, the work [HS96] identifies covert channels jointly exploiting the unused bits of the `Type of Service` field of the IPv4 header and the `Reserved` bits of the TCP to contain up to 1 byte of secret information. The work also proposes to create hidden communications by using the checksum of UDP packets.

Other notable examples to move secret data deal with the Session Initiation Protocol and the work [MS08] highlights the feasibility of using different fields and features of the protocol to implement low-bandwidth storage covert channels. For instance, both the random parts of the

branch and the `Call-ID` fields can be overwritten with secret data, i.e., the “magic cookie” and the hostname of the endpoint can be concatenated with arbitrary information. A similar approach can be used by altering the value of the sequence number contained in the `Cseq` field. A comprehensive survey of hiding techniques targeting VoIP traffic flows is presented in the work [Maz13], which reviews several methods targeting the header/payload of datagrams (i.e., storage channels), the time relations (i.e., timing channels), and an hybrid combination between the two.

Concerning the creation of covert channels targeting the IPv6 protocol, which is becoming widespread and appealing to attackers, the literature already offers some previous attempts. For example, the work [BPK⁺16] showcases how sequential IPv4 and IPv6 sessions can be used to transmit arbitrary data. Moreover, the work deals with the leakage of hidden information through the manipulation of different v4/v6 transitional mechanisms, e.g., `6over4`. The work [LLC05] presents several methods targeting various IPv6 header fields. For instance, the `Flow Label`, a pseudo-random and uniformly generated value, can be used to contain up to 20 bits of arbitrary information. Instead, the `Payload Length` can be increased to append extra data at the end of the packet. The work also shows that the various extension headers can be used as well, e.g., the `Routing Header` contains 4 reserved bytes that can be manipulated to transmit secret data.

In [ROL13] the use of the Dynamic Host Configuration Protocol to establish hidden communication paths is presented. For instance, the `xid` field can hide 4 bytes without altering the normal functioning of the protocol. Instead, the `server name` and the `file` fields can carry data only in the case of Discover, Inform and Request packets coming from the client side. The use of the Stream Control Transmission Protocol to create covert channels is discussed in [FMS12]. For instance, the `Initiate Tag` of each packet can be processed to contain up to 32 bits of arbitrary information. Similarly, the `Number of Inbound Streams`, which defines the maximum number of streams that the sender can handle, can contain 8 bits of secret data. Lastly, also the `Stream Sequence Number` can be used to cloak up to 16 bits but this can be done only if the original stream does not require reordering. The feasibility of hiding data within the Transport Layer Security (TLS) header is investigated in [MJ15]. The work evaluates the possibility of replacing handshake information with arbitrary data. Moreover, the work [HMC20] reviews and designs seven different TLS-based covert channels. For example, it demonstrates how to append a hidden content in the `ClientHello` message or how to alter the `Cipher` and the `Compression` fields to encode data.

Owing to the momentum gained by smart devices and industrial control systems, the recent work [VMWM19] demonstrates several methods to exploit features of the Message Queuing Telemetry Transport (MQTT), e.g., by directly altering `CONNECT`, `PUBLISH`, or `SUBSCRIBE` packets or by modulating the presence/absence of `RETAINED` messages.

Concerning the case of timing channels, they are usually protocol-agnostic and mainly implemented at the network layer or by altering the error rates characterizing the data link [CBS04, BGN17]. The work [WYH16] investigates timing covert channels varying the transmission rate

or the timing statistics characterizing the stream of packets. Possible encoding schemes are based upon the alteration of the throughput, the introduction of statistical signatures in the jitter or the manipulation of the inter-packet time.

Finally, a recent research effort has started to unify and refine the terminology of hiding mechanisms that exploit network traffic by defining “patterns”, i.e., a universal language used to create taxonomies in a generic manner. The work [WZFH15] reviews 109 techniques summarized in 11 hiding patterns defined via the Pattern Language Markup Language. For instance, the “P6. Reserved/Unused Pattern” groups all the techniques that hide data into a reserved/unused header field, whereas the “P8. Interarrival Time Pattern” defines network covert channels that encode an information in the inter-packet time between datagrams. This work has been improved and extended to other steganographic domains, such as digital media and text, in [WCM⁺21]. We point out that, the work [SW22] provides a taxonomy for indirect covert channels, i.e., hidden paths requiring an intermediate node to store or redirect the covert information, which are outside the scope of this Thesis.

1.2.3 Detection of Network Covert Channels

By considering again the reference model of Figure 1.1, a warden can be classified according to its behavior, structure, localization, and the nature of used information [MWCK19]. For instance, wardens can be *passive* or *active*, depending on their capability of monitoring and reporting malicious communications or modifying/dropping traffic without interfering with legitimate packets [AP98, WK12, DD15]. To effectively counteract a covert channel, as a first step, a warden must “discover” the hidden communication attempt. Unfortunately, the complexity of network protocols as well as the heterogeneity of communication technologies prevent to develop a unified framework. As a partial workaround, Chapter 3 will discuss a flexible and extensible method for getting visibility over the network and for collecting data to spot the presence of different network covert channels. Yet, in the following, we present various threat-specific approaches for detecting hidden communications.

Since attackers can hide malicious payloads or commands within several parts of network packets, the detection of storage channels is fragmented and the majority of the solutions can only work in narrow attack scenarios. Owing to its ubiquitous availability and multiple exploitable behaviors, many works address the problem of revealing hidden communication nested within the TCP. The work [ZS13] proposes a statistical model to spot data injected in the `Initial Sequence Number` (ISN) field used to synchronize peers, whereas [TA05] bases the detection approach on a model of the original ISN distribution generated by the particular OS. The work [ZLD10] models the behaviors of TCP flows by using Markov chains to reveal anomalies through a comparison between overt and covert transition probability matrices. The mitigation of channels exploiting the DNS has been largely studied as well, especially due to its wide adoption in data exfiltration campaigns or botnet orchestration. The work [CLL⁺21] proposes a real-time

detection method based on an artificial neural network model for revealing various DNS-based covert channels by inspecting fully qualified domain names.

A relevant amount of works investigates how to mitigate covert channels targeting VoIP conversations. To prevent such attacks, the literature proposes to develop an active warden that “sanitizes” audio signals and limits the bandwidth of the covert channels [TL07]. Unfortunately, such approaches often lead to latency or degradation of the carrier, which could not be acceptable when dealing with audio signals. Therefore, a passive warden could be the preferred choice to inspect the traffic and compute measurements, such as noise quality metrics. Other examples are provided in [Maz13], which reviews a plethora of mechanisms to tame covert communications within VoIP conversations. We mention, among the others, the analysis of audio artifacts to spot data embedded in voice samples, the identification of anomalous traffic features revealing schemes based on packet losses or manipulations of the delays, as well as the deployment of nodes buffering and padding traffic to disrupt parasitic information in a blind manner.

Since IPv6 is becoming popular, various countermeasures for this protocol have been proposed as well. For instance, [SMP17] showcases a machine learning technique based on fuzzy logic to detect covert channels in the IPv6 header. Alas, this requires suitable datasets for training the detector, which are often unavailable. The work [BPK⁺16] investigates the detection of IPv6/IPv4 transitional mechanisms and highlights that current security tools and IDSes present “serious drawbacks when handling IPv6 traffic”.

The protocol-agnostic nature of network timing covert channels leads to a more coherent and homogeneous literature. In this case, the wardens are typically equipped with detection algorithms based on the computation of some statistical indicators. Despite the wide array of used methodologies (e.g., machine-learning-capable frameworks or ad-hoc metrics), the problem of revealing hidden or parasitic conversations within timing features has been better investigated compared to other type of channels (see [BGN17] for a comprehensive survey on the topic). For example, [CBS04] showcases how a regularity measure computed starting from the standard deviation of inter-packet times of a conversation can be used to spot anomalies. Another example concerns the use of non-parametric statistical tests comparing the inter-packet time distributions within different observations [BGC05, RHS17]. The work [GRK13] proposes a detection method based on an entropy measure, i.e., it divides the inter-arrival times into two arrays and calculates their hierarchical entropy. The recent work [WCTS21] deals with a detection algorithm that converts sampled inter-packet times into symbolic time series and calculates the state transition probability matrix. Then, overt and covert traffics are compared to compute a similarity score for revealing the presence of hidden malicious attempts. To face advanced techniques adopted by attackers, e.g., encryption or obfuscation, [Sti08] proposes a warden to detect timing channels by computing the correlation between the packet timing and the data stored in memory. To address attackers hiding data in huge traffic traces, a possible idea is to take advantage of high performance techniques. A possible example is the Worms Distributed Covert Channel Detection Framework (WoDiCoF), which leverages Apache Hadoop to efficiently process a

large volume of network packets [KWZ⁺18]. Artificial Intelligence (AI) paradigms can be used as well to develop more general approaches and mitigate possible obfuscation schemes. For instance, [SHRS15] showcases the usage of Support Vector Machines (SVMs) to detect timing channels and classify covert and overt traffic. Despite its effectiveness, the data used for training the models are limited, which is a relevant problem when dealing with this class of attacks. Another promising approach is presented in [NZCM20], where a combination between data mining algorithms and traffic classifiers is used to spot covert channels based on the reordering of TCP options. The work [DAFB⁺19] combines both statistical analysis, e.g., mean, median, standard deviation, or entropy of inter-packet time, and Deep Neural Networks (DNNs) to obtain better performance compared to SVM-based approaches.

Once the warden has detected the presence of a hidden communication, it can follow two strategies: eliminate or limit the covert channel. The first option is not always possible, especially if eliminating a covert channel leads to disrupt legitimate traffic or reduce the Quality of Experience perceived by the user (e.g., degradation of audio signals in VoIP conversations). Instead, the second approach could be adopted to reduce the usability of the channel, e.g., by impairing the steganographic bandwidth, thus making the channel unusable or not appealing. As hinted, choosing among these options requires different tradeoffs. For example, the processing capabilities of the warden may impact the behavior of the traffic (e.g., introducing unwanted delays or bottlenecks) or it could require expensive resources (e.g., to inspect the traffic).

As regards the limitation of covert channels, the Pump [KM93] is one of the earliest solutions. This mechanism has been originally conceived to prevent the creation of covert channels among two processes/users with different security levels (i.e., “high” and “low”) in a multi-level secure system. In more detail, the Pump acts as an intermediary communication buffer between the processes, requiring that the “low” process receives messages at probabilistic time intervals based on the historical activity of the “high” process. This allows to introduce noise in the reception of the messages and thus reduce the probabilities for creating a covert channel. The Network Pump is an adapted and improved version of the Pump for its use in a network environment [KML96]. Since the complete elimination of exploitable flaws in network protocols is often unfeasible [ZAB07], some approaches aim at “sanitizing” the network traffic, i.e., the covert channel is eliminated by removing the hidden information from the network traffic without interfering with the legitimate conversations. For example, [XKC20] showcases the use of a warden able to inspect and modify headers for mitigating network storage covert channels embedded within TCP connections, whereas [MC14] proposes to restore fields to default values when possible or to pad sequences preventing manipulation attempts. Under specific circumstances, transitional mechanisms (e.g., IPv4/v6) or middleboxes (e.g., network address translation) can unintentionally disrupt possible hidden communications during the normal processing of traffic. Moreover, the growing usage of encryption and security protocols should also be taken into account. In fact, the development of more secure protocols reduces the likelihood that they can be used improperly to establish hidden channels. For example, [SVS13] showcases how the Sequence Number in the Encapsulating Security Payload of the IPsec protocol can be naturally used to mitigate/eliminate

covert channels modulating the order of packets. As regards the specific case of channels that hide data through the alteration of the inter-packet time, an effective countermeasure is to add random delays. Unfortunately, this type of mitigation technique also penalizes the legitimate traffic, which is seldom acceptable in real use cases [BEK16].

To prevent that threats can exploit network covert channels, a promising idea is to consider information-hiding-capable attacks from the very early design stages. By following “secure-by-design” and “secure-by-default” principles, it is possible to prevent network protocols from being exploited by attackers to implement stegomalware. To this extent, standardization plays a significant role: in fact, forcing a flow to comply to its standard implementation or blocking certain features in specific scenarios can disrupt hidden communication attempts.

As a final remark, we point out that being able to collect network information is a core task to support frameworks and algorithms to detect and prevent attacks or engineer security-by-design systems, especially for the case of stegomalware leveraging different types of covert channels. 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, [CRR18] and [RCL19] propose the adoption of an orchestrator for monitoring security hooks embedded in the virtual layers of cloud applications. 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. For instance, [BLC13] proposes an approach for the dynamic placement of DPI-capable software to limit power consumption and costs, while delivering suitable degrees of scalability and performance. A similar blueprint can be used to implement adaptive network security wardens (e.g., IDSes and firewalls) encapsulated within virtual machines [LLW⁺12]. Moreover, due to the nature of stegomalware and other emerging threats (e.g., cryptolockers), a recent trend concerns the gathering and monitoring of some well-defined and low-level features instead of high-level yet specific metrics [ZAB07, MC15]. Being able to gather system-specific information could allow to generalize the detection phase or make it more scalable. 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 network wardens.

1.3 Digital Media Steganography

Compared to network traffic, digital objects and multimedia data offer a wider selection of attractive carriers for cloaking more sophisticated payloads. For example, data can be hidden by manipulating the echo of a certain audio signal or by taking advantage of barely audible low-power tones [DAMH12]. Text steganography is another popular example. Information can be hidden by manipulating the whitespaces between words and paragraphs, marking characters, changing the style or the font, or by mapping symbols into characters [KTB17]. Recently, video






Least Significant Bits	Pixel	Red	Green	Blue
original pixel		11010110	01111110	00101000
1		1101011 <u>1</u>	0111111 <u>1</u>	0010100 <u>1</u>
3		1101000 <u>1</u>	0111100 <u>1</u>	00101 <u>111</u>
5		1100100 <u>1</u>	0110000 <u>1</u>	00110 <u>111</u>
7		1010100 <u>1</u>	0000000 <u>1</u>	010101 <u>11</u>

Table 1.1: Pixel color variation when modifying the least significant bits of the Red, Blue, and Green channels.

files are getting more and more attention [SKM15]. Data can be hidden by using video error correction techniques [YA03] or by transmitting additional information such as subtitles [LLL06]. However, audio, text, and video, are rarely used in real attacks, thus they will not be considered in this Thesis.

On the contrary, images are the most popular carriers for implementing offensive schemes, since they offer both large steganographic bandwidth and adequate undetectability [MEO05]. The literature offers several steganographic mechanisms to conceal messages in digital images [SK15, HWI⁺18]. Example of these techniques include Pixel Value Difference (PVD) methods, where the difference between adjacent pixels is used to decide the amount of secret information that can be hidden [WT03]. Grey Level Modification algorithms, which map a secret information by modifying the gray level of each pixel, are other popular mechanisms (see, e.g., [PC04, MAF⁺15]) along with Edge-based methods, which store secret bits in the edges of the image [LHH10]. Due to its tradeoff between simplicity and effectiveness, the Least Significant Bit (LSB) and its variants are the most common steganographic techniques observed in real attacks. In essence, the LSB technique allows to alter the least significant bit(s) of the color space of the pixels, i.e., Red, Green, Blue (RGB) to contain a secret message. Since changes in the LSBs are often undetectable to the human sight, the method is suitable to conceal information without causing visible alterations in the resulting image. However, the more bits are used, the more artifacts will be present. Table 1.1 depicts the general idea. In particular, the least significant bits of each color component of the original pixel are modified to contain a secret information (the bits composing the secret are underlined in the table). As it can be seen, the fewer bits are changed, the less the resulting pixel differs from the original. With 7 least significant bits modified, the color information is completely disrupted, and thus the presence of secret information can be easily detected even by the human eye.

There are several other techniques based on specific transforms of the original image. For example, some hiding mechanisms can take advantage of the Discrete Cosine Transform (DCT). The latter “converts” the image into a sum of sinusoids of various frequencies and magnitudes, and the coefficients of such components can be used to hide secret messages [PD12]. Other steganographic methods are based on the Discrete Wavelet Transform [AH08], which exploits the frequency domain of images.

As it will be detailed in the next section, the LSB approach is one of the most popular and effective technique used by attackers to conceal malware targeting digital images. Thus, in this Thesis, we address the problem of detecting images containing malicious payloads hidden via various versions of such a technique.

1.3.1 Detection of Malware Hidden in Digital Images

As discussed, multimedia objects are often used as the main carrier for conveying malicious payloads and commands. Even if commercial tools to spot hidden payloads in digital media are emerging, they are either unsuitable to handle the wide-range of cloaking schemes and obfuscation techniques observed “in the wild”, or not mature enough to protect large-scale deployments [HYM⁺20]. The most popular strategies for detecting multimedia hiding arbitrary information fall in the *steganalysis* field [FG02, KKP18]. Concerning digital images, such techniques mainly measure their statistical properties, e.g., histograms of the color values, correlation between pixels or distribution of the DCT coefficients, to search for discrepancies and anomalies. For example, [PLL20] investigates the use of the Chi-Square test to calculate the probability that an image is hiding a secret content and the work [Ker04] introduces a technique to estimate the length of the secret message in 8-bit GIF images by computing entropy-like measurements. Other techniques aim at directly disrupting the hidden content by applying image processing techniques, e.g., flip some bits, change the file format, or use compression methods. As an example, the RS analysis divides the image into groups of pixels. Then, it measures the noise of each group before and after flipping the least significant bits: if the noise increases, the group is classified as potentially malicious [FGD01]. Other works exploit more “standard” file analysis methods. For instance, [VMS22] deals with a technique to spot suitable markers able to reveal alterations, while [PCK⁺20] searches for byte-level signatures indicating the presence of additional information appended at the end of a digital image. The work [CGG⁺22] provides an analysis method developed to reveal images containing PowerShell scripts embedded via the Invoke-PSImage technique. The tool has been implemented within the framework of the SIMARGL (Secure Intelligent Methods for Advanced RecoGnition of malware and stegomalware) European project¹ and integrated in the dedicated toolkit to holistically face the problem of stegomalware.

A relevant part of literature leverages some form of AI to balance the “arms race” between attackers and defenders [CCC⁺20], especially to face new challenges like those offered by Internet of Things (IoT) or zero-day threats [DFP20, WHWS20]. A first attempt to use Deep Learning (DL) has been firstly adopted in [TL14], using a stack of autoencoders to form a Convolutional Neural Network (CNN) to detect malicious images (see [RRG19] for a comprehensive survey on CNNs-based techniques). The recent work [PKSJ22] presents various techniques leveraging machine learning to create predictive models able to discover images potentially altered by various steganographic techniques, mainly in the transform domain (e.g., DCT coefficients). Unfortu-

¹<https://simargl.eu> [Last Accessed, October 2022].

nately, it does not consider real-threats, but focuses on the perspective utilization of data hiding to develop novel attack mechanisms. The work [LWWL08] showcases a more general discussion and demonstrates how different machine learning methods can be deployed to reveal the presence of hidden data. As mentioned, an important aspect to consider is that the detection of stegomalware using some form of AI requires suitable datasets, which are hard to create and distribute in a standardized form, especially when considering threats leveraging information hiding. Moreover, attacks tend to evolve, thus leading to “concept drift” phenomena accounting for degradation of the models [GMP20].

Due to the scalability constraints or lack of suitable models, deploying sophisticated detection frameworks could not be always possible. Moreover, inspecting all the images exchanged in a large-scale deployment could be unfeasible due to hardware requirements or the need of satisfying real-time constraints. In this case, an alternative approach searches for well-known signatures characterizing a specific hiding mechanism or a payload (e.g., a sequence of bytes) in the file structure of the image via a simple and optimized tool [PCK⁺20]. Alas, reverse engineering a malware to grasp its internals is often difficult, hence a “meet in the middle” solution is represented by sanitization, i.e., the image is lightly processed to disrupt the secret content, if any. To this aim, the literature offers solutions using nonlinear transformations [JLE20] or autoencoders to alter anomalous pixels without degrading the perceived quality. Another approach is based on the adoption of AI to locate the area of the image modified via steganography [SZZW19] to trigger the execution of an optimized security pipeline.

1.4 Real Attacks

According to [MC15, CM22], information hiding and steganography are increasingly used by attackers to make malware stealthier and difficult to spot. Unfortunately, quantifying the diffusion of the phenomenon is extremely complex. On one hand, security experts are seldom able to precisely identify a threat taking advantage of information hiding techniques. On the other hand, many attacks remain unnoticed for years. As a consequence, the estimation of the impact of stegomalware provided by the Criminal Use of Information Hiding (CUING) initiative² should be considered a lower bound. In more detail, CUING reports an increasing trend in the diffusion of information-hiding-capable threats for the 2011-2019 period. The most recent measurement reports 16 different threats in 2019. Even if precisely identifying the root of this trend is very difficult, the first malicious activities using network traffic as the steganographic carrier were first observed in 2011 with the W32.Morto worm. In particular, it exploits DNS TXT records to receive commands and additional modules from the attackers. During the same period, the Feederbot botnet has been discovered. It exploits the `rdata` field of TXT resource record in the DNS response to contain hidden messages and secretly communicate towards a C&C server

²<https://cuing.org> [Last Accessed, October 2022].

[DRF⁺11]. Another notable example, discovered in 2013, is the Linux.Fokirtor backdoor. It implements covert communications by exploiting SSH connections to exfiltrate sensitive information towards a remote server. Among the others, exfiltrated data contained passwords and usernames, SSH keys, hostnames and IP addresses. To extract and execute additional commands, the backdoor monitors the incoming SSH traffic and looks for a specific pattern, i.e., “:!:;”. One year later, another sophisticated malware named Regin has been discovered. Regin communicates with a C&C server by using customized versions of the ICMP, UDP, and TCP protocols. Another example regards the TeslaCrypt ransomware, delivered through the Neutrino exploit kit. This is used to redirect users to a malicious landing page to download the TeslaCrypt variant from a remote C&C server via innocent-looking HTTP traffic. A more recent attack is the Sunburst backdoor able to generate HTTP `GET` or `POST` requests to communicate with C&C facilities, hiding data within the response bodies.

Instead, evidences of the use of image steganography within commercially-available software can be rooted back in 2014, e.g., see [STTPL14] for an analysis of applications available on the Google Play Store containing images altered via steganographic mechanisms. More recently, a massive phishing campaign launched in 2021 used the Invoke-PSImage technique to spread the Ursnif banking trojan. To this aim, attackers adopted this variant of LSB steganography to hide a PowerShell script for retrieving additional malicious code [BvEC⁺17]. The recent attack launched by the MageCart group against the Magento e-commerce platform is another major example. In this case, cybercriminals cloaked a web skimmer in favicons, i.e., small images associated to an URL to enhance the user experience of browsers, to steal credit card credentials. A new variant of the Zeus malware appended an additional configuration file (encrypted with Base64, RC4 and XOR) to innocent-looking images. Other recent examples are SteamHide, that uses the game platform Steam to spread malicious payloads hidden in the `PropertyTagIC-CPProfile` metadata field of profile images, and an ongoing Monero crypto-mining malware campaign targeting Docker APIs on Linux servers. In this case, the attackers can run malicious container and fetch a Bash script concealed in a PNG image. At the beginning of 2022, the Serpent backdoor targeted various French entities, such as real estate and government industries. In more detail, a Word macro retrieves a Base64 PowerShell script hidden in a JPG image, which downloads, installs and updates an open-source package installer (i.e., Chocolatey) to configure the backdoor.

At the time of writing (August-October 2022), a new attack campaign combines both image steganography and network covert channels. In particular, an image took by the James Webb telescope has been used to disguise malicious Go code. Specifically, data has been encoded in Base64 and then embedded by mimicking a valid certificate. Once executed, the malware sends the stolen sensitive data encoded in Base64 and hidden in the DNS `TXT` field of “nslookup” queries.

1.5 Research Contributions and Thesis Outline

The contribution of this Thesis is mainly in the detection and possible mitigation approaches of the most recent and popular hiding techniques used by malware. In more detail, it advances in the detection of network covert channels, especially in the design of scalable, privacy-aware and extensible mechanisms. Another relevant contribution is on the use of AI for detecting realistic threats observed in digital images and to provide a workaround when detection is not possible. Throughout the Thesis, emphasis will be put on the performance evaluation of the approaches proposed for taming network covert channels and threats exploiting images, highlighting their impact when deployed in real scenarios. This Thesis also offers some new tools to be used to assess the impact of stegomalware and network covert channels in production-quality scenarios. Lastly, an important outcome is in terms of datasets, which can be used to make feasible the research on threats leveraging advanced information hiding techniques. The rest of the Thesis is organized as follows.

Part II, Detection of Covert Channels

This part of the Thesis focuses on the detection of threats exploiting covert channels. In particular:

- Chapter 2 presents `pcapStego`, a tool for the “offline” creation of network covert channels in various network protocols, starting from real traffic traces. The chapter also introduces `IPv6CC` and `TLSCC`, which allow to create network covert channels targeting IPv6 and TLS conversations in a “online” manner. Finally, the chapter showcases the security capabilities of different detection tools and IDSes to face network covert channels;
- Chapter 3 introduces `bccstego`, an inspection framework for gaining visibility over networks to compute condensed indicators. The framework is based on the extended Berkeley Packet Filter (eBPF) and has been designed to be easily extensible and to guarantee privacy requirements. The scope of this research is to provide a technological foundation to support the detection of stegomalware leveraging network conversations;
- Chapter 4 showcases the use of in-kernel methodologies based on eBPF to programmatically and efficiently trace and monitor processes and network traffic. The chapter takes into account two realistic use cases implementing different steganographic mechanisms, i.e., colluding applications and hidden communication attempts nested within IPv6 conversations;
- Chapter 5 investigates the use of eBPF to create ad-hoc security layers in virtualized architectures without the need of embedding additional agents. The chapter focuses on the detection of both network storage and timing covert channels, demonstrating that the different hidden communications can be revealed while using limited resources;

- Chapter 6 provides a detailed performance analysis of the approaches presented in Chapters 3-5. In particular, it evaluates the development complexity, impact on packet transmission and resource usage of eBPF, Zeek and libpcap;
- Chapter 7 illustrates an approach based on an ensemble of autoencoders to efficiently reveal network covert channels targeting IPv4 traffic in IoT deployments.

Part III, Detection of Malware Hidden in Digital Images

This part of the Thesis focuses on the detection of threats exploiting image steganography. In particular:

- Chapter 8 proposes an approach based on DNNs for the detection and the classification of threats using LSB steganography to conceal malicious PHP scripts and URLs in favicons, as observed in MageCart/Magento-like threats;
- Chapter 9 extends the approach of Chapter 8 by considering larger images, i.e., high resolution icons. The chapter also considers the detection and the classification of different real malicious payloads, i.e., URLs, Ethereum addresses, HTML, JavaScript and PowerShell scripts;
- Chapter 10 presents how a residual convolutional autoencoder can be used to disrupt the information hidden within an image without altering its visual quality. As a test scenario, it considers images containing malicious data hidden via the Invoke-PSImage method.

Part IV, Conclusions and Appendices

This part of the Thesis draws final conclusions and provides additional information. In particular:

- Chapter 11 concludes this work and provides possible future research directions;
- Appendix A lists the software and datasets produced during the PhD;
- Appendix B lists the scientific works that have been used to write this Thesis;
- Appendix C lists the scientific works produced during the PhD.

Part II

Detection of Covert Channels

Chapter 2

Testing Tools and Security Assessment

Despite the increasing volume of attacks, the magnitude of economical losses, the degree of sophistication, and the growing attention from security-oriented software firms, threats leveraging network covert channels are often neglected for a threefold reason:

- the emerging nature of information-hiding-capable malware still requires precise investigation methodologies and conceptual devices;
- the creation of a covert channel is tightly coupled with the specific protocol/feature exploited to conceal the communication;
- network data usually contains confidential information requiring to adhere to strict rules and regulations to not disrupt the privacy of users.

As a consequence, the number of malware samples isolated in production-quality deployments is limited and datasets containing network traffic or execution traces do not consider information-hiding-capable threats. Even if efforts from the research community for sharing data are multiplying (see, e.g., [RWS⁺19] and the references therein), the majority of publicly available collections appear to be tailored for developing IDSEs able to counteract “classical” attacks like Denial of Service (DoS), flooding, port scanning, and botnet orchestration. Moreover, public datasets often require a non-negligible preprocessing effort to isolate or label information of interest [TL20]. Typical workarounds for the lack of traces capturing attacks leveraging network covert channels exploit artificially-generated traffic, non-weaponized malware, and simulations. Despite the level of accuracy, real-world data is almost mandatory to prove the correctness of laboratory trials or to evaluate the performance of a mitigation technique [HP09]. Moreover, the adoption of AI to address security issues (e.g., detection, classification or reverse engineering of malicious code) demands for relevant volumes of data, which should be organized in a standardized form for reproducing and comparing the different techniques [GMP20]. In this vein,

Chapter 7 will address the detection of covert channels in IoT environments via machine learning techniques and will clarify the need of data to engineer a suitable solution and conduct experiments to validate the effectiveness of envisioned countermeasures. As a consequence, the lack of testing samples leads to difficulties when assessing the capabilities of security tools to deal with covert channels. Indeed, evaluating the level of protection of such tools is of prime importance to fully understand the impact of information-hiding-capable malware in real-world conditions. Therefore, a relevant and propaedeutic effort performed within the framework of this Thesis has been devoted to investigate solutions that enable to create and investigate network covert channels with the aim of testing security tools. At the best of our knowledge, the only previous tool with similar capabilities is the CCgen framework [IMAZ22] but was not available when performing the needed preliminary investigations. Other solutions worth mentioning are the Covert Channels Evaluation Framework (CCHEF) [ZA08] and the Covert Channel Educational Analysis Protocol (CCEAP) [WM16]. However, CCHEF does not support IPv6 and CCEAP appears to be more suited for laboratory trials and didactic purposes rather than for large-scale experiments. Another important aspect concerns the creation of mitigation and detection techniques. This requires to fully understand the real “permeability” of pre-existent security tools and solutions to threats exploiting covert channels. As a consequence, the solutions presented in this chapter have been also used to perform a security assessment of de-facto standard IDSes/firewalls.

The remainder of the chapter is organized as follows. Section 2.1 discusses the architectural blueprint, the design choices and the performance of a tool for creating network covert channels starting from real network trace samples. Section 2.2 showcases two tools for the creation of network covert channels within TLS and IPv6 traffic. Section 2.3 investigates whether standard security tools can be considered suitable for detection of hidden communications. Lastly, Section 2.4 concludes the chapter.

2.1 Offline Creation of Covert Channels

To generate large dataset and perform experiments with covert channels, we developed `pcapStego`¹, a tool for creating hidden communications in traffic traces organized according to the `.pcap` file format. Compared to synthetic traffic generation or toy attacks, `pcapStego` offers the following benefits: *i*) data is injected within real traffic to consider production-quality scenarios or perform “what if” analyses; *ii*) the creation of covert channels happens offline enabling to prepare large datasets or exchange attack templates for reproducing experiments; *iii*) `.pcap` traces can be replayed for pentesting purposes through real devices or laboratory equipment. Currently, `pcapStego` can create a hidden communication within IPv4, IPv6, ICMPv4, and ICMPv6 conversations by directly injecting data or manipulating the behavior of the header fields (i.e., network storage covert channels). Moreover, it allows to encode data by altering the

¹<https://github.com/Ocram95/pcap-injector> [Last Accessed, October 2022].

Covert Channel Type	Field	Steganographic Bandwidth [bit/pkt]
IPv4		
Storage	Type of Service	8
Storage	Time To Live	1
Storage	Identification Number	16
Timing	-	1
ICMPv4		
Storage	Payload	48
Timing	-	1
IPv6		
Storage	Traffic Class	8
Storage	Hop Limit	1
Storage	Flow Label	20
Timing	-	1
ICMPv6		
Storage	Payload	8
Timing	-	1

Table 2.1: List of protocols/fields and channels supported by `pcapStego`.

inter-packet time between subsequent datagrams (i.e., network timing covert channels). Table 2.1 reports all the covert channels, protocols and fields supported by the tool.

The `pcapStego` distribution is organized in two sets of tools serving for different purposes. The first set implements an *interactive mode* allowing the user to manually select flows, the hiding mechanism (e.g., the part of the protocol header targeted for containing data) and the payload to be transmitted via the resulting channel. In this manner, users can actively experiment with the tool and gain a deeper understanding of the internals at the basis of network covert channels. The second set, instead, implements a *bulk mode*, which can be used to automatize the embedding of data and thus the creation of various covert communications in the target traffic capture. This working mode is especially suited for creating large `.pcap` files containing various channels at once and in an automatized manner.

Despite the working mode (i.e., interactive or bulk), functionalities of `pcapStego` are implemented in two different groups of Python modules dedicated to inject/extract data in/from `.pcap` files. This design choice allows to provide a simple user interface (especially in terms of command line parameters) and facilitates the development of scripts invoking the tool for the creation of multiple datasets or to compute metrics helpful for testing detection algorithms or perform modelling.

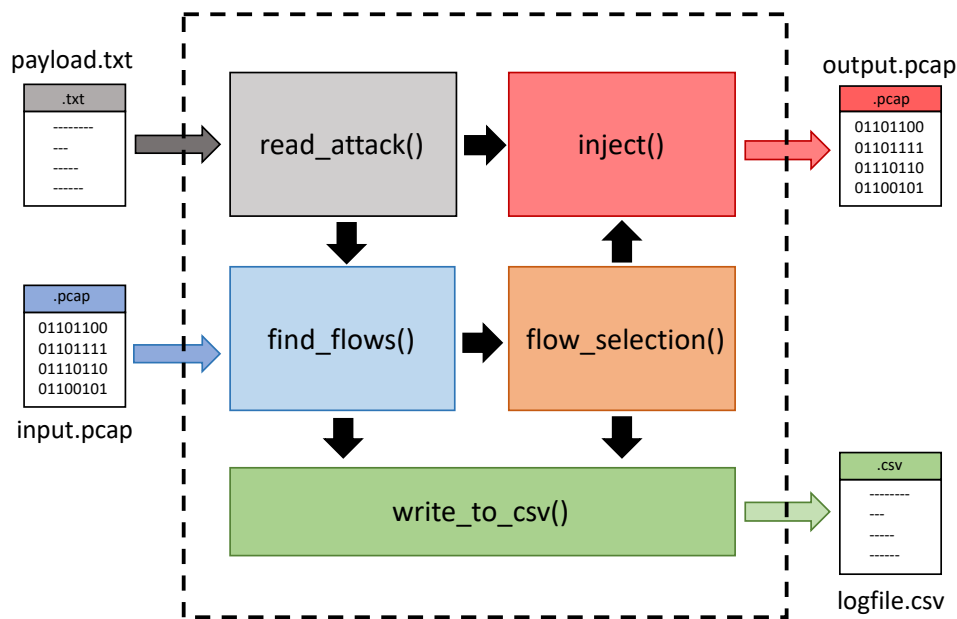


Figure 2.1: Software architecture of the `pcapStego` tool.

2.1.1 Software Architecture

To implement `pcapStego`, we used Python3 and the Scapy 2.4.4 library² for manipulating protocol data units, i.e., to alter fields in the protocol header or the inter-packet time for containing the data. To process `.pcap` files, our tool relies upon functionalities provided by the `tshark` 3.4.5 network analyzer³. Details are hidden to the user since the needed functionalities have been properly encapsulated within the software architecture, which is depicted in Figure 2.1.

As shown, the tool is composed of five main functions. Such a modular design mitigates the complexity and allows to add the support for other channels and data hiding mechanisms in a simple manner. In more detail, the software is composed of the following functions/building blocks:

- `read_attack()`: it parses data contained in the `payload.txt` file. The function then splits the payload into chunks fitting the size of the selected carrier (e.g., 8 bits for the case of the `Traffic Class` of the IPv6 header). When running the tool in interactive mode, the function can also parse the input information to be sent via the covert channel directly from the command line interface. Instead, for the case of bulk mode, each entry in the file specifies a payload that has to be sent through the covert channel along with the desired injection mechanism;

²<https://scapy.net> [Last Accessed, October 2022].

³<https://www.wireshark.org/docs/man-pages/tshark.html> [Last Accessed, October 2022].

- `find_flows()`: it parses data from the *input.pcap* file and prepares a “map” of the available conversations. As a result, the function provides conversations having enough steganographic capacity, i.e., it reports flows with a number of packets able to contain the specified payload, given the specific injection mechanism;
- `flow_selection()`: it allows to identify in a unique manner the traffic flow selected by the user when the tool is running in interactive mode. Depending on the protocol chosen, the function relies upon different header fields to identify the flows, e.g., 5-tuple for TCP/IPv6 conversations. Such details are also used to provide the user appropriate details to prepare filters and to track network covert channels when traffic is replayed through the network;
- `inject()`: it performs the injection using the desired mechanism. In particular, it searches for the packets belonging to the desired flow and changes them according to the selected covert channel, leaving unaltered the remaining parts. This function is modular and can be extended to create other covert channels. In this case, it is sufficient to add an `if-case` and prepare a packet with proper invocations of the Scapy library. The function also populates the *output.pcap* file, both with unaltered conversations and those containing the covert channel(s);
- `write_to_csv()`: it logs information about the flows containing the covert channels, such as the tuple to identify the conversation, the injection mechanisms and the length of the hidden data. Information are stored in a CSV file (denoted as the *logfile.csv* in the figure), which can be used to automatize the extraction phase, replicate experiments or generate filtering/forwarding rules for field trials.

We point out that, the aforementioned functions are general enough to handle the different working modes available for `pcapStego` (i.e., interactive or bulk). Minimal differences in the architecture of the tool are only due to the need of presenting to the user a suitable interface for selecting the flows or to identify the conversations to target when the tool is used in bulk mode. Specifically, when running in bulk mode, the `find_flows()` function automatically searches for all the traffic flows with enough capacity for the creation of the covert channels provided by the user. As an example, Listing 1 provides a code snippet for the `inject()` building block. In more detail, at line 1, Scapy reads the input `.pcap`, which is then parsed in the `for` loop. A packet is then modified according to the selected field, e.g., `Type of Service` at line 9, if and only if it is part of the selected conversation. Finally, at line 19, the (modified) packet is written in the resulting output `.pcap` file.

```

1  pkts = rdpcap(input_pcap)
2  for pkt in range(len(pkts)):
3      #Search for the correct flow
4      if source == pkts[pkt][IP].src and destination == pkts[pkt][IP].dst
5      and src_port == pkts[pkt].sport and dst_port == pkts[pkt].dport
6      and protocol == pkts[pkt][IP].proto:
7      #If there is still something to inject
8      if index < len(payload):
9          if targeted_field == "TOS":
10             pkts[pkt][IP].tos = int(payload[index],2)
11         elif targeted_field == "TTL":
12             if int(payload[index],2) == 0:
13                 pkts[pkt][IP].ttl = 10
14             else:
15                 pkts[pkt][IP].ttl = 250
16         elif targeted_field == "ID":
17             pkts[pkt][IP].id = int(payload[index],2)
18             index += 1
19     wrpcap(output_pcap, pkts[pkt], append=True, linktype=1)
20 return output_pcap

```

Listing 1: Code snippet for the inject () function.

2.1.2 Performance of the Tool

To prove the effectiveness of pcapStego for the preparation of realistic traffic traces containing network covert channels, especially to support the creation of testbeds, repeatable experiments, and suitable datasets to be used with AI-based frameworks, we conduct different trials. In the following we mainly address the case of the IPv6 protocol, however the results can be straightforwardly extended to the other protocols/covert channels supported by pcapStego.

2.1.2.1 Data Injection and Replaying

In the first round of tests, we evaluated the correctness of the embedding/extraction process, including the preservation of the semantic/structural properties of the output .pcap file. To this aim, we used realistic IPv6 traffic traces provided by the Center for Applied Internet Data Analysis (CAIDA)⁴. To avoid burdening the trials, we performed a lightweight preprocessing of traffic. Specifically, we removed single-packet flows and ICMPv6 traffic. Concerning the information transmitted via the various covert channels, we considered a command used by the Astaroth mal-

⁴Traffic dumps have been taken from the CAIDA Anonymized Internet Traces Dataset (April 2008 - January 2019) collected on a OC192 link between Sao Paulo and New York on November, 2018 from 14:00 to 15:00 CET. Available online: https://www.caida.org/data/passive/passive_dataset.xml [Last Accessed: October 2022].

No.	Time	Source	Destination	Protocol	Length	Traffic Class
79	0.012490	d0e7:fb50:fb38:ba5a::	d31c:6ecf:61a7:c2ff::	TCP	76	0x00
165	0.027422	d0e7:fb50:fb38:ba5a::	d31c:6ecf:61a7:c2ff::	TCP	76	0x00
337	0.054405	d0e7:fb50:fb38:ba5a::	d31c:6ecf:61a7:c2ff::	TCP	76	0x00
518	0.085422	d0e7:fb50:fb38:ba5a::	d31c:6ecf:61a7:c2ff::	TCP	76	0x00
620	0.104501	d0e7:fb50:fb38:ba5a::	d31c:6ecf:61a7:c2ff::	TCP	76	0x00
848	0.132486	d0e7:fb50:fb38:ba5a::	d31c:6ecf:61a7:c2ff::	TCP	76	0x00

(a) Input traffic dump

No.	Time	Source	Destination	Protocol	Length	Traffic Class
79	0.012490	d0e7:fb50:fb38:ba5a::	d31c:6ecf:61a7:c2ff::	TCP	76	0x6f
165	0.027422	d0e7:fb50:fb38:ba5a::	d31c:6ecf:61a7:c2ff::	TCP	76	0x73
337	0.054405	d0e7:fb50:fb38:ba5a::	d31c:6ecf:61a7:c2ff::	TCP	76	0x20
518	0.085422	d0e7:fb50:fb38:ba5a::	d31c:6ecf:61a7:c2ff::	TCP	76	0x67
620	0.104501	d0e7:fb50:fb38:ba5a::	d31c:6ecf:61a7:c2ff::	TCP	76	0x65
848	0.132486	d0e7:fb50:fb38:ba5a::	d31c:6ecf:61a7:c2ff::	TCP	76	0x74

(b) Output traffic dump

Figure 2.2: Traffic dumps (in .pcap format) before and after using pcapStego to implement a covert channel targeting the Traffic Class for transmitting a command used by the Astaroth malware.

ware, borrowed from the Fileless Command Lines (FCL) collection⁵. Such a command causes the Windows utility WMIC to download a legitimate file containing an obfuscated JavaScript file, which is then executed to launch a malicious Astaroth routine.

To assess the ability of pcapStego of producing traffic traces that can be used in realistic environments, the obtained .pcap files have been processed via tcprewrite⁶ to rebuild the payload and assign proper MAC addresses. We point out that, such steps can be omitted when in the presence of complete or non-anonymized traffic captures. Yet, using anonymized traces composed of packets deprived of the payload further proves the effectiveness of our approach to conduct research and experiments while complying with privacy of users. Obtained traces have been then transmitted over the network with tcpreplay⁷ and re-collected with tshark in order to verify the “correctness” of the traffic.

Figure 2.2 depicts partial dumps of a .pcap trace before and after the creation of a covert channel targeting the Traffic Class field. For the sake of clarity, the figure only reports

⁵<https://github.com/chenerlich/FCL> [Last Accessed, October 2022].

⁶<https://tcpreplay.appneta.com/wiki/tcprewrite> [Last Accessed, October 2022].

⁷<https://tcpreplay.appneta.com> [Last Accessed, October 2022].

six packets of the original and replayed `.pcap` files. As shown in Figure 2.2(a), before the creation of the channel, the original IPv6 flow is characterized by a `Traffic Class` equal to 0. Instead, after the creation of the covert channel used to deliver the malicious Astaroth command, the `Traffic Class` changes accordingly. Figure 2.2(b) reports the values of the `Traffic Class` of packets containing the “`os get`” portion of the entire command, e.g., `0x6f` and `0x73` correspond to “`o`” and “`s`”, respectively.

2.1.2.2 Generation of Metrics

To face the increasing complexity of threats as well as the large-scale and heterogeneous nature of modern network deployments, a common approach is to use some form of AI to reveal malicious communications, identify malware samples, as well as to perform sanitization of traffic and multimedia contents [CCC⁺20]. Moreover, modern software-defined networks offer a large palette of opportunities both in terms of architectural blueprints (e.g., the possibility of developing controllers to be deployed in edge nodes) and technologies for gathering data [SCPA19]. Hence, being able to produce suitable datasets for the development of machine-learning-based frameworks is of prime importance [SCPA19, CCC⁺20, Cav21].

To prove the effectiveness of `pcapStego` to produce appropriate information for the generation of AI-friendly metrics, we performed several round of tests. Specifically, we used the tool to create two different datasets. The first contains the original traffic and three covert channels targeting the `Flow Label` field of IPv6, which directly stores the secret data. The second, instead, exploits the `Hop Limit` field and contains three hidden channels as well. In this case, to encode data, we used two fixed values, i.e., 10 for the binary value 1 and 250 for the binary value 0. For both datasets, the secret information sent via the channels was the obfuscated payload of the Emotet malware (2,045 bytes), borrowed again from the FCL collection, and two random strings (2,048 bytes, each), cloaked in 60 seconds of network activity.

As regards the metric, we bear with heatmaps computed from the distributions of values of the fields targeted by the considered channels. For the `Hop Limit`, the map has been computed by considering the number of time a specific value has been observed in the overall traffic volume. Instead, for the `Flow Label`, we adopted a bin-based approach. Specifically, the 2^{20} bit space of the `Flow Label` has been mapped to a smaller one composed of 2^8 bins (i.e., ranges of values), each one grouping 2^{12} possible values. Such a mechanism prevents scalability issues and “noisy” heatmaps. Chapter 3 will clarify the benefits of using a bin-based data structure both in terms of performance and scalability, especially to inspect traffic in large-scale network environments.

To conduct experiments, we prepared a network testbed composed of three hosts. Two hosts exchanged a trace containing the covert channels, which has been “rebuilt” with `tcprewrite` and then replayed with `tcpreplay` (denoted as “Output Traffic” in the figures). A third host is responsible of routing the traffic, collecting the values of the `Hop Limit` and `Flow Label`

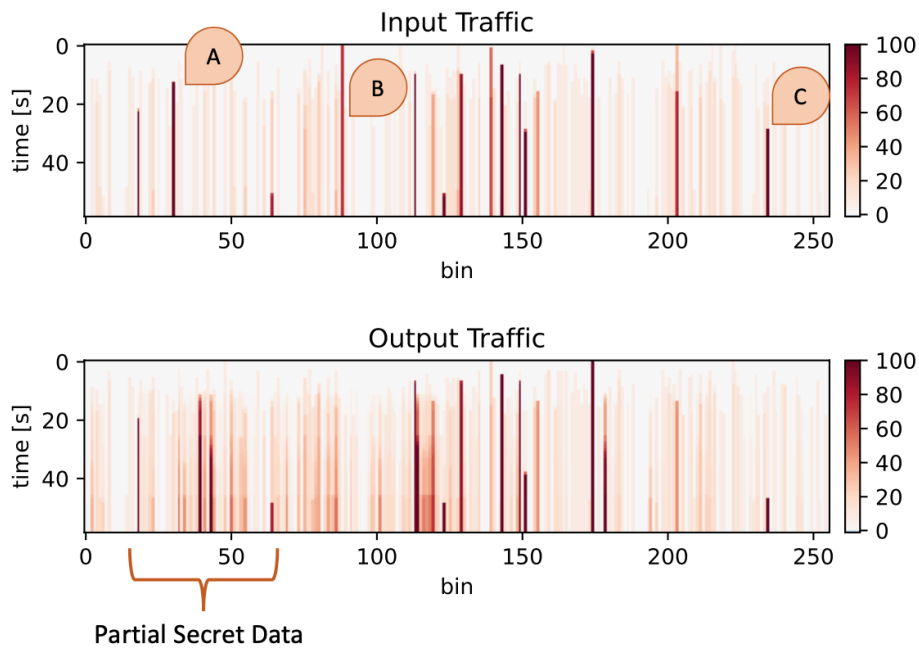


Figure 2.3: Heatmaps generated before and after using `pcapStego` for creating covert channels in the `Flow Label`.

fields and storing them on the file system for further processing. To have a reference scenario, we also computed heatmaps of the original `.pcap` trace, denoted as “Input Traffic” in the figures.

Figure 2.3 depicts example heatmaps both for the input and the output traffic, which have been selected to provide a clear “visual” template when a covert communication is implemented within the `Flow Label`. Since each IPv6 conversation is supposed to be identified via its unique `Flow Label` value, in the following, we refer both to `Flow Label` and flows interchangeably, except when doubts arise. As shown, `pcapStego` is used to embed data within flows denoted as A, B and C of the input traffic. Since original values of the `Flow Label` have been replaced with the secret data, the heatmap of the output traffic exhibits alterations of the corresponding bins. Specifically, the flows A and B are completely “exhausted” by the covert channel, whereas the conversation C is only partially used as a carrier. Moreover, the presence of hidden data leads to many values of the `Flow Label` not present in the original input `.pcap`. This can be viewed in the map in terms of the increasing “heat” of some bins (denoted as “Partial Secret Data” in the figure).

Similar considerations can be drawn when the covert channels are implemented within the `Hop Limit` field. Specifically, Figure 2.4 depicts the heatmaps collected in our testbed. For the case of the original input traffic, the map shows a clusterization around 64, i.e., the default value for the `Hop Limit` defined by the IPv6 standard implementation. Upon creating the covert channel, two different values become visible (denoted in the figure as D and E). Specifically,

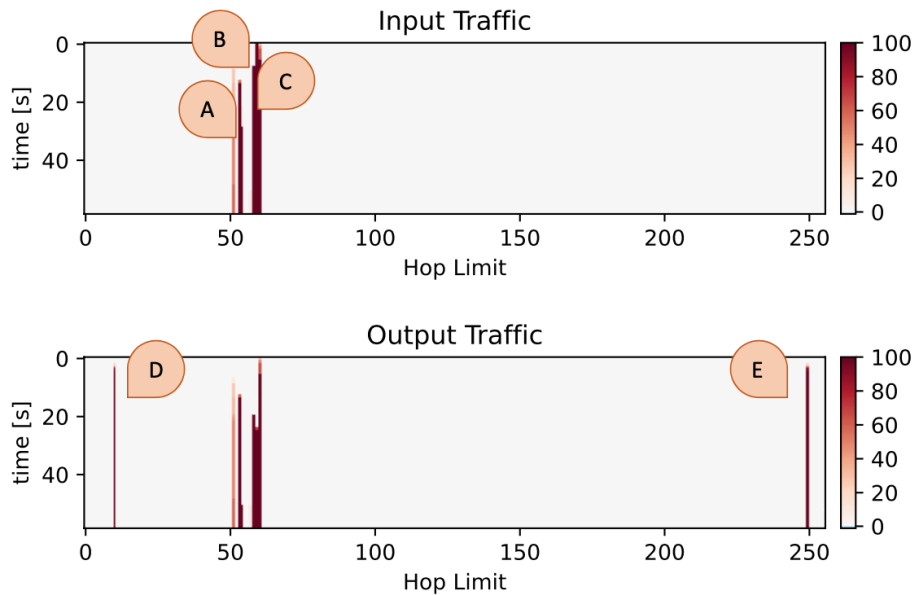


Figure 2.4: Heatmaps generated before and after using `pcapStego` for creating covert channels in the Hop Limit.

they correspond to 10 and 250, which are the values used by `pcapStego` to encode the secret information sent through the covert channels.

2.1.2.3 Resource Usage

The last round of tests aimed at investigating the performance of `pcapStego`. To conduct trials, we used a machine running Ubuntu 20.04 with an Intel Core i9-9900KF @3.60GHz and 32 GB RAM. Table 2.2 summarizes measurements obtained for `pcapStego` running in interactive mode. Specifically, it reports the execution time of the main functions implementing the tool (i.e., the `find_flows()` and `inject()`), to inject a secret of 110 bytes within the Flow Label field. To this aim, we considered different sizes of the `.pcap` file provided as the input. To have a fine-grained evaluation, we also performed tests with `.pcap` files composed of a different number of packets, especially to evaluate if the combination of the size and complexity of the traffic dump plays a role. As shown, the execution time needed by the `find_flows()` function to search for conversations able to contain the secret increases with the size of the traffic trace. Yet, the required time remains bounded, especially for small- and medium-sized datasets. Instead, the `inject()` function turns out to be the real bottleneck of the processing stack implemented by `pcapStego`.

To better understand the limits of the `inject()` function, we performed further trials. Table 2.3 reports a breakdown of the operations performed by the function, i.e., reading the input

<code>.pcap size</code> [kbytes]	<code>packets</code> [x10 ³]	<code>find_flows ()</code> [s]	<code>inject ()</code> [s]
84	1	0.16	0.31
845	10	0.31	2.86
8,557	100	1.65	28.14
84,117	1,000	14.58	276.87
842,105	10,000	142.55	2,835.91

Table 2.2: Performance of `pcapStego` in interactive mode.

<code>.pcap size</code> [kbytes]	<code>packets</code> [x10 ³]	<code>read</code> [s]	<code>inject and write</code> [s]
84	1	0.09	0.22
845	10	0.67	2.19
8,557	100	6.82	21.32
84,1174	1,000	64.86	212.01
842,105	10,000	669.49	2,166.42

Table 2.3: Breakdown analysis of the `inject ()` function in interactive mode.

`.pcap` (denoted as “read”) and preparing the output traffic trace (denoted as “inject and write”). As shown, the performance heavily depends on the size of the dataset, which accounts for a major overhead for a `.pcap` file containing 10 millions of packets. In this case, the preparation of the output `.pcap` requires ~ 35 minutes, which is not acceptable when using the tool for live demonstrations or for training purposes. However, training duties seldom require to deal with such a large traffic capture. Besides, further measurements and a deeper analysis revealed that the injection/writing operations are limited by a twofold bottleneck: the I/O bound nature of read/write operations needed to process/produce the various `.pcap` files and the overheads caused by the Scapy stack for parsing and manipulating all the considered packets.

A similar evaluation campaign has been also done when `pcapStego` operates in bulk mode. To have a realistic reference usage pattern for the tool, we considered a varying population of secret messages to be sent through an equal amount of covert channels. Each secret message is composed of a randomly-generated string in the range of 8 – 64 characters (e.g., a command for activating a backdoor or a sensitive information exfiltrated from a host). For this round of tests, we assessed three IPv6 embedding mechanisms implemented by `pcapStego`, i.e., we considered a balanced mix of covert channels targeting the `Flow Label`, `Traffic Class` and `Hop Limit` fields. Table 2.4 summarizes the obtained results.

Despite the number of channels, the execution time needed by the `read_attack ()` function is always almost negligible, especially by considering that the tool is intended for preparing large-sized datasets in an offline flavor. Instead, the time needed by the `find_flows ()` function increases mainly due to the need of building a larger set of 5-tuples representing the snapshot

<code>.pcap size</code> [kbytes]	<code>packets</code> [x10 ³]	<code>number of channels</code>	<code>read_attack ()</code> [ms]	<code>find_flows ()</code> [s]	<code>inject ()</code> [s]
845	10	10	0.23	0.36	3.04
845	10	100	0.28	1.36	3.25
845	10	1,000	–	–	–
84,117	1,000	10	0.21	15.07	304.9
84,117	1,000	100	1.22	18.37	327.72
84,117	1,000	1,000	10.53	106.72	471.55
842,105	10,000	10	0.26	140.31	2,452.69
842,105	10,000	100	1.34	155.95	2,611.58
842,105	10,000	1,000	10.73	470.57	3,743.88

Table 2.4: Performance of `pcapStego` in bulk mode.

<code>.pcap size</code> [kbytes]	<code>packets</code> [x10 ³]	<code>.pcap size channels</code>	<code>read</code> [s]	<code>inject and write</code> [s]
845	10	10	0.67	2.37
845	10	100	0.66	2.59
845	10	1,000	–	–
84,117	1,000	10	65.03	239.87
84,117	1,000	100	66.34	261.38
84,117	1,000	1,000	66.23	405.32
842,105	10,000	10	661.27	1,791.42
842,105	10,000	100	665.55	1,946.03
842,105	10,000	1,000	655.27	3,088.61

Table 2.5: Breakdown analysis of the `inject ()` function in bulk mode.

of the IPv6 conversations within the input `.pcap` file. Similarly to the interactive case, the `inject ()` function represents the most time-consuming building block of the tool. Specifically, when the number of channels to be created within the traffic trace approaches the 1,000 units, the handling of large `.pcap` files still accounts for reduced performance.

Again, we analyzed the breakdown of the timing behavior of the `inject ()` function when used in bulk mode. Table 2.5 reports the obtained results. In more detail, as the number of channels increases, the number of I/O-bound operations and invocations of the Scapy library increase as well, thus leading to inflated computational times (e.g., 3,088.61 s when processing the largest `.pcap` file considered in our trials). Instead, as expected, the part of the tool processing the input traffic trace is quite insensitive to the complexity of the required embedding operations.

Lastly, we also evaluated the amount of resources used by `pcapStego`. For the sake of brevity, we limit our investigation to the interactive mode case. Results indicate that the memory footprint is proportional to the size of the `.pcap` to be processed. Specifically, the memory consumption

ranges from 3.3 Mbytes to 24 Gbytes, for the smallest and largest considered traffic traces, respectively. This has to be mainly ascribed to the Scapy library having to process all the packets composing the `.pcap`. The size of the input also heavily influences the CPU usage. In fact, it is already equal to 93% when in the presence of medium-sized `.pcap` files (i.e., for a dataset composed of 100,000 packets). Thus, the need of processing through Scapy large volumes of data both accounts for I/O and CPU bound operations impacting the overall performance.

2.2 Online Creation of Covert Channels

Even if traces prepared with `pcapStego` can be replayed and used in realistic network conditions, the lack of processing “live” network conversations could partially void the accuracy assessing the vulnerability of a deployment against covert channels. In fact, exploited flows could be accidentally disrupted or delayed, thus revealing the presence of the covert channels or the traffic manipulation attempts from an attacker. Therefore, a second approach that has been developed and adopted in this Thesis concerns the development of tools able to act in a Man-in-the-Middle fashion, by eavesdropping pre-existent and “live” network conversations. In the following, we will introduce `IPv6CC`, a tool able to lively intercept legitimate IPv6 network traffic and create network covert channels. A similar approach has been also adopted for the case of TLS traffic, by using the `TLSCC` tool.

2.2.1 Covert Channels in IPv6

As discussed in Chapter 1, the IPv6 protocol is increasingly used and appealing to attackers, since it provides several features that can be exploited to implement covert channels or support attack routines of stegomalware [LLC05, MPC19]. For these reasons, we developed `IPv6CC`⁸, a suite of network covert channels targeting the IPv6 protocol, which has been used within the framework of the project `SIMARGL` to test a wide range of security devices. Its main scope is supporting penetration test campaigns to evaluate the security of a system against emerging information-hiding-capable attacks or steganographic malware. Specifically, the `IPv6CC` suite can be used to:

- comprehend the menace: the technology-dependent nature of stegomalware prevents one-fits-all security roadmaps. This tool can help to understand whether a deployment is already protecting itself from covert communications;
- assess security: testing the adopted security policies/solutions allows to plan upgrades and adjust configurations or the design of the network;

⁸<https://github.com/Ocram95/IPv6CC-SoftwareX> [Last Accessed, October 2022].

- make decisions: understanding the performance of a covert channel helps to quantify the menace and the economical impact of required solutions.

The tool is written in Python3 and uses Scapy 2.4.3 for handling data injection and Netfilterqueue 0.8.1 for intercepting the overt IPv6 flows. Currently, the tool supports Traffic Class, Flow Label, and Hop Limit IPv6 fields for hiding purposes. Moreover, IPv6CC offers four working modes for implementing covert communications with different degrees of sophistication:

1. Naive Mode: the covert endpoints decide offline the number of packets containing the secret to be transmitted. The covert peers exfiltrate the information for the number of agreed packets starting from the beginning of the transmission;
2. Start/Stop: two “magic values” indicating the start and the end of a sequence of packets containing secret information are agreed offline. Such values are then injected in the targeted field of the IPv6 header. To avoid collisions, hidden data is inserted via character stuffing;
3. Packet Marking: both endpoints generate a set of encrypted signatures produced by a pseudo-random number generator built on a shared seed. A mark is then injected into a field not containing secret information: for the Flow Label, Traffic Class and Hop Limit channels, the mark is inserted in the Traffic Class, Flow Label, and Flow Label, respectively;
4. Reliable Marking: this is a cross-layer variant of the previous mode and exploits information from the transport layer to implement some form of error detection. The TCP sequence number is used to detect loss of marked packets.

For all methods, it is also possible to use a more fine-grained injection strategy for simulating an attacker/malware trying to bypass potential (un)known detection mechanisms. In more detail, IPv6CC allows to specify a suitable interleaving between legitimate and “stego” packets, as to reduce the chance of spotting the channels due to anomalous bursts [MPC19]. Further details on the tool, along with its performance, are available in [CSZM22].

2.2.2 Covert Channels in TLS

TLSCC⁹ is a tool for creating network covert channels targeting the TLS protocol. This tool has been developed by Corinna Heinz within her work with the FernUniversität in Hagen [HMC20]. The covert channels are implemented in C, using libcrypto (for a wide range of cryptographic

⁹<https://github.com/CoriHe/TLSCC> [Last Accessed, October 2022].

algorithms used in various Internet standard) and libpcap¹⁰ (for packet capturing). In essence, TLSCC establishes a connection to a TLS-enabled service and injects data into the conversation according to the selected hiding mechanism. Currently, TLSCC supports the following covert channels, which have been selected as they are characterized by a good tradeoff between the steganographic bandwidth and the undetectability (see [HMC20] for a more comprehensive discussion on possible TLS-based covert channels):

- **Record Length Encoding:** since the overt TLS messages can be split into an arbitrary number of records, a secret information can be hidden by modulating their size and number. To this aim, the covert sender encodes the secret message by properly generating record lengths. The overt TLS traffic is then artificially split into records of the needed lengths and forwarded to the receiver. This modification has no influence on the TLS payload data and the overt TLS endpoints are still able to correctly decrypt the information;
- **Initialization Vector:** this field can be targeted for data hiding purposes when a Cipher Block Chaining (CBC) cipher (e.g., the AES256-CBC) is used to encrypt each TLS data record. This can be only modified by the overt TLS nodes, otherwise the manipulation of the TLS stream will be detected and the connection will be shut down;
- **Record Content Type:** the Record Layer of the TLS protocol uses the Content Type to specify the content of a record. An attacker wanting to create a covert channel within the TLS stream can use non-critical alert messages by exploiting unassigned alert identifiers (see, e.g., [DR08] for a list of TLS Alerts maintained by the Internet Assigned Numbers Authority) or an empty record. Then, by interleaving records of various types, the secret information can be encoded. As an example, a simple encoding scheme could consider the alternation of TLS data messages and TLS alert messages. For instance, a data message represents a 1 bit, while an alert message a 0 bit. The covert receiver inspects the sequence of alert and data records and then decodes the secret information according to the agreed scheme.

2.3 Security Assessment

As said, network covert channels are seldom considered and many security tools are not capable to perform the detection “out-of-the-box”. In fact, risks arising from network covert channels are largely underestimated both by security experts and users [MC15, CCM⁺18, CCC⁺20]. Moreover, extending the functionalities of network security tools could lead to inefficient behaviours since each protocol requires ad-hoc rules. To avoid bottlenecks, security mechanisms are often event-based and they could not offer detection schemes based on a per-packet granularity.

¹⁰Part of the tcpdump analyzer: <https://www.tcpdump.org> [Last Accessed, October 2022].

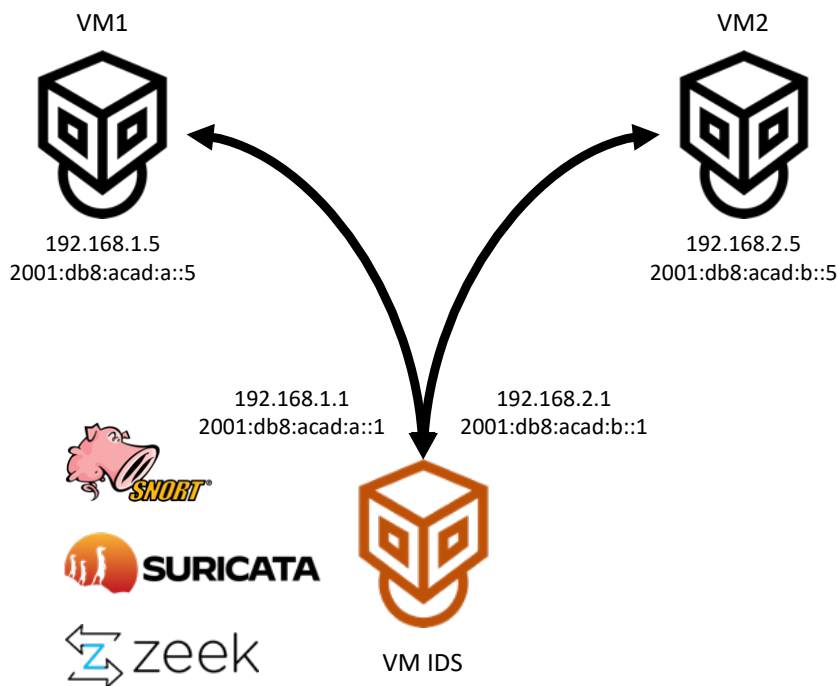


Figure 2.5: Reference testbed for the security assessment.

As a result, several implementations are not suitable to handle all the possible protocol-hiding combinations.

Therefore, in the following we will evaluate the “level of insecurity” of covert channels when deployed in realistic network scenarios. To this aim, we investigate three popular network IDSes when handling the various network covert channels supported by `IPv6CC`, `TLSCC`, and `pcapStego`, mainly to highlight their inadequacy when used to counteract information-hiding-capable threats [MC15, KWE⁺16, CCM⁺18].

2.3.1 Experimental Setup and Results

To conduct trials, we prepared the testbed depicted in Figure 2.5, which is composed of two virtual machines, denoted in the figure as VM1 and VM2, both running Debian GNU/Linux 10 (kernel 4.20.9). The endpoints communicate through a third virtual machine, denoted in the figure as VM IDS. The latter, with the same technical specifications, is in charge of routing traffic and running the different security tools. In our tests, we considered three different, de-facto

standard security frameworks: Snort¹¹, Suricata¹², and Zeek¹³. To prove that risks arising from network covert channels are often underestimated, we decided to test them in an “out-of-the-box” fashion. Specifically, we considered Snort with two different set of rules (the “community” and the “registered” sets), while for the case of Suricata and Zeek we considered only standard configurations.

To test the firewalls in real traffic conditions, we used traces collected by CAIDA and modified via `pcapStego` to contain various covert channels generated in an “offline” flavor. In particular, we considered a random string of 125 bytes, which can be representative of a malicious configuration file, and stored within the `Type of Service`, `Time to Live`, and `Identification Number` fields of IPv4, in the `Traffic Class`, `Hop Limit`, and `Flow Label` of IPv6, and in the `Payload` field of ICMPv4 and ICMPv6. Moreover, we evaluated timing covert channels encoding the random string in the inter-packet time for all the aforementioned protocols. To investigate a wide range of network conditions, we considered both UDP and TCP transmissions sent over IPv4 and IPv6. Throughout all the trials, i.e., TCP over IPv4 and IPv6, UDP over IPv4 and IPv6, as well as ICMPv4 and ICMPv6 traffic, the considered tools never spot the presence of hidden data by means of any alert, warning, or other flag.

To make our investigation more comprehensive, we also evaluated “online” covert channels targeting the TLS protocol created via the TLSCC suite. For our experiments, we considered three different types of hidden data in order to model realistic attacks. The first case deals with a channel used to deliver a malware of 179 bytes, i.e., a PowerShell script allowing to infect the victim with the CryptoWorm threat. The second case considers an obfuscated payload of 2,046 bytes retrieved via the covert channel, i.e., a collection of malicious instructions to run the Emotet malware. Both the CryptoWorm and Emotet payloads are borrowed from the FCL collection. The third case models the exfiltration of sensitive data via the covert channel from the host of the victim towards a remote server. Specifically, we modeled the stolen data with a string composed of 1,000 randomly-generated bytes to consider the presence of some form of encryption or scrambling to improve the undetectability of the hidden content. The different payloads have been transmitted via the `Initialization Vector`, the `Record Content Type`, and `Record Length Encoding` covert channels available in TLSCC.

Finally, we tested covert channels created via the IPv6CC suite. We evaluated the exfiltration of 625 and 1,250 bytes messages within the `Traffic Class`, `Hop Limit`, and `Flow Label` of IPv6. Since the IPv6CC acts in a man-in-the-middle fashion on overt communications, we considered different legitimate traffic, i.e., traffic generated via an SCP file transfer between VM1 and VM2 and a mix of TCP/UDP traffic generated via `iPerf3`¹⁴. To avoid burdening results, in the following we will not report other tests performed with the IPv6CC, including those varying

¹¹<https://www.snort.org> [Last Accessed, October 2022].

¹²<https://suricata.io> [Last Accessed, October 2022].

¹³<https://zeek.org> [Last Accessed, October 2022].

¹⁴<https://iperf.fr> [Last Accessed, October 2022].

Protocol	Field	Snort		Suricata	Zeek
		community	registered		
TLS	Initialization Vector	–	–	–	–
	Record Content Type	–	–	–	–
	Record Length Encoding	–	–	–	–
SCP traffic					
IPv6	Traffic Class	○●	○●	–	–
	Hop Limit	○●	○●	–	–
	Flow Label	○●	○●	□	–
iPerf3 traffic					
IPv6	Traffic Class	○	○	■	–
	Hop Limit	○	○	■	–
	Flow Label	○	○	■	–

- “Consecutive TCP Small Segments Exceeding Threshold”
- “Challenge-Response Overflow Exploit”
- “Packet with Invalid Timestamp”
- “Applayer Protocol Detection Skipped”
- “No Alerts”

Table 2.6: Security assessment when considering “online” covert channels generated with TLSCC and IPv6CC suites.

the working modes presented in Section 2.2.1 or trying different injection strategies.

Since the outcomes did not change among the different payloads tested, Table 2.6 summarizes the results in a “condensed” manner. Similar to previous results, all the tools did not raise any flags when dealing with TLS-based covert channels. Concerning IPv6 hidden communications, Snort reported the flags “Consecutive TCP Small Segments Exceeding Threshold” and “Challenge-Response Overflow Exploit” and denoted in the table with ○ and ●, respectively. Both warnings are mainly due to the usage of SSH traffic, rather than the presence of a covert channel. In fact, they are also raised when Snort processes legitimate SSH connections. Similarly, Suricata raised the flags “Packet with Invalid Timestamp” (for the case of SCP) and “Applayer Protocol Detection Skipped” (for the case of iPerf3) and denoted in the table with □ and ■, respectively. Also in this case, the warnings cannot be considered representative of the presence of the covert channels since they are also raised during legitimate conditions. Finally, Zeek did not raise any flag for all the covert channels despite they were injected using overt SCP or iPerf3 traffic.

Summarizing, the selected security tools were not able to detect the covert communications or raise relevant warnings. Such a finding reinforces the general idea that many firewalls and IDSes, with a standard set of rules, cannot be considered a valid countermeasure against information-hiding-capable threats without massive configuration and tweaking efforts. In fact, writing new detection rules for dealing with network covert channels could require to modify the core of the

detection mechanism. Even if possible, it is necessary to deal with the limits imposed by the rules matching structures of the tools. On the other hand, Zeek is a scriptable network IDS, thus it makes possible to write new detection mechanisms for covert channels. Unfortunately, due to its event-based nature, it also requires to write ad-hoc event handlers for dealing with the all possibilities and combinations for the creation of covert channels (e.g., protocols and fields), which could be unfeasible in many realistic conditions.

2.4 Conclusions and Future Works

In this chapter we introduced three tools for testing network covert channels, i.e., pcapStego, TLSCC, and IPv6CC. The tools allowed to create covert channels both in an “online” and “offline” manner. Then, we tested three production-quality network security tools (i.e., Snort, Suricata and Zeek) and results showcased their “inability” to detect network covert channels, highlighting the urgency of developing novel countermeasures and rules to anticipate such a threat. Future works aim at extending the assessment by considering ad-hoc rules, testing more sophisticated security frameworks, and types of channels.

In the next chapter, we will discuss a scalable, extensible and privacy-aware data gathering approach for revealing network covert channels.

Chapter 3

Data Gathering for Covert Channels

As discussed in Chapter 1, the detection of network covert channels is a nontrivial and poorly generalizable problem. In fact, spotting hidden communications within a bulk of network flows typically requires to implement attack-specific methodologies or to deploy DPI strategies, which pose scalability problems [Cav21]. The latter should be considered as a design constraint, since inspection processes should not penalize legitimate traffic flows, e.g., by adding additional delays or disrupt the perceived Quality of Experience [ZAB07, CCC⁺20, Cav21]. To have a foundation for gathering the needed information to tackle the problem of detecting covert channels hidden within network traffic, we developed `bccstego`. Specifically, `bccstego` is a framework that allows to collect statistics for specific fields of network protocols by creating and injecting custom eBPF programs within the Linux kernel.

The contribution of the research ideas presented in this chapter is the design of a tool that collects statistical indicators for spotting covert channels within network traffic and a performance evaluation considering the impact in terms of packet processing overhead and CPU/network footprints.

The remainder of the chapter is organized as follows. Section 3.1 explains the methodology to collect measurements, and the reasoning behind this choice. Section 3.2 describes `bccstego` whereas Section 3.3 reports the functional and performance evaluation of the tool. Section 3.4 showcases other solutions and alternative technologies that could be adopted. Lastly, Section 3.5 concludes the chapter.

3.1 Collecting Statistics From Protocol Headers

The most effective and straightforward way to detect storage covert channels targeting protocol headers (introduced in Section 1.2.2) requires to track the values assigned to relevant fields

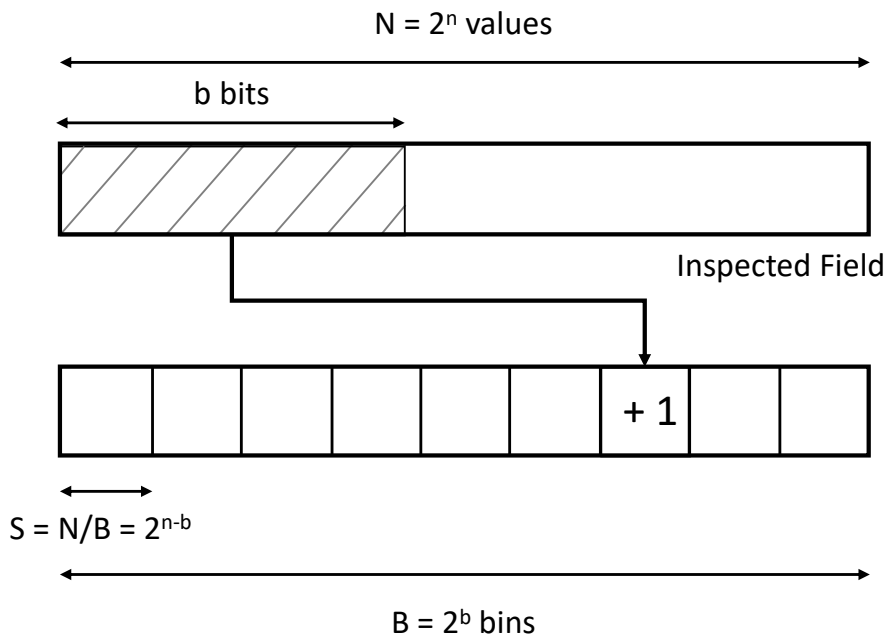


Figure 3.1: Mapping field values to bins.

within the same network flow, an approach that is usually denoted as “flow tracing.” Usually, the `<src addr, dst addr, protocol, src port, dst port>` tuple is used for identifying a conversation. For each stream, several parameters are recorded, e.g., the number of packets, the number of bytes, or the average inter-packet delay. It would be simple to extend such approach to also consider fields vulnerable to covert communication attempts (e.g., `Flow Label` or `Time To Live`) and use this information to detect anomalous usages. However, the overhead due to tracing increases linearly with the number of flows, and may be unfeasible for large Internet links. Indeed, common tools for network monitoring cannot sustain high bitrates, and sometimes make use of sampling techniques to “estimate” the number of active flows [EKMV04].

Based on this consideration, we introduce an alternative technique able to efficiently scale independently from the number of flows and save-up memory. Rather than keeping the “state” for each flow, we only measure general statistics about the usage of “vulnerable” fields. In essence, our approach counts the number of occurrences for the different values that a given field assumes. To make this approach scalable, multiple values may be grouped together into what we call a *bin*, and a single *counter* is assigned for each group. We will refer to this technique as “counters” to emphasize the differences with the flow tracing approach based on the tuple. The general idea is depicted in Figure 3.1. Specifically, $N = 2^n$ is the number of possible values for a given field, where n is its length in bit, e.g., $n = 20$ for the `Flow Label`. Such values are grouped into

$B = 2^b$ equally-spaced bins, thus $S = 2^{n-b}$ values are grouped into the same bin. The counter associated to each bin is incremented by 1 every time a specific field value of a packet falls in the corresponding range. In our design, the number of bins is always a power of 2, which is necessary to have uniform bins. In addition, mapping the value of a field to the corresponding bin is reduced to a simple bitwise operation, i.e., only the first b bits of a field value are used to search for the index of the corresponding bin.

As an example, let us suppose to monitor the `Time To Live` field of the IPv4 header. Since the `Time To Live` has 8 bits in total, the possible values are $N = 2^8 = 256$ (i.e., $n = 8$). Such values can be then grouped in a number of bins $B = 2^3$ (i.e., $b = 3$), each of which is $S = N/B = 2^8/2^3 = 32$ values wide. This leads to the fact that the first bin considers the range of values 0–31, the second bin 32–63 and so on, until the last bin, which deals with the interval 224–255. Let us suppose that an inspected packet has a `Time To Live` value equal to 237, i.e., “11101101” in bits. The first $b = 3$ bits, i.e., “111”, are used to increment by 1 the counter of the 8-*th* bin.

For what concerns resource consumption, the lower the number of bins, the less the memory needed. However, this requires to map a larger number of values in the same bin, hence resulting in coarse-grained statistics that limit the efficiency of the detection when many flows are present. To make a rough comparison of memory consumption, in the following we estimated the memory required by classical flow tracing approaches and by our counters methodology.

For the case of flow tracing, the computation of the required memory is rather straightforward. Let us showcase an example for an IPv6 flow. First, we should consider the tuple for identifying the conversation, i.e., source and destination addresses (128 bits each), protocol number (8 bits), and source and destination ports (16 bits each). Then, we also need memory for storing the state (8 bits), timestamp (64 bits), and the IPv6 header field that is supposed to contain secret data (32 bits, which allow to contain the larger field, i.e., `Flow Label`). Thus, each flow requires a minimum of 400 bits.

When using counters, only 32 bits of memory are required for each bin, independently of the length of the considered field. However, the number of bins B is not fixed. For the `Traffic Class` and the `Hop Limit`, the memory consumption accounts to 8,192 bits in the worst-case scenario, namely when 256 bins are used (one bin for each value). Instead, for the `Flow Label`, using a number of bins equals to the entire value space (e.g., $b = 20$) would led to an excessive overhead. From our experiments, we found the empirical rule of thumb that the number of bins B should be at least twice the average number of flows, in order to capture meaningful trends that could be used for the detection of anomalies caused by a network covert channel nested within a flow. To be more conservative in our estimation, we considered different scenarios, where the number of bins is 2, 4, 8, and 16 times the number of active flows.

Figure 3.2 compares the estimated memory consumption of our approach versus flow tracing while varying the number of active flows. As shown, the memory requirement for standard flow

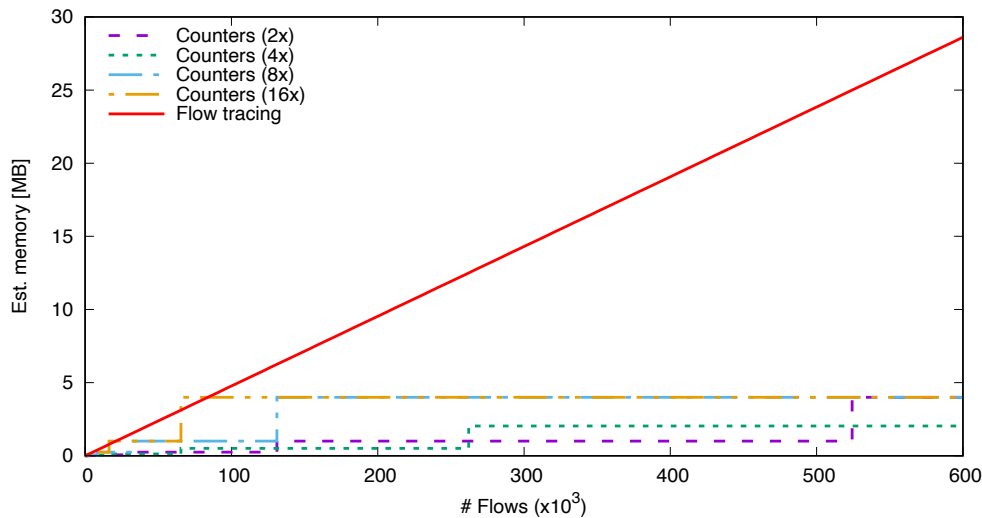


Figure 3.2: Estimation of memory consumption with different number of flows.

tracing increases linearly with the number of flows (i.e., each flow requires a 400 bit long record), whereas our method scales much better and the estimated memory consumption is larger only in case of few flows.

3.2 The `bccstego` Framework

The main objective is to design an inspection tool able to run with a low execution footprint in different environments, both physical and virtual. Since the range of possible covert channels is virtually unlimited, extensibility is an important design constraint, mainly for handling additional protocols or considering new attacks. To fit these requirements, we take advantage of the eBPF technology. In this section, we will introduce the eBPF framework and we will provide the implementation details of the `bccstego` framework.

3.2.1 The eBPF Technology

The Berkeley Packet Filter has been originally conceived to monitor and filter network traffic in a stateless manner. The eBPF is its extended version with the support of key/value data structures, helper functions, the ability to “stack” multiple programs, and an enhanced set of registers and instructions, just to mention the most important features. Moreover, interactions with user-space applications enable to process and monitor network traffic in a stateful manner. Natively designed to run on Linux kernels, the framework is supported by companies such as Facebook,

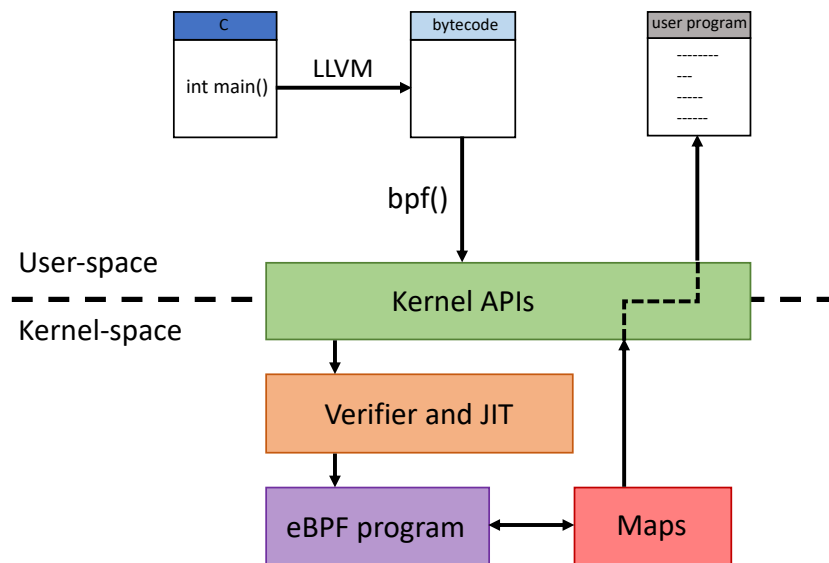


Figure 3.3: The eBPF architecture.

Google, Isolavent, and Netflix¹. By extending the visibility over the entire host, eBPF allows to improve observability, security, tracing, and network monitoring operations. For example, it can be used to inspect network packets, to trace specific kernel functions or to monitor the CPU or the memory usage. Owing to its flexibility, the eBPF can collect a wide range of data at runtime, from behaviour of software processes to network events, addressing both local and network covert channels.

In essence, eBPF is a virtual machine able to safely execute sandboxed programs within the kernel. The eBPF programs are attached to specific hook points: once the hook points are passed by the kernel or by an application, the programs are triggered and executed. There are several predefined kernel and application hooks, including system calls, function entry/exit points, kernel tracepoints, as well as network hooks such as XDP (the eXpress Data Path) or the `tc` (the traffic control subsystem). Hook points can be also created according to the needs (e.g., `kprobe` or `uprobe`). This event-driven architecture allows to execute programs only when needed, without impacting the overall performance of the kernel.

Figure 3.3 depicts the eBPF architecture. The programs are typically wrote in C and compiled in eBPF bytecode (for example by leveraging suite like LLVM²). The bytecode is then loaded in the kernel-space using the `bpf` system call. To guarantee the safety, an in-kernel verifier ensures that the program always terminates (e.g., it does not contain deadlocks or forever-loops), it does not crash or harm the system (e.g., by accessing memory out of bounds or using uninitial-

¹<https://netflixtechblog.com/how-netflix-uses-ebpf-flow-logs-at-scale-for-network-insight-e3ea997dca96> [Last Accessed, October 2022].

²<https://llvm.org> [Last Accessed: October 2022].

ized variables), and it has a finite complexity (i.e., the number of instructions that the verifier is allowed to check before rejecting the program). Once it is validated, a Just-in-Time (JIT) compiler optimizes the program by translating the bytecode into the machine-specific instruction set. Due to performance and security constraints, the only way to interact with an eBPF program is through data structures called *maps*, e.g., hash tables, arrays, ring buffers, or stack traces, able to store data. For this reason, the typical use of eBPF requires the development of both an eBPF program to collect measurements and a user-space utility to load the program in the kernel-space and interact with data through the maps.

As hinted, the most resource-consuming operation is the “periodical” data transfer between the kernel- to the user-space for a twofold reason. First, the more data an eBPF map contains in the kernel-space, the more resources are required to “move” the data to the user-space to perform the additional analysis. Second, the “period” to read the data must be properly chosen in order to avoid bottlenecks and overheads. This mainly leads to correctly devise a map capable of efficiently scaling with the inspected data. In Chapter 5 we will provide a thorough analysis of these parameters, highlighting their impact in term of performance.

Although the implementation of an eBPF program consists of a limited number of instructions, there may be a non-negligible overhead in the external constructs that are necessary to compile the code, load it into the kernel, and exchange data. Among the alternative loaders available (e.g., `bpftool`, `tc` and `ip` utilities), the BPF Compiler Collection (BCC)³ emerged as a flexible yet powerful framework for running eBPF programs, including a rich collection of kernel tracing tools for investigating the performance of the OS (e.g., CPU usage, TCP/IP active connections, system calls). The programming model for BCC revolves around a Python class delivering functionalities for compiling, loading, and running eBPF programs. The class also takes care of creating shared maps, and provides specific methods to read and write data. Therefore, the source code of the eBPF program is usually embedded into the user-space Python application (e.g., it is statically-stored as a string within the module), which simplifies the portability of the application. Alternatively, eBPF programs can also be loaded from an external file.

3.2.2 The `ipstats.py` Tool

The `bccstego`⁴ framework is an umbrella of various tools targeting specific traits of the traffic and providing support for the detection of network covert channels. The idea is to share a common pattern for parsing packets and collecting data, while different eBPF programs are developed for specific protocols or steganographic threats. Within the `bccstego` framework, the `ipstats.py` tool builds the usage statistics for multiple header fields, adopting the “counters” method introduced in Section 3.1.

³<https://github.com/iovisor/bcc> [Last Accessed, October 2022].

⁴<https://github.com/mattereppe/bccstego> [Last Accessed, October 2022].

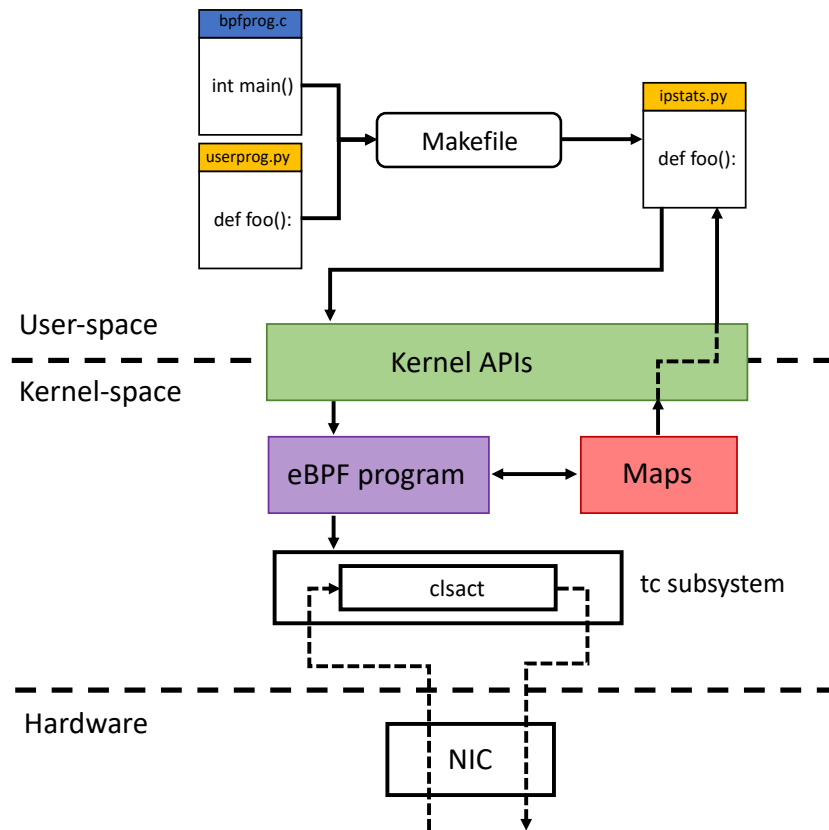


Figure 3.4: Software architecture of the `ipstats.py` tool.

Figure 3.4 depicts the software architecture and the build process of the `ipstats.py` tool, which is explicitly designed to facilitate the maintenance of complementary programs (i.e., the eBPF filter and user-space utility). There are two distinct files: `bpfprog.c` providing the skeleton of the eBPF program and `userprog.py`, which is the user-space utility written in Python. A `Makefile` automatizes the merge of the different components and builds the monolithic `ipstats.py` executable Python module. As soon as new programs will be added to `bccstego`, the same implementation pattern should be followed. There are different types of eBPF programs, based on the specific context where they are executed. For our purposes, we develop programs for the `tc` subsystem, which inspects network packets via the special `clsact` qdisc. This approach gives access to both ingress or egress traffic, and to richer kernel metadata with respect to the XDP subsystem.

Since the eBPF code is executed for each packet, it is important to use as few instructions as possible, mainly to avoid unnecessary computation overheads and delays. For this reason, the `ipstats.py` script dynamically creates the eBPF code for monitoring a specific header field, which is indicated on the command line as a parameter. The current version is able to collect data

and statistics for the protocols/fields that can be likely used for creating covert channels: IPv6 (Flow Label, Traffic Class, and Hop Limit), IPv4 (Type of Service, Identification Number, Time To Live, Fragment Offset, and Internet Header Length), TCP (Acknowledgement Number, Reserved Bits, and Timestamps), and UDP (Checksum). Other parameters that can be given from the command line include the number of bins B , the network interface to be monitored, the direction of the traffic (i.e., ingress or egress), the period to read data from kernel- to user-space, and the name of the file for saving the obtained statistics. Listing 2 showcases a code snippet of the `ipstats.py` tool. In particular, the C program `bpfprog.c` is stored in the Python variable `bpfprog`, which changes according to the number of bins set at line 2 and to the field length specified at line 4. Moreover, the `src` variable contains the code that depends on the specified field, in this case the `Flow Label` (lines 6–7), which is properly substituted in the `bpfprog` variable at lines 9–12. The resulting program is used to create a BPF object in order to be loaded in the kernel (lines 14–15). Finally, the variable `hist` contains the results stored in the `nw_stats_map` eBPF map, which is periodically queried to print out the values of the inspected field (lines 18–29).

3.3 Numerical Results

To evaluate the performance of the `ipstats.py` tool in the `bccstego` framework, we prepared a testbed composed of three virtual machines running Debian GNU/Linux 10 (kernel 4.20.9), with 1 virtual core and 4 GB of RAM: two machines exchanged traffic, while the third one acted as a router and ran the eBPF software. All the virtual machines were running on a 3.60 GHz Intel i9-9900KF host with 32 GB of RAM and Ubuntu 20.4 (Linux kernel 5.8.0). To test our tool in real-world network conditions, the overt IPv6 traffic load was created by replicating (via the `tcpreplay` tool) traffic collected on a OC192 link between Sao Paulo and New York on January 17, 2019 from 14:00 to 15:00 CET, and made available CAIDA⁵. The sampling interval, i.e., the frequency with which the user-space program reads the counters updated by the eBPF, was set equal to 1 second.

Since the eBPF map is copied from the kernel- to the user-space in a periodical manner, a useful metric is how many bins are incremented between two subsequent reads. Figure 3.5 depicts this idea. In case of `Flow Label` (see, Figure 3.5(a)), this metric provides an approximate information about the volume of active IPv6 flows populating the overt traffic. In fact, every packet belonging to the same IPv6 flow is expected to use the same `Flow Label` value. When an attacker uses this field to bear some secret, the number of changing bins will abnormally increase, hence this metric may be directly used for the detection of covert channels. A similar consideration can be drawn for the evolutions observed for the `Traffic Class` and `Hop`

⁵The CAIDA Anonymized Internet Traces Dataset (April 2008 - January 2019) - Used traces: Jan. 17th 2019. Available online: <https://www.caida.org/data/monitors/passive-equinix-nyc.xml> [Last Accessed: October 2022].

```

1  # Set the required number of bins in the source file
2  bpfprog = re.sub(r'SETBINBASE',r'#define BINBASE ' + str(binbase), bpfprog)
3  # Set the length of field to be monitored
4  bpfprog = re.sub(r'IPFIELDLENGTH',str(ipfieldlength), bpfprog)
5  # Set the specific code to read the required field
6  if prog == 'fl':
7      src = """
            for(short i=0;i<3;i++) {
                ipfield |= iph6->flow_lbl[i];
                if(i==0) {
                    /* Remove DSCP value */
                    ipfield &=0x0f;
                }
                if(i!=2)
                    ipfield <<= 8;
            }
            """
8  elif [...]

9  if prog in ipv6_fields:
10     bpfprog = re.sub(r'UPDATE_STATISTICS_V6',src, bpfprog)
11     bpfprog = re.sub(r'UPDATE_STATISTICS_V4',"", bpfprog)
12     bpfprog = re.sub(r'UPDATE_STATISTICS_L4',"", bpfprog)
13 elif prog in ipv4_fields: [...]

14 prog = BPF(text=bpfprog)
15 fn = prog.load_func("ip_stats", BPF.SCHED_CLS)
16 if direction == "ingress":
17     ipr.tc("add-filter", "bpf", idx, ":1", fd=fn.fd, name=fn.name,
            parent="ffff:fff2", classid=1, direct_action=True)

18 hist = prog.get_table('nw_stats_map')
19 try:
20     prev = time.time()
21     while True:
22         # Wait for next values to be available
23         time.sleep(output_interval)
24         hist_values = hist.items()
25         now = time.time()
26         num = len(hist_values)
27         print("Bin value\tTotal")
28         for i in range(0,num):
29             print("{0:05x}".format(i), "\t\t", (hist_values[i][1]).value)
30 except KeyboardInterrupt: sys.stdout.close()

```

Listing 2: Code snippet of the ipstats.py tool.

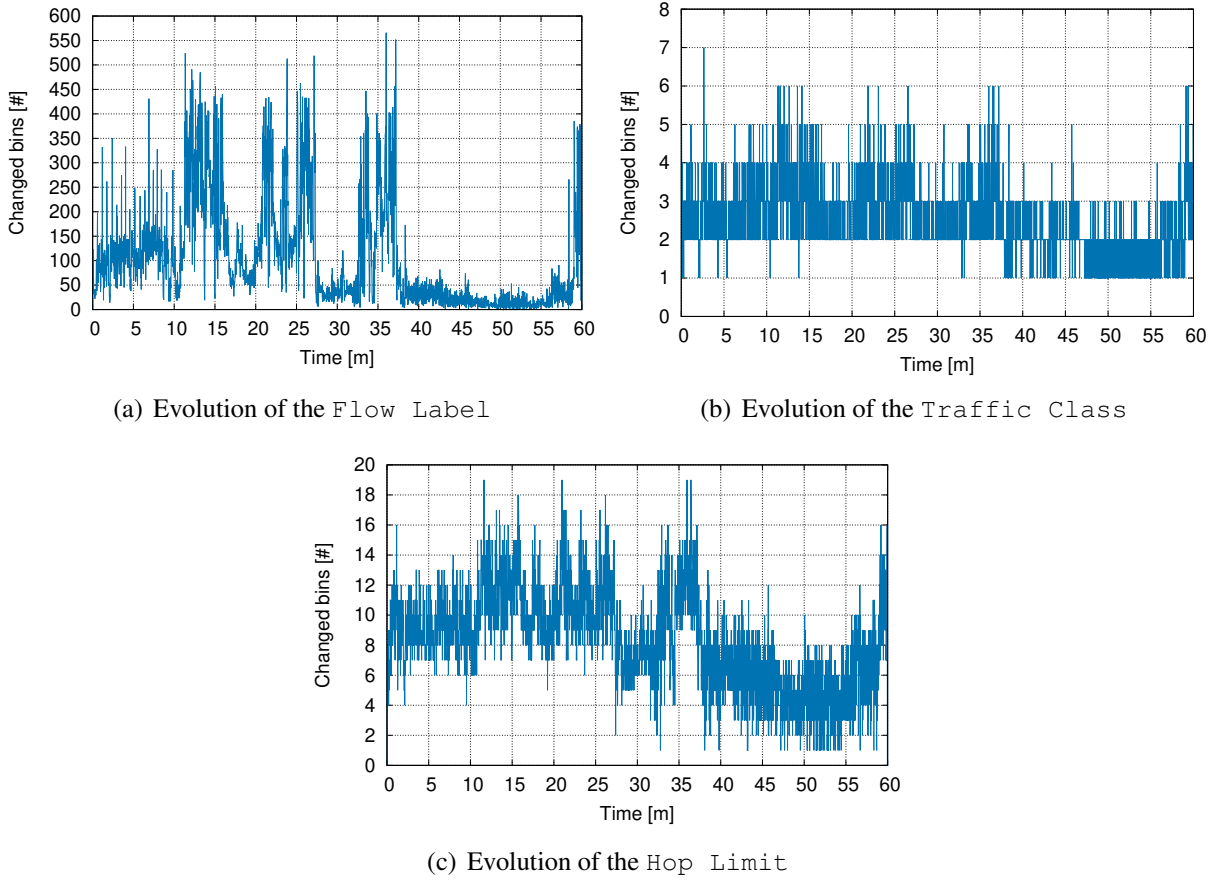


Figure 3.5: Number of changing bins of the observed traffic when gathering data for different fields.

Limit. Specifically, Figure 3.5(b) shows that the number of values observed for the `Traffic Class` is limited. Thus, an attacker wanting to inject data in this field without producing visible alterations should adopt some form of encoding (e.g., mapping 1 and 0 values into a sequence of `Traffic Class` values that are present in the overt traffic), or he/she should keep the data rate as small as possible (see [MPC19] for a thorough investigation on the embedding capacity of real-world IPv6 traffic). Similarly, using the `Hop Limit` as a carrier requires the attacker to modulate values without deviating too much from the observed average value (see Figure 3.5(c)).

Concerning resources used by the `ipstasts.py` tool, we considered performance of both the eBPF and the user-space programs. The eBPF code is composed of about 120 assembly instructions, which are executed only when a packet is processed. As a consequence, providing a simple estimation for the CPU usage is not straightforward. Instead, we can provide an estimation of the additional latency introduced when processing a packet. It turned out that such a quantity is minimal, 104.48 ns, on average. The maximum and minimum execution times observed were

Interval [s]	CPU Usage [%]			Memory Usage [Kbyte]		
	1	10	30	1	10	30
Flow Label	12	7	3	179,748	176,160	174,472
Traffic Class	3	3	3	164,108	163,752	164,180
Hop Limit	3	3	3	163,680	164,032	163,920

Table 3.1: CPU and memory usage for the user-space program.

19, 715 ns and 49 ns, respectively. To give an idea of the impact of the latency introduced by our eBPF code on a realistic case, we measured the total delay introduced when transferring a file of 1.2 Gbytes. Results indicate a very minor impact, as the additional delay was equal to ~ 7 ms. As regards the memory usage, the stack size is limited to 512 bytes. Instead, the amount of shared memory depends on the number of bins, but it is insensitive to the length of the monitored field, as already discussed in Section 3.1. Accordingly, the maximum memory occupancy occurs for 2^{20} bins, and it is equal to ~ 8 MB, which is limited. For a lower number of bins (i.e., in the range of 4 – 8 entries), the memory consumption is constant and equal to 4 Kbytes. The used memory increases proportionally to the number of bins.

Concerning the user-space program, we measured both the memory and CPU usage. Table 3.1 reports the data obtained via the system tools (`top` and `time`). Similar to the case of eBPF code, the size of the shared memory area impacts on resource consumption, since data is copied in user-space. For the sake of brevity, we only consider a limited number of bins, corresponding to what has been used in Figures 3.5 and 3.6. For the case of CPU usage, the relevant parameter is the sampling interval, i.e., the time frame between consecutive reads from the user-space utility, denoted as “interval” in the table. As it can be seen, the larger number of bins, the higher the utilization or resources. For `Traffic Class` and `Hop Limit`, which only uses 256 bins, there is no meaningful difference in CPU utilization when changing the sample time.

3.3.1 Towards the Detection of Covert Channels

Even if `ipstasts.py` and possible additional tools/features in the `bccstego` framework can be used as general methodologies for investigating different anomalies in network traffic, the prime goal is to use them for the detection of network covert channels. To evaluate this possibility, we conducted an additional round of tests. Specifically, we used the same testbed and traffic conditions of the previous round, but we added two malicious endpoints that hide a secret in the different fields of the IPv6 header. The covert channels are implemented via the `IPv6CC` suite presented in Section 2.2.1. The two endpoints injected data within an SCP file transfer with a rate of 500 kbit/s. Figure 3.6 depicts the number of changing bins when comparing “clean” traffic, i.e., traffic with no covert channels, with traffic containing a covert communication, during 10 minutes of network activity. Specifically, Figure 3.6(a) shows the

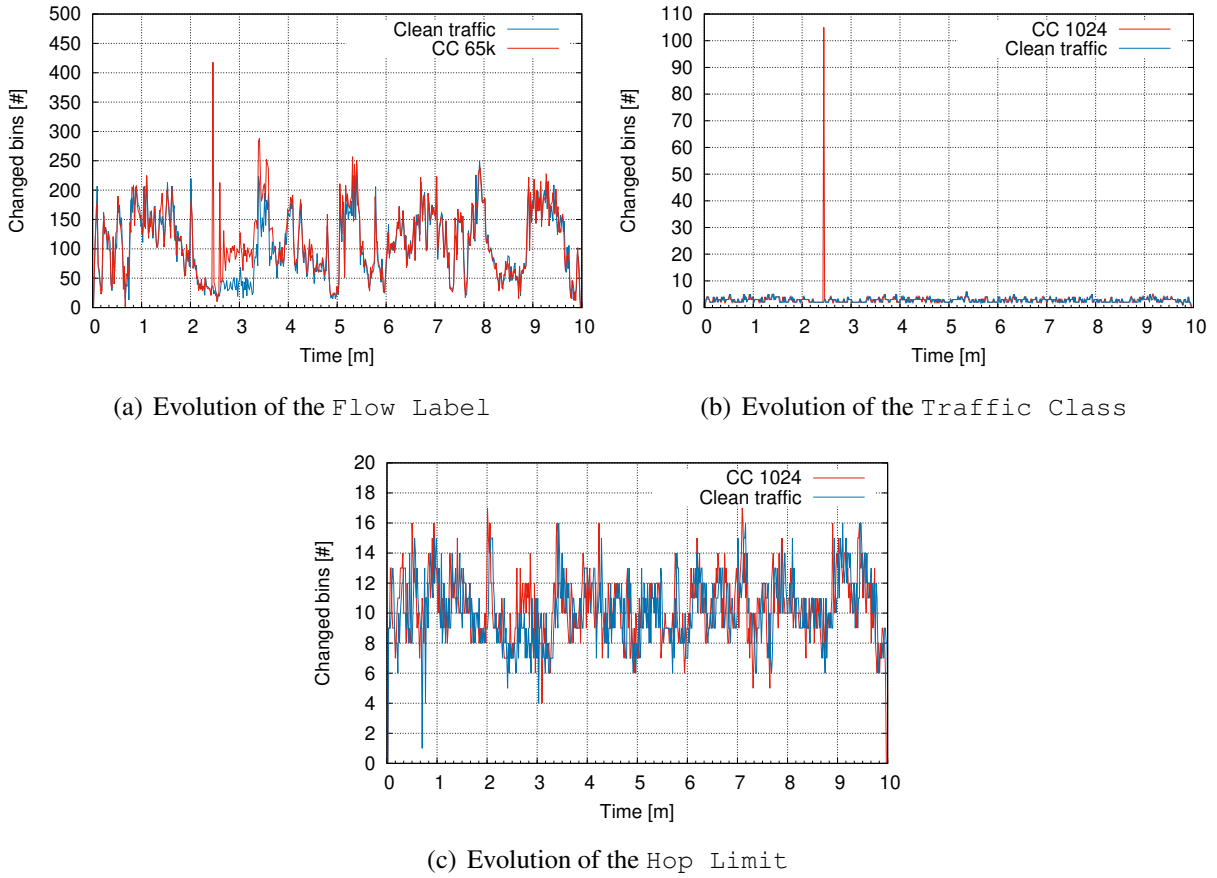


Figure 3.6: Different behaviors of the number of bins changed between clean overt traffic and traffic containing a hidden communication for different fields.

case where the `Flow Label` has been used to covertly transfer a 65 kbyte file (representative of a chunk of sensitive data or the retrieval of additional attack routines). As previously hinted, changes in the number of bins indicate the arrival of new IPv6 conversations. This is due to the “natural” evolution of randomly-generated `Flow Label` values that are expected to fall under different bins during the sample period. This approximation is more precise when the number of new flows is smaller than the number of bins (hence, B and the sampling interval must be chosen accordingly). As a consequence, the anomalous increase of the number of bins used can indicate the presence of a hidden communication.

Different considerations can be done for other fields of the IPv6 header. For the case of a covert channel using the `Traffic Class`, the limited amount of values used in practice makes the detection trivial, as indicated by the “spike” at ~ 3 minutes shown in Figure 3.6(b). To elude the detection, the attacker should be able to use a suitable encoding or to slow the channel down to only few bits per minute. Finally, the “modulating” flavor used to hide a cover channel in

the `Hop Limit` makes the detection more difficult. As depicted in Figure 3.6(c), the specific behavior is less visible since the secret information is not directly injected in the field and the alterations are spread over the various bins in a more regular manner.

3.4 Other Tracing Technologies

High-rate inspection of network packets with software tools has always been challenging, especially since the architecture of general-purpose computers is not designed for this scope. In this vein, several technologies have been proposed to improve the performance of packet processing. They usually leverage hardware acceleration capabilities present in network interface cards and CPUs (e.g., checksum offloading or CPU pinning), reducing the implicit overhead in system calls (due to the context switching between kernel- and user-space) and packet copies in memory.

The most effective approach is kernel bypass, which replaces the networking stack with an alternative path. Notable examples are `PF_RING` [Der04] and `Netmap` [Riz12], which map Network Card Interface (NIC) memory and registers to user-space to avoid the need of copying packets. To support such a paradigm, applications must re-implement common networking utilities and protocols. To partially overcome this issue, `DPDK`⁶ provides a large set of libraries for common packet-intensive tasks, whereas `OpenOnload` [PR11] uses a hybrid architecture, which dynamically selects between user-space and kernel mode for any network flow. To further reduce the impact of context switches in the hardware, `Vector Packet Processing` [BLM⁺18] exploits the persistence of common information in the processor caches. In this case, it collects and processes large batches of packets (called vectors). From the viewpoint of supporting security appliances and operations, [Der04, PR11, Riz12, BLM⁺18] have been largely used in middleboxes for intrusion detection, firewalling, flow monitoring, and mitigation of DoS attacks.

Even if kernel bypass is a very effective mechanism for simple networking processes (e.g., packet forwarding and routing), the implementation of generic communication channels is not trivial. Hence, many frameworks make use of kernel bypass technology to create common processing patterns: `Click` [KMC⁺00], `BESS` [HJP⁺15], `Snabb` [PNFR15], just to mention a few. In this case, the adoption of fixed processing patterns may jeopardize the implementation of tailored monitoring and detection features.

Even if eBPF cannot reach the performance of kernel bypass mechanisms, it represents a very flexible and efficient solution for making custom operations on the traffic processed by the host. eBPF has been mainly conceived for investigating the kernel performance, while security-related tools are largely missing. Interestingly, many eBPF-based tools are being integrated in the `Cilium` platform [MG19b]. The flexibility of eBPF and the possibility to precisely monitor and trace the kernel make this framework a really promising technology for discovering and investigating

⁶https://doc.dpdk.org/guides/prog_guide [Last Accessed, October 2022].

a large variety of stegomalware [CCRZ20, CMR⁺21] within single hosts or complex digital infrastructures [RCR21].

3.5 Conclusions and Future Works

In this chapter we introduced the `bccstego` framework. Differently from other approaches, it leverages in-kernel code augmentation to reduce the development effort without impacting the packet processing performance provided by Linux. The proposed solution should not be conceived as a tool working “out of the box” but it has to be considered as the part of a larger framework that aggregates information from complementary sources. Future works aim at extending the framework to consider other protocols and network features.

In the next chapters, we will discuss how `bccstego` can be used to reveal network covert channels, also focusing on its impact in terms of network performance.

Chapter 4

Detection of Covert Channels via Kernel-level Tracing

As discussed, identifying anomalies and spotting suspicious activities require to have deep visibility over the behavior of software processes. However, 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 [RR18]. Thus, detecting sophisticated attacks or stegomalware is an emerging challenge that should properly balance the depth of inspection with resource consumption [MC15, CCM⁺18].

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., [AMH16] 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 [YY18] or interconnected with a cyber-physical system [DHX⁺18]. However, as shown in Chapter 1, information-hiding-capable threats pose new challenges, as they exploit bandwidth-scarce channels and their detection depends on the used carrier and steganographic technique.

In this chapter, we show how to take advantage of kernel-level techniques for code augmentation introduced in Chapter 3 and we investigate their usage for detecting stegomalware. First, we consider two processes running on the same host and colluding to “evade” typical security controls, i.e., sandboxing (see Section 1.2.1 for a detailed discussion of this attack template).

Second, we gather low-level measurements that are not available from common tools to detect hidden communication attempts nested within network traffic (see Section 1.2.2 for the various types of network covert channels considered in this Thesis).

Summarizing, the contributions of this chapter are: *i)* the development of eBPF programs to collect data from the Linux kernel in an efficient way; *ii)* the creation of realistic use cases for colluding applications and IPv6-capable covert channels; *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.

The remainder of the chapter is organized as follows. Section 4.1 showcases the use of eBPF to detect local covert communications via the manipulation of permissions of files, while Section 4.2 investigates its ability to spot network covert channels targeting IPv6 traffic. Section 4.3 elaborates on the use of kernel-based measurements both in terms of overheads and perspective integration with other frameworks. Lastly, Section 4.4 concludes the chapter.

4.1 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, Section 4.1.1 discusses an attack based on the `chmod-stego`¹ technique and Section 4.1.2 presents the obtained numerical results.

4.1.1 `chmod`-based Stegomalware and its Detection

To model a malware implementing a colluding applications offensive template, we leveraged a local covert channel using the `chmod-stego` as depicted in Figure 4.1. The `chmod-stego` is a Python application made of two peers, i.e., the covert sender and the covert receiver. To communicate, the sender encodes the secret message by changing the access permissions of various files stored in a directory shared between the endpoints. Moreover, it can also modulate the steganographic bandwidth by imposing a delay between two consecutive invocations of the needed `chmod(s)`. As a first step, the sender saves the initial permissions “state” for all the files (by using the `stat` system call in the OS module). Then, 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

¹<https://github.com/operatorequals/chmod-stego> [Last Accessed, October 2022].

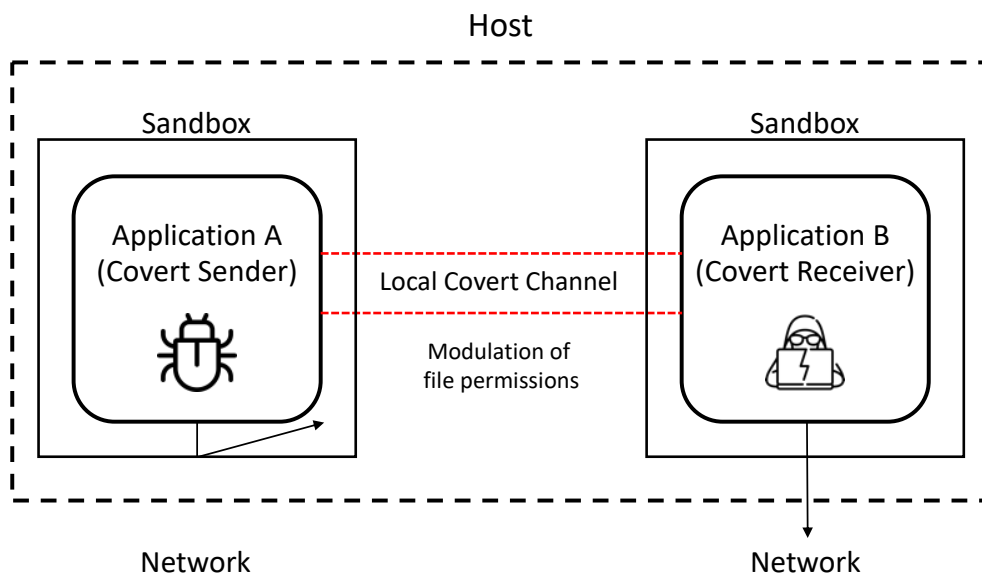


Figure 4.1: Reference scenario for the colluding applications technique.

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, the secret sender restores the file permissions to the original state at the end of the transmission.

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`² utility from BCC, which periodically reports the number of times a given kernel function is invoked.

4.1.2 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

²<https://github.com/iovisor/bcc/blob/master/tools/trace.py> [Last Accessed, October 2022].

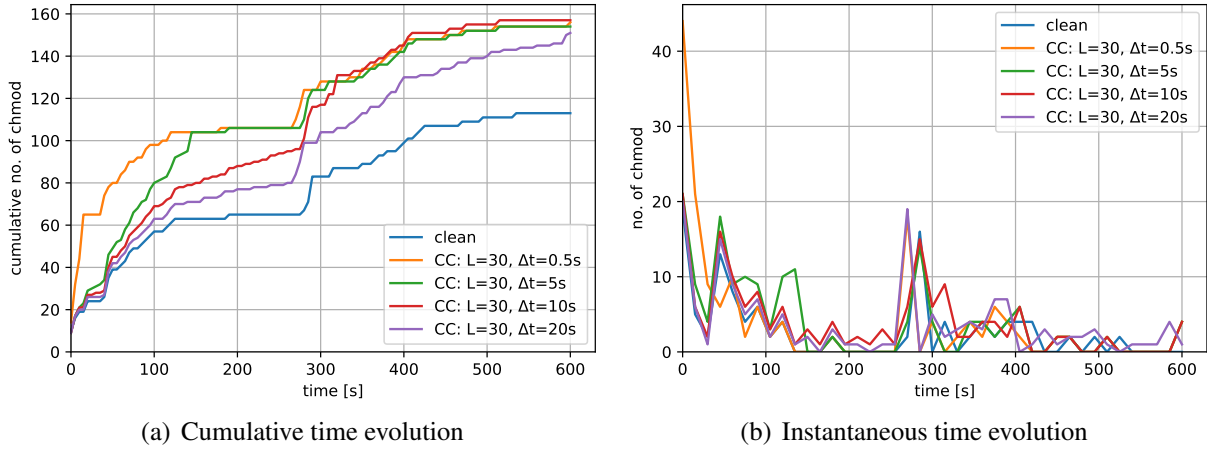


Figure 4.2: Detected invocation of the `__x64_sys_chmod` kernel function with $L = 30$ and $\Delta t = 0.5, 5, 10, 20$ s.

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 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 in the following 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 minutes 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 APTs or malicious applications wanting to exfiltrate as quick as possible sensitive information) while guaranteeing the adequate statistical relevance. Figure 4.2 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 Figure 4.2(a)). We note that the higher the steganographic bandwidth (i.e., Δt decreases), the higher is the load of `__x64_sys_chmod` ker-

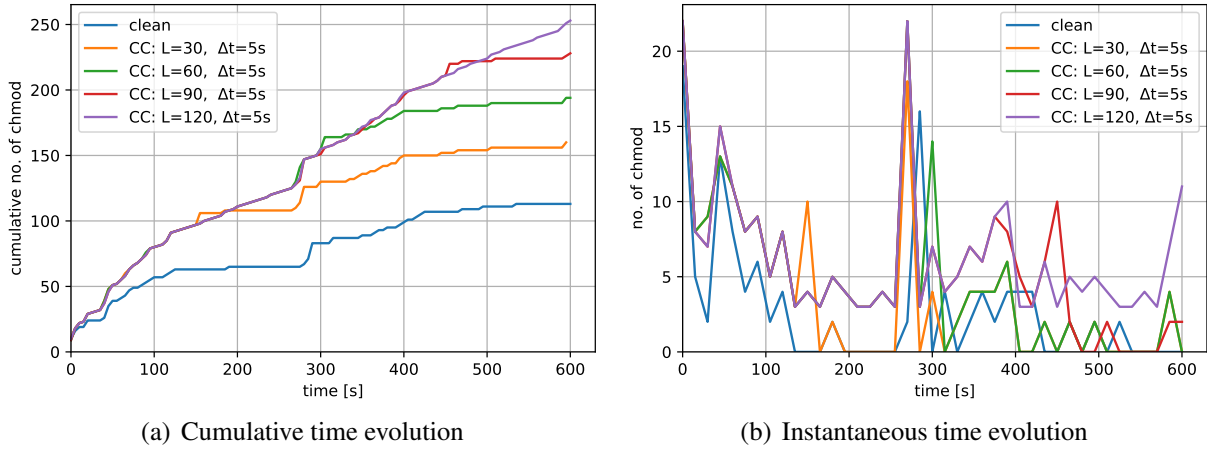


Figure 4.3: Detected invocation of the `__x64_sys_chmod` kernel function with $\Delta t = 5$ s and $L = 30, 60, 90, 120$.

nel functions at the begin. In fact, higher transmission rates reduce the time needed to transmit the secret message. This can be viewed in Figure 4.2(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 Figure 4.3. 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, channels with a higher steganographic bandwidth and longer messages are usually easier to detect. Indeed, they imply either sudden peaks or larger volumes of `__x64_sys_chmod` kernel functions. Clearly, online 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.

4.2 Data Gathering for Network 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. Section 4.2.1 discusses the implementation of the attack and Section 4.2.2 presents the obtained numerical results.

4.2.1 IPv6 Covert Channels and Their Detection

To implement a realistic IPv6 network covert channel, we used `IPv6CC` (see Section 2.2.1). We directly stored 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 [MPC19, MSWC19]. 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 [BKP20]. The `bccstego` framework presented in Section 3.2 has been used to inspect IPv6 datagrams and gather `Flow Label` values.

4.2.2 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 generated by an SCP file 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 will be done in Chapter 5. 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 performance 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 or a PIN), $L = 4,096$ bit and $L = 65$ kbit (e.g., multiple address book entries or sensitive data in a textual form), and $L = 1,000$ kbit (e.g., a highly-

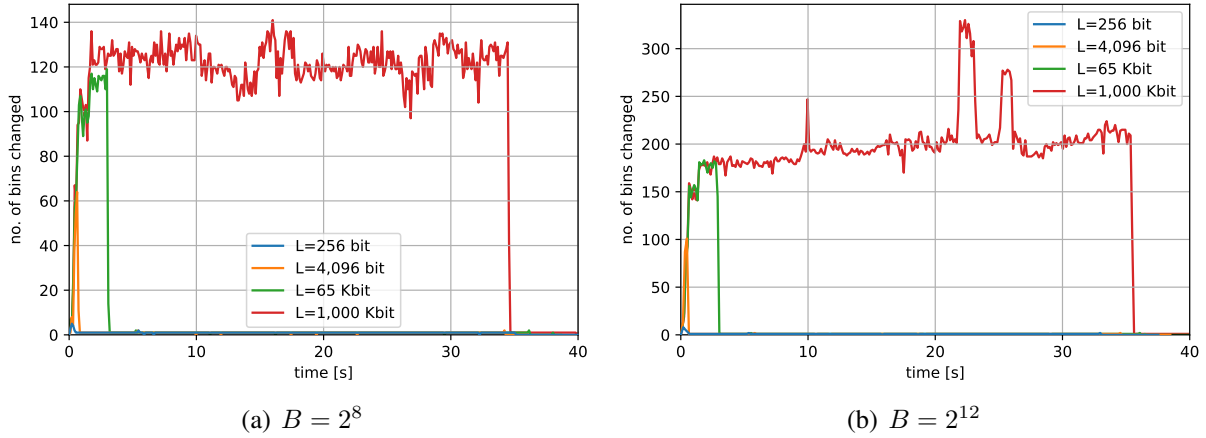


Figure 4.4: Numbers of changing bins for covert messages with various lengths L and number of bins B , with $\Delta s = 100$ ms, $k = 0$.

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 $1,000$ packets, with $k = 0$ denoting a flow without interleaving. The third round of tests addressed the performance 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 . At the time of writing this part of the research, we were not able to map the `Flow Label` with a resolution greater than 2^{16} bins for security requirements forced by eBPF. Since this constraint has been relaxed in later updates, Chapter 5 will provide additional results when considering larger resolutions.

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 APTs). 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.

Figure 4.4 shows the number of bins that change between two consecutive sampling intervals. In the case of a legitimate behavior, we expect 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

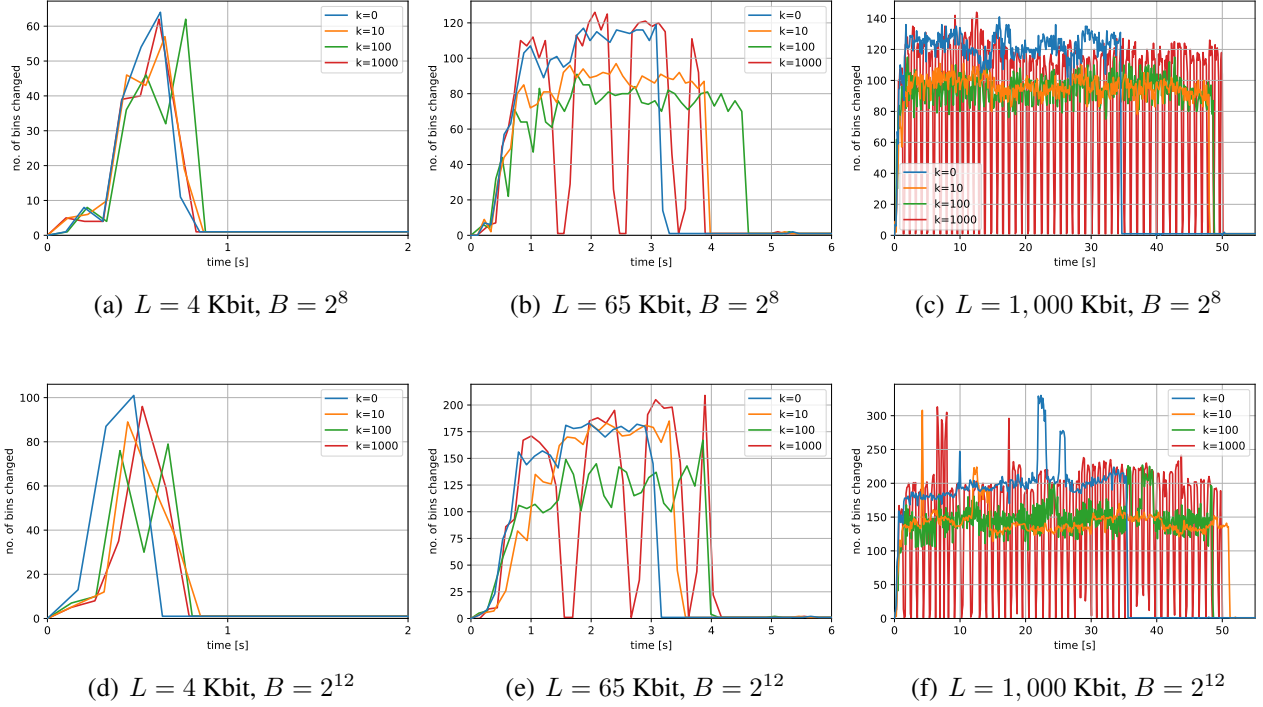


Figure 4.5: Impact of the message length L and the interleaving scheme k on the number of changing bins with various B and $\Delta s = 100 \text{ ms}$.

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 $L = 256 \text{ bit}$) are very hard to be detected especially with additional background traffic.

A similar investigation is presented in Figure 4.5. In this case, we consider a steganalware 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

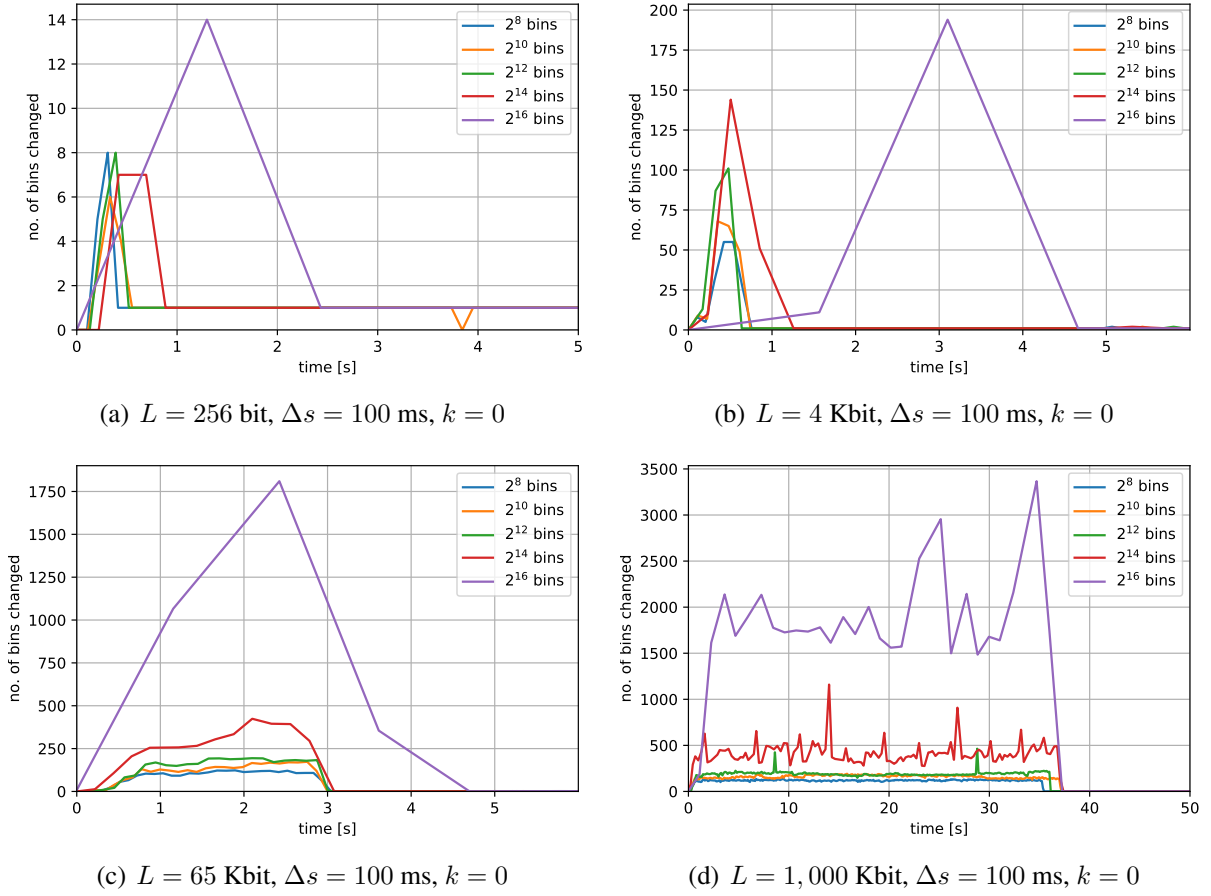


Figure 4.6: Impact of the granularity of B on the number of changing bins for different message lengths L with $\Delta s = 100$ ms and $k = 0$ (no interleaving).

multiple consecutive samples should be considered, or a more fine sampling Δs should be used.

Lastly, Figure 4.6 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., [MPC19, BKP20]) 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,

B	0	2^8	2^{10}	2^{12}	2^{14}	2^{16}
Max entries	–	256	1,024	4,096	16,384	65,536
Mem size [Kbyte]	–	4.10	12.30	36.96	135.17	528.39
N. of active slabs	5,777	5,899	5,928	5,938	5,990	6,004
Active slab size [Mbyte]	48.49	49.85	49.94	50.14	50.47	50.53
Total slab size [Mbyte]	50.01	51.35	51.47	51.67	52.01	52.06
Free memory [Mbyte]	1,715.27	1,620.13	1,620.81	1,619.62	1,619.00	1,618.26
Mapped memory [Mbyte]	81,644.00	81,728.00	81,720.00	81,740.00	81,772.00	81,808.00

Table 4.1: Memory usage with different number of bins B .

especially for shorter message lengths. Delays and performance issue will be further discussed in Section 4.3.

4.3 Deployability and Additional Results

Kernel-level tracing can be considered an effective enabler for detecting steganographic attacks that target 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 the SIMARGL project) and they can be dynamically orchestrated at run-time to support multiple detection techniques (as proposed in project ASTRID³ - AddreSsing ThReats for virtualized services). We then consider additional aspects related to efficiency and resource consumption, outline possible usage for advanced detection techniques, and identify some open issues. We point out that, Chapter 6 will present a thorough investigation of the footprint of our approach, especially when compared to de-facto standard solutions.

4.3.1 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.

Table 4.1 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 (`bpf_tool`, in our case). The middle

³www.astrid-project.eu [Last Accessed, October 2022].

N. of bins	2^8	2^{10}	2^{12}	2^{14}	2^{16}
$\Delta s'$ [ms]	109.40	120.40	233.48	351.13	1,391.91

Table 4.2: Real sampling time experienced by the user-space program.

section shows selected relevant measures of the used cache (number of active slabs⁴ 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.

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 4.2 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.

4.3.2 Envisioned Applications

Information gathered by kernel-level inspection and tracing 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 AI techniques [BG15]. In fact, AI-capable frameworks require heterogeneous and rich information to provide satisfactory statistical performance 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 model is probably the most challenging aspect for application of machine learning, that usually requires 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.

⁴Slabs 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, or symmetric multi-processors.

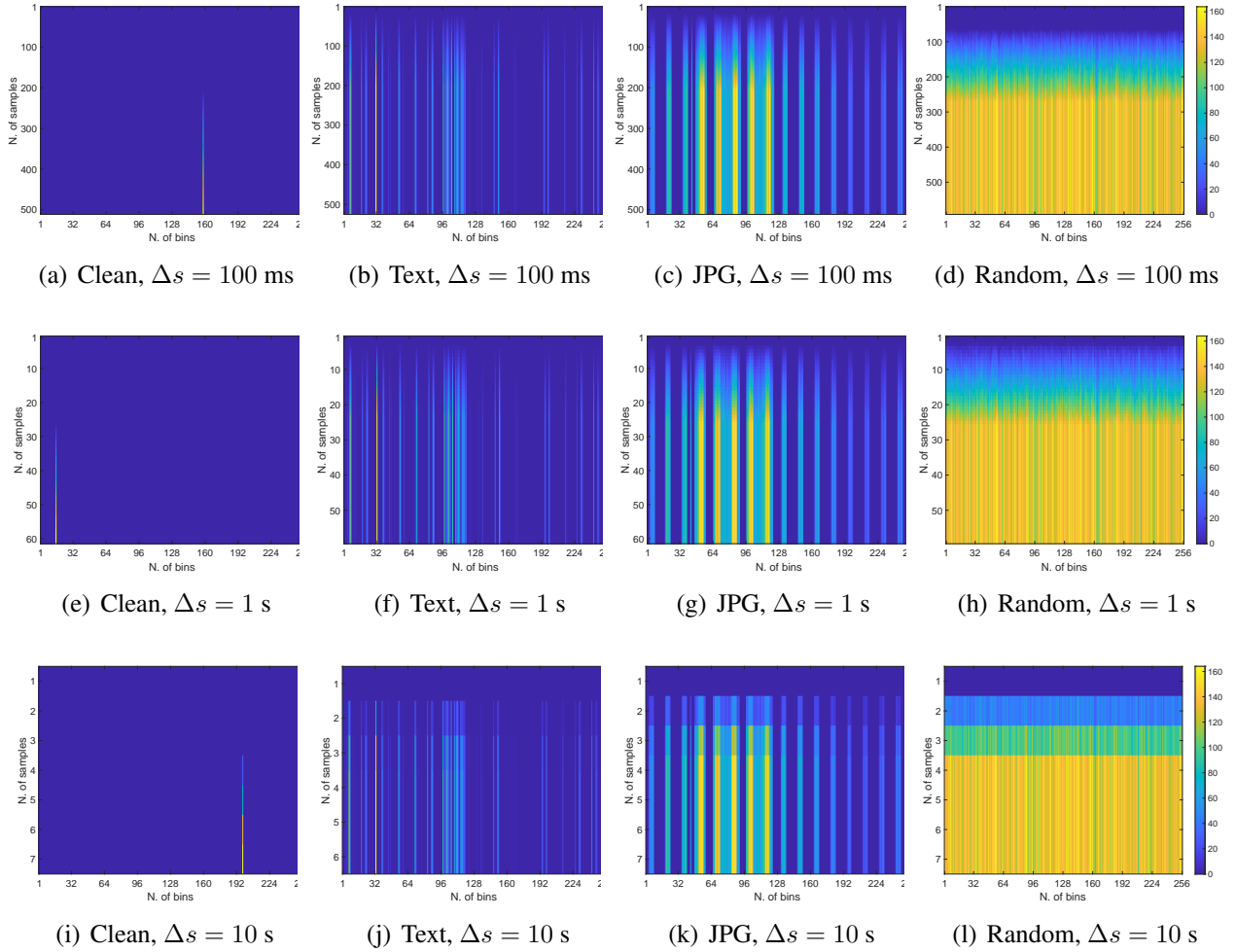


Figure 4.7: Heatmaps for various covert transmissions, with $B = 2^8$, and different granularities Δs used for populating bins via eBPF.

To understand whether our set of measurements could be used to feed a machine learning 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 4.2, 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. Figure 4.7 showcases selected results with different sampling times of the user-space program. Figures 4.7(a), 4.7(e), and 4.7(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 Figures 4.7(b) - 4.7(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 IDS.

As a final remark, we argue that the additional code injected in the kernel should not introduce bottlenecks, mainly to avoid degradation in the performance 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: hence, we now only 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, Figures 4.7(j) - 4.7(l) computed from data gathered by the eBPF program with $\Delta s = 10$ s still offer informative insights.

4.3.3 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 part of the research, which is rather focused on understanding the “brute force” of eBPF to support the task of mitigating emerging threats endowed with steganographic features. We then 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 labelling, the lack of scalability and the composite nature of datasets are prime obstacles for successfully mixing AI and cybersecurity [BBCC19, MG19a]. 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., [PSW16] 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 colluding applications schemes) and IPv6 covert channels collected in-the-wild are very limited as widely discussed in Chapter 2. Thus, as it often happens in the literature, our investigation is based on non-weaponized colluding applications and covert channel attacks [ZAB07, MC14, MC15, WZFH15, CCM⁺18, MPC19], but this limitation will be relaxed in the next chapters.

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 detecting 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). 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. Yet, as it will be detailed later, efforts to port eBPF outside the Linux world are ongoing, thus making this constraint less tight.

4.4 Conclusions and Future Works

In this chapter, we showcased how eBPF 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 evaluated 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 AI-capable techniques. Future works aim 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, future developments deal also with finding threat-independent signatures such as energy consumption, RAM usage patterns, and the time statistics of running processes. Ongoing research aims at extending and generalizing the eBPF approaches to detect a wider array of threats such as cryptolockers and APTs.

In the next chapter, we will further elaborate on the detection of network covert channels via code layering approaches.

Chapter 5

Detection of Network Covert Channels via Code Layering

As discussed in Chapter 1, the usage of virtualization can ease data gathering operations in networking scenarios and face the heterogeneity of modern deployments and threats. To this aim, we now investigate the detection of network covert channels specializing what presented in Chapter 4 by the mean of virtualization concepts.

Following the ground-breaking innovation wave that has led to the network function virtualization era, the telecommunication industry now requires the agility to rapidly deliver new services and reduce their time-to-market. In this respect, the growing interest in “cloud-native” solutions pushes the evolution from Physical Network Functions to Virtual Network Functions (VNFs) and Container Network Functions (CNFs)¹. This trend has been observed in recent open-source platforms, including CORD, OSM, ONAP and SONATA, not to mention the transition from traditional function-reference points to service-oriented architectures in the control plane of the 5G core [Eur20]. Unfortunately, moving network functions from physical hardware to virtual machines is easier than containerizing the software (e.g., due to the lack of kernel acceleration). Monitoring and inspection for security purposes is more difficult as well, especially for immutable software images that cannot be modified at runtime.

Cloud-native cybersecurity platforms usually provide proactive controls at deployment time on the integrity and safety of the software. Yet, monitoring, inspection, and tracing remain three crucial requirements for telco-grade transition to Platform-as-a-Service (PaaS), especially to detect and mitigate attacks at the network boundary [RCL⁺21]. To this aim, in this chapter we explore the concept of code layering via the eBPF framework to instrument VNF/CNF entities with monitoring and inspection capabilities. As hinted in Chapter 3, the framework is supported by several

¹<https://5g-ppp.eu/wp-content/uploads/2020/02/5G-PPP-SN-WG-5G-and-Cloud-Native.pdf> [Last Accessed: October 2022].

companies such as Google or Facebook and recently it has been also ported on Windows. To meet the typical demand for safe, immutable, and certified software images for telco-grade services, we propose a framework for the management of a broad class of eBPF programs. The approach goes in the direction of agentless systems in order to guarantee the ability to address challenging and emerging security threats. As a paradigmatic example, we further investigate the detection of network covert channels. Since modern IDSes have major drawbacks when handling IPv6 traffic and seldom can detect covert channels out of the box [BPK⁺16, MPC19], assessing such a class of threats is of prime importance. Besides, the widespread adoption of IoT and industrial control systems requires flexible mechanisms against timing channels [KWJ21]. Unfortunately, embedding detection capabilities in resource-constrained devices is extremely challenging, therefore suggesting to address them within VNFs.

As shown in Chapter 1, there are virtually unlimited opportunities to implement covert channels by altering protocol headers or packet timings, thus making their detection an open research question [ZAB07, MC14, WZFH15, CCC⁺20, Cav21]. Specifically, a comprehensive and general solution to address covert channels would require to continuously adapt inspection processes to new protocols and hiding patterns, which is almost unfeasible with static agents in a conventional security framework. The framework proposed in this chapter allows to run a rich set of eBPF programs for gathering condensed statistics on header fields and timings that can be further processed and combined with additional data to spot the presence of covert channels.

In this perspective, the contributions of this part of the research are under the umbrella of the “deployability” of countermeasures against network covert channels in real settings. Specifically, the chapter showcases the development of a scalable and privacy-preserving method to spot covert communications in IPv6 headers, and also presents an extensive vis-à-vis comparison among the proposed code layering approach and de-facto standard tools, i.e., Zeek and libpcap. We also point out that, this part of research has been carried out by using real traffic traces, differently from other works only focusing on theoretical analysis or data obtained in experimental setups (see, e.g., [MPC19] and [LLC05]).

The remainder of the chapter is organized as follows. Section 5.1 showcases the reference architecture, Section 5.2 introduces the threat model and covert channels, while Section 5.3 describes the experimental setup. Section 5.4 discusses the detection of storage covert channels, whereas Section 5.5 considers timing channels. Section 5.6 evaluates the performance of our approach compared to other tools and Section 5.7 reviews the related literature. Lastly, Section 5.8 concludes the chapter.

5.1 Reference Architecture

Code layering is a technique that stratifies the software into a number of functional layers, which can be modified in an independent manner. This allows to perform changes without having to re-

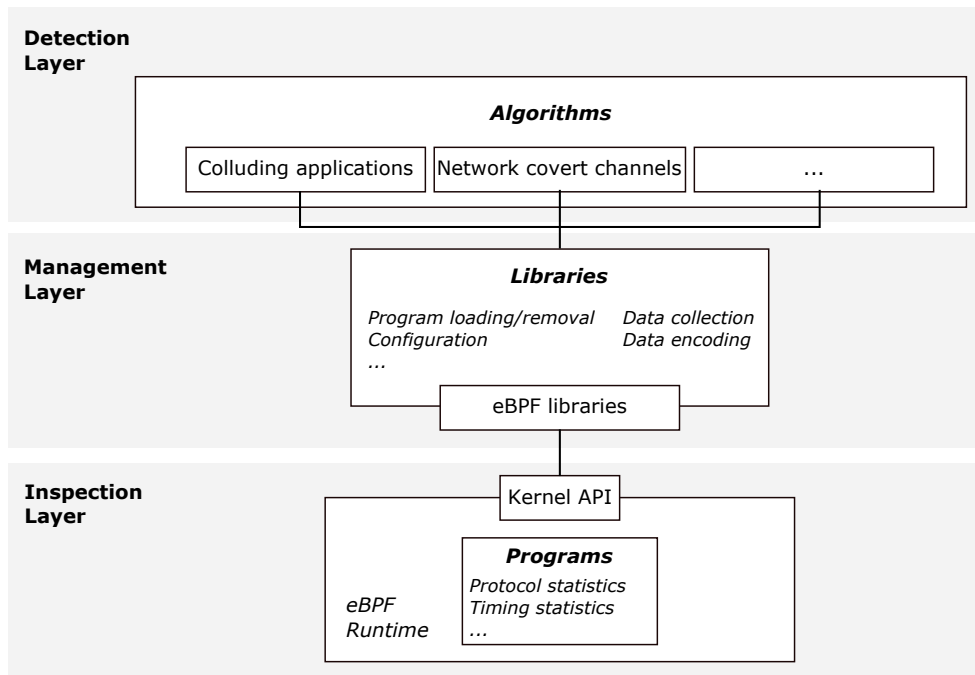


Figure 5.1: Reference layered architecture for the agentless monitoring and detection of various threats.

build and re-deploy the whole software infrastructure. Such a property is highly desirable, since the disruption of a running service is an unacceptable practice for telco-grade operations. To this aim, the approach exploits the eBPF technology to implement low-level inspection and trace operations at run-time both in conventional or PaaS/serverless environments, with negligible impact on service continuity. We point out that, the proposed approach is an updated version of the framework presented in Chapter 3.

Figure 5.1 depicts the reference layered architecture of the proposed framework for monitoring and detection purposes. In particular, the *Inspection Layer* is located in kernel-space and contains various eBPF programs implementing simple monitoring and inspection tasks. It is explicitly designed to run multiple eBPF programs without the need of changing the guest OS. The *Inspection Layer* offers functionalities for parsing protocol headers, recording inter-arrival times, as well as for creating custom statistics. As previously discussed, an eBPF program should be simple and with a reduced footprint since it is triggered at the reception of each packet and it could lead to hangs or scalability issues. Interaction with eBPF programs (including management operations and data exchange) is possible through a specific Kernel API.

The *Management Layer* runs in user-space and represents a sort of middleware entity responsible for loading/unloading eBPF programs and collecting their data. To support the broadest range of inspection and monitoring tasks without having to perform changes, it should be loosely-

coupled with the data structures used by eBPF programs to collect and store information. Indeed, the Management Layer is the most critical block for building an agentless system, because it is expected to collect generic data without any *a-priori* knowledge of their structure. For instance, tools using eBPF such as Cilium and Suricata put tight constraints on data structures, hence jeopardizing the possibility to shape the inspection tasks to evolving threats and attacks. Notwithstanding, there are also some examples of monitoring services that allow the collection and creation of custom metrics from generic eBPF programs, see, e.g., the dynamic network monitoring service of Polycube². Such a design choice allows to include this layer in closed-source, verified, and certified software images of VNFs/CNFs or hosting infrastructure without precluding the possibility to collect additional or different measures at run-time. As said, interactions with eBPF programs can be carried out through the Kernel API. However, it is also possible to exploit higher-layer eBPF libraries, which can include bindings for many languages, e.g., C, Python, Go, and Lua.

Finally, the *Detection Layer* entails specific algorithms running in user-space to reveal and mitigate various threats and attacks. Algorithms implemented in this layer are not strictly part of standard security agents, since most security information and event management architectures deploy them in a remote centralized location. The Detection Layer can be used to engineer a wide range of security tasks. As possible examples of services using eBPF, we mention: tracking traffic with a per-flow granularity with a reduced footprint [BFZ21], identification of processes or nodes contacting malicious servers without degrading the performance of the inspected traffic/processes [DSM⁺19], and support of DPI operations [RCL⁺21]. For the case of hidden communications, this layer can be used to detect network covert channels as well as processes or threads locally leaking data [CMR⁺21].

5.2 Threat Model

The threat model considered in this work deals with two endpoints, i.e., the covert sender and the covert receiver, trying to remotely communicate via a covert channel. Figure 5.2 depicts the two major classes of covert channels, i.e., storage and timing, introduced in Section 1.2 and used in this chapter.

Recalling that the use of IPv6 has been partially neglected and it is expected to become a major target for covert communications in the future, we consider the most effective storage network covert channels exploiting IPv6 traffic, especially those targeting the `Traffic Class`, `Flow Label`, and `Hop Limit` [LLC05]. For the case of `Traffic Class` and `Flow Label`, we consider an attacker directly writing data within such fields. Instead, for the case of the `Hop Limit`, the secret is encoded by introducing a pre-shared offset between two consecutive

²<https://polycube-network.readthedocs.io/en/latest/services/pcn-dynmon/dynmon.html> [Last Accessed, October 2022].

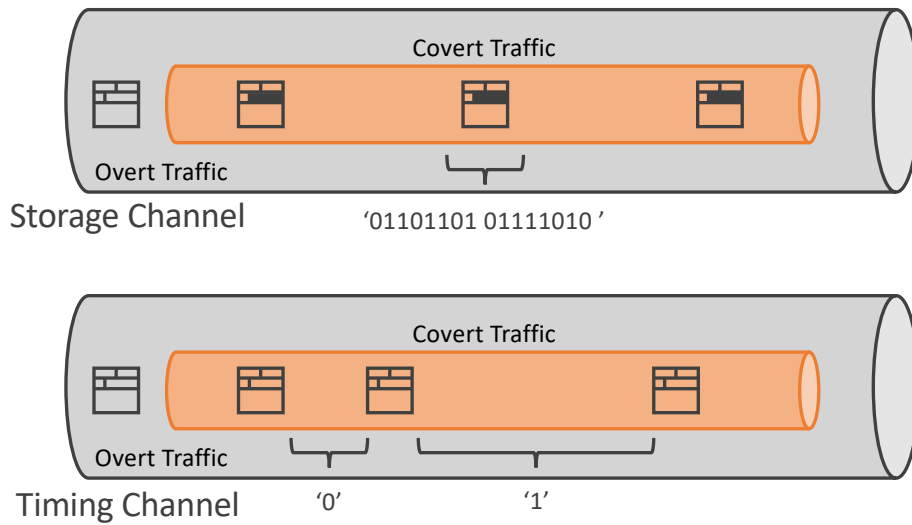


Figure 5.2: Storage and timing network covert channels.

values to encode 1 or 0. In Section 5.4 we will mostly concentrate on revealing channels in the `Flow Label` since it offers more space to embed secrets (i.e., 20 bits compared to the 8 bits of the `Traffic Class` and 1 bit of the `Hop Limit` value modulation). Moreover, `Quality of Service` is often enforced in border routers causing the disruption of the secret hidden in the `Traffic Class` as well as its detection owing to the presence of anomalous values. Similar considerations can be drawn for the case of the `Hop Limit`, especially for modern networks engineered via fewer but longer links, thus reducing the range of values for the field and making the presence of arbitrary values easier to spot. Therefore, the `Traffic Class` and `Hop Limit` will be briefly addressed in Section 5.4.3. Moreover, since we are interested in covert channels with an Internet-wide scope, in Section 5.5 we will address timing channels exploiting the alteration of the time gap between consecutive datagrams. Compared to storage channels, the detection of timing channels is more coherent and investigated [ZAB07, BGN17]. Thus, we will resort to a known approach instead of proposing novel mechanisms.

5.3 Experimental Setup

For the sake of evaluating code layering for the detection of network covert channels, we developed the reference implementation depicted in Figure 5.3, which is composed as follows:

- **Inspection Layer:** it contains a set of eBPF programs that can create statistics on the usage of header fields and packet inter-arrival times for both IPv4/v6 traffic. Programs collecting data to address storage channels are based on `bccstego` (see Chapter 3 for more details

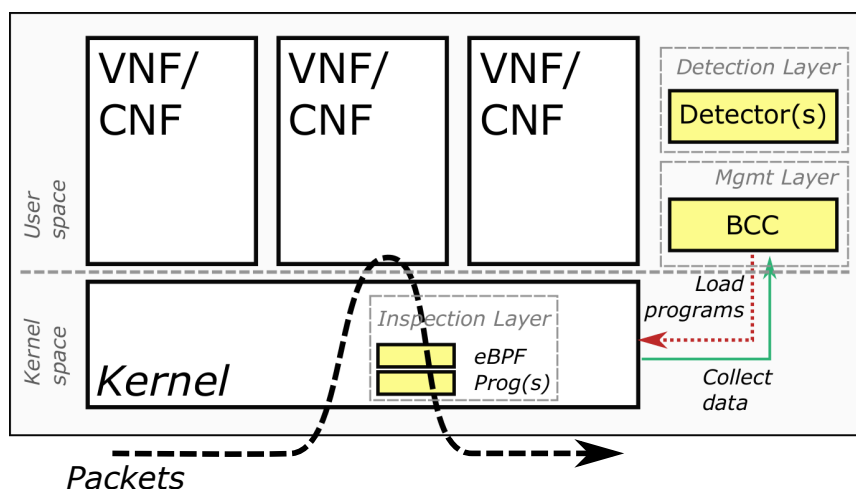


Figure 5.3: Experimental setup leveraging run-time code augmentation for the detection of covert channels.

on the implementation). Instead, to address timing channels, we created a novel eBPF program³ collecting time information and implementing the approach presented in [CBS04] within the kernel;

- **Management Layer:** to load and unload eBPF programs as well as to collect measures, we implemented ad-hoc scripts and userland utilities taking advantage of the BCC library (see Section 3.2 for additional details);
- **Detection Layer:** to spot storage covert channels, we developed a method based on “condensed” statistical indicators, e.g., the frequency/number of values for a specific field provided by the Inspection Layer. Instead, for the case of timing channels, we consider regularity metrics presented in [CBS04]. Details on the detection methodology will be provided in Section 5.4 and Section 5.5, respectively.

Concerning the threat model, we considered malicious endpoints communicating through several types of network covert channels in different scenarios targeting large traffic aggregates. To run tests, the communicating peers have been implemented via two virtual machines running Debian GNU/Linux 10 (kernel 4.20.9), with 1 virtual core and 4 GB of RAM. A third virtual machine with the same characteristics has been deployed to route and inspect traffic as well as to implement the code layering approach depicted in Figure 5.3. In our trials, the various eBPF programs have been attached to the output queue, thus inspecting the egress traffic. However, this does not lead to a loss of generality, since our implementation can also handle programs attached to the input queue without any meaningful difference in terms of performance. For the

³https://github.com/Ocram95/cabuk_eBPF [Last Accessed, October 2022].

sake of comparison, the intermediate node has been also used to run a modified version of Zeek, i.e., `zeek-stego`⁴, and a pure user-space tool for gathering data with `libpcap`. To run the virtual machines, a host with a 3.60 GHz Intel i9-9900KF CPU, 32 GB of RAM and Ubuntu 20.4 (Linux kernel 5.8.0) has been used. In all trials, to quantify the footprints in terms of CPU and memory, we used `pidstat`, which is part of the `sysstat` collection⁵. Apart eBPF programs written in ANSI C, we used Python to implement loading functionalities, the various user-space daemons as well as supporting tools for gathering and analyzing obtained data.

To conduct tests in realistic network conditions, we used traffic collected on an OC192 link in different conditions/periods made available by CAIDA⁶. Without loss of generality and to prevent burdening our trials, we removed packets with a `Flow Label` value equal to 0, ICMPv6 traffic, and single-datagram UDP conversations. In our experiments, we used the slice captured on March 15, 2018 from 14:00 to 15:00 CET between Sao Paulo and New York. After processing, we obtained a 30-minute long dataset composed of $\sim 15,000$ TCP and UDP conversations. To implement storage covert channels, we directly injected via `pcapStego` (see Section 2.1) various secret messages in the dumps provided by CAIDA. Instead, for the case of timing channels, we used `iPerf3` to generate ad-hoc flows and `pcapStego` has been used again to modulate inter-packet times and encode the secret information. Traffic generated via `iPerf3` has been also used to compare the performance of the proposed agentless approach against Zeek and `libpcap`.

As it will be detailed later, the detection of storage covert channels can also take advantage of other network monitoring tools. To this aim, in our trials we adopted `nProbe Enterprise M v. 9.5.210715`⁷ to inspect the traffic in real-time and compute the number of active IPv6 flows. According to preliminary tests, the number of active flows reported by `nProbe` is insensitive to the presence of IPv6 covert channels. This further supports the need of teaming up with a specific solution when such channels have to be detected.

5.4 Detection of Storage Covert Channels

This section showcases the detection of storage covert channels targeting IPv6 conversations. As a paradigmatic example, we will discuss the case of the `Flow Label`, since it requires to handle a 20-bit space leading to a significantly higher steganographic bandwidth compared to other fields. Thus, for the `Hop Limit` and the `Traffic Class` we limit to a simpler analysis. We point out that, the proposed mechanisms could be further extended to tackle channels targeting other fields/protocols.

⁴<https://github.com/mattereppe/zeek-stego> [Last Accessed, October 2022].

⁵<http://sebastien.godard.pagesperso-orange.fr/index.html> [Last Accessed, October 2022].

⁶The CAIDA Anonymized Internet Traces Dataset (April 2008 - January 2019) - Available online: <https://www.caida.org/data/monitors/passive-equinix-nyc.xml> [Last Accessed, October 2022].

⁷<https://www.ntop.org/products/netflow/nprobe/> [Last Accessed, October 2022].

5.4.1 Detection of Channels Targeting the Flow Label

The detection of storage channels targeting the `Flow Label` is based on the coarse-grained estimation of the number of IPv6 conversations. Since each IPv6 conversation is identified via a fixed, unique `Flow Label` value generated according to a uniform distribution [ACJR11], this can provide a rough estimation of the number of flows. The resulting metric can be then compared against measurements collected by network monitoring tools or used to “reinforce” indicators provided by standard firewalls or IDSes. Without loss of generality, we assume to have periodical measurements on the number of active IPv6 flows in the network, denoted as F , which is commonly provided by tools for network monitoring.

To compute such an estimation, we used the bin-based approach described in Chapter 3. Recalling, the kernel has been extended to count the occurrence of `Flow Label` values by setting a hook point in the `tc` queue management. To guarantee privacy and scalability requirements as well as to prevent performance degradation for large traffic volumes, the 20-bit space of possible values is mapped into a bin-based data structure composed of B equally-capable bins. Accordingly, each bin has a size of $2^{20}/B$ values. Data is then periodically collected by a user-space utility every Δt seconds and the bin-based structure is periodically emptied to avoid saturation: this is ruled via a time window with a duration denoted with T seconds. Parameters Δt and T allow to adjust the proposed approach to “follow” the dynamic of birth/death of covert communications and match measurements/feedback information provided by external tools with different timings, respectively. Therefore, the number of “dirty” bins, i.e., bins with a non-zero value, provides an estimate of the number of IPv6 conversations, denoted in the following with N . This is only an approximation: if different `Flow Label` values share the same bin, this will cause a collision. Greater values of B reduce such a probability and improve the precision, but at the price of a higher memory burden. As an example, let us consider the case of $B = 2^{12}$ bins with a size of 2^8 values. If a packet with a `Flow Label` value equal to 337 (i.e., `0x00151`) is observed, the second bin is flagged since it is the one containing values in the `256 – 511` range (indexed by the `0x001` prefix). Accordingly, N is incremented by 1.

The presence of a covert communication could be revealed by comparing N and F , e.g., to understand if the relation $N > F$ holds. However this could be inaccurate, especially due to the saturation of a bin and the coalescing of entries caused by a limited value of B . For this reason, we introduced a scale factor denoted with α to balance the flow/bin proportion. The resulting detection relationship is then $\alpha N > F$. Unfortunately, using only a threshold could lead to an unstable behavior, that is, the detector over/under reacts when in the presence of minimal fluctuations in the number of flows. For this reason, we added a hysteresis parameter ξ .

For the sake of illustrating the proposed detection mechanism, Figure 5.4 showcases an example considering the exfiltration of 21.25 kbytes of data. In more detail, Figure 5.4(a) depicts the outcome of the detection for different values of B when $\alpha = 0.9$ and $T = 30$ seconds. As shown, smaller bins (i.e., when B increases) allow to better spot the covert channel but at the

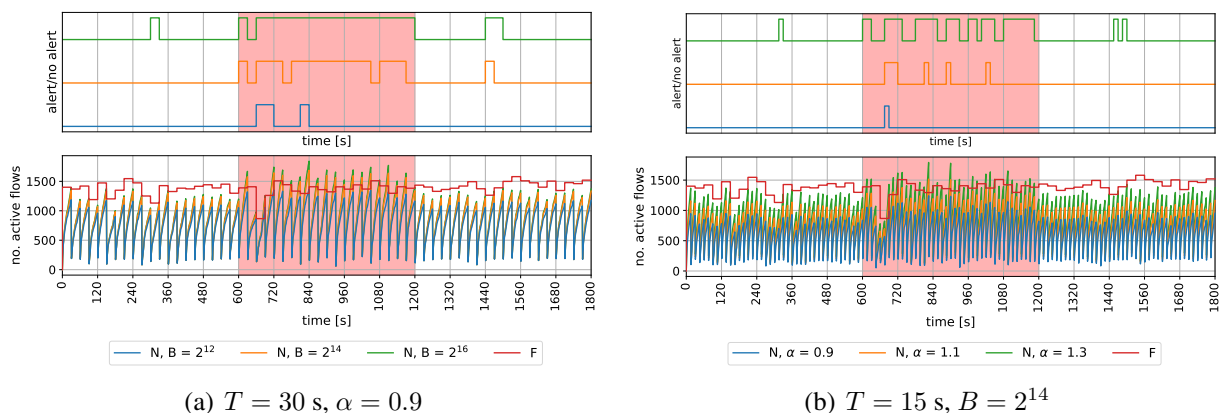


Figure 5.4: Detection of a covert channel targeting the Flow Label when transmitting 21.25 kbytes of secret data. The red area denotes when the covert communication is present.

price of more false positives. On the contrary, coarse-grained bins (e.g., for $B = 2^{12}$) tend to underestimate the presence of an hidden communication. A possible workaround could exploit a tradeoff between the number of bins and the “frequency” of measures. Figure 5.4(b) reports the results for $T = 15$ seconds. As shown, smaller timeframes cause a more frequent “reset” of the bin-based scheme leading to an underestimation of the number of active IPv6 conversations. Thus, it is not possible to directly compare N with the measurement F provided by `nProbe`. The parameter α can correct this mismatch by “magnifying” the obtained values but at the price of errors leading to false positives. In general, the “optimal” matching between the observed traffic and the number of bins is critical since it influences both the “stability” and the performance of the detection. Thus, Section 5.4.2 discusses in detail the design of the various parameters.

5.4.2 Sensitivity Analysis

Detecting storage covert channels is subject to many tradeoffs. For the case of the Flow Label, there is the need of balancing the granularity of the gathering phase (i.e., Δt , T and B), the quality of the estimation (i.e., N and α), as well as the resources required to run additional logic. Therefore, this round of tests aims at performing a sensitivity analysis of the framework.

For the sake of considering a wide-range of use cases, we designed three different attack scenarios. Specifically, Scenario 1 considers an exfiltration attempt modeled via the transmission of a file requiring to target 8,500 IPv6 packets (i.e., 21.25 kbytes). The used overt IPv6 conversation had an average bitrate of 12 kbit/s leading to an exfiltration time of ~ 10 minutes. Scenario 2 models different channels alternating in time, as it happens in the case of the orchestration of a botnet [MC15]. To this aim, we used three different covert channels activating in a timeframe of 15 minutes to exchange data requiring 2,500, 6,000, and 7,500 IPv6 packets (i.e., 6.25,

B	Δt [s]	CPU Usage [%]					Memory Usage [Mbyte]				
		1	5	15	30	60	1	5	15	30	60
2^8		0.13	0.02	0.01	0.00	0.00	181	180	180	180	181
2^{12}		1.84	0.40	0.13	0.01	0.03	184	185	183	183	183
2^{16}		20.80	6.31	2.25	1.08	0.48	237	227	225	223	221
2^{17}		29.81	11.26	4.36	2.16	0.99	282	272	268	264	260
2^{18}		36.97	18.14	7.86	4.21	1.88	379	370	360	350	336
2^{19}		42.73	26.55	13.30	7.22	3.68	584	569	544	522	494
2^{20}		45.81	34.31	21.88	12.41	7.61	1,012	970	896	880	818

Table 5.1: Combined CPU and memory usage for different values of B and Δt .

15, 18.75 kbytes, respectively) within overt flows of 12 kbit/s, 1,600 kbit/s, and 100 kbit/s, respectively. Lastly, Scenario 3 considers an APT targeting a datacenter or a subnetwork, thus producing multiple covert channels towards a C&C server. In this case, we used 10 concurrent covert communications targeting each one of 800 IPv6 packets (i.e., 2 kbytes). After 10 minutes the number of connections is halved, for instance, due to reboots or crashes/shutdowns of compromised nodes.

As a first step, we evaluated the impact of the number of bins B and the sampling time Δt ruling the kernel-to-user-space copy of collected values to elaborate on constraints of the granularity of the detection process. To this aim, we replayed the considered traffic trace towards the node running the eBPF framework. The related CPU and memory usage have been collected with a granularity of 10 samples per minute and average values have been computed. Table 5.1 shows the obtained results. To avoid burdening the table, we report values for $B = 2^8$ (as they represent the case of measuring the `Hop Limit` and `Traffic Class`), $B = 2^{12}$ for an intermediate reference, and for $B > 2^{16}$. As shown, the footprint of the user-space program collecting results increases with the “precision” of the data gathering (i.e., B and Δt). Despite the absence of configurations leading to an unbounded utilization of resources, a major bottleneck is caused by the operations needed to copy data from the kernel-space to userland. This is especially true for $B = 2^{20}$: in fact the copy requires ~ 14 seconds, thus causing a “misalignment” from real `Flow Label` values and those collected in the meantime. Indeed, also the granularity of Δt is subject to careful design choices. Even if a precise tracking of the abused flow is desirable, this should be impeded by difficulties in gathering data in a fine-grained manner. For instance, a typical timeframe for computing analytics of large-scale links/networks is in the range of 30 – 90s, thus relaxing tight constraints on Δt (see, e.g., [ZNA12] for timing constraints for scalable classification). Therefore, for the sake of brevity, in the rest of the chapter we will limit our analysis to $\Delta t = T = 30$ s.

Concerning the possible tradeoff among B and the ability of spotting hidden communications within the bulk of traffic, Figure 5.5 provides a comprehensive overview for the impact of B on the accuracy. In general, as shown in Figure 5.5(a), best results are achieved for Scenario

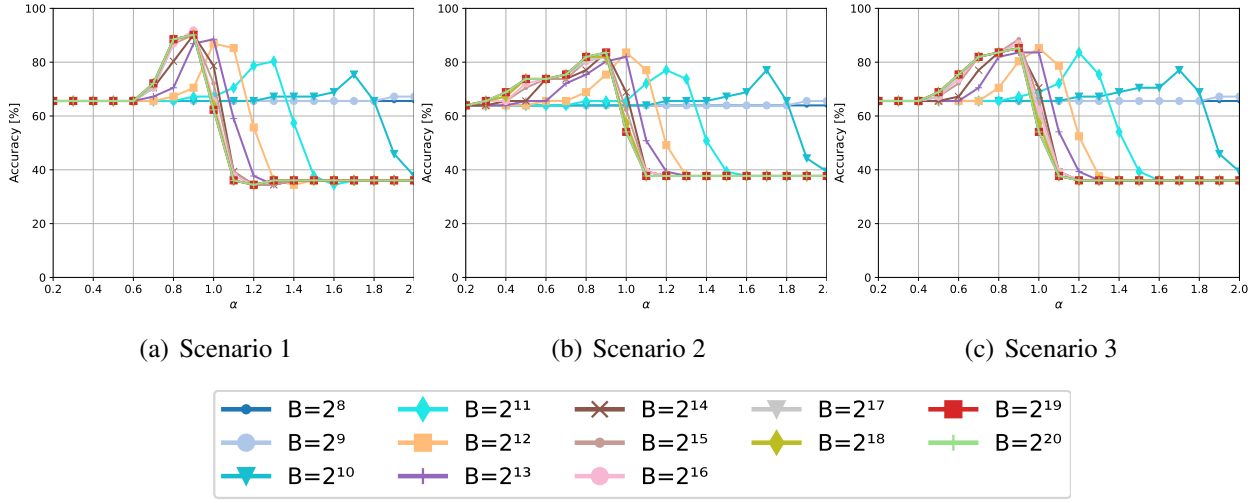


Figure 5.5: Accuracy of the bin-based detection mechanism with different values of B for all the considered scenarios.

1 mainly owing to the presence of a unique covert communication leading to a non-negligible volume of artificial `Flow Label` values. Instead, when in the presence of hidden transfers characterized by ON/OFF or “fading” behaviors, the accuracy decreases accordingly, as reported in Figures 5.5(b) and 5.5(c). Even if higher values of B typically lead to a better accuracy, the proposed approach is able to capture the presence of storage covert channels also with a reduced number of bins (see Figure 5.5 for the case of $B = 2^{14}$). This can be ascribed to the parameter α , which can compensate the under/overestimation of the observed values of the `Flow Label` used to flag the various bins.

The accuracy may not be sufficient to capture the performance of the proposed approach in terms of false/true positive/negative events. Therefore, Table 5.2 reports the true positive rate (TPR) and the true negative rate (TNR) collected when using various values of B for $\alpha^* = 0.9$, i.e., the “optimal” α leading to the best performance. Moreover, as depicted in Figure 5.4, the presence of a threshold-based rule may lead to an unstable behavior of the detection. To mitigate such an issue, we also investigate the impact of ξ implementing a sort of hysteresis for the comparator rule $\alpha N > F$, i.e., the outcome of the detection changes according to $+\xi$ and $-\xi$ switching thresholds à-la Schmitt. Specifically, it is a lower/upper bound considering $F \pm \xi$ with $\xi = 1\%$, 5% , and 10% of its current value. For the sake of brevity, we limit our analysis to $B > 2^{15}$.

As shown, for the case of Scenario 1, the parameter ξ allows to improve the overall detection, especially in terms of TNR. However, greater values of ξ may cause a decay of the accuracy as they make harder to switch the outcome of the detector, thus remaining in a “wrong” state. For the case of Scenario 2, the poor performance of the TPR affects the accuracy, despite the

B	ξ	Scenario 1			Scenario 2			Scenario 3		
		ACC [%]	TPR [%]	TNR [%]	ACC [%]	TPR [%]	TNR [%]	ACC [%]	TPR [%]	TNR [%]
$B = 2^{15}$	0%	90.16	80.95	95.00	83.61	59.09	97.44	88.52	71.43	97.50
$B = 2^{15}$	1%	91.80	80.95	97.50	83.61	59.09	97.44	88.52	71.43	97.50
$B = 2^{15}$	5%	91.80	76.19	100.00	85.25	59.09	100.00	86.89	61.90	100.00
$B = 2^{15}$	10%	88.52	66.67	100.00	80.33	45.45	100.00	83.61	52.38	100.00
$B = 2^{16}$	0%	91.80	90.48	92.50	83.61	63.64	94.87	81.97	63.64	92.31
$B = 2^{16}$	1%	90.16	80.95	95.00	85.25	63.64	97.44	88.52	71.43	97.50
$B = 2^{16}$	5%	90.16	76.19	97.50	83.61	59.09	97.44	85.25	61.90	97.50
$B = 2^{16}$	10%	90.16	71.43	100.00	83.61	54.55	100.00	85.25	57.14	100.00
$B = 2^{17}$	0%	90.16	90.48	90.00	81.97	63.64	92.31	85.25	71.43	92.50
$B = 2^{17}$	1%	93.44	90.48	95.00	85.25	63.64	97.44	88.52	71.43	97.50
$B = 2^{17}$	5%	91.80	80.95	97.50	83.61	59.09	97.44	85.25	61.90	97.50
$B = 2^{17}$	10%	90.16	71.43	100.00	83.61	54.55	100.00	85.25	57.14	100.00
$B = 2^{18}$	0%	90.16	90.48	90.00	81.97	63.64	92.31	85.25	71.43	92.50
$B = 2^{18}$	1%	93.44	90.48	95.00	85.25	63.64	97.44	88.52	71.43	97.50
$B = 2^{18}$	5%	91.80	80.95	97.50	83.61	59.09	97.44	85.25	61.90	97.50
$B = 2^{18}$	10%	90.16	71.43	100.00	83.61	54.55	100.00	85.25	57.14	100.00
$B = 2^{19}$	0%	90.16	90.48	90.00	83.61	68.18	92.31	85.25	71.43	92.50
$B = 2^{19}$	1%	91.80	90.48	92.50	83.61	63.64	94.87	86.89	71.43	95.00
$B = 2^{19}$	5%	91.80	80.95	97.50	83.61	59.09	97.44	85.25	61.90	97.50
$B = 2^{19}$	10%	90.16	71.43	100.00	83.61	54.55	100.00	85.25	57.14	100.00
$B = 2^{20}$	0%	90.16	90.48	90.00	83.61	68.18	92.31	85.25	71.43	92.50
$B = 2^{20}$	1%	91.80	90.48	92.50	83.61	63.64	94.87	86.89	71.43	95.00
$B = 2^{20}$	5%	91.80	80.95	97.50	83.61	59.09	97.44	85.25	61.90	97.50
$B = 2^{20}$	10%	91.80	76.19	100.00	83.61	54.55	100.00	85.25	57.14	100.00

Table 5.2: Impact of B and ξ over the accuracy of the detection (ACC), the true positive rate (TPR), and the true negative rate (TNR).

various B and ξ . This can be ascribed to the presence of a low-throughput channel reducing the effectiveness of the detection mechanism, i.e., the TPR remains in the 45.45 – 68.18 range. A similar behavior characterizes Scenario 3: again, ξ improves the TNR. Yet, the presence of many covert channels halving their activity influences the TPR and the accuracy mainly due to the reduced volume of altered `Flow Label`. Similarly for the case of Scenario 1, higher values of ξ prevent to switch from positive to negative (and viceversa) when the throughput of covert data changes in time.

5.4.3 Channels Targeting Other IPv6 Fields

When handling less capacious fields, the bin-based approach can still be used to implement simpler yet effective counters to reveal hidden communications. Specifically, for the case of the `Traffic Class` and `Hop Limit`, by creating a structure with $B = 2^8$, it is possible to

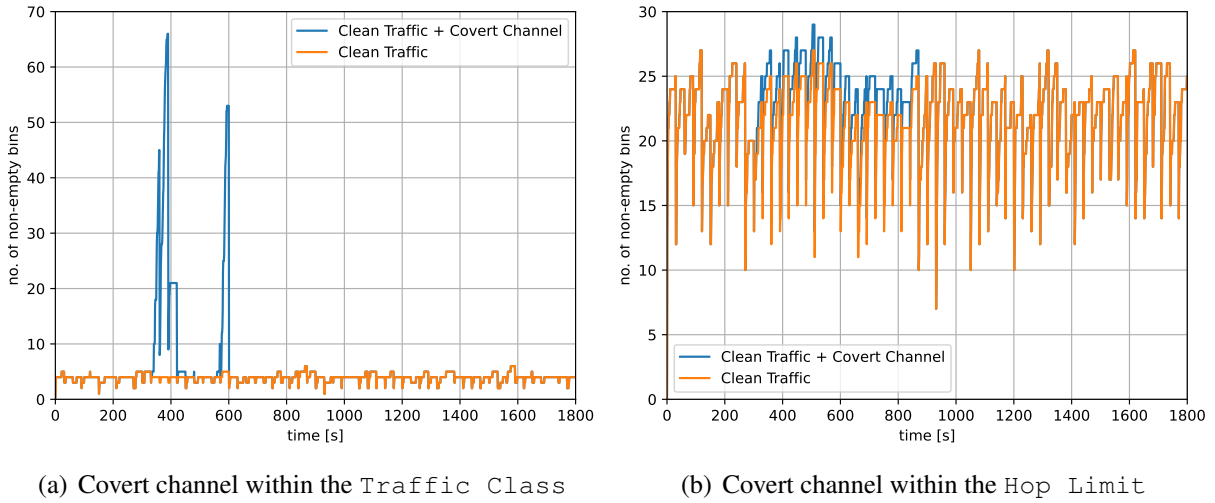


Figure 5.6: Number of “dirty” bins observed when inspecting the Traffic Class and Hop Limit.

perform a one-to-one map between observed values and “dirty” bins. To this aim, we performed an experimental campaign considering the same background traffic used for trials in Section 5.4.2. For the Traffic Class, we considered a threat embedding a malicious command of 512 bytes as used by the Silence Trojan to download and execute a PowerShell script within a flow with the average throughput of 750 bytes/s. Instead, for the Hop Limit, we assumed an attacker wanting to deliver a stage of the Emotet malware of 964 bytes within a flow with the average throughput of 1.5 Mbytes/s. The bits 0 and 1 have been encoded by modulating the Hop Limit with values 10 and 250, respectively. Both malicious payloads are borrowed from the FCL collection.

Figure 5.6 depicts the collected results. Specifically, Figure 5.6(a) deals with a covert channel built by embedding secrets in the Traffic Class. As shown, the number of non-empty bins varies according to the different Traffic Class values within the bulk of traffic. When the targeted IPv6 conversation is active, the number of bins is higher due to the presence of the secret, leading to a sort of “signature”. On the contrary, for the case of Hop Limit (see Figure 5.6(b)), this is less evident, especially due to the used hiding strategy not directly storing the secret. Hence, a more sophisticated approach is needed: this is part of our ongoing research.

5.5 Detection of Timing Covert Channels

This section investigates how the proposed agentless approach can be used to reveal the presence of timing covert channels. Specifically, we are interested in understanding whether the detec-

Bitrate	1 Mbit/s			10 Mbit/s			1 Gbit/s			
	Q	5	10	15	5	10	15	5	10	15
Packet Size [bytes]	$W = 100$									
16	0.07	0.06	0.04	0.03	0.04	0.10	0.09	0.02	0.05	
1,470	0.08	0.04	0.05	0.04	0.05	0.07	0.09	0.04	0.04	
8,192	0.06	0.03	0.06	0.06	0.05	0.04	0.07	0.04	0.05	
65,507	0.07	0.04	0.03	0.05	0.06	0.04	0.05	0.06	0.06	
$W = 250$										
16	0.09	0.06	0.03	0.03	0.04	0.03	0.05	0.06	0.04	
1,470	0.05	0.06	0.04	0.06	0.04	0.04	0.03	0.05	0.03	
8,192	0.05	0.08	0.06	0.04	0.03	0.05	0.04	0.05	0.04	
65,507	0.09	0.03	0.03	0.04	0.03	0.03	0.06	0.02	0.05	

Table 5.3: CPU usage for various traffic rates, W , Q , and different packet size.

tion logic can be partially embedded within eBPF mainly to avoid the need of further moving and processing data in user-space. To this aim, we implemented a de-facto standard algorithm borrowed from the literature. Originally introduced in [CBS04], the idea is to compute a measure of regularity for a set of variances built by grouping packets to make pattern-like behaviors emerge. Patterns can then be used to reveal the presence of hidden information causing “anomalous” inter-packet times. In essence, for each window composed of W packets, the algorithm in [CBS04] calculates the standard deviation σ of the related inter-packet time values. Then, it computes the pairwise differences between σ_i and σ_j , for each pair i, j . The final regularity measure is given by computing the overall standard deviation for all the pairwise differences. Unlike the original version, our implementation checks the regularity metric on-line, i.e., a flow is evaluated on a semi-continuous basis. Unfortunately, due to eBPF limitations in terms of stack size and number of instructions, the regularity measure has been approximated (e.g., the lack of `sqrt()` and other mathematical operations required to implement approximate counterparts). Moreover, to tame memory consumption, the regularity indicator is periodically reported to prevent the need of “unrolling” too many operations. In the following, we define such a “control” parameter as Q , i.e., the number of values for σ considered for each computation of the regularity metric.

5.5.1 Numerical Results

To evaluate our code layering mechanism when used to detect timing channels, we performed trials with hidden conversations nested within the inter-packet time of a $\sim 7,000$ datagrams flow. To make our investigation more comprehensive, we present results obtained with IPv4 traffic: similar results have been obtained with IPv6. To test the covert channel, we sent a malicious

command of 304 bytes used by the GZipDe malware. According to [CBS04], to encode the value 1, we inflated the inter-packet time for two adjacent datagrams by 0.06 seconds, which ensures a character accuracy of 98%. Instead, the 0 value is encoded by maintaining the original timing of the overt traffic. To evaluate the impact of the in-kernel detection algorithm, we measured the CPU and the memory usage, as well as the packet loss and the jitter of the processed traffic. For each trial, we considered flows with various bitrates, i.e., 10 kbit/s, 100 kbit/s, and 1 Mbit/s. We also evaluated the impact of the number of packets W used to compute the standard deviation, which somewhat constitutes the granularity of the approach. Specifically, we considered $W = 100$ and $W = 250$ as suggested in [CBS04]. Results indicate that our agentless approach does not introduce further delay or packet loss on the inspected traffic. Indeed, the low bitrate characterizing timing channels plays a major role, especially it does not require tight computational constraints. This is further supported by CPU and memory consumptions, which are limited to $\sim 0\%$ and ~ 114 Mbytes, respectively, throughout all the trials.

To understand the ability of the eBPF-based code layering approach to handle large traffic volumes, we performed an additional round of tests considering different packet sizes and higher traffic rates. Specifically, we considered datagrams ranging from 16 to 65,507 bytes, in order to consider both worst and best cases in terms of packet processing. Although the fragility of timing channels limits the allotted throughput, the proposed approach could be deployed to monitor Internet-scale deployments (e.g., a datacenter). In this perspective, we also investigated the impact of W and Q to assess the scalability of the proposed agentless implementation. Table 5.3 contains the CPU utilization. In more detail, the code layering mechanism does not account for major overheads. Concerning the overall memory consumption, it is always bounded to ~ 110 Mbytes and almost constant. This is due to the limited amount of memory needed to store the Q standard deviations and the W inter-packet times. Lastly, we also measured the delay and jitter of the traffic processed with the eBPF code. Overheads caused by the inspection are almost negligible for all the considered configurations.

5.6 Performance Comparison

A main goal of our framework is to run agentless detection processes without relevant performance degradation. Hence, this section presents a comparative analysis with well-known monitoring tools and technologies for network inspection. Specifically, we compared our agentless approach against implementations of the bin-based technique with Zeek and ANSI-C/libpcap. For the sake of comparison, we also considered a reference scenario, denoted as “baseline” in the following, where no traffic inspection is performed (i.e., no tools were running) in order to have a lower bound. Indeed, monitoring network traffic could interfere with the overall Quality of Service/Experience (especially by impacting on the packet loss, bitrate, and latency of delay-sensitive applications) or require non-negligible computing and storage resources. Therefore, we considered the impact of the packet size and transmission rate for UDP flows as well as the

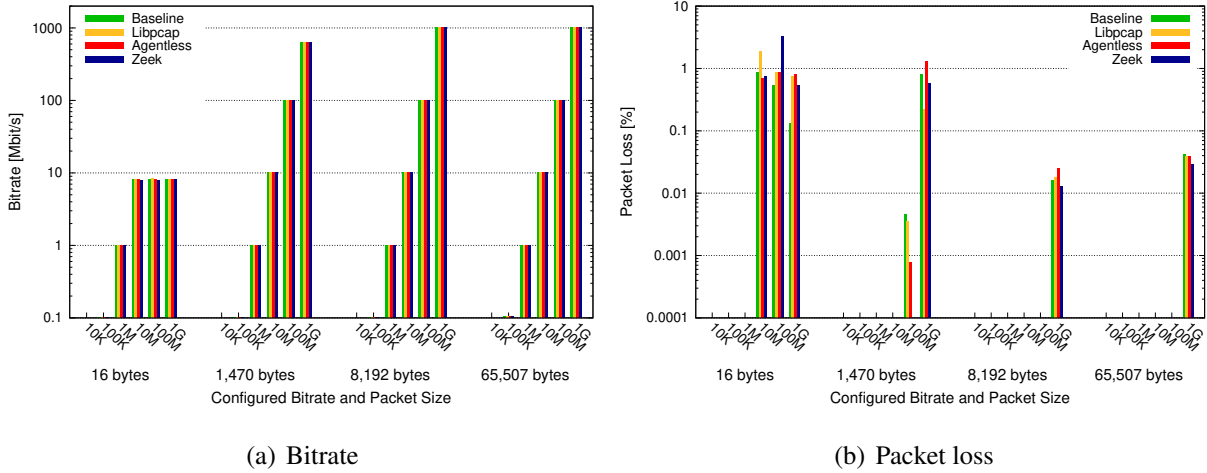


Figure 5.7: Characteristics of the UDP flow after the inspection (the graph is in log scale).

Maximum Segment Size (MSS) for TCP streams. For the packet, we considered four different values: 16 bytes modeling tiny and fragmented traffic, 1,470 bytes modeling a full utilization of the Ethernet frame, 8,192 bytes modeling IPv6 jumbo frames, and 65,507 bytes modeling maximum size allowed by UDP and representing the “best” condition for forwarding⁸. For the MSS, we selected four different values as well, by doing similar considerations for the case of UDP, i.e., we used 88 bytes, 536 bytes, which is the minimum value that should be used on IP links, 1,460 bytes for the full Ethernet utilization, and 9,216 bytes.

Concerning the transmission rates, we considered traffic loads ranging from 10 kbit/s to 10 Gbit/s. However, it turned out that our testbed was not able to sustain rates higher than 3 Gbit/s. This has to be ascribed to a limitation of our softwarized implementation but does not represent a constraint. In fact, production-quality deployments usually rely upon some form of acceleration that can sustain more than 10 Gbit/s of traffic [MLR⁺07].

For the sake of brevity and to avoid burdening results, in the following we only report and discuss the case of gathering information for the Flow Label when $B = 2^{12}$ and for loads up to 1 Gbit/s. Yet, similar results have been observed for the case of the Traffic Class and Hop Limit.

5.6.1 Impact on Packet Transmission

Figure 5.7 investigates how the inspection process behaves in the presence of different packet sizes and bitrates. In more detail, Figure 5.7(a) shows that the proposed method has a very

⁸The maximum size of 65,507 bytes is only feasible on loopback interfaces. Yet, it is of interest since is often used by virtual machines running on the same host.

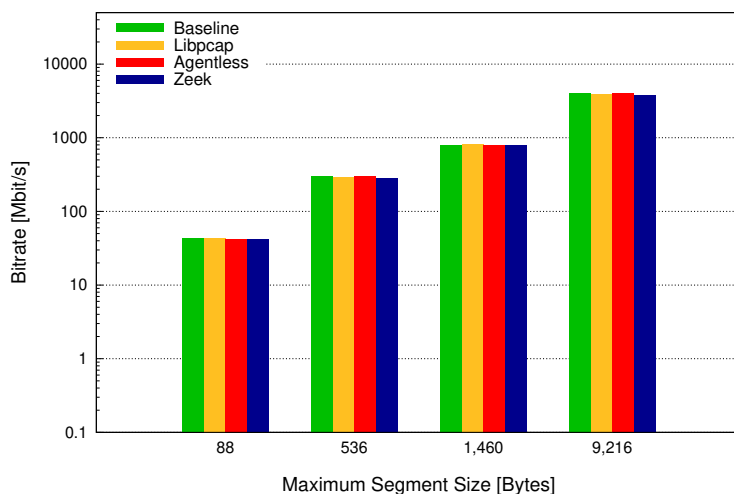


Figure 5.8: Measured bitrate for a TCP flow after the inspection (the graph is in log scale).

limited impact on the transmission rate, for the whole range of relevant parameters. Specifically, libpcap-based tools duplicate packets via raw sockets, hence decoupling additional processing from forwarding operations (i.e., inspection is done on a copy of the packet). However, even if eBPF programs act on the forwarding path, the impact is limited, thus the resulting behavior does not deviate from the considered baseline condition. Figure 5.7(b) depicts results for the packet loss, which is affected by the bitrate, as expected. In general, the causes of the losses are due to tiny packets causing a major overhead, and limitations of our setup to handle rates in the 1 Gbit/s range. For the sake of brevity, we omit results concerning the jitter. The measured variation for the inter-packet delay is ~ 0.1 ms for all the considered tools, thus making our approach feasible also to search for covert channels in multimedia or time-sensitive flows.

Finally, Figure 5.8 showcases the performance in terms of rates achieved when using the TCP/IPv6 traffic. Coherently, higher bitrates are possible with larger MSS especially due to a beneficial impact on the TCP flow control mechanism. Again, our approach performs similar to the case of Zeek and libpcap. Thus, our eBPF-based mechanism does not affect packet transmission in a significant way.

5.6.2 CPU and Memory Usage

CPU and memory utilizations are important to understand the footprint of the various frameworks used for the detection of network covert channels. Figure 5.9 reports a detailed breakdown of the used CPU. Our eBPF-based approach accounts for a small overhead with respect to the baseline. Both the libpcap-based tool and Zeek require more CPU at higher bitrates, whereas

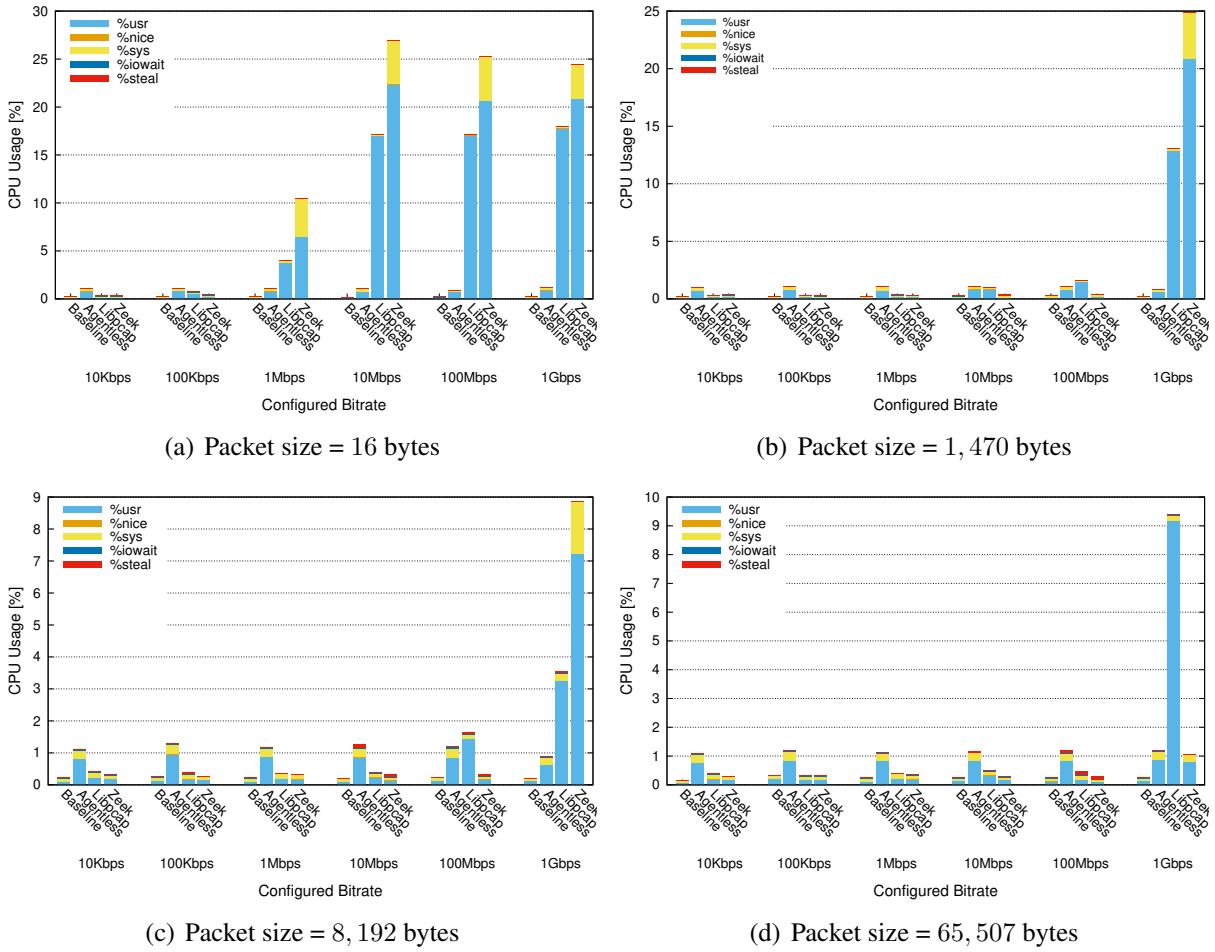


Figure 5.9: Cumulative CPU usage of the intermediate node for various packet sizes and bitrates when inspecting a UDP flow.

our framework has a more “stable” demand. Similar considerations hold for the case of TCP as reported in Figure 5.10. Even if the Python language is not the best option in terms of processing speed, our agentless mechanism performs better than the other tools and limits the used CPU compared to the baseline.

Concerning the used memory, in our trials we investigated the overall memory utilization including the Virtual Memory Size (VMS), the Resident Set Size (RSS) representing the size of physical memory including shared libraries, the Proportional Set Size (PSS) capturing the size of physical memory with proportional attribution to shared libraries, and the Anonymous utilization (Anon) containing the stack and other allocations. Figure 5.11 depicts the obtained results. As shown, Zeek has a larger memory footprint, but only a minimal part is allocated to the RAM. The memory allocated for our eBPF-based approach is larger because of the many libraries needed

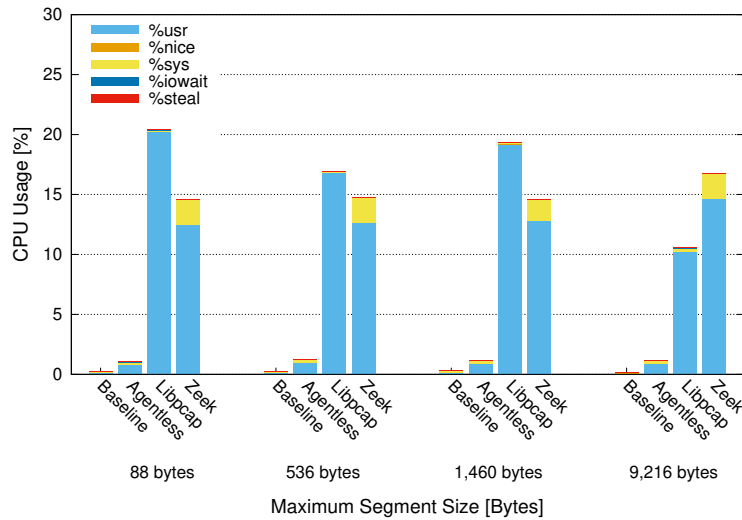


Figure 5.10: Cumulative CPU usage for a TCP flow.

by the Python runtime. Instead, the ANSI-C implementation based on libpcap has a negligible memory requirement owing to the efficient nature of the library and the use of a very minimal fraction of other calls, mainly for I/O operations.

5.7 eBPF for Security Tasks

The eBPF has been already considered for a variety of security-related tasks or to improve various software components. As an example, [DSM⁺19] investigates how to extend the `ntopng` network monitoring tool with events generated by the `libebpf_flow`, which allows to enrich network-layer data with system metadata (e.g., source and destination IP addresses are matched against source and destination processes and system users). The goal is to support the definition of custom policies to drop unwanted connections. Besides, eBPF can be used to break up the conventional packet filtering model in Linux. This can be achieved by moving the inspection process in the XDP, where ingress traffic can be processed before the allocation of kernel data structures, thus leading to performance benefits [HJBB⁺18]. This paradigm can be used to provide a “first line of defense” against unwanted traffic such as flows with spoofed addresses or DoS/Distributed DoS (DDoS) attacks [Ber17].

In the context of network tracing, [SZCR18] proposes a framework where a master node translates user inputs into configuration files to feed eBPF agents for monitoring network packets of specific connections at given tracepoints (e.g., virtual network interfaces). Obtained measurements are then collected and analyzed in a centralized manner. In [HJYH18], the authors

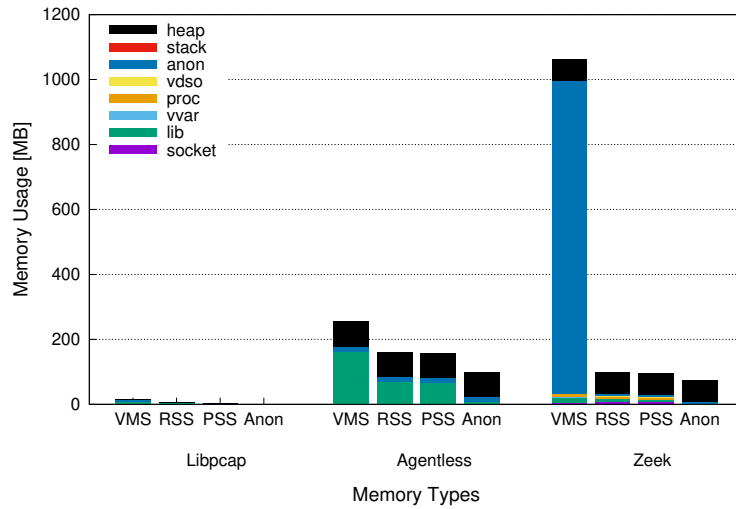


Figure 5.11: Memory allocation for the different tools.

propose an eBPF-based implementation for monitoring the traffic exchanged between the virtual machines without the need of specific hardware appliances. Results indicated that duplicating packets with an eBPF program attached to a hook in the `tc` achieves better throughput than native port mirroring of Open vSwitch, especially for large data units. The idea closest to our approach is presented in [CTJS20] showcasing a system for deploying eBPF programs and collecting their measurements in containerized user-space applications. To this aim, the framework exploits tools like Prometheus, Performance Co-Pilot, and Vector, as well as specific eBPF programs and various userland counterparts. However, differently from our work, [CTJS20] does not consider covert communications or manipulations of network artifacts. Rather, it focuses on monitoring the garbage collector, identifying HTTP traffic, and implementing IP whitelisting.

Compared to previous approaches, our idea allows to consider different covert channels within a unique framework. Owing to the flexibility of eBPF in handling various traffic features, the inspection process can be extended or adapted to consider different types of covert channels. Differently from past works available in the literature only addressing a single protocol or pursuing generalization via AI and data-intensive approaches, our bin-based data structures prevents to store and process sensitive details even with a per-flow granularity. This design allows to guarantee privacy requirements, while taming the computational burden. For what concerns the detection, revealing a class of channels only accounts for the creation of a simple detection rule, which can also take advantage of measurements already provided by network monitoring tools commonly deployed in medium- and large-sized scenarios.

5.8 Conclusions and Future Works

In this chapter, we presented a code layering framework for the detection of storage and timing covert channels. Specifically, we engineered an agentless monitoring architecture and developed various eBPF programs to gather data, map obtained values in suitable data structures, and implement a reference detection mechanism. Collected results indicate that code layering can be effectively and efficiently used to implement monitoring mechanisms in PaaS/serverless environments, as well as to implement a complete detection “pipeline” for covert channels. Moreover, the required resources make the use of eBPF a convenient choice, especially if compared with tools like Zeek or libpcap. Future works aim at using eBPF for actively manipulating traffic, e.g., to sanitize flows and disrupt the channels by overwriting fields or restoring them to a standard value. Concerning the technological viewpoint, future developments aim at refining the engineering and implementation of the agentless framework, especially for its deployment in production-quality environments.

In the next chapter, we will further investigate on the efficiency of the eBPF approach, in terms of used resources and impact on network transmission, when compared with alternative implementations and tools. Moreover, to remove some limitations of the detection scheme proposed in Chapters 3-5, in Chapter 7 we will investigate the use of AI to face realistic threats.

Chapter 6

Towards a Real-World Deployment

In Chapters 3-5, we discussed how eBPF can be considered a valuable building block to engineer a framework for spotting the presence of network covert communications. In this chapter, we further investigate its efficiency in terms of resource usage and impact on network transfers. Specifically, we consider alternative implementations for detecting covert channels built via libpcap, which is one of the most widespread library for packet processing outside of the kernel, and extensions of Zeek, a de-facto standard network analysis platform. The comparison offers a deep analysis of resource consumption, including CPU, memory and disk usage.

The remainder of the chapter is organized as follows. Section 6.1 discusses the difficulties to create efficient monitoring processes for covert channels with existing cybersecurity appliances. Section 6.2 describes the alternative technologies introduced before and used to implement the same data gathering mechanism. Section 6.3 provides a complexity analysis highlighting the level of expertise and difficulty for further extending the proposed tools whereas Section 6.4 reports the numerical results. Lastly, Section 6.5 concludes the chapter.

6.1 Pros and Cons of Monitoring and Inspection Processes

Traditionally, network appliances implement packet processing in hardware, because general-purpose computing architectures do not fit well the workflow for receiving, inspecting, and forwarding network packets. This approach has been largely used for cybersecurity purposes and many routing and switching devices today report flow-level statistics and measurements. Although this approach perfectly suits the need for flow analysis and reporting, it lacks the flexibility to adapt to new protocols and semantics, especially for cloud-native applications and service-oriented architectures [RR18].

A more flexible approach is also required to effectively tackle the growing sophistication of at-

tacks, for example in the case of stegomalware. One typical problem is the need to continuously update the database of known signatures and rules. Besides, when the scope is narrowed to single applications or few hosts, the amount of network traffic is not comparable with large infrastructures where hardware appliances are still needed. Today, several technologies are widely used both in commercial and industrial environments, e.g., Suricata, Ossec, Snort, Zeek just to mention the most popular. Unfortunately, as highlighted in Chapter 2, detection of covert channels is not a standard feature of security tools, even if they usually provide some mechanisms to extend the basic capabilities and to define monitoring tasks tailored to different use cases. This is, for instance, the case of Suricata and Zeek. As it will be detailed later in this chapter, the Zeek scripting language is more powerful when compared to rule-based tools such as Suricata. However, Zeek scripts are interpreted at run-time and are not suitable for high packet rates. Moreover, Zeek is not able to efficiently inspect fields with a per-packet granularity.

Because of performance issues, security technologies often leverage packet processing acceleration frameworks that “bypass” the native kernel networking stack and give direct access to hardware queues and functionalities in the NICs, e.g., PF_RING, Netmap, or DPDK, posing two main issues. First, the main processing code is usually developed in user-space for simplicity and it does not harm the stability of the whole kernel. Unfortunately, this means that all the well-tested configuration, deployment and management tools developed over the years within the built-in stack become useless, and should be re-implemented as well. Second, the direct access to hardware queues brings great advantage when the processing delay is mostly due to packet reception and transmission, but forwarding operations are very simple and limited to few table look-ups. However, such DPI-based methods often require several parsing operations and the need to continuously polling the NIC, which lead to large wasting of CPU time [RC21].

Finally, following the agile and lightweight nature of modern applications require easily extensible and manageable solutions. The latter are usually general-purpose and conceived to cover a broad number of threats. This is the best approach for monolithic systems, where all functions are grouped together on a single system, but does not fit distributed and service-oriented architectures largely used in cloud and cyber-physical systems [RCR21]. In fact, such architectures may change their topology at run-time due to management actions, e.g., for scaling or migration operations.

6.2 Monitoring Technologies

Implementing monitoring processes to detect covert channels is not trivial, especially when modern computing paradigms are taken into consideration (e.g., cloud-native applications, service-oriented architectures, or IoT deployments). To understand the feasibility of creating effective detection algorithms, we investigated and compared the following approaches implementing the bin-based mechanism used in Chapters 3-5:

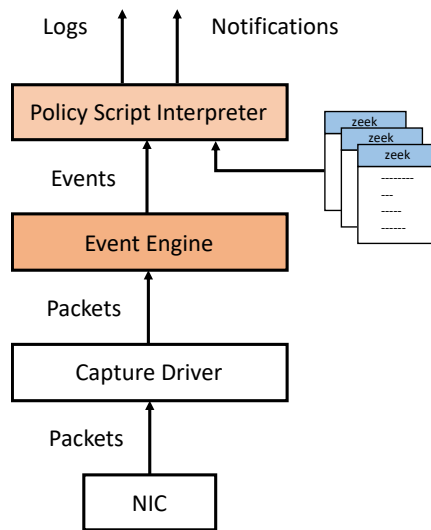


Figure 6.1: Zeek architecture and processing model.

- an extension of Zeek through a patch developed by using its native scripting language;
- a raw implementation in the C language based on libpcap;
- the `bccstego` approach leveraging the eBPF framework discussed in Chapter 3.

Differently from Chapter 4 and Chapter 5, the main purpose of this portion of research is to compare the performance of the different implementations in terms of development complexity, impact on packet transmission and resource usage, while specifically targeting modern computing paradigms.

6.2.1 Extending Zeek to Handle Network Covert Channels

Zeek is a fully open-source tool optimized for interpreting network traffic, generating logs based on events and mainly acting as a network probe for security management. Figure 6.1 depicts the architecture of Zeek. A Capture Driver (e.g., raw socket or PF_RING) delivers network packets to the core of the Zeek framework, called the Event Engine. This component inspects any protocol header and reduces the incoming packet stream into a series of higher-level events. Examples of events include connection initiation/termination, DNS query/response, or HTTP request/response. Finally, the Policy Script Interpreter derives the semantics related to the events, by executing a set of event handlers written using the custom scripting language provided by Zeek, and generates Logs and Notifications for external consumers.

Beyond the extensive set of logs describing network activity and built-in functionality for a range of analysis and detection tasks, Zeek comes with a domain-specific scripting language for expressing arbitrary analysis tasks. Indeed, analysis and logging are done via scripts. Following the above model, it is possible to create custom scripts for collecting data and enable the detection of network covert channels. Unfortunately, Zeek does not generate events on the reception of individual IP packets for performance reasons. Therefore, we patched the source code and created a custom version named `zeek-stego`. In essence, `zeek-stego` enables Zeek to compute statistical indicators with a per-packet granularity rather than a per-flow one. In addition, the patch can populate the bin-based data structure that has been introduced in Section 3.1.

6.2.2 Implementing the bin-based Approach with `libpcap`

The second approach we decided to use leverages `libpcap`, a C/C++ library developed for Unix-like OSes for traffic monitoring. The library allows to retrieve and capture packets from a live network device in a simple and straightforward manner.

The implementation¹ developed for this Thesis is composed of a single C program, in charge of opening the network device for packet live capturing, retrieving packets, parsing them according to the parameters set by the user and updating the counters of the data structure described in Section 3.1. The current version is able to collect data and statistics for the protocols/fields that can be likely used for creating covert channels: IPv6 (Flow Label, Traffic Class, Hop Limit, Next Header, Payload Length), IPv4 (Type of Service, Identification Number, Time To Live, and Fragment Offset), TCP (Acknowledgement Number, and Reserved Bits), and UDP (Checksum). The program is designed to be easily extended for considering other protocols and fields by simply adding cascading `if-case` conditions in the main function.

6.3 Complexity Analysis

Zeek is a well-known tool and its scripting language is rather simple. Moreover, the system comes with a large set of pre-built functionalities (the “standard library”), which can be used as a solid base. The main limitation is that the scripting language can only process events generated by the Event Engine, which requires a non-negligible effort, especially due to lack of documentation. As a partial workaround, recent Zeek versions implement the Packet Analysis feature that allows to customize the packet analyzer for the IP layer and include the necessary code for generating an event for each packet. We leveraged this feature for our purposes, even if it is mostly conceived to parse new protocol headers rather than extending existing ones.

¹<https://github.com/Ocram95/libpcap-filter> [Last Accessed, October 2022].

In general, the development of plain C code based on libpcap is the easier approach. In fact, capturing packets, parsing the headers and managing the code to handle the asynchronous reception and processing of network packets are straightforward and well documented operations. Extending such implementations may require full-understanding of the original code, which in any case is unlikely to reach the maturity of existing tools (e.g., Zeek). Nevertheless, writing C code is probably the most efficient alternative to find an optimal balance between performance and resource usage.

As discussed in Chapter 3, writing eBPF programs poses some challenges. First, the relationship and interaction between user-space utilities and eBPF programs must be fully understood, and this could be challenging especially due to a fragmented documentation. Second, eBPF programs have a small available stack size, which limits the number of functions and instructions. This practically narrows the protocols that can be parsed within a single program. Lastly, eBPF programs are checked by the kernel verifier before being loaded. The process ensures that there are no loops and the program always terminates. On the other hand, this has many limitations, including poor scalability and lack of support for loops [GAG⁺19].

On the contrary, handling packets is very easy, since eBPF programs are automatically run by the kernel on each packet reception. Moreover, the usage of frameworks like BCC in user-space facilitates many operations, including the dynamic creation of eBPF code, which might mitigate some of the aforementioned limitations. Even if eBPF programs have the steeper learning curve among the selected alternatives, we think it is a promising approach for this kind of applications.

6.4 Performance Evaluation

To conduct experiments, we prepared a testbed composed of three virtual machines, hosted on an OpenStack installation. All nodes ran on the same hypervisor, two Intel Xeon CPU E5-2660 v4@2.00GHz with 14 cores, 128 GB RAM, 64 GB SSD storage. Two of them were used as traffic source and destination, whereas the third virtual machine was used as the router and hosted the monitoring tools (i.e., Zeek, libpcap filter and eBPF framework). Both the traffic source and destination were created with 1 virtual core and 1 GB of RAM; the router had 4 virtual cores and 2 GB of RAM. All the virtual machines ran Debian GNU/Linux 11 with kernel 5.10.

Performance evaluation was intended to understand the impact of the alternative implementations on network operation. To this purpose, we considered both UDP and TCP streams generated with iPerf3 while changing the following parameters:

- packet size for UDP: 16, 1,470, 8,192, and 65,507 bytes, to account for the minimal packet size, and Maximum Transfer Unit for Ethernet, Gigabit Ethernet, and the loopback interface, respectively;

- transmission bitrate for UDP: 10, 100 Kbit/s, 1, 10, 100 Mbit/s and 1, 10 Gbit/sec, hence taking into account multiple rates up to the nominal bandwidth of the fastest technology available in common installations;
- the MSS for TCP: 88, 536, 1,460 and 9,216 byte, which reflects the smallest value accepted by the tool, the minimum value that should be used on IP links, the typical value used for Ethernet links and jumbo frames, respectively.

The impact was evaluated in terms of both network performance and resource usage. Network performance was monitored by the `iPerf3` tool itself, whereas resource usage was collected by the `sar`² utility for CPU (e.g., user-space, kernel-space) and `pmap`³ for memory allocation. In addition to run the three monitoring implementations, we also took measurements when no tool ran, which is taken as the baseline scenario. We limited our investigation to two different fields, the `Acknowledgment Number` and the `Hop Limit`, in order to consider fields both in two different layers of the protocol stack, i.e., TCP and IP header, respectively. Since parsing operations take more CPU instructions than reading specific fields, there is no need to replicate the experiments for every field covered by our implementation. For the two fields, we took into account two different amount of bins, i.e., 2^{12} and 2^8 , to understand their impact in the overall performance.

Experiments were run for 60 seconds, but measurements were taken for a slightly shorter interval, in order to avoid any transient. The sampling time interval to report the values of the internal counters was set to 1 second for each implementation.

6.4.1 Impact on Packet Transmission

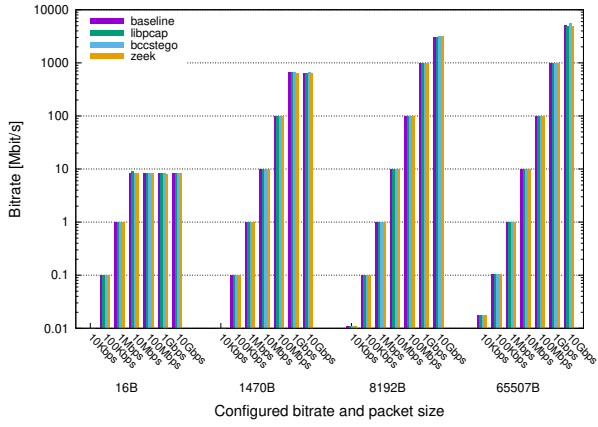
First, we considered how monitoring operations affect the transmission of network packets, by measuring the transmission rate, packet error rate and jitter.

Figure 6.2 shows how the bitrate for UDP flows changes at the receiver while varying both the packet size and the transmission bandwidth, in case of inspection of the `Acknowledgment Number` and the `Hop Limit`. As expected, smaller packet sizes lead to lower bitrates, but there are not meaningful differences with respect to the baseline when using the proposed monitoring mechanism, neither for inspection of the `Acknowledgment Number` nor of the `Hop Limit` (Figures 6.2(b) and 6.2(a), respectively). We can also notice that in our setup it is impossible to transmit at 10 Gbit/s, even with the largest packet size. This has to be ascribed to a limitation of our setup but does not represent a constraint.

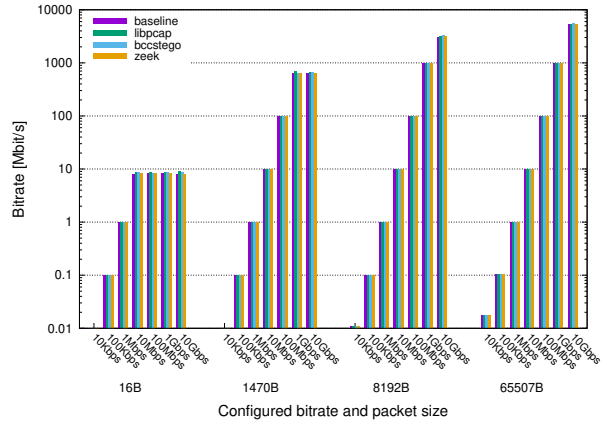
Figure 6.3 shows the packet loss percentage at the receiver in the same conditions aforementioned. The trends are correlated to the bitrate just seen. In fact, when the desired transmission

²<https://linux.die.net/man/1/sar> [Last Accessed, October 2022].

³<https://linux.die.net/man/1/pmap> [Last Accessed, October 2022].

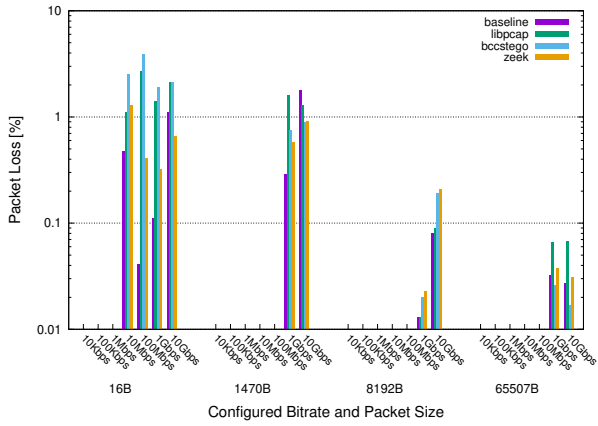


(a) Acknowledgement Number

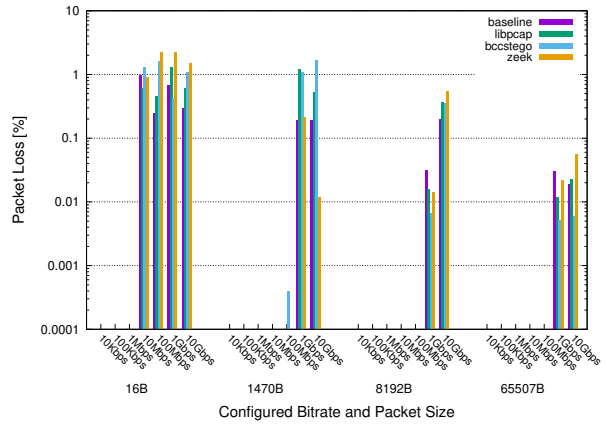


(b) Hop Limit

Figure 6.2: Measured bitrate at the receiver, while varying packet size and the transmission bitrate for a UDP flow.



(a) Acknowledgement Number



(b) Hop Limit

Figure 6.3: Measured packet loss at the receiver, while varying packet size and the transmission bitrate for a UDP flow.

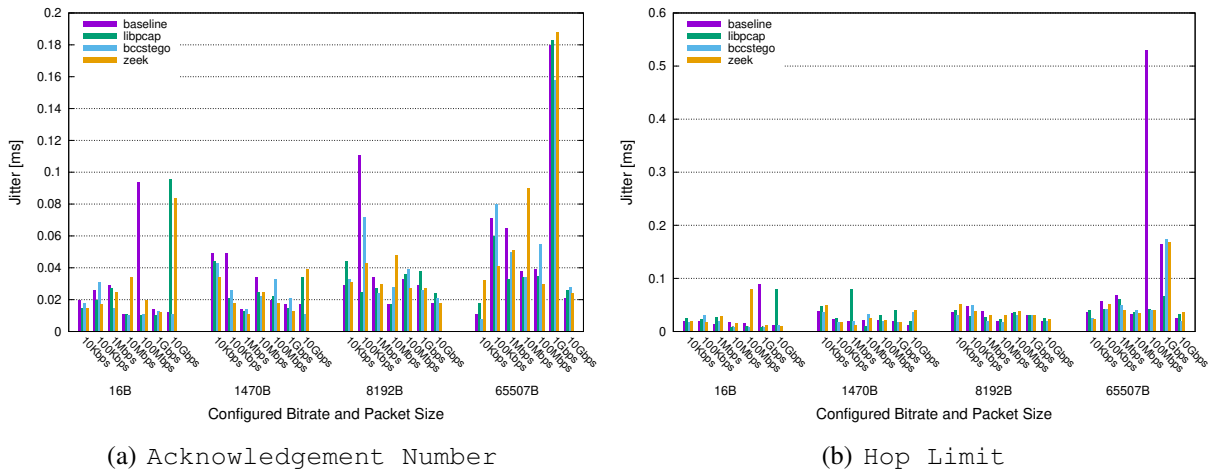


Figure 6.4: Measured packet jitter at the receiver, while varying packet size and the transmission bitrate for a UDP flow.

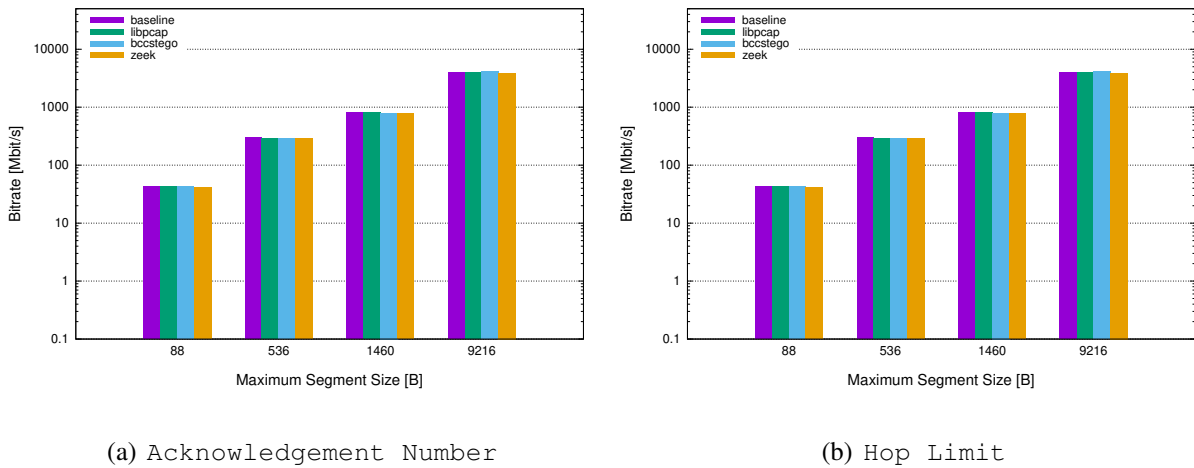


Figure 6.5: Measured bitrate at the receiver, while varying the MSS for a TCP flow.

rate cannot be reached, we see a higher packet loss. Excluding the case of smaller packets, the packet loss percentage introduced by the tools is limited and similar to each other.

Figure 6.4 depicts the average packet jitter for a UDP flow. Also in this case, the inter-packet delay generated by all the technologies is negligible (always under 0.2 ms) for most of practical applications. Again, no meaningful differences can be found in the inspection of the two fields.

For TCP flows, we measured the average transmission bitrate while varying the MSS (see Figure 6.5). In general, a larger MSS results in a higher bitrate, because the TCP congestion control protocol works better. No significant differences can be seen for inspection of the different fields

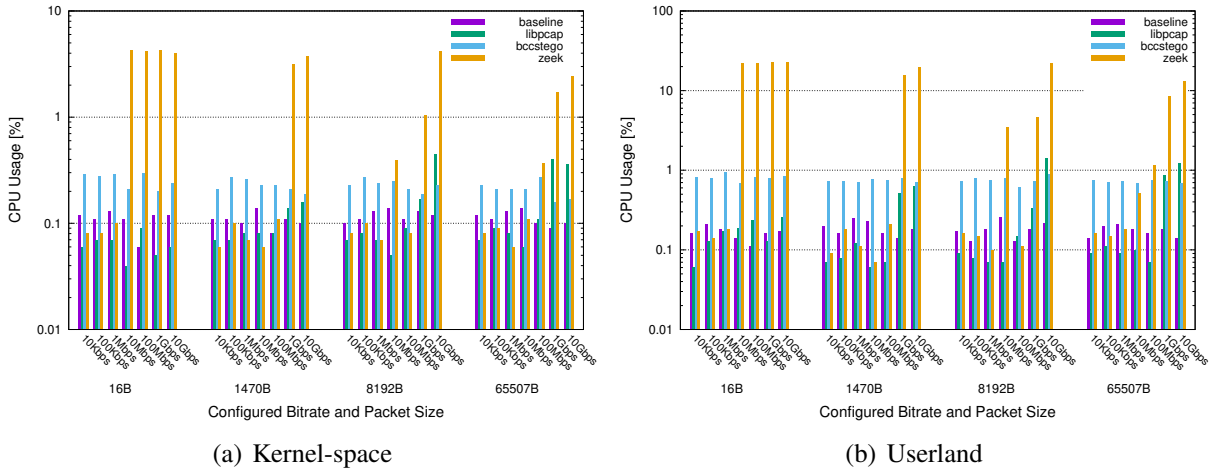


Figure 6.6: CPU usage measured at the intermediate node, while varying the packet size and transmission bitrate for a UDP flow. The monitored field is the `Acknowledgment Number`.

or the different monitoring mechanisms.

In conclusion, the measurements required by our detection technique do not impact over packet transmission. Even if eBPF programs are called as part of the forwarding operations, there is no practical differences with respect to the tools that duplicate packets with `libpcap` and process them in parallel to the kernel.

6.4.2 CPU Usage

As a second step, we measured the CPU usage for the different technologies. Since Zeek and the tool based on the `libpcap` library are user-space applications, the amount of CPU usage in userland is expected to behave in a different way compared to eBPF approach, which leverages in-kernel programs. Figures 6.6 and 6.7 show the CPU consumption in case the `Acknowledgment Number` or the `Hop Limit` are monitored, respectively.

As expected, both Figures 6.6(a) and 6.7(a) show a kernel CPU usage for `bccstego` higher with respect to `libpcap` and the baseline traffic, but still lower than Zeek, especially when higher bitrates are considered. However, while kernel CPU usage of the eBPF approach is rather constant with different packet sizes, the same measurement increases for the other tools, because duplicating packets takes more time in case of larger packets (e.g., 65 Kbytes). Zeek also requires more CPU in the user-space (i.e., 20% in the worst case), with respect to the other tools. Instead, `bccstego` performs quite well, remaining close to the baseline in almost all scenarios.

Figure 6.8 depicts a breakdown of the cumulative CPU usage, when varying both the packet size and the bitrate. For smaller packets, i.e., 16 bytes, and higher bitrate, i.e., 10 – 100 Mbit/s

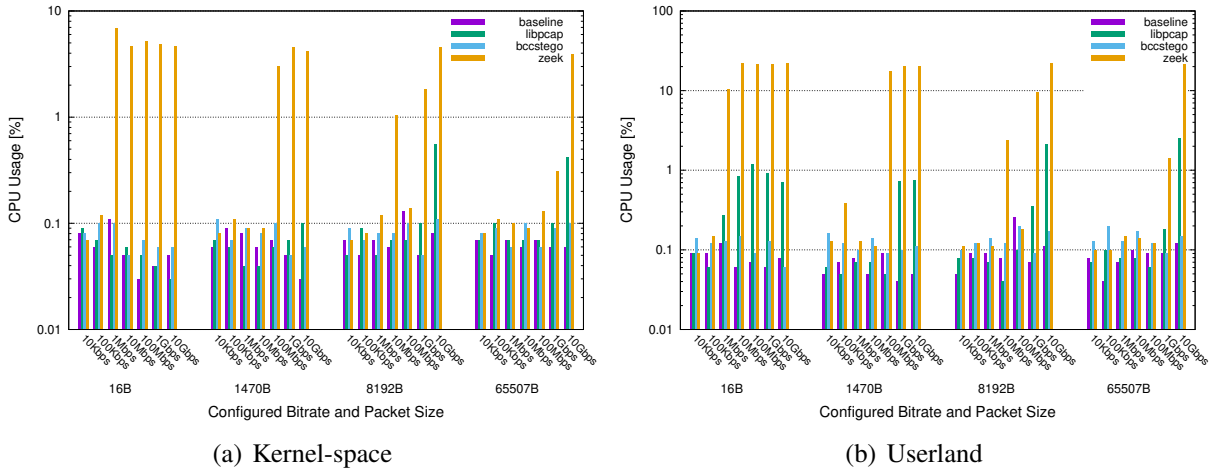


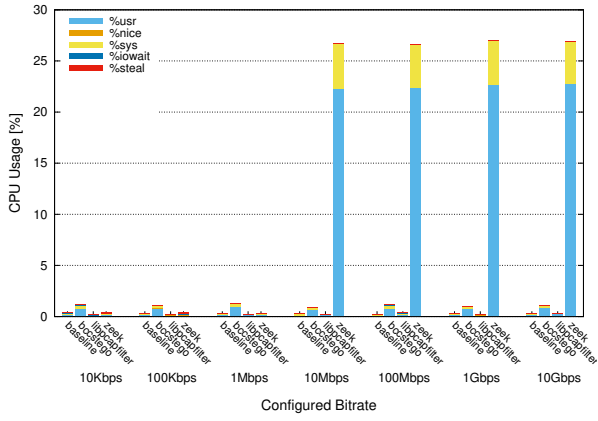
Figure 6.7: CPU usage measured at the intermediate node, while varying the packet size and transmission bitrate for a UDP flow. The monitored field is the `Hop Limit`.

and 1 – 10 Gbit/s, Zeek uses up to 25% of available CPU. Instead, `bccstego` makes a limited usage of CPU for almost every case, while the `libpcap` tool CPU usage increases especially in case of 65-Kbyte packets. We only show the results for monitoring the `Acknowledgement Number` in this case, since data for the `Hop Limit` leads to similar considerations (see Figure 6.7). These considerations can also be extended for the case of TCP flows while varying the `MSS`. Figures 6.9 and 6.10 show the kernel-space, the user-space and the cumulative CPU used by all the tools.

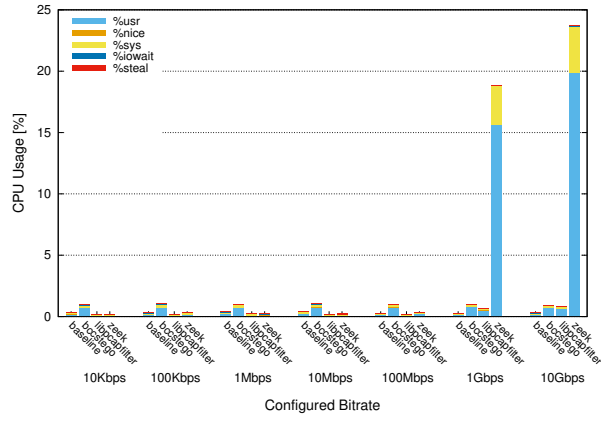
6.4.3 Memory Allocation

Finally, Figure 6.12 shows the amount of memory allocated by each technology. Similarly to Chapter 5, we considered a breakdown of memory allocation, i.e., the `VMS`, `RSS`, `PSS`, and `Anon`. The implementation based on the `libpcap` library has a minor impact on the memory used, since it only uses minimal system libraries for input/output. Zeek, instead, needs a large memory space with a minimal allocation in the RAM. Finally, since `bccstego` leverages Python features, it requires larger memory compared to `libpcap` implementation. This suggests that a more lightweight implementation is possible, for example by switching to a pure C implementation that leverages `libebpf` instead of the BCC framework.

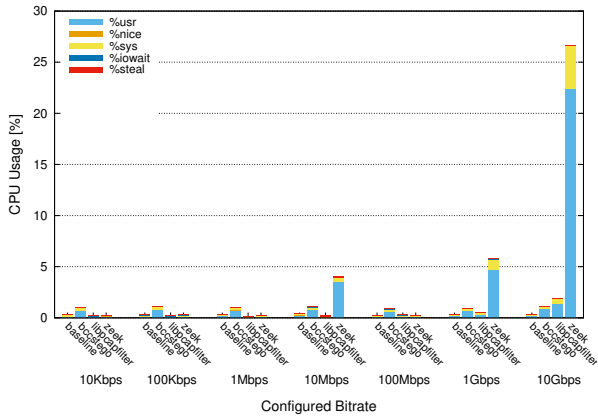
There are no significant differences when using 2^{12} or 2^8 bins, which are the cases for the `Acknowledgement Number` and `Hop Limit` fields, respectively. Again, this confirms that the overall monitoring approach is really lightweight and scalable, because most of the memory consumption is due to the implementation and libraries used.



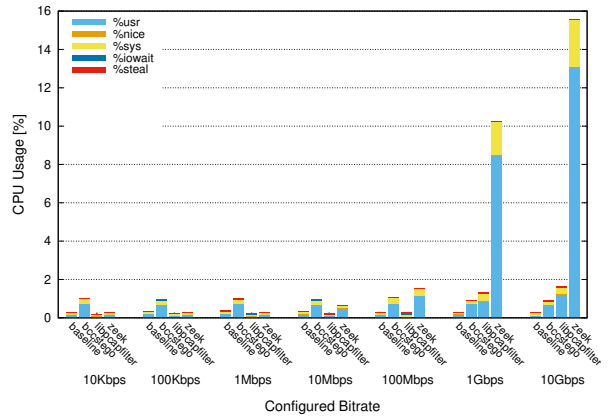
(a) 16-bytes payload



(b) 1470-bytes payload



(c) 8192-bytes payload



(d) 65507-bytes payload

Figure 6.8: Cumulative CPU usage measured at the intermediate node, for a UDP flow. The monitored field is the Acknowledgement Number.

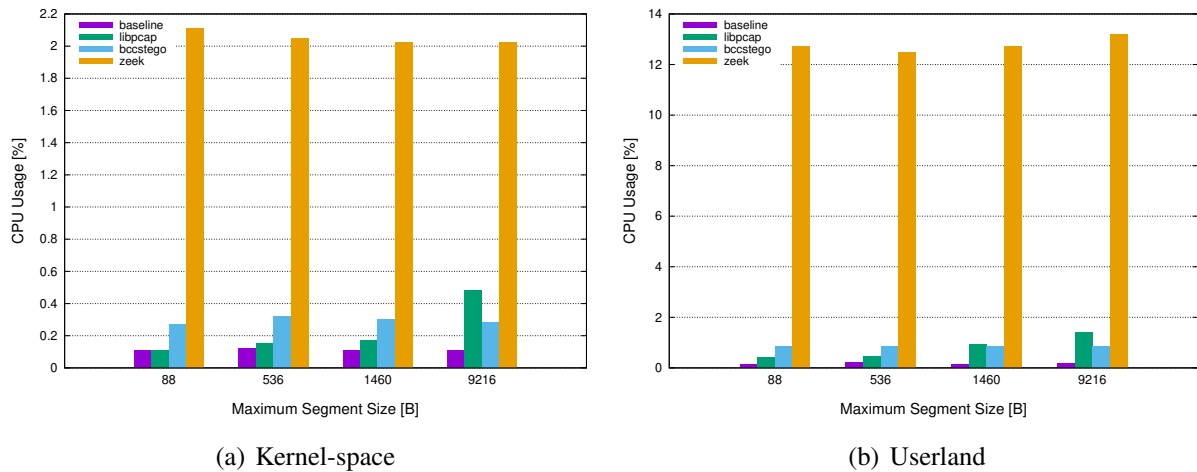


Figure 6.9: CPU usage measured at the intermediate node, while varying the MSS for a TCP flow. The monitored field is the Acknowledgement Number.

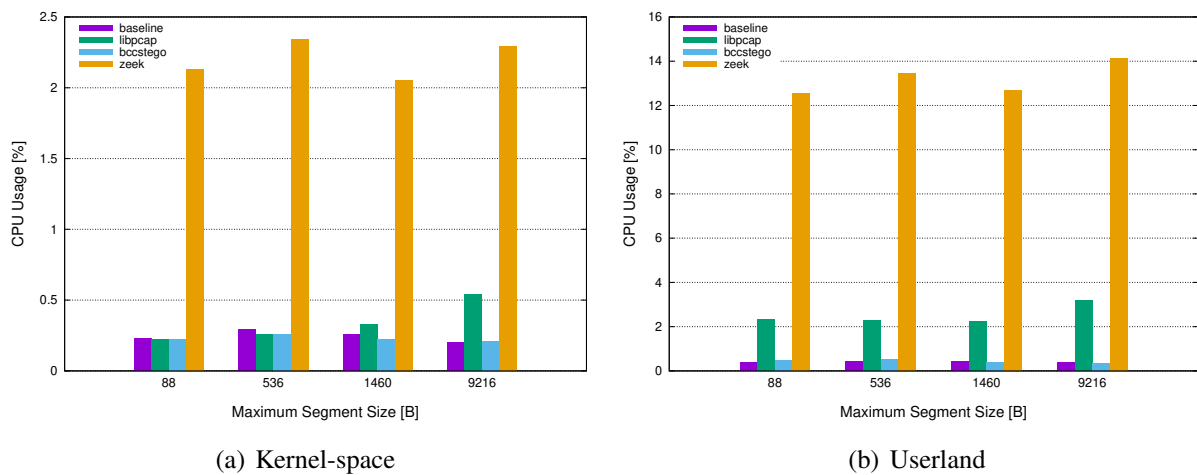
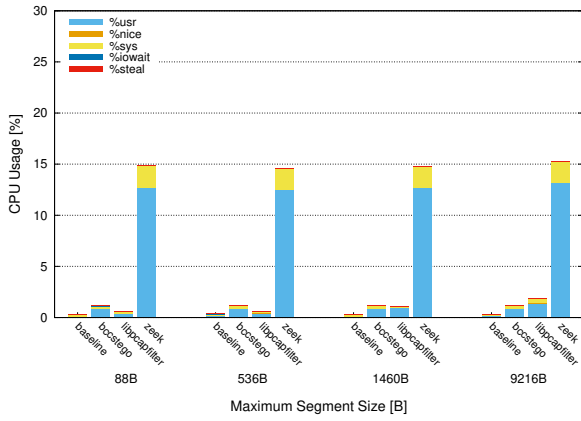
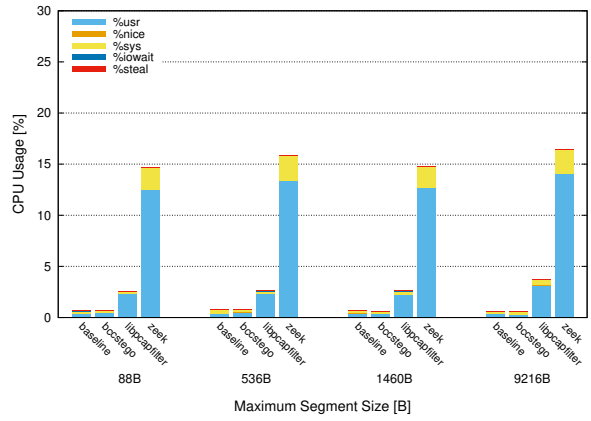


Figure 6.10: CPU usage measured at the intermediate node, while varying the MSS for a TCP flow. The monitored field is the Hop Limit.

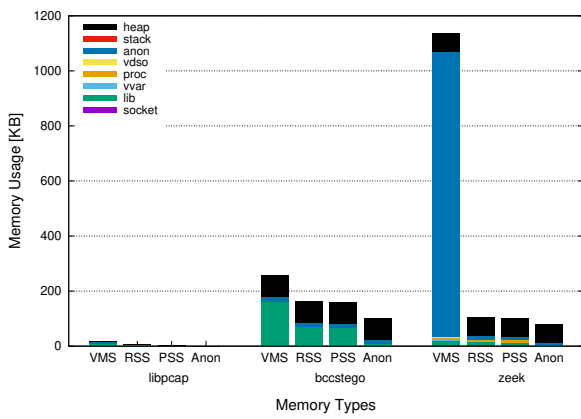


(a) Acknowledgement Number

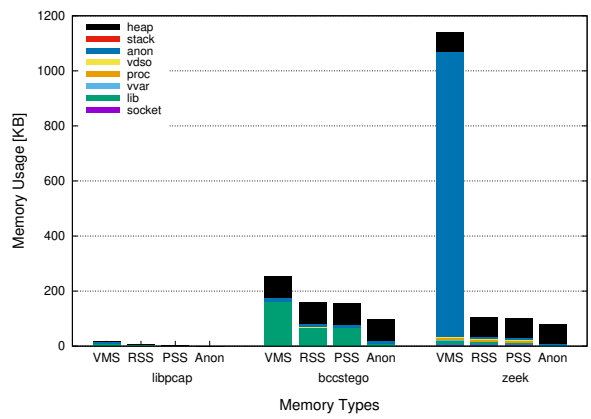


(b) Hop Limit

Figure 6.11: Cumulative CPU usage measured at the intermediate node, for a TCP flow.



(a) Acknowledgement Number



(b) Hop Limit

Figure 6.12: Memory allocation for the different inspection tools.

6.5 Conclusions and Future Works

In this chapter we analyzed different implementations for a lightweight packet inspection mechanism that aims at detecting network cover channels. Among the technologies used to implement the bin-based mechanism proposed in Chapter 3, the results for the eBPF-based approach indicate a negligible impact on packet transmission in terms of bitrate, packet loss percentage, and jitter, when considering different network conditions. In terms of resource usage, eBPF programs demonstrate to be lightweight and scalable. However, the implementation of the userland utility in Python largely increases memory requirements, and suggests to switch to pure C code for memory-constrained systems (this may be the case for containers and IoT devices). Future works aim at understanding the feasibility for an automatic code generation approach, in order to provide more flexibility for network monitoring purposes.

With a suitable technological foundation for gathering data, in the next chapter we will investigate the use of AI for detecting network covert channels in IoT environments.

Chapter 7

Detection of Network Covert Channels via AI

As discussed in Chapter 1, the use of AI for detecting threats is taking momentum. The case of detecting network covert channels is not an exception. Thus, this part of the Thesis addresses the problem of using AI to reveal the presence of covert communications in a wide-spread and challenging scenario, i.e., the IoT case. In fact, the IoT paradigm allows to create advanced services able to interact with the physical world and to remotely operate large-scale infrastructures. As a result, the number of applications taking advantage of IoT technologies is now almost unbounded. For instance, cost-effective sensors and devices are used for entertainment and health purposes, to access and manage industrial control systems, as well as to automatize homes and buildings. Unfortunately, the tight coupling between devices and physical entities, the resource-constrained nature of many nodes, and the lack of rigorous development or configuration processes, are at the basis of countless security and privacy flaws [NBHC⁺19].

Despite IoT nodes are often considered simple devices, they can be used to implement effective threats. As an example, the Mirai malware allows to create a large-scale botnet of devices with limited computing and connectivity resources, which has been used to launch DDoS attacks against many international organizations and sensitive targets [AAB⁺17]. In addition, IoT nodes can be enumerated to infer details on the physical deployment [SGL⁺18] and the resulting traffic can be inspected to implement various side-channel-based techniques for partially invalidating defensive frameworks or to conduct reconnaissance campaigns [MC21]. Therefore, a major effort is devoted to make IoT ecosystems more secure, but this could be partially voided by stegomalware. Due to the ubiquitous availability of devices always connected to the Internet, their intrinsic interaction with sensitive data, as well as several design flaws and limitations, completely assessing the security of IoT deployments requires also to consider threats endowed with network covert channels capabilities (see Section 1.4 for examples of real attacks exploiting covert channels). To develop suitable mitigation techniques, machine learning approaches

demonstrated to be effective for detecting a multitude of network attacks and to implement general intrusion detection mechanisms [ASKWS⁺21].

Therefore, in this chapter we address the problem of detecting network covert channels targeting the TTL field of IPv4 datagrams. In fact, the resource-constrained nature of IoT devices, including the use of “lean” TCP/IP protocol stacks to tame complexity, prevent malware to implement sophisticated covert channels or computing-intensive network steganography algorithms. To develop our detection methods, we take advantage of autoencoders, which are neural networks where the target of the network is the data itself. Autoencoders allow to reduce dimensionality and learn efficient encoding, whereas they are a convenient choice when in absence of a labeled training set and can be incrementally updated in order to be deployed also on devices with limited computational and storage resources. This is of prime importance when addressing malware exploiting network covert channels, since it often remains undetected or undocumented until major reverse engineering attempts or forensics investigations [MW17]. As regards prior works considering covert channels targeting IoT scenarios, the literature mainly focuses on timing channels, for instance to detect cloaked communications in Supervisory Control And Data Acquisition applications [ABP⁺19] or in the Constrained Application Protocol [VMS19].

Summing up, the contributions of this chapter are the design of a machine-learning-capable approach for detecting covert channels targeting IoT ecosystems, and a performance evaluation campaign based on realistic traffic traces commonly used in the literature. Since countermeasures could be also deployed at the border of the network in nodes with limited capabilities (e.g., home gateways) emphasis has been put on the footprint required by the proposed approach.

The remainder of the chapter is organized as follows. Section 7.1 provides details on the considered attack model, Section 7.2 introduces our detection approach to detect covert channels targeting the TTL of IPv4 datagrams, and Section 7.3 showcases numerical results. Lastly, Section 7.4 concludes the chapter.

7.1 Attack Model and Design of the Covert Channel

This section discusses the attack model taking advantage of a network covert channel. Figure 7.1 showcases the general reference scenario. Specifically, we consider an attacker able to take control of one or more IoT nodes, for instance by dropping a malicious payload via a phishing campaign [NBHC⁺19]. The infected device will then create a network covert channel to exfiltrate data towards a remote C&C server or to exchange commands with the attacker, e.g., to configure a backdoor or operate a botnet. Relying upon a network covert channel allows to bypass a firewall or specific security policies enforced by a middlebox, such as a home gateway.

Even if the literature abounds of techniques for creating cloaked communication paths within network flows and real-world threats taking advantage of information hiding are multiplying

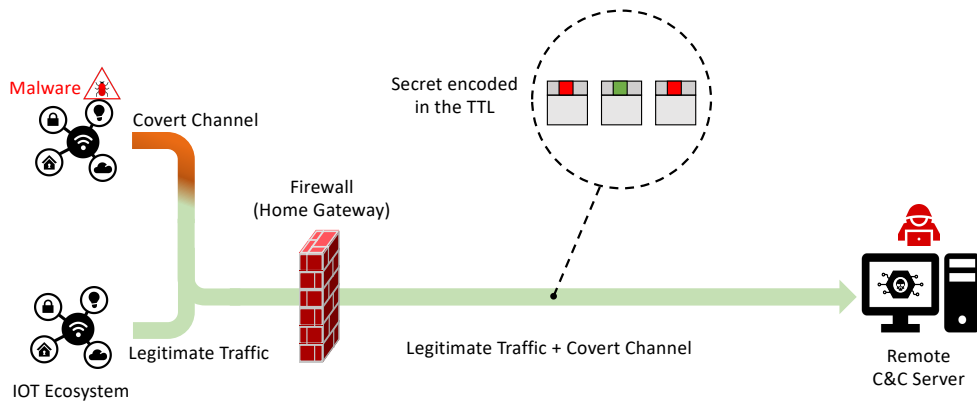


Figure 7.1: Attack model considering a malware sending data towards a remote C&C facility via a network covert channel created within the TTL field of IPv4 traffic.

[ZAB07, MC15, CCC⁺20], the resource-limited nature of IoT nodes poses constraints on the complexity of the covert channel. As a consequence, the embedding mechanism should be simple in order to not disclose the presence of the malware due to perceptible lags or anomalous depletion of batteries. At the same time, since IoT traffic often requires some form of Quality of Experience (e.g., to not postpone the execution of commands sent by the user), traffic alterations and the introduction of additional delays should be limited. Therefore, we consider a malware cloaking data within the TTL field of the IPv4 header [ZAB07].

In more detail, the TTL is manipulated to implement a storage network covert channel and transport arbitrary information. Due to the varying nature of the TTL and to not appear suspicious, the malware should not directly write the secret data in the field [ZAB06]. Rather, it can encode the bits 1 and 0 by increasing or decreasing the observed TTL of a suitable threshold or by using most common values as “high” and “low” signals. Finding proper TTL values is not trivial, since their difference should be ample enough to absorb fluctuations caused by alterations of the routing and to prevent decoding errors, while not reducing the stealthiness of the channel.

To design the covert channel, the attacker usually investigates the targeted network to understand “clean” traffic conditions and adapt the hiding mechanism accordingly. To discuss how to tune the channel used in our attack model, we considered the collection of IoT traffic made available in [SGL⁺18]. As an example, we showcase results for the 24-hour slice of data captured from September 22, 2016 at 16:00 to September 23, 2016 at 16:00, CEST¹. Without loss of generality and to prevent burdening results, we removed IPv6, ICMP, DNS and NTP conversations, in addition to multicast/broadcast traffic. Figure 7.2 depicts heatmaps for the collected TTL values. As depicted in Figure 7.2(a), the values observed for the TTL are clusterized, especially in the 32 – 64 and 208 – 224 ranges. This requires the attacker to encode information without using

¹Data collected for IEEE TMC 2018, University of New South Wales, Sydney. Available online at: <https://iotanalytics.unsw.edu.au/iottraces.html> [Last Accessed: October 2022].

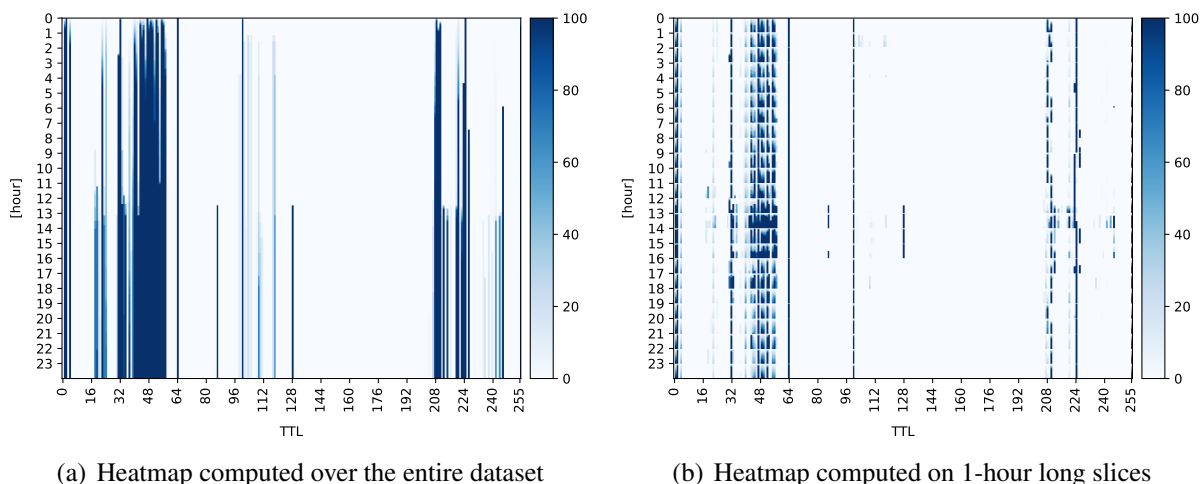


Figure 7.2: Various heatmaps computed over a 24h traffic trace.

values never observed in normal conditions. Yet, traffic conditions are not static, hence, we refined our analysis by resetting the observed values each hour. Figure 7.2(b) portraits results. As shown, some values of the TTL are always present in the traffic (e.g., those around 48), whereas others have an intermittent behavior. For instance, datagrams with a TTL equal to 128 are present only for 3 hours (i.e., from 13-th to the 16-th hours). This puts constraints on the temporal location of channels using a TTL equal to 128 to encode information as well as on their duration.

In general, channels targeting the TTL should alter datagrams in a limited manner in order to avoid macroscopic per-flow signatures [ZAB06]. Moreover, TTL values highly depend on the type of nodes, hosts and appliances exchanging traffic through the network. In fact, Android and iOS devices as well as Linux hosts generate traffic with a default TTL of 64, whereas Windows nodes use a default TTL of 128 [CLB⁺14]. Thus another important trade off should aim at avoiding to make the channel detectable via simple host/OS fingerprinting mechanism.

7.2 Deep Ensemble Learning Scheme

In this section, we illustrate a solution based on DL to spot the presence of covert channels within traffic flows. Our detection model takes the form of an ensemble of unsupervised neural network models. The main benefit in using the unsupervised approach relies on the capability of the models to raise alarms also on never seen attacks: this represents a frequent scenario when dealing with covert channels, since they are often undocumented and unknown *a priori*. We first illustrate the detection mechanism for a single model and then we describe how this technique can be extended to learn effective and scalable ensembles.

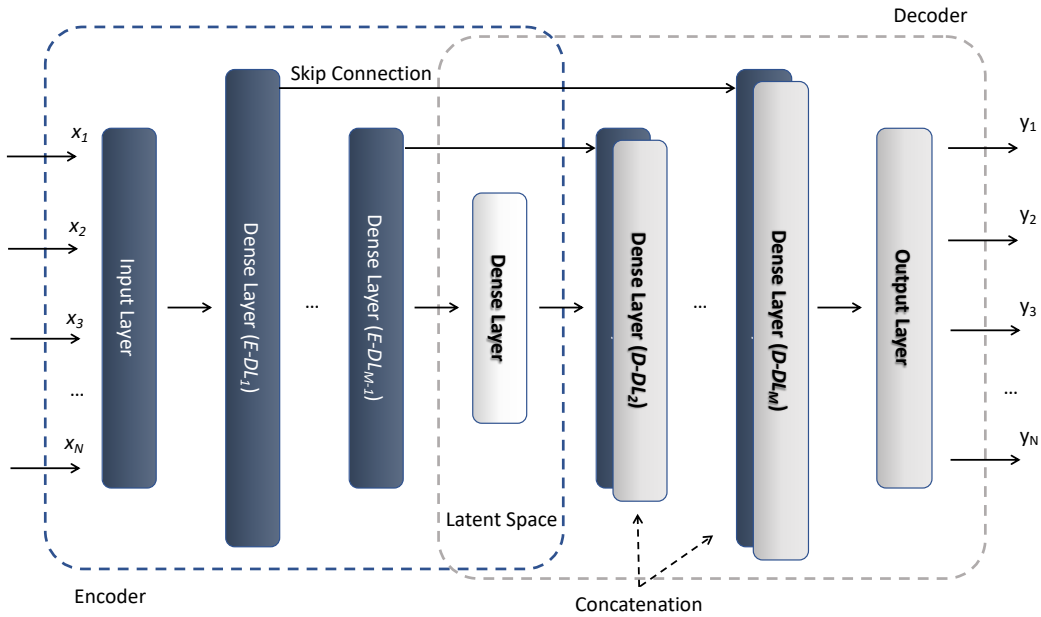


Figure 7.3: Neural architecture (Sparse Autoencoder) used to perform the detection of covert channels targeting the TTL of IPv4 traffic.

7.2.1 Detection Through a Single Autoencoder

The idea behind our solution is to adopt a neural encoder-decoder architecture trained against traffic data. Basically, an autoencoder is an unsupervised (i.e., trained without any information concerning the nature of the attack/normal behavior) neural network model performing two main operations: first, it compresses the input data (i.e., a number of statistics computed over the traffic generated by the IoT network and described in Section 7.3.2) within a latent space, then it reconstructs the original information provided as input. In our setting, the model is only trained against the normal behavior. The underlying intuition is that the legitimate input data should be (almost) correctly reconstructed by the autoencoder, in other words, the encoding/decoding phases should not introduce a heavy distortion in the output. By contrast, outliers and anomalous values in the input will yield a deviant output.

The usage of the reconstruction error as a measure of outlierness to discover abnormal behaviors has already been proposed in the literature, but the adoption of unsupervised techniques (and in particular of encoder-decoder architectures) for revealing covert channels is quite unexplored [BG15, DAFB⁺19, ASKWS⁺21]. As discussed in [BLPL06, HS06], autoencoders are considered as a valid solution to the problem of effectively summarizing the main information of a given input into a low-dimensionality representation. In essence, these neural network models aim at yielding as output a duplicate similar to the input data.

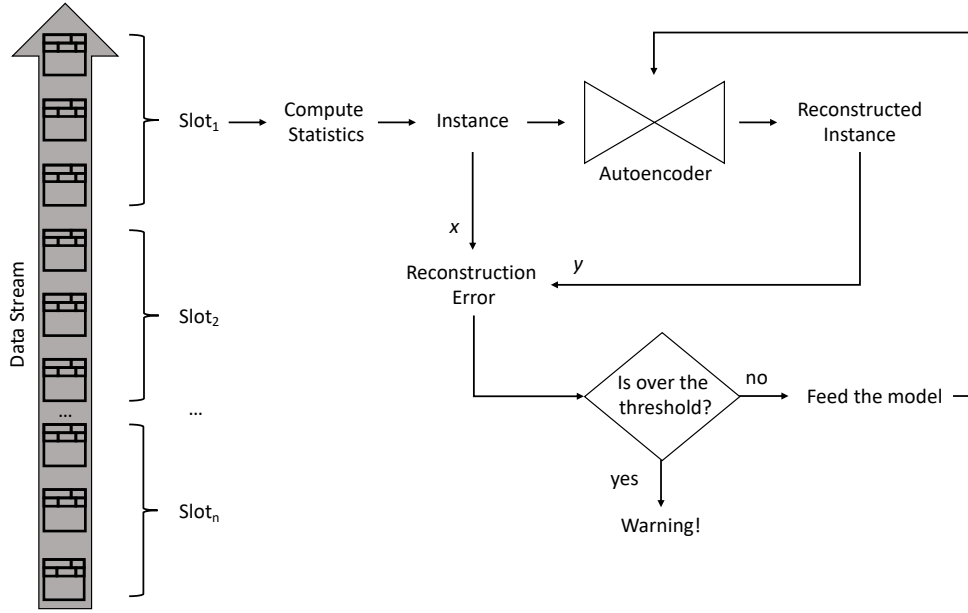


Figure 7.4: Detection mechanism for revealing the presence of network covert channels.

In this work, we employ the neural model shown in Figure 7.3. Basically, it includes two main components, named Encoder and Decoder, respectively. Let $\mathbf{x} = \{x_1, \dots, x_N\}$ be a set of numeric features (in our case, a number of traffic flow statistics computed for a time slot). The former sub-network is devoted to map the input data with a latent space (encoding), i.e. learning a function $\mathbf{z} = enc(\mathbf{x})$, whereas the second one yields the overall network output by reconstructing the input from the features extracted by the encoder $\mathbf{y} = dec(\mathbf{z})$ (decoding). Gradient descent is employed to learn the model weights by minimizing a suitable loss function. In our approach, the *Mean Square Error* (MSE), i.e., $Loss_{MSE}(\mathbf{x}) = \frac{1}{N} \sum_i \|x_i - y_i\|_2$, is used as loss.

Notably, the architecture of Figure 7.3 exhibits two main differences with respect to a standard encoder-decoder model: (i) Skip Connections are used to boost the predictive performance of the model and to reduce the number of iterations required for the learning algorithm convergence, and (ii) a hybrid approach including the usage of Sparse Dense Layers is adopted to make the autoencoder more robust to noise, especially since attacks often exhibit slight differences compared with normal behaviors. Both, encoder and decoder are composed of M hidden layers, therefore we adopted a symmetric architecture. In more detail, the adoption of the skip connections simplifies the learning process of the network by providing as input to each layer of the decoder ($D-DL_i$), except for the shared latent space, both the previous ($D-DL_{i-1}$) and the correspondent encoder layer ($E-DL_{M-i+1}$). As regards the Sparse Layers, they are used to generate a wider number of discriminative features, which allow for extracting a more representative latent space.

Figure 7.4 depicts the detection process of covert channels targeting the TTL field of IPv4 datagrams. Without loss of generality, we assume to monitor an “infinite data stream”, i.e., the traffic

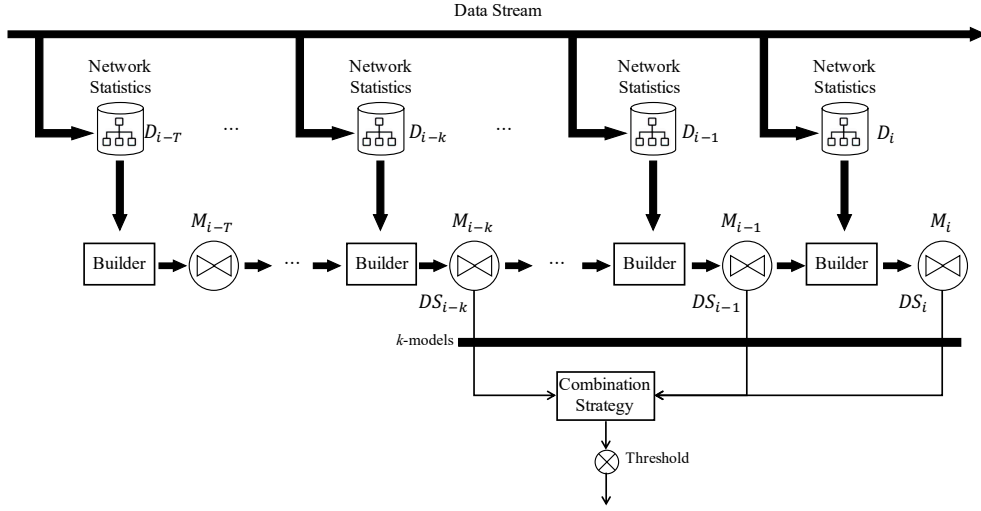


Figure 7.5: Incremental Deep Ensemble model approach.

of the various IoT nodes feeds our detection mechanism in a continuous manner. At pre-fixed time intervals (corresponding to a time slot in Figure 7.4), we compute a number of statistics (e.g., min, max, or average) to describe the behavior of the TTL fields composing the aggregate traffic flow. This can be done without impairing the overall traffic and by using limited computing resources [RCZ21]. Specifically, we compute metrics starting from TTL values gathered from the packets composing the inspected traffic aggregate. First, an autoencoder, pretrained only against legitimate data flows, is used to reproduce the statistics, then reconstruction error is calculated for the current example as the MSE between x and y . As a last step, if the error is smaller than a given outlierness threshold, the current data are labeled as “normal” and the model is updated, otherwise a warning is raised.

7.2.2 Learning and Combining Different Detectors

A main limitation of the above described approach relies on the necessity to learn the neural network model against the whole training set (that could be unfeasible in IoT networks with tight computational resources). Moreover, in real scenarios the limited resources of the device where the detector is deployed and the presence of concept drifts in the observed behaviors [FGP19] can affect the predictive performance of the autoencoder. To mitigate such issues, we devised an incremental learning scheme based on an ensemble of encoder-decoder architectures shown in Figure 7.5. Basically, we consider the case where only a limited number of training examples \mathcal{D} can be gathered and stored in a data chunk (named D_i in the figure).

The ensemble solution relies on building up a series of k base DNN detectors (denoted as

$M_i, M_{i-1}, \dots, M_{i-k}$) sharing the same neural architecture described in Section 7.2.1. These autoencoders are trained from disjoint data chunks (denoted as $D_i, D_{i-1}, \dots, D_{i-k}$, respectively), which are fed with data instances gathered in different temporal intervals. Specifically, the learning process is loosely inspired to Transfer Learning (TL). The main idea of TL consists in re-using DL models, learned on different domains/datasets or for tackling different tasks, by fine-tuning them for addressing the own learning problem. Differently from a standard TL approach, in our solution the model M_i is trained (i.e., fine tuned) from the weights of the model M_{i-1} and the sample D_i , and so way for all the models composing the ensemble. In this way the detection model will be able to gradually adapt to normal concept drifts that can occur, for instance, due to the deployment of new devices in the network that can modify the network statistics. This approach can also reduce the risk of “catastrophic forgetting” [PKP⁺19] that affects DL models (and also different types of shallow architectures) when learned incrementally. The final anomaly score for each instance is then computed as the median value of the k reconstruction errors, i.e., the Deviance Scores (DS), $\{DS_i, \dots, DS_{i-k}\}$ yielded by the base models.

7.3 Performance Evaluation

In this section, we illustrate experiments for evaluating the performance of our deep ensemble based approach to spot covert channels in IoT scenarios.

7.3.1 Dataset Preparation

To evaluate the effectiveness of our approach for detecting network covert channels targeting IoT ecosystems, we prepared a dataset starting from the traffic traces made available in [SGL⁺18]. In more detail, we used datasets containing traffic collected from September 22, 2016 at 16:00 to September 29, 2016 at 16:00, CEST. Similarly to the example of Section 7.1, we removed IPv6, ICMP, DNS and NTP conversations as well as multicast/broadcast traffic. To avoid unwanted signatures/fingerprints, we also removed traffic generated by non-IoT devices, such as mobile phones and laptops. As a result, we obtained a 1-week long dataset with an overall throughput in the 5 – 36 kbit/s range, generated by 28 IoT endpoints, such as smart speakers, smart lights, cameras, and hubs.

To implement the considered attack template in a realistic manner, we modeled the presence of a threat tampering a single IoT device. As an example, the attacker could gain access to the assets of the victim via phishing or by exploiting some ad-hoc CVEs². In our scenario, we considered a malicious software targeting the Dropcam camera, which has been used to

²List of CVEs targeting IoT nodes/devices maintained by MITRE. Available online at: <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=iot> [Last Accessed: October 2022].

send/exfiltrate sensitive data towards a remote C&C facility. To have a dataset containing fair amounts of “legitimate” and cloaked conversations, we assumed that the IoT device has been tampered on September 27, thus the Dropcam has been under control of the attacker for 3 days.

To create the various storage network covert channels, we used `pcapStego` to directly rewrite the traffic captures and implement realistic attack conditions. As discussed in Section 7.1, to not make the detection trivial, we encoded bits 1 and 0 in TTL values equal to 64 and 100, respectively. Moreover, we randomly interleaved packets containing hidden data with legitimate/unaltered packets in order to prevent long bursts of manipulated TTL values. In fact, the latter could reduce the stealthiness of the covert channel leading to a trivial detection [ZAB06]. Such a behavior can be ascribed to an attacker switching the hidden communication among two states (i.e., exfiltrate data and not manipulate traffic) to remain unnoticed via elusive mechanisms. To avoid further statistical signatures, the secret data transmitted over the covert channel has been modeled with randomly-generated strings: this is representative of an attacker using some obfuscation technique, such as encryption or scrambling [MRB17]. Concerning the volume of data transmitted within the covert channel, we modeled each day of attack with a different template. Specifically, we considered the exfiltration of 69, 80, and 64 kbit of data. Such volumes can represent a file containing sensitive information like several username+password pairs or configuration details of a specific IoT device or smart hub. Moreover, assuming covert transmissions in the 64 – 80 kbit range allowed to have an IoT node accounting for a variable amount of steganographically-modified traffic. In more detail, the compromised IoT node manipulates the 18%, 1%, and 12% of the overall daily traffic, respectively.

7.3.2 Preprocessing, Parameters and Evaluation Metrics

To assess the quality of the proposed approach in detecting the presence of network covert channels within traffic aggregates, we developed a prototype in Python based on the TensorFlow³ library. From the traffic dataset, we extracted the following fields: a progressive timestamp, the number of incoming packets within a given time slot, the average and median values of observed TTLs, the values of the 10th, 25th, 75th and 90th percentile, minimum and maximum TTLs, as well as a label indicating the presence of the attack (i.e., for testing purposes). Recalling that our approach exploits a “slotted” architecture (see Figure 7.4), in this work we consider a time slot with a duration of 5 seconds.

Data have been partitioned in training and test sets by using a temporal split: (i) the data gathered in the first 96 hours only contains legitimate traffic and has been used for the learning phase of the ensemble, whereas (ii) the remaining instances compose the test set. As a result, the training and the test set contains 69, 116 and 51, 837 instances, respectively. Moreover, input data are further preprocessed through a normalization procedure: a MinMax normalization has been adopted to

³<https://www.tensorflow.org> [Last Accessed: October 2022].

map each feature in the range $\{-1, 1\}$ to improve the stability of the learning process.

As described in Section 7.2, the base model composing the ensemble is an autoencoder model. The Encoder is composed of 4 fully-connected dense layers. Three layers have been instantiated with 32, 16, and 8 neurons and equipped with a ReLU (Rectified Linear Unit) activation function. The fourth layer is the latent space and it is instantiated as a dense layer (shared between the encoder and the decoder) including 4 neurons and equipped with a ReLU activation function. Symmetrically, the Decoder consists of three fully-connected dense layers with the same dimensions and activation functions. The output of the model is yielded by a Dense Layer with the same size of the input, and equipped with a Tanh activation function since the input is normalized in the range in $\{-1, 1\}$. The model is trained over 16 epochs with a batch size of 16. Finally, Adam is adopted as optimizer with learning rate $lr = 1e^{-4}$. As regards the ensemble parameters, we consider different values of k . In particular we tested the approach by including the last 3 and 5 base models while we consider data chunk size of $\sim 5,000$ instances. Notably, k can influence the capability of the model to keep the memory of past behaviors.

To evaluate the quality of the solution, we computed the following metrics. Let us define TP as the number of positive cases correctly classified, FP as the number of negative cases incorrectly classified as positive, FN as the number of positive cases incorrectly classified as negative, and TN as the number of negative cases correctly classified. Moreover, we considered:

- *Accuracy*: defined as the fraction of cases correctly classified, i.e., $\frac{TP+TN}{TP+FP+FN+TN}$;
- *Precision* and *Recall*: metrics used to estimate the detection capability of a methodology, since they provide a measurement of the accuracy in identifying upcoming attacks and avoiding false alarms. Specifically, *Precision* is defined as $\frac{TP}{TP+FP}$, while *Recall* as $\frac{TP}{TP+FN}$;
- *F-Measure*: condenses the overall system performance and is calculated as the harmonic mean of *Precision* and *Recall*.

Lastly, to perform experiments, we used a machine with 32 Gb RAM, an Intel i7-4790K CPU @4.00GHz and a 1Tb SSD drive.

7.3.3 Numerical Results

Since the outlierness threshold can influence the detection capability of the proposed approach, its sensitivity analysis is important to assess the robustness of the solution. Thus as a preliminary step, we investigated its impact. As the autoencoder model is trained only against legitimate data (i.e., clean traffic produced by IoT nodes), we computed the outlierness degree for each slot composing the training set. We then selected as the anomaly threshold the values corresponding to the 90th, 95th and 99th percentiles. A detailed breakdown is depicted in Figure 7.6. Moreover,

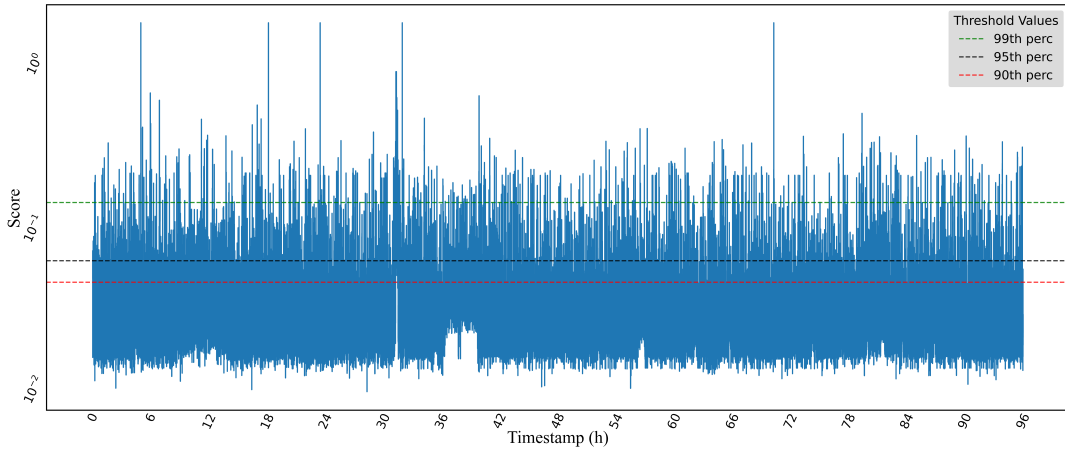


Figure 7.6: Training-set outlieriness with different threshold values.

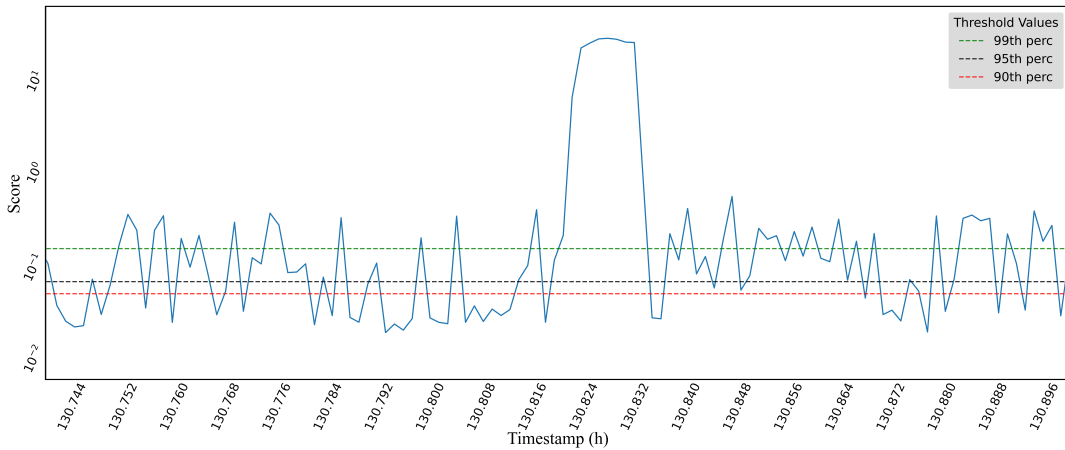


Figure 7.7: Test-set outlieriness.

Figure 7.7 portrays the distribution of the outlieriness degree for a window including a marked number of compromised time slots. As it can be seen, the outlieriness degree exhibits higher values than the ones reported in Figure 7.6. In some cases, the outlieriness is one order of magnitude higher than the outlieriness max value computed on the training set. This event represents the presence of a covert communications within the bulk of traffic, thus leading to a “deviation” in the output of the neural network.

Instead, Table 7.1 reports the results of the trials by comparing the performance of a single autoencoder with respect to the ensemble model. The performance metrics are computed by ranging different ensemble sizes and sensitivity thresholds. As regards the anomaly score, it is estimated by computing the reconstruction error for each instance contained in the training set and extracting the values corresponding to 90th, 95th and 99th percentiles. As expected, for all

Model Type	Ensemble Size	Threshold	Accuracy	Precision	Recall	F-Measure
Single Model	—	90 th perc.	0.882	0.743	0.993	0.850
Single Model	—	95 th perc.	0.921	0.822	0.976	0.893
Single Model	—	99 th perc.	0.936	0.942	0.865	0.902
Ensemble	3	90 th perc.	0.894	0.771	0.979	0.863
Ensemble	3	95 th perc.	0.947	0.902	0.948	0.924
Ensemble	3	99 th perc.	0.955	0.950	0.915	0.932
Ensemble	5	90 th perc.	0.890	0.764	0.977	0.858
Ensemble	5	95 th perc.	0.933	0.863	0.954	0.906
Ensemble	5	99 th perc.	0.952	0.944	0.911	0.927

Table 7.1: Experimental results for different outlier thresholds and ensemble size. Values have been selected by computing the outlier scores against the training set and by extracting the correspondent percentile values.

the model types, the usage of a looser threshold (e.g., the 90th percentile) allows for improving the probability of detection (i.e., the recall) but at the price of a higher number of false alarms. By contrast, a higher threshold (e.g., the 99th percentile) allows to limit the number of FP but a lesser recall value is obtained. Moreover, the adoption of the ensemble strategy improves the overall predictive capabilities of the detection systems. In fact, the best F-measure is obtained with an ensemble size equal to 3 and by considering the threshold value corresponding to the 99th percentile. Finally, the slight reduction of the predictive performance when increasing the ensemble size appears to be mainly due to the evolving nature of traffic characterizing IoT ecosystems. In fact, asynchronous activations of nodes, external triggers, or periodical synchronizations account for broad changes in traffic conditions. Therefore, recent data in some cases could be more informative for revealing an attack as the “past history” could not be representative of the actual traffic exchanged over the network.

Lastly, as regards the feasibility of deploying our approach in realistic settings, we point out that its resource footprint is very limited. In more detail, apart the training phase, which can be done offline, the average prediction time is 0.0132 ms. Another important aspect concerns the “stateless” nature of the approach. In fact, the used neural architecture performs the detection of covert communications by using information on the overall traffic (grouped in time slots), which prevents memory consumption due to the need of storing information with a per-flow granularity. Thus, the proposed approach should be considered suitable for being implemented in home gateways often used in production-quality IoT ecosystems.

7.4 Conclusions and Future Works

In this chapter, we showcased the use of ensemble of autoencoders for detecting network covert channels targeting IoT scenarios. Our approach has been designed to be lightweight and required a limited number of training examples at time to be effective. Numerical results demonstrated its effectiveness: the method can achieve a probability of detection (i.e., recall) of $\sim 91\%$ while exhibiting a good precision $\sim 95\%$. Future works aim at considering other types of network covert channels. To this aim, part of the ongoing research is devoted to develop some form of “intermediate” representations that can be used to develop a more general mechanisms for facing different threats, e.g., protocol-agnostic representations.

Next part of this Thesis will be devoted on investigating the use of AI to detect stegomalware from a different perspective, i.e., the use of images as the main carrier.

Part III

Detection of Malware Hidden in Digital Images

Chapter 8

Revealing Threats in Favicons via AI

As discussed in Chapter 1, the majority of recent attack campaigns primarily utilizes steganography to conceal malicious code or configuration files within innocent-looking images. Thus, being able to detect malicious payloads hidden in digital images is of prime importance to completely assess the security of modern scenarios and to mitigate the impact of the growing wave of stegomalware [MC15, CCC⁺20]. Therefore, from now to Chapter 10, this Thesis will deal with the problem of using AI to detect and mitigate realistic threats exploiting steganography to hide data in digital images. In fact, AI can support the arms race between attackers and defenders, especially to face new challenges like those offered by IoT [WHWS20] or zero-day threats [DFP20].

An important contribution of this part of the Thesis concerns the creation of suitable datasets built on malicious samples observed in production-quality scenarios and the performance of AI-based methods when used in realistic conditions. Specifically, in this chapter we want to spot Magento/MageCart-like attacks, while providing a more general framework. Hence, we consider the case of the widely-used LSB steganography to embed in favicons both malicious scripts and URLs (see, e.g., the case of the financial malware Vawtrak). To perform detection, we exploit DNNs, which turned out to be effective for analyzing and combining raw data from compromised images, as well as to update models when new attacks are available. To the best of our knowledge, the literature lacks of previous approaches using AI in such a scenario. Specifically, [AIS16] focuses on revealing URLs hidden in images via pattern-matching, while [PKKK16] considers favicons but exploits general heuristics. Besides, [SZZW19] addresses the general problem of locating steganographically-modified areas of an image, but does not consider real attacks or threat models.

The remainder of the chapter is organized as follows. Section 8.1 introduces the attack model and the general approach, Section 8.2 showcases the design of the DNNs, and Section 8.3 presents numerical results. Lastly, Section 8.4 concludes the chapter.

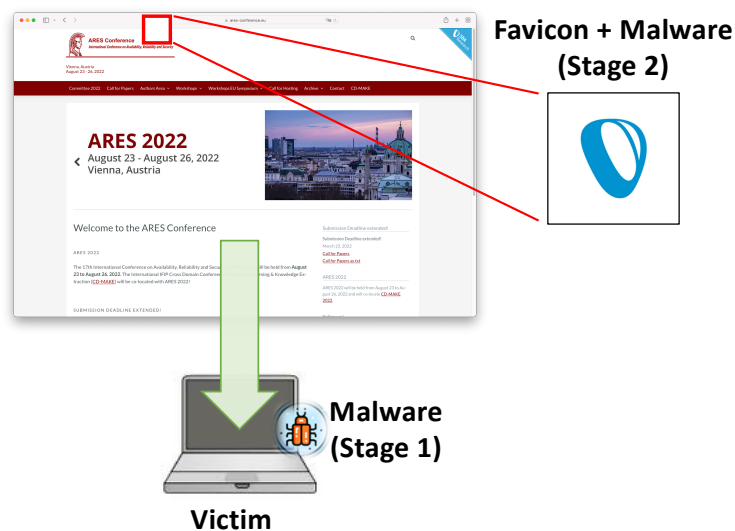


Figure 8.1: General attack model: a malware retrieves an additional payload hidden in a favicon.

8.1 Attack Model and Solution Approach

Figure 8.1 depicts the reference attack model and generalizes a scheme observed in stegomalware like MageCart and Vawtrack. In more detail, we assume a malware (denoted as “Stage 1”) wanting to retrieve an additional payload or a configuration file (denoted as “Stage 2”). The first stage of the threat could be already present in the host of the victim. For instance, a loader could have been dropped via phishing or embedded within a web page (e.g., hidden in an `iframe`) along with other offensive assets [CCC⁺20]. To avoid detection, prevent blockages, or make the forensics analysis harder, the malware extracts content hidden in a favicon. The latter could be provided by the original webserver or delivered by a third-party host controlled by the attacker [MC15, CCC⁺20].

The malicious payload is typically cloaked via a simple LSB steganography approach, i.e., the least significant bit(s) of the color channels of a pixel are replaced to encode arbitrary data without further considering surrounding values [HKR18, CCC⁺20]. Usually, “Stage 2” cannot detonate itself: rather, the contained information is decoded and used by previous stages. Without loss of generality, this chapter focuses on favicons, despite the other stages used in the attack chain. As discussed, detecting stegomalware leads to scarcely reusable approaches, since each hiding method is tightly coupled with the exploited carrier or the considered attack model. Therefore, creating general mechanisms that can be suitable for addressing a family of threats, without being limited to the specific implementation/version, is a challenging task. This is why, progresses of AI frameworks in learning from data or image processing should be considered prime tools towards the engineering of more general detection pipelines [ME21]. To detect the hidden data,

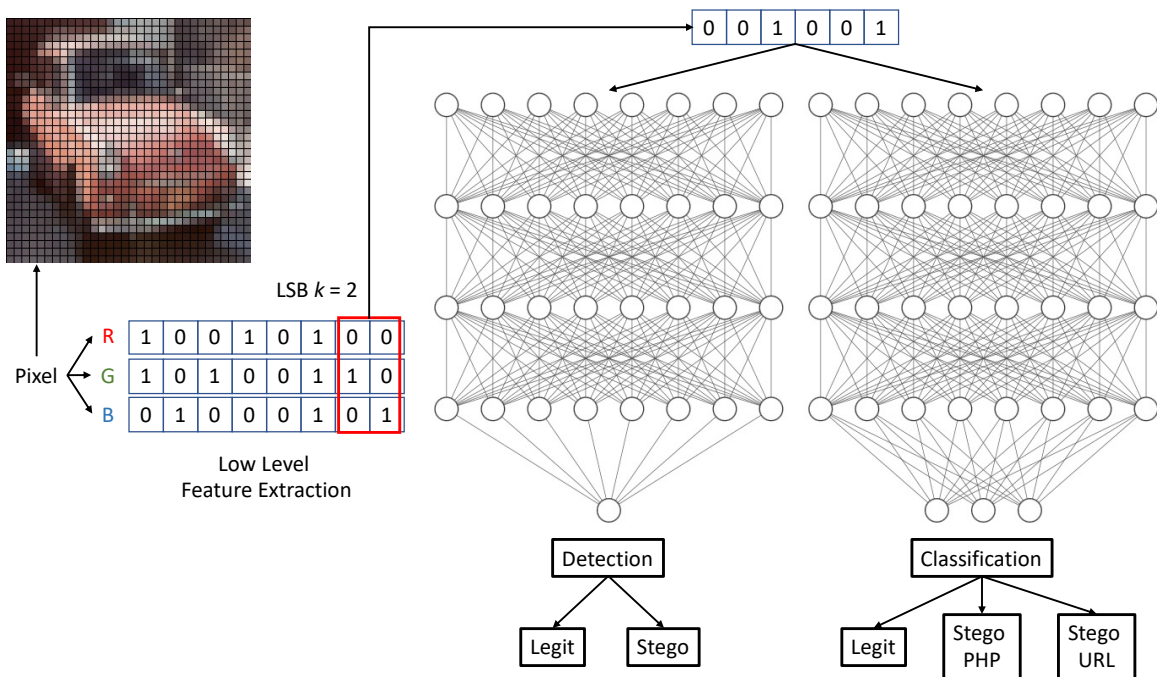


Figure 8.2: Methodological approach for the detection/classification of a stegomalware cloaked in a digital image via LSB steganography.

we consider DNNs, which represent an emerging approach for discovering compromised images via steganographic techniques (see, e.g., [HKR18] and the references therein). In more detail, DL allows to learn detection or classification models by directly combining raw low-level data, such as pixels, bits, and sensor data streams [LBH15]. Indeed, these models learn in a hierarchical fashion: several layers of non-linear processing units are stacked in a single network and each subsequent layer of the architecture can extract features with a higher level of abstraction compared to the previous one. Therefore, data abstractions and representations at different levels are automatically learned leading to effective solutions for analyzing and combining raw data from images. Moreover, discovered models can be updated incrementally or retrained only by considering the new data, thus overcoming limitations of classical approaches (see, e.g., [OCL17] and [HYM⁺20] for a comparison). In the following, we use the terms image and favicon in an interchangeable manner, except when doubts arise.

Figure 8.2 depicts the general approach. Concerning the input, we consider each image as a matrix with dimensions $X \times Y$. The pixel is the smallest manageable element of this matrix and stores color information, which is further decomposed in the three primary components, i.e., Red, Green and Blue. Each value represents the intensity of a given component that can vary in the $[0, 255]$ range. Accordingly, each pixel is stored by using three bytes, one for each primary color. We denote with N the size of the image computed as $N = X \times Y \times 3$.

As discussed in Chapter 1, the LSB technique can be used to hide malicious code or data by modifying the least significant bits of each color component of each pixel of the image (denoted as k and equal to 2 in the figure). If manipulations are under a given threshold, the resulting compromised image will not exhibit any visible alteration, i.e., pixels will appear as homogeneous with respect to the surrounding space [ZMCG21]. As a result, state-of-the-art approaches may fail in revealing the presence of the hidden content, leading to weak detection models unable to identify the slight differences between legitimate and compromised images. Therefore, we devised an approach that processes and analyses the k LSBs of each pixel of the given image. In essence, the k bits (represented in a vectorial manner) are offered to the DNN for the learning phase. The hidden layers combine this raw information and extract high-level discriminative features. As it will be discussed, this enables to effectively address both the detection of malicious data and its classification. We point out that, the use of a limited value for k allows to not dilute the amount of information contained in the attack pattern (see [MP21] for an example considering monochromatic images).

8.2 Detection via Deep Learning Models

To mitigate the impact of MageCart-like threats, we developed two DNNs for the detection and classification of various malicious payloads, i.e., PHP and URLs, hidden in favicons. However, the approach can be easily extended to other types of digital pictures. To this aim, we exploited the architecture presented in Figure 8.3(a), which provides accurate predictions for both tasks. The two models (i.e., detector and classifier) share the same topology, except for the last layer characterized by a task-specific activation function and a suitable number of outputs.

In particular, the DNNs consist of a stack of different subnets. The first layer, denoted as the `Input Handler`, simply propagates the input data, i.e., $N \times k$, through the rest of the architecture for further processing via subsequent layers of artificial neurons.

Then, both models include a variable number m of `SubNets` composed of three main parts: a `Fully-Connected Layer` equipped with a ReLU activation function [NH10], a `Batch Normalization Layer` for improving stability and performance of the current `Fully-Connected Layer` [IS15], and a `Dropout Layer` for mitigating the problems due to the overfitting [SHK⁺14]. The detailed structure of the generic subnet is depicted in Figure 8.3(b). Batch normalization allows to standardize the input with respect to the current data batch (specifically by considering the average μ and the variance σ of each feature) for the other layers of the neural network. Additionally, the dropout mechanism resets a random number of neurons during the training phase. As discussed in [HSK⁺12], dropout induces the neural network to behave in an ensemble flavor, i.e., the overall DNN can be regarded as a combination of the different subnetworks resulting from this random masking, which disables some paths of the whole DNN.

Finally, the `Output Layer` is instantiated on the basis of the specific task. For the case of

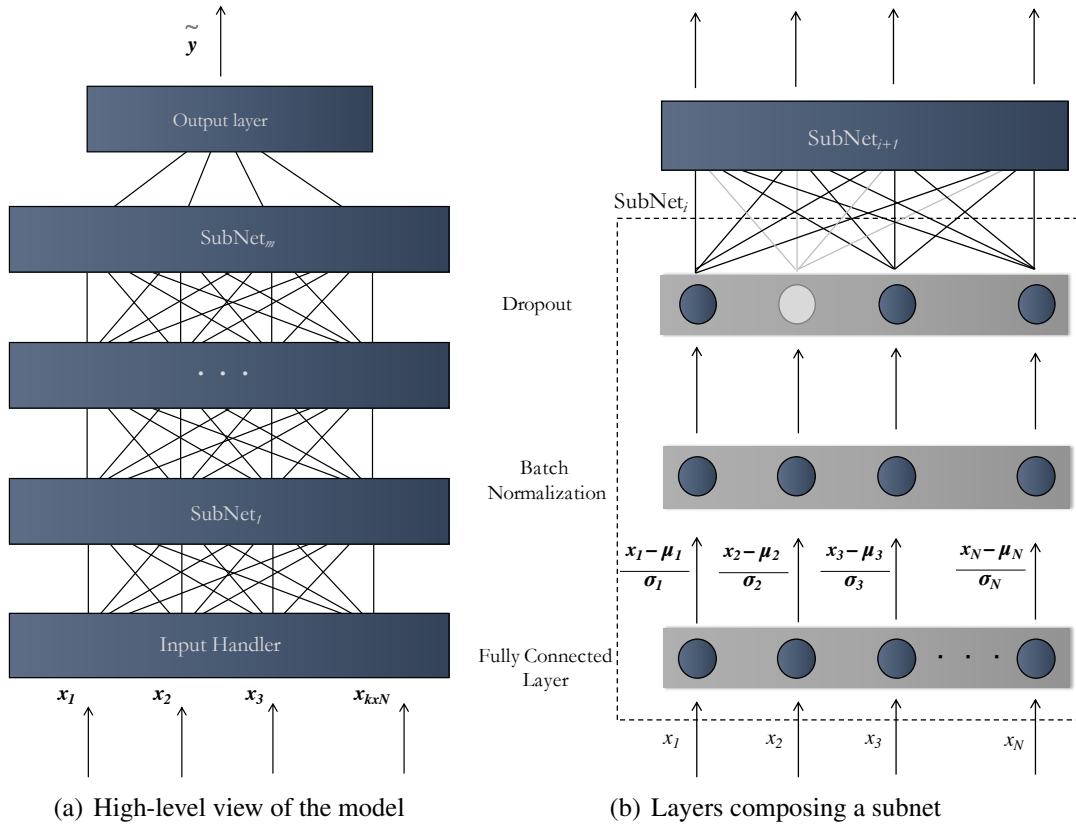


Figure 8.3: Model architecture of used DNNs: overview (a) and details of the internal structure for each subnet (b).

detection, it is endowed with a *sigmoid* activation function [GMR18], which maps any given data instance $\mathbf{x} = \langle x_1, \dots, x_{k \times N} \rangle$ to an anomaly score \tilde{y} (i.e., the estimate of the probability that x is an image containing some hidden information). By contrast, when used for the classification task, the layer is instantiated with c neurons (one for each class) and equipped with a *softmax* activation function [GMR18]. In more detail, the network is learned on a set $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \dots, (\mathbf{x}_n, \mathbf{y}_n)\}$, where \mathbf{x}_i represents the k -LSB-based representation of the image, while \mathbf{y} is the label associated with the image. For the detection task, \mathbf{y} is a binary value indicating the legitimate/compromised nature of the image. Instead, for the classification task, \mathbf{y} is an one hot encoded representation of the c classes, i.e., clean images, and favicons containing a malicious PHP script or URL. The learning phase optimizes the network weights by minimizing a suitable loss function. For the detection task, the *Binary Crossentropy* (BCE) is used as the loss function during the learning phase and it is defined as follow:

$$BCE(\mathbf{y}, \tilde{\mathbf{y}}) = -\frac{1}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} \mathbf{y}_i \log \tilde{\mathbf{y}}_i + (1 - \mathbf{y}_i) \log(1 - \tilde{\mathbf{y}}_i).$$

Instead, for the classification task, the *Categorical Crossentropy* (CCE) is used as the loss function and it is computed as follow:

$$CCE(\mathbf{y}, \tilde{\mathbf{y}}) = -\sum_{i=1}^{|\mathcal{D}|} \mathbf{y}_i \log \tilde{\mathbf{y}}_i.$$

8.3 Performance Evaluation

In this section we present the performance of the DNN-based approach for detecting MageCart-like threats. First, we showcase the dataset, as well as the considered parameters and performance metrics. Then, we discuss numerical results.

8.3.1 Dataset Preparation

To evaluate the effectiveness of our approach, we prepared a dataset starting from 60,000 32x32 favicons (i.e., $X, Y = 32$) borrowed from the CIFAR-10¹ collection. To conceal secrets, we used LSBSteg², which embeds data in the LSB of each RGB component of a given pixel. If the payload exceeds the available capacity, the tool will use other bits until the embedding is complete. To have a suitable tradeoff between the visual quality of the favicon (i.e., the undetectability of the steganographic scheme) and the cloaking volume, as well as to avoid a loss of generality, in this work we only consider payloads fitting in 1 bit per RGB channel.

We then considered two different payloads. First, since PHP is one of the the most relevant source of exploits [HY21], we used PHP malicious scripts from the PHP-Malware-Collection³. In order to prevent trivial detection caused by visible artifacts, we selected PHP scripts (e.g., webshells and backdoors) that can be hidden in favicons by only using 1 bit per RGB channel of each pixel. This led to 28 different malicious PHP scripts. This is not a major limitation: in fact, real scripts mainly exploit known CVEs or bugs, which are limited. To have a broader scenario, we also taken into account malicious URLs, which are often cloaked and at the basis of multi-stage loading schemes [CCC⁺20]. In this case, we shortlisted 1,000 real, malicious entries from

¹<https://www.cs.toronto.edu/~kriz/cifar.html> [Last Accessed, October 2022].

²<https://github.com/RobinDavid/LSB-Steganography> [Last Accessed, October 2022].

³<https://github.com/marcocesarato/PHP-Malware-Collection> [Last Accessed, October 2022].

the URLhaus⁴ collection. Since real malware samples often mix hard-coded IP addresses and DNS entries, we considered 500 numeric entries and 500 URLs, respectively. In the following, we will simply refer to such quantities as URLs.

As a result, we obtained 240,000 images containing malicious URLs (each image has been the target of LSB steganography for four different entries), and 60,000 images embedded with a malicious PHP script, which has been randomly-selected with a uniform distribution. The resulting dataset is then divided into train, test, and validation sets. The train set is composed of 180,000 favicons (30,000 legitimate images, 120,000 obtained by combining each legitimate image with 4 different URLs, and 30,000 obtained by combining each legitimate image with a randomly-picked PHP script). The validation set consists of 90,000 favicons, divided into 15,000 legitimate images, 60,000 images containing different URLs, and 15,000 images generated using the PHP scripts. The same proportion applies to the test set.

To further evaluate our approach, we prepared an additional dataset containing URLs and PHP scripts completely unknown to the DNN. Specifically, we considered 1,000 new DNS/IP entries borrowed again from the URLhaus collection. Recalling that realistic, malicious PHP scripts that can be hidden in favicons exist in a very limited quantity, we decided to model a “lateral movement” attacking scheme. Thus, we encoded the original 28 scripts in Base64 in order to consider a sort of basic “obfuscation” mechanism. We point out that, part of our ongoing research is devoted to use more realistic schemes, such as XORring with a pseudo-random sequence. Since such encoding inflates their sizes, we only used scripts still able to fit the favicon with 1-bit LSB steganography, leading to 21 valid PHP threats. The resulting set is composed of 90,000 favicons: 15,000 legitimate images, 60,000 images with URLs, and 15,000 images with Base64-encoded PHP scripts.

8.3.2 Parameters, Evaluation Metrics, and Testbed

To evaluate the idea, we implemented a prototype⁵ in Python and the Tensorflow library. Specifically, each model is composed of $m = 3$ subnets. Each fully-connected layer exploits a ReLU activation function with 128 neurons and a dropout percentage of 2.5%. The RMSprop is used as the optimizer with an initial learning rate of 0.001. Finally, each model is trained over 20 epochs and the value of the batch size is equal to 256. In our experiments, we considered $k \in \{1, 2, 3\}$ to understand whether the number of bit planes composing the image plays a role. We point out that, even if the DNN could be used to find the best value of k , i.e., it can be used to “choose” the most relevant features to drive the detection process, this requires to pay a price in terms of resources that can be unfeasible in the presence of large volumes of data.

⁴<https://urlhaus.abuse.ch> [Last Accessed, October 2022].

⁵<https://github.com/massimo-guarascio/FaviconStegoDetection> [Last Accessed, October 2022].

k	Task	Test Type	Accuracy	F-Measure	AUC-PR
1	Detection	known payload	1.000	1.000	1.000
		unseen/obfuscated payload	0.993	0.987	1.000
	Classification	known payload	1.000	1.000	1.000
		unseen/obfuscated payload	0.907	0.811	0.995
2	Detection	known payload	1.000	1.000	1.000
		unseen/obfuscated payload	0.990	0.983	1.000
	Classification	known payload	1.000	1.000	1.000
		unseen/obfuscated payload	0.911	0.843	0.999
3	Detection	known payload	1.000	1.000	1.000
		unseen/obfuscated payload	1.000	0.999	1.000
	Classification	known payload	1.000	1.000	1.000
		unseen/obfuscated payload	0.912	0.834	1.000

Table 8.1: Experimental results on different test cases by ranging different values of k .

Apart the performance metrics already introduced in Chapter 7 for spotting channels in IoT environments, i.e., *Accuracy*, *Recall*, *Precision*, and *F-Measure*, in this chapter we also consider the *AUC-PR*, that is the area under the Precision-Recall curve computed considering the different class probability values. Although the F-Measure and AUC-PR are defined for a binary classification problem, they can be extended for a multi-class scenario by averaging, the obtained results for each class. The literature provides two approaches named *macro* and *micro* averaging [SL09]. In the former, the performance measure is computed for each class and then it is averaged, whereas in the latter, the cumulative sum of the counts of various true/false positive/negative is computed, and then the overall measure is calculated. While macro-averaging weights all classes equally, micro-averaging favors bigger classes. Since in our dataset the positive attack examples overwhelm the legitimate ones, we adopted a macro-averaging strategy.

Lastly, to perform experiments we used a machine equipped with an Intel Core I9-9980HK CPU @2.40GHz and 32 GB RAM. The validation set has been exploited to select the model guaranteeing the best loss value from the training phase.

8.3.3 Numerical Results

Table 8.1 reports experimental results obtained by taking into account different scenarios and by varying the number k of bits used to detect/classify payloads hidden within favicons. Specifically, detection and classification tasks have been evaluated by considering two different test

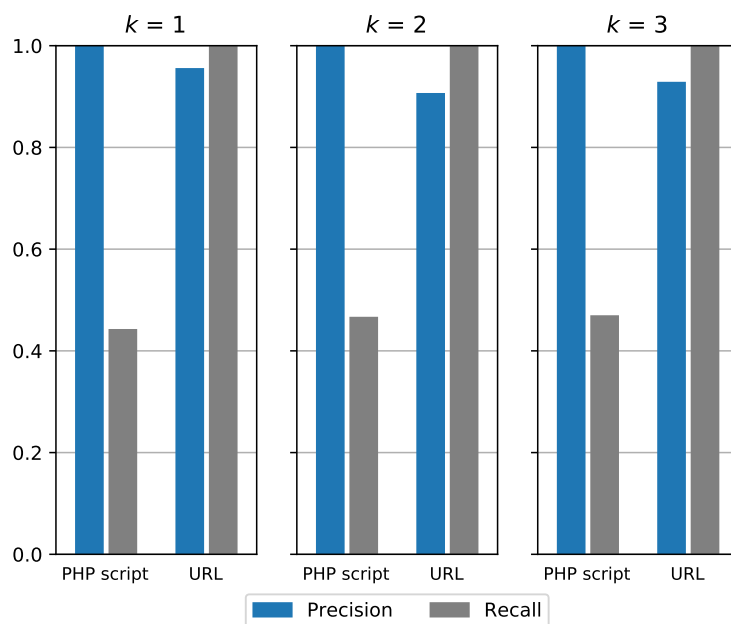


Figure 8.4: Precision and Recall results for each payload class obtained by varying different values of k in unseen/obfuscated payload setting.

cases. The first deals with favicons embedding known payloads (i.e., those contained in the training set), whereas the second addresses favicons cloaking “unseen” URLs or Base64-obfuscated PHP scripts. As shown, results highlight the capability of the approach to provide very accurate predictions ($\sim 100\%$ in terms of all the metrics) for both detection and classification tasks. However, the last test case is more challenging and the number of bits k used by the DNN to encode the image affects the prediction, especially when detecting the presence of malicious payloads. In particular, the initial value of 0.987 for the averaged F-Measure increases to 0.999, when k varies.

For the second round of tests, we conducted a more detailed analysis. Specifically, considering only unseen URLs or obfuscated PHP scripts allowed to model the presence of an attacker aware of the AI-based countermeasure and thus deploying elusive techniques, such as the generation of new URLs via domain fronting and the adoption of different content encoding as “obfuscation” mechanisms [CCC⁺20]. Figure 8.4 depicts precision and recall values for each type of malicious payload while using different values of k (we point out that, the legitimate class has been omitted since we want to focus on the behavior of attack cases). Since the recall quantifies the capability of the model to detect examples belonging to a specific class, the precision provides an estimate of the prediction accuracy. In particular, high values for the precision lead to a low false alarm rate and limits the number of misclassifications. Always referring to Figure 8.4, the model exhibits excellent performance in correctly classifying favicons containing URLs, both in terms of precision and recall, whereas $\sim 50\%$ of the favicons embedding PHP scripts is erroneously clas-

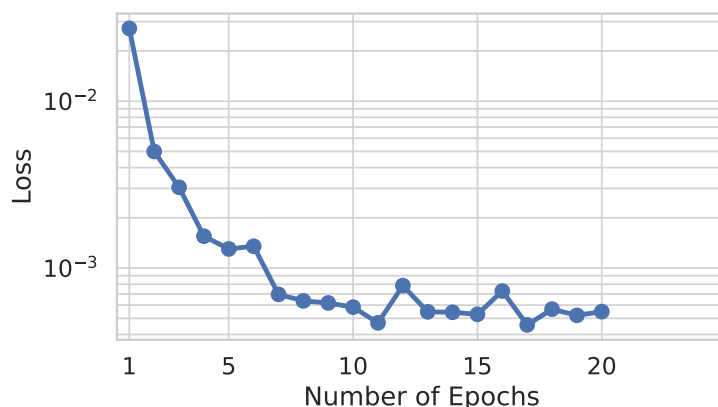


Figure 8.5: Loss behavior on the training set for the classification task (logarithmic scale).

sified and assigned to the other two classes, i.e., legitimate and URLs. However, the high values of precision (for each value of k) denote the robustness of the model against false alarms, i.e., a favicon assigned to this class has a high probability of containing some malicious payload.

We also studied the number of iterations (i.e., epochs) required for convergence of the training algorithm. In particular, Figure 8.5 shows the training loss value computed per epoch. As it can be seen, already after a limited number of iterations (~ 7), the loss exhibits a stable behavior meaning that the algorithm has been able to discover an accurate predictive model.

Finally, to understand whether the proposed approach can be deployed in production-quality scenarios as well as to be implemented in “as-a-Service” paradigm, we performed tests on its computational footprint. In this vein, it turned out that, processing a batch of 32 images requires ~ 4 ms on average, without the need of a dedicated hardware like GPUs. For all these reasons, our DNN-based architecture can be considered effective for detecting malicious favicons in medium-sized networks or to be directly implemented through softwarization (e.g., in a containerized micro service) along with the webserver without penalizing the Quality of Experience perceived by end users.

8.4 Conclusions and Future Works

In this chapter, we presented a framework based on DNNs for detecting MageCart-like threats able to cloak malicious payloads in favicons via LSB steganography. Our approach has been also designed to classify the type of payload, i.e., malicious URLs and PHP scripts. Results showcases its ability to detect the $\sim 100\%$ of compromised favicons (with examples provided during the learning phase), as well as $\sim 90\%$ of payloads obfuscated via a Base64 encoding. Owing to a reduced computational footprint, our method could be deployed to inspect online favicons ex-

changed in medium-sized networks or delivered by high-traffic web servers. A current limitation of our approach is its ability of solely spotting payloads directly hidden in bitplanes. Thus, future works aim at extending it to reveal payloads cloaked within non-zero DCT coefficients.

In the next chapter, we will investigate the use of DNNs to detect/classify other payloads (e.g., HTML or PowerShell) hidden in full-sized icons.

Chapter 9

Revealing Threats in Icons via AI

The boundaries between mobile and desktop applications are progressively blurring. For instance, iOS/iPadOS applications can run over macOS by using a transparent and lean emulation layer, and Android software can be deployed in set top boxes, mobile devices, and various appliances with minimal modifications. Moreover, the diffusion of the Software-as-a-Service paradigm leads to a common back-end, which can be shared by different users and classes of devices [BHGG18]. As a consequence, malware and threat actors can target a limitless population of devices often sharing the underlying hardware architecture or a non-negligible amount of software assets, such as libraries or network protocols. Even if the uncontrolled diffusion of malicious software characterized by a “write once, attack anywhere” nature could be not an imminent danger, the use of digital images should be considered an important feature unifying the majority of devices and hosts connected to the Internet, especially when considering stegomalware.

Therefore, this chapter advances with respect to Chapter 8 and further investigates on image steganography when considering the massification of mobile devices, the large-scale deployment of IoT nodes and smart frameworks, as well as the fragmentation of various software sources (e.g., ad-hoc and official stores vs side-loaded applications). Specifically, it deals with an ecosystem for the detection of steganographic threats targeting digital images in a general and efficient manner. In more detail, we propose the usage of a similar DNN architecture to reveal the presence of a wide array of malicious payloads hidden in larger digital images. Moreover, we evaluate the cases when the attacker tries to implement basic evasion techniques, i.e., via obfuscation, compression or alternative encoding schemes.

The contribution of this chapter is twofold: it introduces a framework using AI to detect different malware leveraging steganography, and it showcases the creation of a realistic dataset for modeling various threats and attack schemes.

The remainder of the chapter is organized as follows. Section 9.1 introduces the attack model as well as background information on malware and its mitigation via AI techniques, Section

9.2 presents the proposed ecosystem and the detection approach, and Section 9.3 showcases numerical results obtained via a thorough performance evaluation campaign. Lastly, Section 9.4 concludes the chapter.

9.1 Attack Model

The general attack model considered in this work deals with an attacker wanting to cloak a malicious payload into a digital image to bypass a secure perimeter. As an example, a threat actor wants to drop a payload on the host of the victim or build a stealth chain to reduce the effectiveness of forensics investigations [MC15]. In general, infiltrating a malicious content or making difficult to locate the source of the attack can be done by using different techniques. For instance, in the case of platforms based on Android, malware can be repackaged within applications to evade detection algorithms, even exploiting machine learning [CLW⁺19]. Instead, when considering simpler targets like IoT nodes, malicious routines can be directly injected or obfuscated in binaries [YY10].

Once the payload is hidden in the image, the malware has to be distributed to the victim. As today, different attack vectors exist, but the most popular are [CCC⁺20]: *i*) the payload is sent as an email attachment, for instance via phishing campaigns; *ii*) the target is decoyed to retrieve some software, for instance by clicking a malicious link or tricked with social engineering techniques; *iii*) a third-party malicious stage already running on the device of the victim retrieves the payload from a remote server; *iv*) the payload is cloaked in an asset delivered via a publicly-accessible platform, such as a store or a web server. According to the used method, different checks and countermeasures are enforced within security services implemented through a variety of architectural/functional blueprints. In more detail, for attack vectors *i*) and *ii*) a local antivirus could inspect files searching for known signatures. In the case of *iii*), a firewall or an IDS could block/spot the network conversation between the remote C&C server and the compromised node. Lastly, for the case *iv*), security checks enforced in a web server, a cloud provider, or an application store could spot the presence of malicious assets packed within in-line objects composing a web page or a `.dll` bundled with an application. To have a reference use case, Figure 9.1 depicts the considered general attack model. Specifically, we consider an attacker hiding a malicious content in a digital image (i.e., a high-resolution icon) by exploiting steganography. Similarly to what discussed in Chapter 8, we consider the use of LSB steganography. When the malicious payload is cloaked via LSB, it can be delivered via methods *i*)-*iv*), each one aiming at bypassing a specific security service.

With image processing, we refer to the technical analysis of an image through complex algorithms, which are usually exploited to address a variety of tasks. Improving the human understanding on information content of an image as well as extracting, storing and transmitting pictorial information [GW18] are typical examples of problems that can be tackled via image

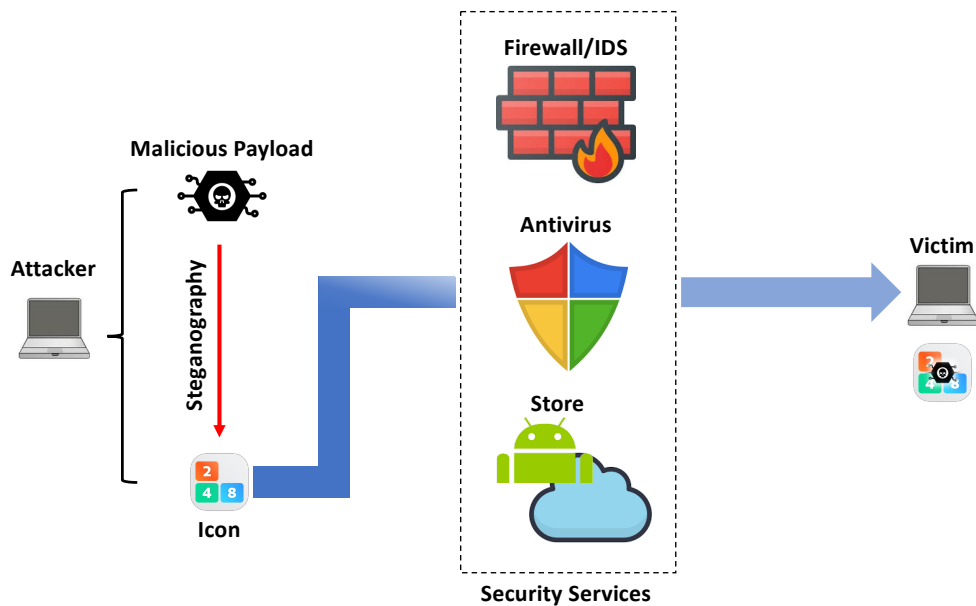


Figure 9.1: General attack model considered to design the ecosystem for the mitigation of steganographic threats.

processing. The recent advances in AI and (in particular) in machine learning allowed to further boost the capabilities of traditional frameworks and generate reliable and effective models. Among the others, the DL paradigm [LBH15] is particularly suited to reveal the presence of malicious information in digital images. Indeed, an emerging solution for identifying hidden contents relies on the usage of DNN, which allows for detecting and classifying compromised images through steganographic tools (see, e.g., [HKR18]). Hence, accurate detection and classification models can be directly learned from raw low-level data (e.g., pixels, bits, and sensor data streams) by using approaches exploiting DL methodologies. Recalling that these models can learn in hierarchical fashion, data abstractions and representations at different levels are automatically learned leading to effective solutions for analyzing and combining raw data. This is especially true for the case of processing digital images, for instance, high-resolution icons used in several commercial software such as Android. Moreover, discovered models can be updated incrementally or retrained only by considering new data coming from the observation of additional threats or variants of well-known attacks.

9.2 Detection Approach

Even if many concepts have been already introduced in Section 8.2, for the sake of completeness we introduce again the software architecture of our ecosystem for revealing the presence

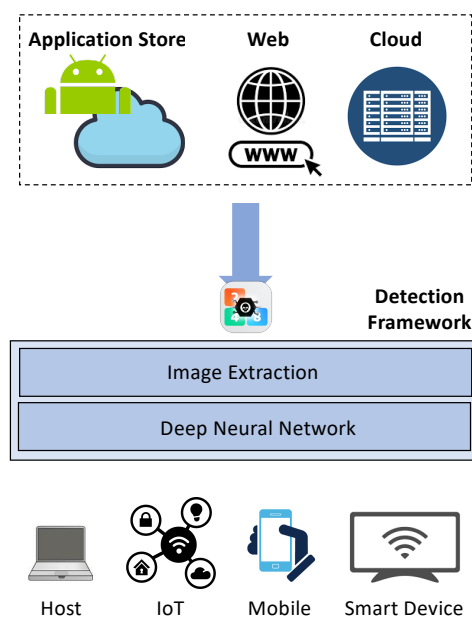


Figure 9.2: Layered architecture of our approach for detecting malware targeting malicious images.

of steganographic threats targeting images. Then, we will discuss the design of the detection framework based on DNNs.

9.2.1 Architectural Blueprint

As discussed, the proposed detection framework aims at revealing the presence of malicious payloads embedded in images contained in different software artifacts, which can be retrieved from different sources. Figure 9.2 portrays the layered software architecture. As a first step, the detection framework “intercepts” the content and extract the digital image(s). For instance, this could require to extract assets from the resource bundle of an application or capture in-line objects composing an HTML hypertext. As soon as the image is retrieved, the detection logic exploiting a DNN checks for hidden contents. Malicious images can be discarded or an alarm can be raised.

In general, this type of test can be done in two different portions of the network. In the first case, the detection framework can be deployed at the border of the network closer to devices to be protected. As a possible example, it can be implemented in edge nodes, home/smart gateways, or as an ad-hoc service running over local appliances. If performance is not a tight constraint, the detection can also happen directly in end nodes. In the second case, the framework can be

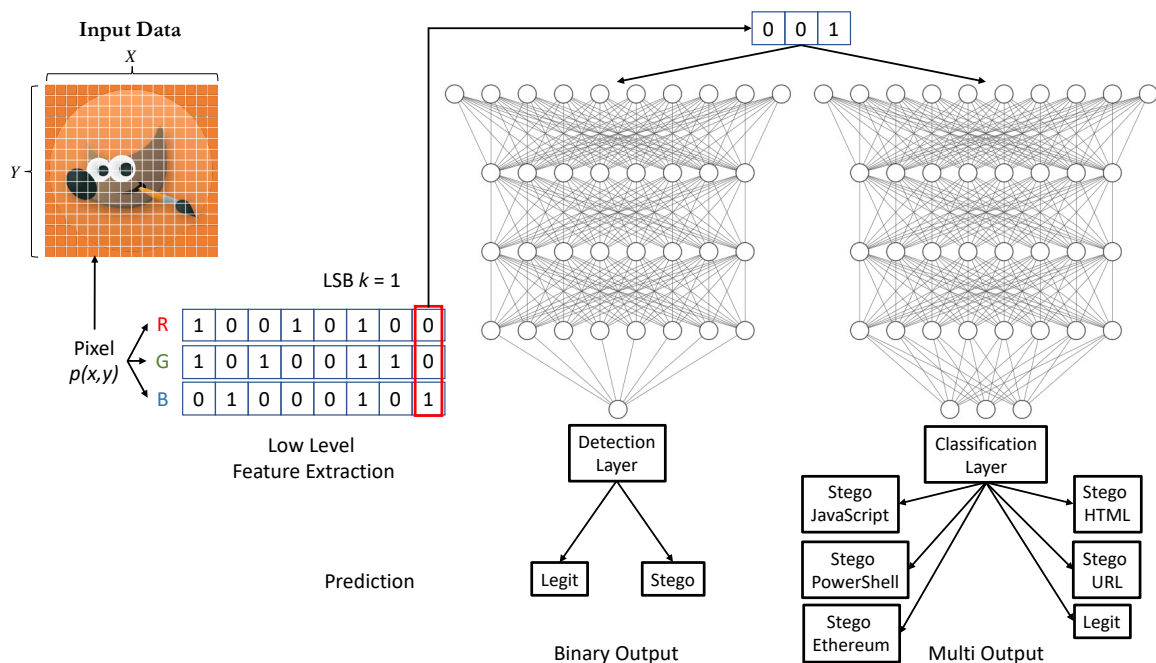


Figure 9.3: Methodological approach for the detection/classification of a stegomalware cloaked in a digital image via LSB steganography.

deployed within the service to protect. For instance, it can be implemented as a software layer inspecting applications submitted by a developer before they become publicly available, or can be a component periodically checking assets stored in a datacenter.

Despite the placement, the detection framework should be properly trained and its model should be periodically updated. When considering edge-like deployments, such requirements could be too narrow. Thus, data gathering and training of the neural network can be done in a centralized manner in order to deliver only the model to devices with limited capabilities.

9.2.2 Hidden Content Detection Approach

Figure 9.3 shows the general approach adopted to address the problem of revealing images targeted via steganographic or information-hiding-capable techniques. The core architecture is the one used in Chapter 8 and it is presented again for the sake of completeness. Basically, the input data are images, i.e., matrices with dimension $X \times Y$. Each image is composed of pixels, the smallest manageable element of these matrices storing information about the color. The color of each pixel is obtained by combining the three RGB components. As it will be detailed later, in this work we focus on high-resolution icons as they offer a sort of “unified playground” for

various threats. Indeed, this does not account for a loss of generality, as the approach can be applied and scaled also to address regular-sized images. Thus, in the following, we consider that the values associated to RGB components represents the intensity of that color and ranges in the interval $[0, 255]$. Specifically, three bytes are used to store the intensity value for each primary color. Hereinafter, we denote with N the size of the image computed as $N = X \times Y \times 3$.

As previously hinted, LSB steganography represents a prominent approach to hide malicious code or data in legitimate pictures by changing the value of the (k) least significant bit(s) of each color composing the pixel of the image (see, Figure 9.3 for the case of $k = 1$). It is worth noting that, when only a limited number of changes are performed on the image, it will not exhibit any visible alteration, i.e., pixels will appear as homogeneous with respect to the surrounding elements [ZMCG21]. Therefore, a high number of the approaches proposed in literature fail in detecting the presence of hidden content as they produce weak detection models unable to discover the slight differences between licit and compromised contents.

To overcome the limitations of traditional frameworks, in this work we designed a machine-learning-based solution that focuses on processing and analyzing the k LSBs of the images under investigation. Basically, a vectorial representation is yielded by extracting the k least significant bits of each pixel, hence this representation is used to feed the learning phase of a DNN. The raw information is automatically combined by the DNN hidden layers that allows for extracting high-level discriminative features.

The proposed approach permits to deal with both the main issues investigated in this work i.e., the effective detection of malicious data and its classification. Notably, the predictive performance of the detection model are generally not affected by the number k of LSBs analyzed as demonstrated in [MP21] for monochromatic images.

9.2.3 Neural Detector Architecture

To mitigate the risks arising from threats leveraging information hiding and steganography, we devised two (deep) neural models for their detection and classification. Specifically, we adopted the deep architectures depicted in Figure 9.4, which allow for yielding accurate predictions for both tasks. Basically, the two DNNs (i.e., detector and classifier) share the same backbone (i.e., same hidden layers and activation functions), but differ for the output layer, which is instantiated with a different number of neurons and activation function on the basis of the specific task to address. Essentially, our solution is composed of a stack of several subnets. The first layer has the role of handler for the input provided to the network (denoted in Figure 9.4 as `Input Handler`) and it propagates the raw data to the subsequent layers of the DNN for further processing. The size of this level is $k \times N$ i.e., the product between the number of least significant bits to analyze and the size of the image.

Both networks are composed of a variable number m of `SubNets` obtained by stacking three

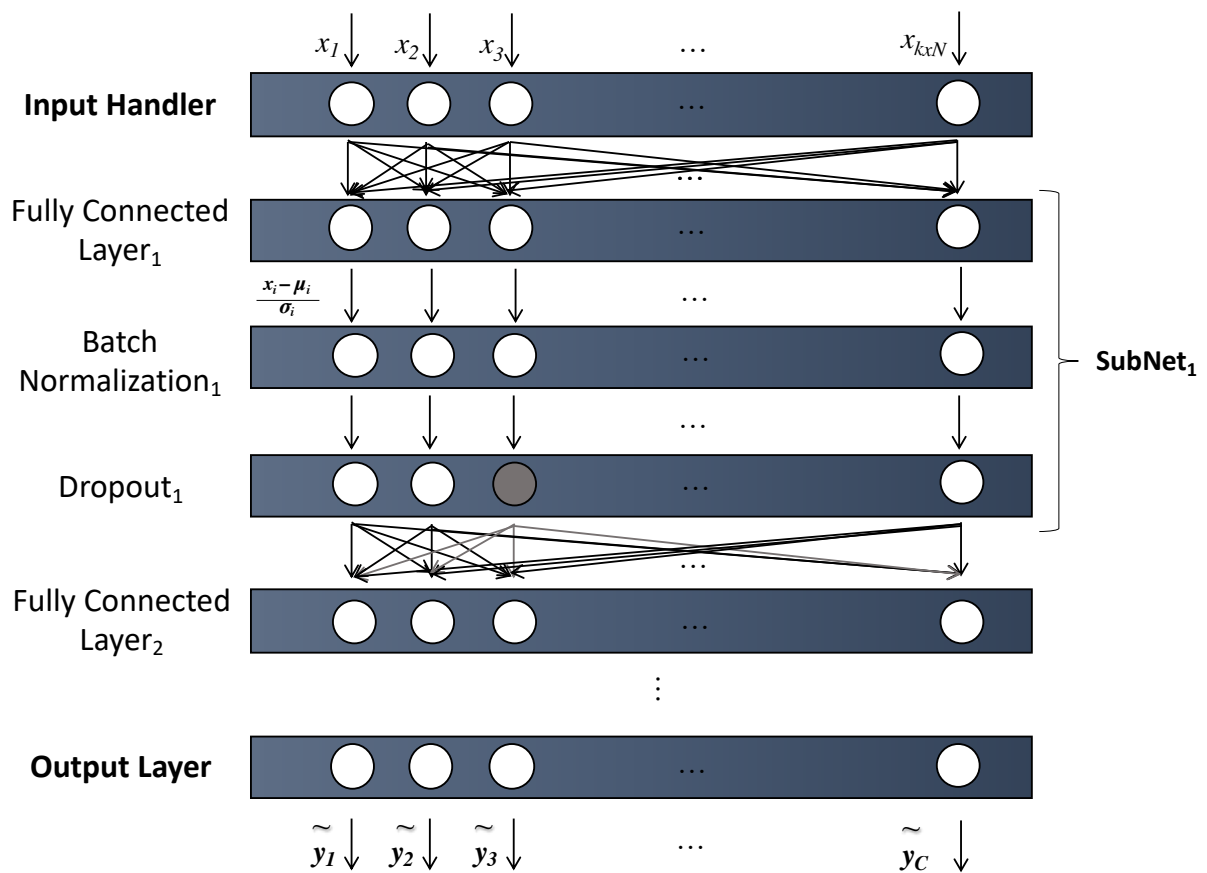


Figure 9.4: Neural architecture for hidden content detection and classification.

main components: (i) on top, a fully-connected dense layer (equipped with a ReLU activation function [NH10]) is instantiated, (ii) then, a batch-normalization layer is stacked to the previous one in order to improve the stability of the learning phase and to boost the model performance, and finally, (iii) to mitigate the risk of overfitting, a dropout layer is added to the subnet [SHK⁺14].

As an example, Figure 9.4 illustrates the overall model architecture. The first instance of this specific configuration has been labeled as SubNet₁. In more detail, the Batch Normalization implements the role of standardizing the data to be offered to the subsequent layers of the DNN with respect to the current batch (by considering the average μ and the variance σ of each input), whereas a reset of a random number of neurons during the learning stage is performed by applying the dropout mechanism. As pinpointed in [HSK⁺12], the adoption of a dropout technique induces in the DNN a behavior similar to an ensemble model: in a nutshell, the overall output of the whole neural network can be considered as the combination of different subnetworks resulting from this random masking, which disables some paths of the neural

architecture.

Finally, the number of neurons and the type of activation function of the `Output Layer` depends on the specific task to address. As regards the detection, a single neuron providing the attack probability score is required. In particular, the `Output Layer` is equipped with a *sigmoid* activation function [GMR18], which maps any given data instance $\bar{\mathbf{x}} = \langle \mathbf{x}_1, \dots, \mathbf{x}_{k \times N} \rangle$ to an anomaly score $\tilde{\mathbf{y}}$ (i.e., the estimate of the probability that x is an image containing some hidden information).

By contrast, for the classification task the `Output Layer` will include C neurons (one for each class) and will be equipped with a *softmax* activation function [GMR18]. Basically, we can consider the detection task as a sub-case of classification where $C = 1$. The proposed neural model is trained against a set $\mathcal{D} = \{(\bar{\mathbf{x}}_1, \mathbf{y}_1), (\bar{\mathbf{x}}_2, \mathbf{y}_2), \dots, (\bar{\mathbf{x}}_D, \mathbf{y}_D)\}$, where \mathbf{x}_i is the k -LSB-based representation of the image while \mathbf{y} is the class of the image. In particular, as concerns the detection \mathbf{y} takes a binary value specifying the legitimate/compromised nature of the image. For the classification task, an One-hot Encoding based on C classes is used to model the different labels each one indicating a specific malicious payloads. As it will be detailed later, in our work we considered C classes representing “clean” images and images cloaking JavaScript, HTML, PowerShell, Ethereum wallets, and URL/IP addresses. Finally, the training stage is responsible for optimizing the network weights by minimizing the same loss function presented in Chapter 8. In particular, for the detection task, the BCE is exploited to compute the network weights, which is defined as follow:

$$BCE(\mathbf{y}, \tilde{\mathbf{y}}) = -\frac{1}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} \mathbf{y}_i \log \tilde{\mathbf{y}}_i + (1 - \mathbf{y}_i) \log(1 - \tilde{\mathbf{y}}_i).$$

By contrast, the CCE is adopted for the classification task and it is calculated as follow:

$$CCE(\mathbf{y}, \tilde{\mathbf{y}}) = -\sum_{i=1}^{|\mathcal{D}|} \mathbf{y}_i \log \tilde{\mathbf{y}}_i.$$

9.3 Performance Evaluation

In this section, we preliminary present how the dataset has been prepared to model the attack template introduced in Section 9.1. Then, we showcase numerical results.

9.3.1 Design of Attacks and Dataset Preparation

To model a wide range of threats leveraging steganography with emphasis to real attacks presented in Section 1.4, we consider an attacker cloaking different malicious payloads within high-resolution icons. This can be representative of an APT or a Crime-as-a-Service toolkit offering cross-platform offensive functionalities or generic hiding capabilities. Moreover, icons are an ubiquitous digital asset deployed in software running in mobile nodes (e.g., Android and iOS devices), tablets and set top boxes (e.g., Windows and Android ecosystems), as well as in desktops. In general, icons are provided in different sizes or dynamically scaled by the guest OS according to the resolution or the intended usages, for instance to identify file types or to show applications on the desktop/dashboard. Thus, without loss of generality and to take into account possible future scenarios, we considered an attacker targeting icons of a size of 512×512 pixels.

Concerning payloads, we want to model threats using steganography or information hiding to conceal different malicious assets, as it happens in real-world attack campaigns and APTs. To not limit our investigation and to reflect the increasing trend of endowing malware with some stealthy capabilities, we utilized the following realistic malicious payloads [MC15, CCC⁺20]:

- JavaScript code¹: threats that can target victims in a cross-platform manner. For instance, such scripts can be used to retrieve an additional payload, de-obfuscate weaponized contents stored in the filesystem, or implement file-less malware;
- Obfuscated JavaScript in HTML²: many scripts are usually obfuscated within an hypertext. On one hand, this allows to trigger their execution when used in Web-based attack chains. On the other hand, scripts can be concealed within chunks of text or comments, via a wide array of text-based obfuscation mechanisms. Hence, they are expected to proliferate for making detection and forensics attempts harder;
- PowerShell scripts³: malicious PowerShell code has been observed in many steganographic malware. For instance, the Invoke-PSImage technique used to hide PowerShell contents in images has been at the basis of several attack campaigns, such as those against the Pyeongchang Olympic Games or for the diffusion of Bandoon malware [HYM⁺20];
- Ethereum Addresses⁴: ransomware and cryptojackers in many cases contain data to programmatically reach a remote wallet or to instruct a victim for paying the ransom. To avoid

¹JavaScript Malware Collection. Online: <https://github.com/HynekPettrak/javascript-malware-collection> [Last Accessed, October 2022].

²Malicious Javascript Dataset. Online: <https://github.com/geeksonsecurity/js-malicious-dataset> [Last Accessed, October 2022].

³PowerShell dataset. Online: <https://github.com/denisugarte/PowerDrive> [Last Accessed, October 2022].

⁴Ethereum-lists. Online: <https://github.com/MyEtherWallet/ethereum-lists> [Last Accessed, October 2022].

	Train	Test	Validation
Clean	4,000	2,000	2,000
JavaScript	2,363	1,188	1,214
JavaScript in HTML	2,284	1,167	1,162
PowerShell	2,468	1,164	1,213
Ethereum addresses	2,473	1,247	1,193
URL/IP addresses	2,412	1,234	1,218
Total	16,000	8,000	8,000

Table 9.1: Breakdown of the dataset used to model a malware exploiting steganography and test our detection approach.

detection or to update the malware during its lifespan, fixed-length hash values identifying crypto wallets can be hidden into innocent-looking contents and periodically retrieved from a C&C server;

- URL and IP Addresses from URLhaus database: the majority of threats contacts a remote facility to exfiltrate data as well as to retrieve additional payloads or configurations (see, e.g., the ZeusVM banking trojan using steganography to conceal a list of addresses and URLs belonging to financial institutions [MC15]). In general, such information is hard-coded in the malware, thus making the creation of signatures for binary analysis easy [CCC⁺20]. Hence, many recent threats cloak both IP addresses and DNS entries in digital images to escape detection.

To conceal the payloads within the images, we used the LSB steganography technique, which has been observed in many real-world threats [MC15, WCM⁺21]. To this aim, we used LSBSteg. In essence, the tool hides the source payload (i.e., the malicious content) in the RGB color channels of each pixel. To avoid trivially-visible artifacts, we considered payloads that can be hidden by only using 1 bit per channel, i.e., only payloads with a size of $512 \times 512 \times 3$ bits.

To have a realistic condition, we selected images from different open source repositories (and released under GPL3 licence) to build a dataset composed of 8,000 equally-sized images. To obtain ample test settings, we combined each image with three different payloads, which have been selected randomly. The selection process has been modeled with an uniform distribution among the different payloads, i.e., JavaScripts, obfuscated JavaScripts in HTML, PowerShell scripts, Ethereum addresses, and IP/URLs.

As a result, we obtained 32,000 images containing the various malicious contents. The dataset has been further divided into train, test, and validation sets. The overall breakdown is reported in Table 9.1. As indicated, each entry represents the amount of images generated for each set, considering the particular payload type.

To verify the effectiveness of our approach when revealing the presence of hidden payloads

within “unseen” digital images, we prepared two additional test sets. Such sets have been used to model an attacker aware of the countermeasure, thus trying to escape the detection via obfuscation or lateral movements [MW17]. In more detail, the first additional test set considers payloads encoded in Base64 with the `base64` Linux utility version 1.13.4. The second instead models attacks à-la LokiBit, which exploits zip compression to further obfuscate the hidden data. In this case, we used the `zip` Linux utility version 3.0 with the deflation compression method. To generate the two additional datasets (denoted in the following as “Base64 Test” and “ZIP Test”, respectively), we considered again the same 8,000 images with the same proportions for the payloads.

9.3.2 Parameters, Evaluation Metrics, and Testbed

As illustrated in Section 9.2, the proposed approach relies on the usage of a DNN architecture for detecting and classifying compromised images. To validate the approach, a prototypal implementation written in Python based on the Tensorflow library has been developed. As regards the DNN architecture instantiated for the experimentation, it includes $m = 3$ SubNets. Specifically, the hidden fully-connected layers are composed of 128 neurons and equipped with ReLU activation functions, while the reset probability for the dropout is 2.5%. RMSprop is adopted as optimizer with a learning rate of $1e - 3$. Both models used for detection and classification tasks are learned over 20 epochs with a batch size of 256.

To assess the quality of the results obtained by the proposed approach, we consider the *Accuracy*, *Precision*, *Recall*, and *F-Measure* defined in Chapter 7. Additionally, we evaluated the *Area Under the Curve (AUC)* for the the *Receiver Operating Characteristic (ROC)* curve. Specifically, the latter is obtained by plotting the False Positive Rate (i.e., the ratio between the number of false alarms signaled and that of all the licit images) and the True Positive Rate (i.e., the Recall) for different class probability values. As a result, the AUC is the area under the ROC curve.

Lastly, to perform experiments we used a machine equipped with an Intel Core I9-9980HK CPU @2.40GHz and 32 GB RAM. The validation set has been exploited to select the model guaranteeing the best loss value from the training phase.

9.3.3 Numerical Results

Table 9.2 showcases the results obtained for both detection and classification tasks and by considering the different attack scenarios presented in Section 9.3.1. In the first scenario, the hidden information has been encoded in a plain ASCII format, whereas in the second and third scenario, the payload has been encoded in Base64 or compressed in zip format.

As regards the detection, the proposed framework is characterized excellent results, i.e., $\sim 100\%$

Task	Test Type	Accuracy	F-Measure	AUC
Detection	Plain	0.992	0.994	1.000
	Base64 Encoding	0.999	0.999	1.000
	Zip Encoding	1.000	1.000	1.000
Classification	Plain	0.820	0.829	0.969
	Base64 Encoding	0.683	0.670	0.886
	Zip Encoding	0.487	0.324	0.704

Table 9.2: Experimental results in terms of detection and prediction capabilities obtained by varying the type of encoding for the payload.

of accuracy, in every scenario. Instead, for the classification task, the overall performance decay, especially when the payload is compressed before being cloaked in the image. The same behavior can be observed in Figure 9.5, where the values for the precision and recall are reported for all the three considered scenarios/cases. As shown, it seems that the encoding/compression operation reduces the capability of the model to distinguish among the different classes, since both the precision and recall exhibit a decreasing trend.

To investigate such a behavior in more detail, a further analysis has been performed. Specifically, we quantified the prediction capabilities of the model when dealing with each class for each scenario. To this aim we plotted the corresponding ROC curves, which have been grouped according to the use of “normal” hiding, the adoption of an additional encoding, or the use of zip compression to obfuscate the payload. In more detail, Figure 9.6 shows the ROC curves when the payload is directly embedded in the images, i.e., no elusive mechanisms are deployed by the attacker. As shown, the neural classifier exhibits good performance for each class, although it could misclassify URL/IP addresses with Ethereum ones. This behavior can be explained by considering the close similarity of such payloads (i.e., both are characterized by short string containing alphanumerical characters). As a consequence, the learning process becomes more difficult.

The case of an attacker using the Base64 encoding before the LSB steganographic injection is the depicted in Figure 9.7. Specifically, we can observe the slight degradation of the performance for each class (except the legitimate one). It must be noted that, the PowerShell class is the one suffering most of the encoding: essentially, PowerShell scripts tend to be misclassified with the JavaScript counterpart. This behavior can be ascribed to the fact that the prose of both PowerShell and JavaScript shares the use of statements (e.g., `if-then` clauses), parenthesis and specific punctuation.

Finally, Figure 9.8 deals with the case of an attacker deploying obfuscation via zip compression, i.e., the payload has been compressed and then embedded in the image. As it can be seen, a further performance degradation can be observed, although the model is however able to distinguish

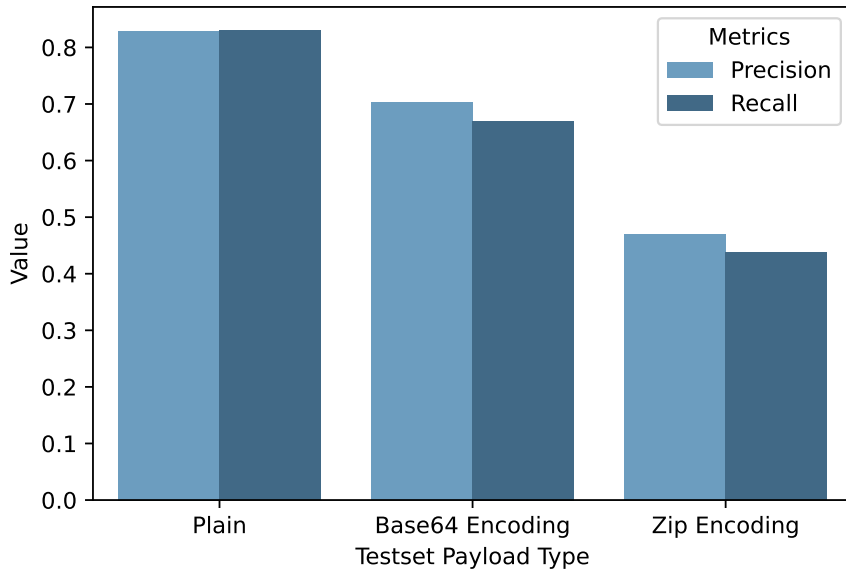


Figure 9.5: Precision and recall values for the classification task and grouped by payload type.

between compromised and legitimate images. In essence, payloads containing an “address” (i.e., IP/URLs and Ethereum pointers) are misclassified each other, while PowerShell and Javascript in HTML are labeled as Javascript payload. According to further investigations, this behavior is mainly due to the fact that the compression operation reduces the differences among the payloads. In fact, the metadata added to the compressed representation further contribute to make harder the classification task as they are similar for all the classes. In other words, the data structure imposed by the zip algorithm constitutes the majority of the information compared to the original data.

Lastly, we point out that our approach can be deployed in a simple manner in many realistic scenarios, especially owing to its limited resource footprint. In more detail, the average prediction time for a single image is ~ 5 ms calculated on the same machine used for the experimental campaign. Hence, the architecture of Figure 9.2 could be implemented over commodity hardware to protect small- and medium-sized networks in a centralized manner (e.g., by deploying a specific appliance). Besides, if timing constraints are not too tight, our approach can be also deployed in edge nodes protecting SOHO networks or granting access to various smart devices. Another possible deployment flavor could exploit the DNN to equip local antivirus with some form of AI to reveal steganographic threats: in this case, resource-intensive operations (e.g., training) can be done in a centralized manner and updates about the configuration of the neural network can be delivered locally.

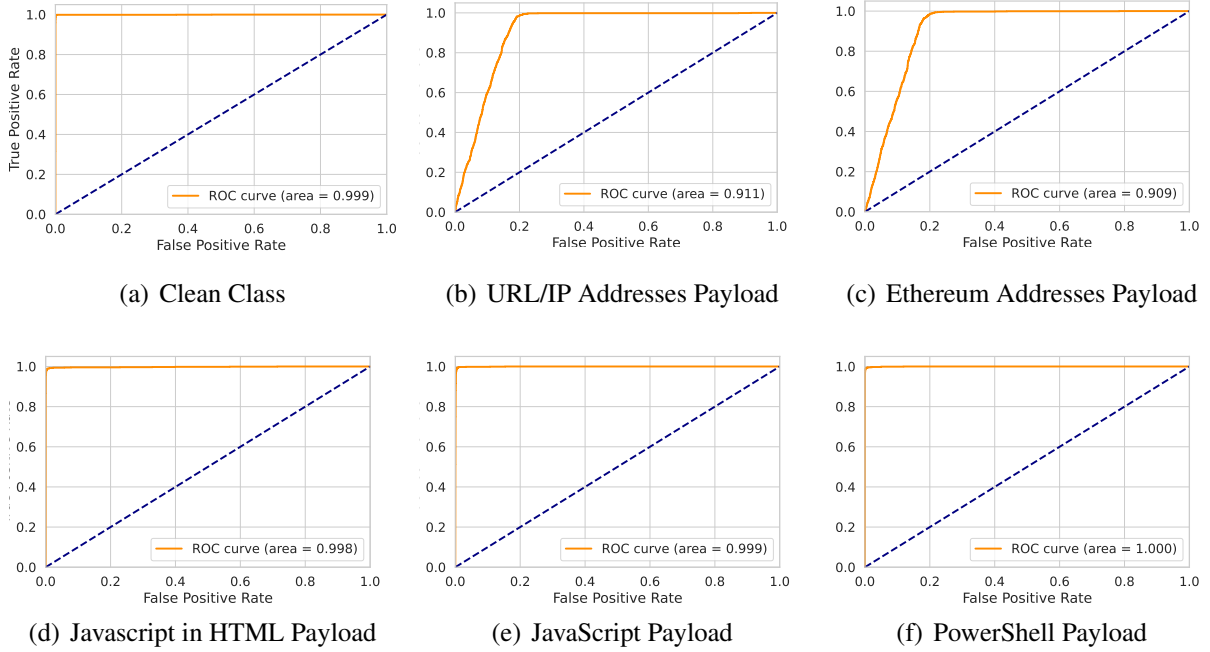


Figure 9.6: ROC Curves for Plain Testset.

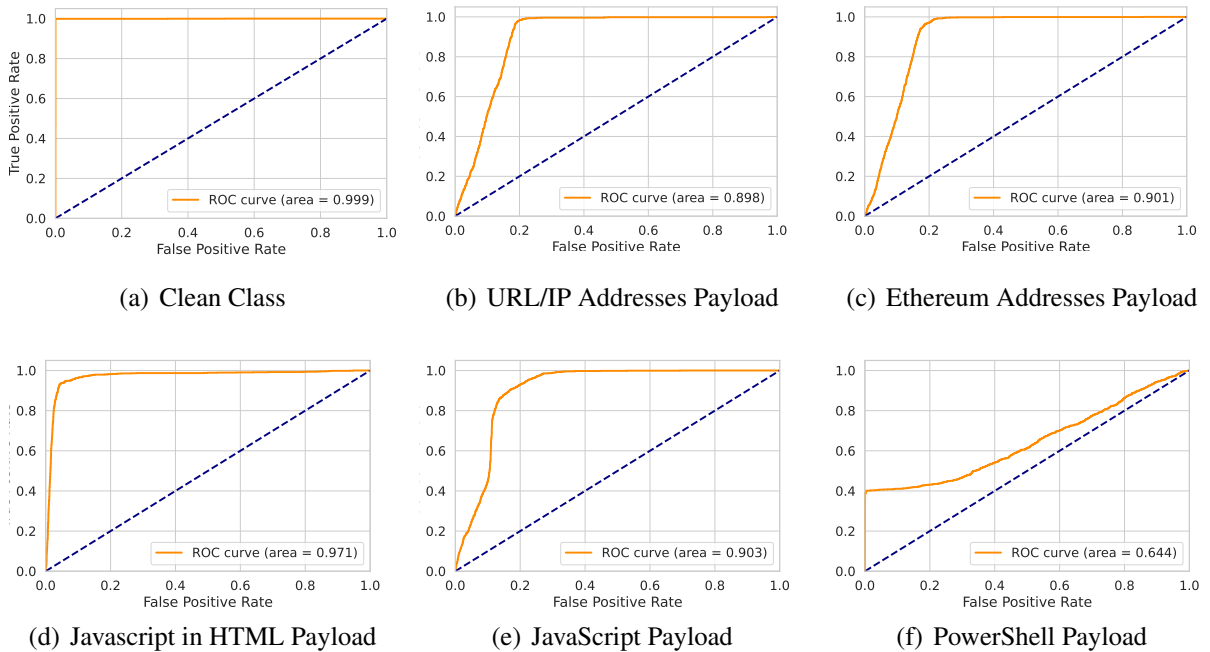


Figure 9.7: ROC Curves for Base64 Encoded Testset.

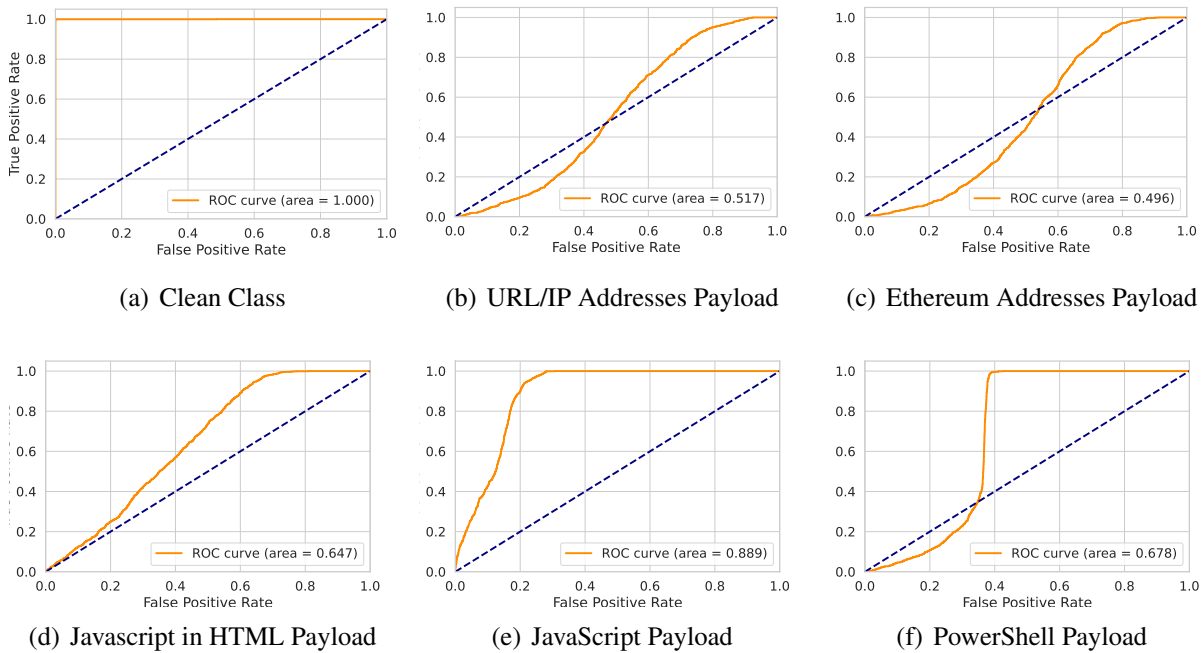


Figure 9.8: ROC Curves for ZIP Encoded Testset.

9.4 Conclusions and Future Works

In this chapter, we presented an approach for detecting digital pictures containing malicious payloads hidden via LSB steganography. To this aim, we used DNNs and demonstrated the feasibility of our approach by considering malware exploiting icons with a size of 512×512 pixels. This allowed to consider threats targeting an heterogeneous population of devices/ecosystems, which share the use of icons. Results showcased the effectiveness of our approach also when handling payloads obfuscated via zip or further processed with an alternative encoding, i.e., the attacker deployed some elusive technique. Future works aim at refining the proposed idea. Specifically, part of our ongoing research is devoted to understand the impact in terms of scalability of the approach, especially to understand its feasibility in protecting Internet-scale services or cloud datacenters in an effective manner. In addition, future research aims at performing detection of a wider array of digital images, also by considering threats using different steganographic techniques (e.g., DCT steganography) or information hiding approaches (e.g., manipulation of metadata).

In the next chapter, we will consider the case when detection of malicious images is not possible and propose the use of autoencoders for the sanitization.

Chapter 10

Sanitization of Images Containing Stegomalware via AI

When preserving privacy is not possible or when stegomalware prevents the detection performed by classical tools, other approaches for mitigating images containing hidden malicious payloads are required. Differently from the detection mechanisms presented in Chapters 8 and 9, we propose a framework based on autoencoders to “sanitize” images by disrupting the hidden payloads without degrading their quality. We point out that, compared to other works leveraging AI to implement security features (see, e.g., [DFP20, GMP20, CCC⁺20] and the references therein), we do not focus on detection. Rather, we sanitize image files assuming the presence of a third-party tool able to “flag” a content as malicious or by sanitizing all the assets of a well-defined service, for instance images hosted on a webserver acting as the front-end for a critical infrastructure. The choice of autoencoders has been driven both by their properties and their performance when handling security-related tasks, such as the extraction of features to discriminate malicious calls of APIs [DFP20]. This chapter considers images containing PowerShell scripts embedded via the Invoke-PSImage technique, which has been observed in many real-world threats and malicious campaigns [CCC⁺20].

Therefore, the contributions of this chapter are the design of an architectural framework for the mitigation of information-hiding-capable threats, and a preliminary performance evaluation campaign considering the Invoke-PSImage.

The remainder of the chapter is organized as follows. Section 10.1 provides background details on stegomalware targeting digital media and machine learning techniques for image processing. Section 10.2 deals with the proposed approach for preventing steganographic attacks via image sanitization, while Section 10.3 showcases numerical results. Lastly, Section 10.4 concludes the chapter.

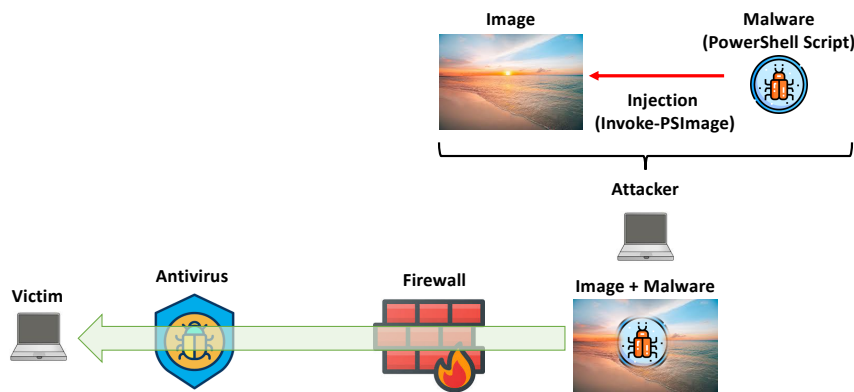


Figure 10.1: Reference attack model considering an attacker that embeds a malicious script into an innocent-looking image to evade a security tool.

10.1 Background

This section introduces the considered attack model and provides details on the machine learning approaches used to manipulate images.

10.1.1 Stegomalware and Attack Model

Figure 10.1 depicts the reference attack model. In particular, an attacker wants to hide a malware within a digital image to evade detection mechanisms, e.g., antivirus, or to allow data exchange with the victim even in the presence of a firewall or other blocking mechanisms [MC15]. To this aim, as a preliminary step, the attacker utilizes some information-hiding-based approach to inject a secret information within a digital picture. The latter should not appear as an anomaly and the embedding process should not generate visible artifacts. In this chapter, we will consider a variant of the LSB embedding technique leveraging the `Invoke-PSImage` tool. Put briefly, it allows to encode a PowerShell script within an input image. To hide the payload, the embedding method uses the 4 least significant bits of the green and the blue channels of each pixel. As a result, each pixel of the processed image will contain 1 byte of secret information.

According to threats observed in the wild, the secret data could be an attack routine, a script, a configuration file or a sequence of commands [MC15, CCM⁺18]. To have a realistic scenario, in the following we will consider an attacker hiding a PowerShell script within the digital image, as it has been observed in various malware samples [CCC⁺20] and APTs¹. The obtained image is seldom delivered to the victim in a direct manner: rather, a third-party vector is exploited. For

¹<https://cyware.com/news/new-malware-strain-abuses-github-and-imgur-e29bc6f6> [Last Accessed, October 2022].

instance, the attacker can send the image via a phishing campaign, or manipulate the content of a web page: such an altered resource can be then fetched by the victim. As an example, the MageCart malware exfiltrated payment information by hiding them into images implementing an e-commerce site [CCC⁺20] and its detection has been discussed in Chapter 8.

Upon reaching the victim, the malicious payload can be retrieved and then utilized to complete the attack. Usually, part of the malware has already been injected in the host of the victim and awaits for a specific image to be scanned for extracting the information (e.g., a list of IP addresses to contact). Another typical mechanism exploits some form of social engineering where the victim is decoyed as to use the infected image in a way that the payload detonates.

10.1.2 Machine Learning for Image Processing

Image processing represents the technical analysis of an image by means of complex algorithms. Roughly, it can be considered as a process where both the input and the output are an image. Image processing techniques can be used to improve the information content of an image for human understanding, as well as for extracting, storing and transmitting pictorial information [GW18]. Although the field is generally considered loosely separated from image analysis and computer vision, the rapid acceleration of new AI methods within the latter fields has opened new opportunities even in the former. In particular, the recent establishment of DL techniques [LBH15] has fostered significant improvements in various applications of image processing and computer vision, e.g., image enhancement, restoration and morphing. By exploiting multiple levels of abstractions, deep architectures allow to discover highly accurate models by capturing interactions between set of features directly from raw and noisy image data. The capability of learning such abstractions represents one of the most important and disruptive aspects introduced by the DL framework: no feature engineering or interaction with domain experts are required to build good representative features.

As discussed, CNNs [LBD⁺89, LB95] represent a particularly relevant variant of traditional neural networks, where the connectivity between neurons is delved on a local basis, thus allowing to capture the invariance of patterns to distortion or shift in the input data. Under this perspective, CNN architectures are particularly well-suited for the analysis of image data. A basic CNN can be devised as a stacking of layers where each of them transforms one volume of activations to another. A convolutional layer produces a higher-level abstraction of the input data, called a feature map. Units in a convolutional layer are arranged in feature maps, within which each unit is connected to local regions in the feature maps of the previous layer and represent a convolution of the input. Each neuron represents a receptive field, which receives as input a rectangular section (a filter) of the previous layer and produces an output according to the stimuli received from this filter.

The intuition within the architecture of a CNN is that convolutional layers detect high-level fea-

tures within the input, which are hence used to represent the key features of the input data and properly represent the latter at a higher abstraction level. For example, within an image, convolutional layers can progressively detect edges, contours and borders, which can be ultimately exploited to interpret the image. Because of this, deep CNN architectures are extensively used in image classification [SLJ⁺15, SZ14, HZRS16] or object detection [Gir15, LAE⁺16, RDGF16].

Another deep architecture of interest for image processing and analysis is the Encoder-decoder framework [HS06, NKK⁺11]. An autoencoder is a particular neural network where the target of the network is the data itself, and it is mainly exploited for dimensionality reduction tasks or for learning efficient encoding. The simplest structure includes three components: *i*) an input layer of raw data to be encoded; *ii*) a stack of hidden layers mapping the input data into a low-dimensional representation and viceversa; and *iii*) an output layer with the same size of the input layer. Autoencoders find several applications in image processing. For example, they can be used for regularization: a denoising autoencoder [VLBM08] tries to reconstruct the original information from noisy data. By optimizing the reconstruction loss, the denoising autoencoder learns to extract features from a noisy input, which can be used to reconstruct the original content at the same time disregarding the noise. More advanced architectures based on a combination between convolutional and Encoder-decoder architectures [LSD15, NHH15, RFB15] can also be exploited for tasks such as enhancement, morphing and segmentation.

10.2 Sanitization Through Machine Learning

In this section we present the approach to sanitize images containing malicious PowerShell scripts injected via the Invoke-PSImage technique. First we introduce the reference architecture, then we discuss the methodology used to process the various digital media.

10.2.1 Architectural Blueprint

To process a digital image for disrupting the hidden information without altering the perceived quality, we take advantage of convolutional autoencoders. To this aim, the proposed framework could be implemented as a middlebox able to intercept the flow of data and process the images. Figure 10.2 showcases a reference deployment. In general, processing huge volumes of information in a centralized manner poses some scalability issues, introduces a unique point of failure, and can account for additional delays or degradation of the Quality of Experience perceived by end users. Moreover, intercepting digital images from the bulk of traffic could not be possible, e.g., due to encrypted conversations. A possible idea to implement the proposed image sanitizer framework concerns the use of a proxying architecture only targeting specific protocols. For instance, it can be implemented as an HTTP proxy to prevent an attacker to distribute malicious code via innocent looking web pages or contents published on online social networks

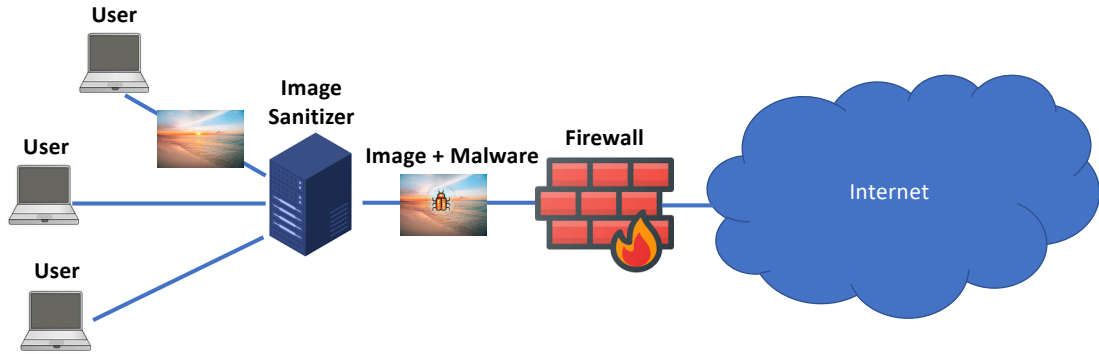


Figure 10.2: Reference scenario leveraging the proposed sanitization framework based on autoencoders.

[BHCdFR12]. Moreover, since many attacks observed in the wild exploit web pages as the attack vector, the proposed framework can be engineered as a plug-in to be deployed in the web server in charge of “scrubbing” images and other in-line objects before they are sent to end users.

10.2.2 Methodology

Within the proposed framework, the image sanitization can be accomplished as follows. We assume that \mathbf{x} is an input image and ϵ is the *a priori* unknown malicious attack compromising the content of the image into $\mathbf{x} + \epsilon$. The objective is to devise a neural functional N such that $N(\mathbf{x} + \epsilon) \approx \mathbf{x}$, while contemporarily guaranteeing that $N(\mathbf{x}) \approx \mathbf{x}$ for uncompromised images. The functional N represents the sanitizer to be exploited in the reference scenario of Figure 10.2.

The architecture for N is loosely inspired to Unet [RFB15]: the input image is progressively halved in size and doubled in volume by means of convolutional blocks, until a core compact representation of 512 layers of size 60×60 is obtained. The decoding phase is characterized by a series of deconvolution blocks, each of them combined with the corresponding residual block from the encoding phase and transformed in volume through an additional convolutional block. The final image is reconstructed from the final block by exploiting a sigmoid activation layer. Each block is composed of: a convolution/transposed convolution, a rectified linear unit, a dropout and a batch normalization layer. The overall design is illustrated in Figure 10.3: the grey blocks on the right-hand side represent the corresponding blocks in the encoding phase, stacked with the outputs from deconvolutional blocks.

The network is learned on a set $\mathcal{D} = \{(\mathbf{y}_1, \mathbf{x}_1), (\mathbf{y}_2, \mathbf{x}_2), \dots, (\mathbf{y}_n, \mathbf{x}_n)\}$ of image pairs, where \mathbf{x}_i represents the original image and $\mathbf{y}_i = \mathbf{x}_i + \epsilon_i$ the (possibly) compromised input. The learning phase aims at optimizing the network weights by minimizing the reconstruction loss. We studied two possible choices for the latter. The MSE is a natural choice for the reconstruction, as it

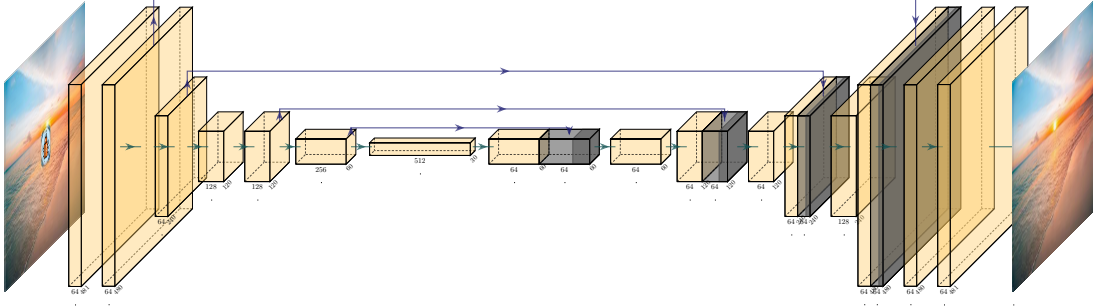


Figure 10.3: Reference architecture for the autoencoder. The input (compromised) image is processed through the convolutional layers and its sanitized version is produced as output.

measures divergences at a pixel level:

$$MSE(N, \mathcal{D}) = \frac{1}{n} \sum_i \|\mathbf{x}_i - N(\mathbf{y}_i)\|^2.$$

Notice that here we consider normalized images, such that the values of the pixels within each channel ranges within the interval $[0, 1]$. This can set a potential problem, especially when the attack ϵ_i represents a negligible distortion of the original image: if the amount of displaced pixels is small, the loss becomes small and the gradient vanishes, thus making the learning phase stationary.

To avoid this, we also studied the adoption of the BCE loss,

$$BCE(N, \mathcal{D}) = -\frac{1}{n} \sum_i \sum_j (x_{ij} \log N_j(\mathbf{y}_i) + (1 - x_{ij}) \log (1 - N_j(\mathbf{y}_i))).$$

where, j represents the coordinate of the j -th pixel within the image. The rationale is that, whenever \mathbf{x}_i and $N(\mathbf{y}_i)$ diverge on a pixel, the contribution to the loss is significantly amplified by the logarithmic term. This can stabilize the learning phase making it faster and in principle more reliable.

10.3 Performance Evaluation

To prove the effectiveness of the sanitization approach, we performed a preliminary performance evaluation campaign. To this aim, we generated an ad-hoc dataset of images containing PowerShell scripts embedded via the Invoke-PSImage tool. We considered 500 legitimate images taken from the publicly available Berkeley Segmentation Data Set [AMFM10]. Specifically, the

dataset consists of 500 natural images, explicitly separated into disjoint train, validation and test subsets. Images have been processed with a custom Python module to automatize the embedding of 110 different PowerShell scripts taken from the Lazywinadmin repository².

The resulting dataset is once again divided into train, test, and validation sets. The train set is composed of 12,000 images (200 legitimate images combined with 60 different PowerShell scripts), the test set contains 2,500 images (100 legitimate images combined with 25 different scripts), and the validation set contains 5,000 images (200 legitimate images combined with 25 PowerShell scripts). As a result, the final dataset is composed of 19,500 “dirty” images, i.e., digital pictures embedding a PowerShell script.

To evaluate the performance in terms of sanitization, we used StegExpose³, which is a Java tool for detecting contents embedded in images via steganographic techniques [Boe14, Bal17]. Specifically, StegExpose combines four different detection methods, i.e., Sample Pairs [DWW02], RS Analysis [FGD01], Chi-Square Attack [WP99], and Primary Sets [DWM02]. For each algorithm, it calculates the likelihood of an input image of being “malicious” with the acceptance of being the target of some steganographic alteration. Returned values are then averaged and the result is compared to an empiric threshold value. StegExpose implements two execution modes: default and fast mode. The default mode executes all the four detectors in sequence, whereas the fast mode perform a decision as soon as a detector returns an alarm. To guarantee the accuracy of the detection, in this work we use StegExpose in default mode. To conduct tests, we used a machine running Ubuntu 20.04 with an Intel Core i9-9900KF CPU @3.60GHz and 32 GB RAM. In our trials, the threshold value used by StegExpose to flag an image as malicious was set to 0.2 since it provides the best tradeoff between false positive and true positive rates [Boe14].

The model devised in Section 10.2.2 has been implemented in PyTorch⁴. The experiments were executed on an NVidia DGX Station equipped with 4 GPU V100 32GB. The model was learned by optimizing the weights in batches of 32 images from the training set using the Adam optimizer with a learning rate $lr = 0.001$. The validation set was exploited to select the model guaranteeing the best reconstruction loss from the training phase. Finally, the the evaluation performed on the test set images measures the capability of the model in cleaning the images from malicious codes and simultaneously reconstructing the original image.

Figure 10.4 showcases an example outcome of the proposed approach when an animal with a complex background is depicted. Specifically, Figure 10.4(a) reports the original image, whereas Figure 10.4(b) shows the same image after Invoke-PSImage is used to embed a script. According to our results, our tool makes the embedded PowerShell information unreadable at the price of a limited variation of the visual quality of the media (see, Figure 10.4(c)). Yet, when alterations happen, our approach allows to “improve” the overall quality, e.g., by mitigating artifacts caused

²<https://github.com/lazywinadmin/PowerShell> [Last Accessed, October 2022].

³<https://github.com/b3dk7/StegExpose> [Last Accessed, October 2022].

⁴The code is available on <https://github.com/gmanco/stegomalware> [Last Accessed, October 2022].

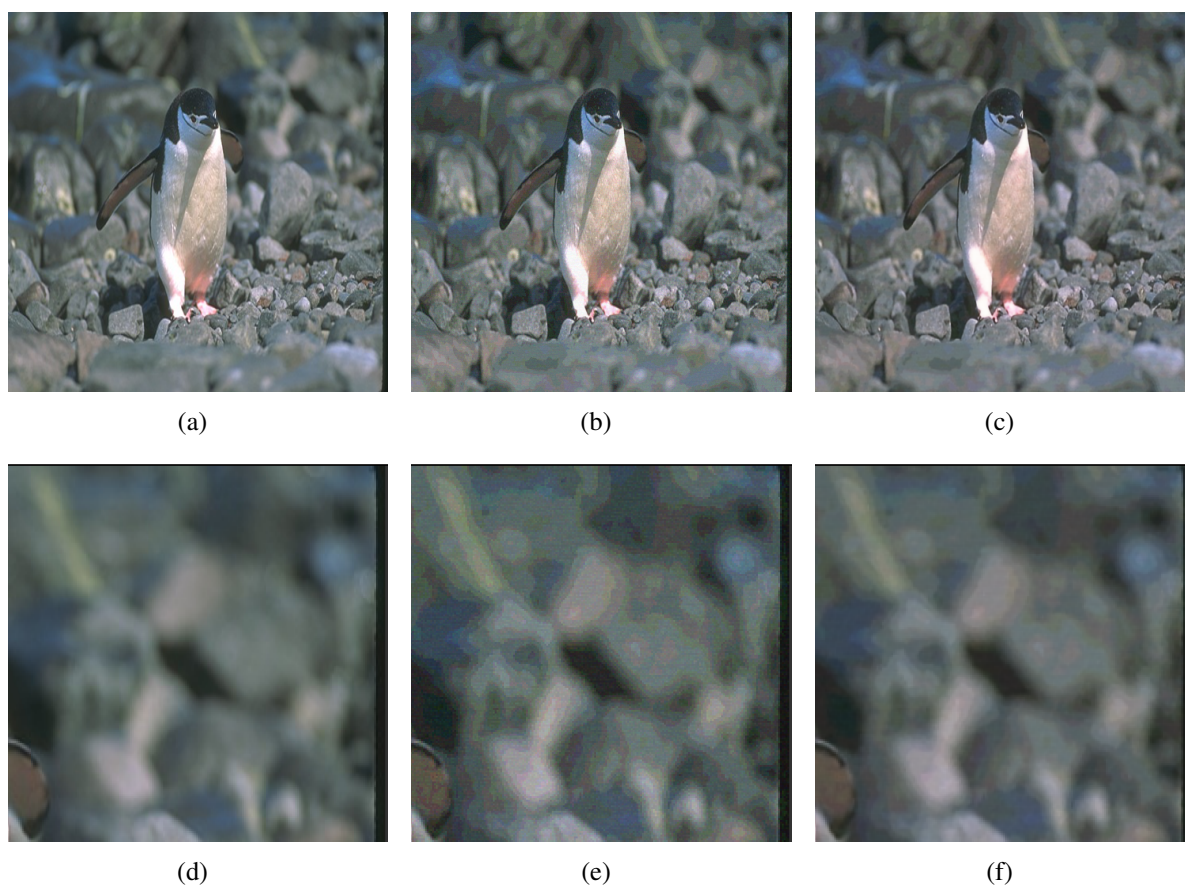


Figure 10.4: Example of the sanitization process - an animal with a complex background: (a) original image, (b) image containing a PowerShell script embedded via Invoke-PSImage, (c) sanitized image. Magnifications of relevant details for images (a), (b) and (c) are provided in (d), (e) and (f), respectively.

by the embedding process via Invoke-PSImage (see, Figures 10.4(d), 10.4(e), and 10.4(f)). In this perspective, our approach should not be perceived only as a sanitization technique: in fact, it also performs a sort of restoration enabling users to receive contents closer to their original form. Another example, considering a landscape is reported in Figure 10.5. A key success to deliver malware and evade detection is to use images that do not appear as anomalous. To this aim, attacks like those launched by the Zeus/Zbot Trojan exploited an image depicting a sunset to exchange data while remaining unnoticed [MC15]. Similarly to the previous case, magnifications of major details (reported in Figures 10.5(d), 10.5(e), and 10.5(f)) demonstrate the ability of the approach in mitigating alterations introduced by the Invoke-PSImage tool.

Table 10.1 reports summary statistics which quantify the effectiveness of the sanitization. In the experiment, we compared the original (clean) images with both their compromised and sanitized

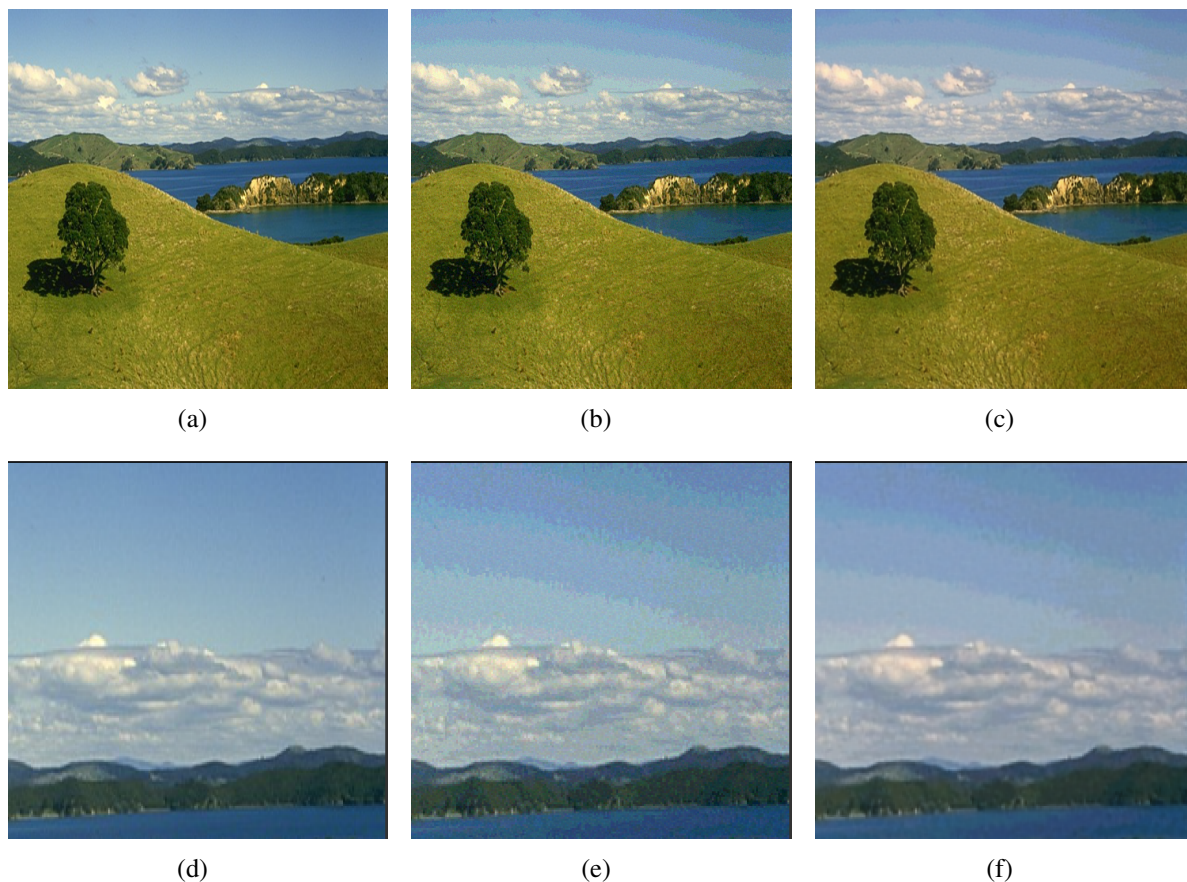


Figure 10.5: Another example of the sanitization process - landscape: (a) original image, (b) image containing a PowerShell script embedded via Invoke-PSImage, (c) sanitized image. Magnifications of relevant details for images (a), (b) and (c) are provided in (d), (e) and (f), respectively.

counterpart. The comparison was done by computing the mean absolute difference among pixels. For the sanitization, we exploited models learned with either MSE and BCE loss. Within the table, the second column represents the cumulative absolute difference on the whole test set, while the third column represents the average number of pixel distortions that can be perceived. We can see that the sanitization recovers on average more than 35% of the compromised pixels. Notably, the MSE loss seems to work better, compared to the BCE loss. This behavior is counter-intuitive with respect to the initial hypotheses and deserves further study. Better tuning strategies, (e.g., based on weighting schemes), can possibly recover performance for the BCE and further improve the reconstruction of the original image.

To further prove the effectiveness of our method, we tested images of the validation dataset against the StegExpose tool. It turned out that, for all of them, the embedded information has been disrupted and the entire dataset was flagged as clean. However, according to additional tri-

	Cumulative Difference	Avg. Pixel Distortions
Malicious samples	3.235	448
MSE sanitization	2.016	279
BCE sanitization	2.36	327

Table 10.1: Comparison between original and compromised/sanitized images.

als on the 5,000 images of the validation set containing the various PowerShell scripts, StegExpose correctly classified them as “clean” or “malicious” with an error of 19.36%. Thus, even in the presence of sanitized images, solely using a tool like StegExpose could bring to false positive/negative phenomena. As a consequence, a more reliable idea could be using some sort of extracting mechanism as to check the presence of a working script. This is part of our future work.

Finally, processing a batch of 32 images with our autoencoder-based methodology requires ~ 25 ms on average, without the need of a dedicated architecture based on GPUs. Therefore, our method can be considered effective for disrupting malicious payloads embedded in images also when deployed in realistic, production-quality scenarios.

10.4 Conclusions and Future Works

In this chapter, we presented an approach leveraging autoencoders to sanitize digital pictures containing PowerShell scripts injected via the Invoke-PSImage tool. Such a scenario captures a new-wave of threats defined as stegomalware exploiting steganography and information hiding to remain unnoticed and avoid detection. Results prove the effectiveness of our approach allowing to disrupt the embedded information while improving the image quality as to match its original form. Future works aim at refining the proposed idea. In particular, we want to understand the “degree of disruption” of a script, i.e., to quantify if some instructions or functions survived the sanitization process. We also aim at identifying the portion of the image containing the malicious content. In this case, an important part of the future works will be devoted to quantify the performance in terms of false negatives/positives as to understand the feasibility of deploying the proposed approach in production-quality scenarios.

The next part of this Thesis draws the final conclusions of this work and provides additional information.

Part IV

Conclusions and Appendices

Chapter 11

Conclusions and Future Works

The increasing diffusion of malware endowed with information hiding and steganographic techniques demands for efficient algorithms to collect data and implement new mitigation strategies. Therefore, this Thesis has addressed the detection of network covert channels and malware cloaking malicious payloads within digital images.

In order to conduct the needed research, we first addressed the lack of suitable traffic traces containing covert communications observed in real attack campaigns. To overcome this issue, part of the Thesis focused on the creation of two ad-hoc tools, i.e., `IPv6CC` and `pcapStego`. The first creates hidden communication paths within IPv6 traffic in a Man-in-the-Middle fashion, while the second manipulates pre-collected traces to contain various storage and timing covert channels. Then, `IPv6CC` and `pcapStego` have been used to assess the capabilities of Snort, Suricata, and Zeek to detect several types of network covert channels. Specifically, tests have been conducted with different traffic volumes and loads and by considering storage and timing methods, as well as strategies embedding data in various protocol fields. Despite the configuration, security tools turned out to be insensitive to this class of attacks. For the sake of completeness, Snort, Suricata and Zeek have been also tested against covert channels within TLS traffic by using the `TLSCC` suite. Also in this case, the selected security tools were not able to spot the presence of hidden data. Thus, the assessment demonstrated the need of improving the standard countermeasures or creating “tweaked” rules and configurations to properly mitigate the impact of stegomalware. However, as showcased during this Thesis, the amount of possibilities for cloaking data in terms of protocols, header fields and hiding strategies is almost unbounded. Thus, a relevant part of future activities will be devoted to extend the functionalities of `pcapStego` and `IPv6CC` to support a wider array of protocols, especially the MQTT used in IoT settings or ubiquitous VoIP/DNS services. Moreover, another important aspect concerns the ability of hiding data via advanced schemes, e.g., by encoding information in the modulation of the throughput, in the artificial creation of retransmissions, or in increased error rates. The main limit of the investigation presented in this Thesis concerns the lack of consideration of both

commercial and academic tools. Thus, part of future research will try to conduct tests with solutions such as the Security Event Manager by SolarWinds or the WoDiCoF platform. Moreover, ad-hoc sets of rules that are starting to emerge for open source solutions (e.g., BroCCaDe for Zeek) will be evaluated as well.

Unfortunately, malware can conceal information in almost every protocol or traffic feature, thus extending tools or “stacking” rules should not be considered the preferred solution. For instance, the zeek-stego patch required to write suitable hooks to inspect traffic with a per-packet granularity and to collect information for the specific protocol field. To overcome this problem, we proposed a code layering framework based on the eBPF technology, i.e., `bccstego`. In more detail, `bccstego` takes advantage of kernel events to easily inspect network packets and then to collect statistic indicators useful for revealing the presence of covert communications. However, this process requires to periodically transfer data from kernel- to user-space, thus leading to possible bottlenecks. Hence, the solution presented in this Thesis exploited a bin-based data structure, i.e., several values are grouped and associated to a specific bin. Even if this accounts for a less precise “snapshot” of the observed traffic, the approach turned out to be effective. For instance, for the case of covert channels targeting the `Flow Label`, the number of non-empty bins can be used to estimate the amount of active IPv6 flows. By using a simple detection rule (i.e., a comparison against the number of active flows), hidden communication attempts are revealed in various scenarios with $\sim 90\%$ of accuracy. The use of in-kernel methodologies turned out to be efficient and lightweight, especially when compared to technologies such as Zeek and libpcap. In fact, the needed processing influences the network traffic in a minimal manner, i.e., the average latency is ~ 105 ns on a per-packet basis. Similarly, the CPU and memory footprints are limited, thus proving the scalability of `bccstego` to spot covert channels starting from network inspection. Results collected during the Thesis also suggest that some performance improvements could be achieved by rewriting the user-space part in C instead of Python. Unfortunately, this would not prevent the major degradation due to memory copy operations. In this perspective, an idea that will be considered in future developments regards the use of more performing and optimized data structures. A possible idea to explore concerns the use of Bloom filters, which can reduce the time needed to lookup and “fill” the proper bin (e.g., by means of hash functions) as well as the overall size of data to be moved from kernel- to user-space.

In general, our research demonstrated that eBPF can be considered a good technological solution to gain visibility over network and software and to collect information useful to spot the presence of different types of stegomalware. In fact, during the Thesis, eBPF has been also used to implement the well-known algorithm from Cabuk to detect timing channels. This is a promising research path to pursue, since being able to perform the detection directly within the kernel prevents kernel- to user-space data movements. Similarly, eBPF also showcased its ability to deal with local covert channels through a unique technological solution. In this case, the functionalities of eBPF have been also used to trace a kernel function, i.e., the `_x64_sys_chmod`, to reveal data transfers between two processes implementing a colluding applications attack scheme based on the `chmod-stego` technique.

Summing up, results contained in this Thesis suggest to further pursue the use of kernel tracing techniques to counteract stegomalware or disrupt covert channels by means of suitable sanitization techniques. However, the main limitation of the approach is that it heavily relies upon the specific threat/hiding mechanism, which is seldom known *a-priori*. To face this issue, a main research topic for the next years should be the investigation of more general indicators that are not strictly coupled with the carriers used by the steganographic threat. In this vein, possible metrics should consider signatures in the CPU usage, in the volume of operations within the file system, as well as the use of protocol-agnostic features. Luckily, eBPF can still be used to develop multiple kernel filters for the creation of threat-independent data collectors.

When specific or general indicators/signatures are not available, mitigating the impact of steganographic threats by using some form of AI should be considered a valid alternative. During the Thesis, this approach has been applied to two different class of threats. First, unsupervised AI methods have been investigated with the aim of spotting covert channels targeting IoT nodes. Despite the ensemble of autoencoders achieved a probability of detection equal to $\sim 91\%$ and a precision equal to $\sim 95\%$, further refinements should be taken into consideration. As an example, the detection has been performed by considering a static threshold, which could fail to reveal hidden communications when in the presence of “changing” network conditions, e.g., due to the churning of devices or IoT nodes. To mitigate such a problem, a simple workaround could exploit dynamic thresholds computed on different time windows of reduced lengths. However, to really advance in the use of AI for the detection of covert communications, future research should solve the issue of too many false alarms raised by unsupervised approaches. A promising path is to adopt hybrid approaches able to combine the benefits of supervised and unsupervised techniques.

Concerning the detection of malware hidden within digital images, this Thesis proposed the use of a DNN for mitigating various threats modeled via realistic synthetic datasets. The first reference scenario considered the detection of MageCart-like threats hiding malicious payloads (i.e., URLs and PHP scripts) within favicons of 32×32 pixels via a plain LSB steganographic technique. The proposed approach was able to spot all the modified favicons with an accuracy equal to $\sim 100\%$ and was also able to classify the payload type. An important outcome of this part of research dealt with the ability of the DNN to face also “unknown” payloads or the presence of an attacker performing some form of “lateral movements” or elusive maneuvers. In fact, the DNN approach was able to also detect and classify payloads obfuscated via Base64 encoding, even with an accuracy reduced to $\sim 91\%$. The second reference scenario considered a generic stegomalware hiding information in larger images. In more detail, the Thesis has addressed the case of attackers trying to take advantage of the massive diffusion of mobile applications, which are often distributed in a cross-device manner. Thus, high-resolution icons of 512×512 pixels hiding a wide selection of payloads, e.g., JavaScript, HTML, PowerShell, and Ethereum addresses, have been considered. Despite some misclassification issues due to similarities in the prose of some scripting languages (e.g., PowerShell code is often recognized as JavaScript), obtained results demonstrated that the approach should be considered valid to spot the presence of hidden pay-

loads. Additionally, the DNN was able to identify malicious contents even in the presence of an attacker using basic obfuscation techniques or alternative encoding schemes, such as Base64 or zip compression methods. Yet, to effectively deploy such mechanisms in production-quality scenarios, further research needs to be done. For instance, there is the need of understanding whether DNNs can be also used to detect and classify more sophisticated steganographic methods, such as the Invoke-PSImage technique observed in Ursnif as well as DCT or PVD algorithms. Moreover, the “robustness” of approaches leveraging AI needs to be fully understood. In fact, it is still unclear the performance when dealing with adversarial machine learning schemes trying to compute “best” hiding strategies. Similarly, the impact of adversaries providing malicious samples labeled as legitimate (i.e., data poisoning) should properly be investigated. Finally, when the detection or the classification are not possible (e.g., due to privacy issues or scalability constraints), this Thesis has also proposed the use of autoencoders to sanitize malicious contents concealed in digital pictures. Yet, the approach is still not mature enough: for example, the amount of hidden information that survived the sanitization process has not been quantified. In fact, even if an incomplete script is almost useless, its residues could still contain data useful for the attacker (e.g., a URL for downloading an additional payload). Finally, to really deploy AI-based frameworks in real settings for counteracting the multifaceted range of offensive mechanisms characterizing modern steganographic malware, research should advance in terms of privacy and performance. As regards privacy, GDPR-like regulations would make almost impossible to collect traffic or to deeply inspect network conversations for collecting data. Similarly, it is quite difficult that users will give permissions to inspect their images or upload them to third-party providers. To this aim, a promising idea is to explore some form of federated approach, i.e., AI models are locally trained at the border of the network or in devices of end users and only parameters are remotely uploaded. A second aspect concerns the performance, i.e., the ability of deploying AI in real conditions. Even if this Thesis has not fully investigated the computational requirements of the used AI frameworks, observed estimates hint at the possibility of deploying some detection mechanisms (e.g., those revealing covert channels in IoT ecosystems) at the border of the network, by using the edge paradigm. Similarly, sanitization of images containing hidden payloads can be done with commodity hardware in a centralized manner for small/medium settings.

As a concluding remark, one of the most important outcomes of the Thesis is represented by the various datasets released during the PhD. In fact, the main challenge to be addressed for investigating modern steganographic threats concerns the lack of comprehensive collections of samples or execution traces (as it happens for the mobile case). In this perspective, tools aimed at producing suitable traffic collections and datasets of images containing real payloads should be considered a starting point for developing countermeasures against steganographic malware. Therefore, a relevant part of our future work will be devoted to improve the various publicly-available datasets for making the research possible.

Bibliography

- [AAB⁺17] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai Botnet. In *Proc. of the 26th USENIX Security Symposium (USENIX Security)*, pages 1093–1110, Vancouver, Canada, August 2017.
- [ABP⁺19] Cristina Alcaraz, Giuseppe Bernieri, Federica Pascucci, Javier Lopez, and Roberto Setola. Covert Channels-based Stealth Attacks in Industry 4.0. *IEEE Systems Journal*, 13(4):3980–3988, May 2019.
- [ACJR11] Shane Amante, Brian Carpenter, Sheng Jiang, and Jarno Rajahalme. IPv6 Flow Label Specification. RFC 6437, RFC Editor, November 2011.
- [AH08] Ahmed A. Abdelwahab and Lobna A. Hassaan. A Discrete Wavelet Transform Based Technique for Image Data Hiding. In *Proc. of the 2008 National Radio Science Conference (NRSC)*, pages 1–9, Tanta, Egypt, June 2008. IEEE.
- [AIS16] Moudhi M. Aljamea, Costas S. Iliopoulos, and Mohammad Samiruzzaman. Detection of URL in Image Steganography. In *Proc. of the International Conference on Internet of Things and Cloud Computing (ICC)*, pages 1–6, Cambridge, United Kingdom, March 2016. ACM.
- [Alp11] Dmitri Alperovitch. *Revealed: Operation Shady RAT*, volume 3. McAfee, August 2011.
- [AMFM10] Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. Contour Detection and Hierarchical Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(5):898–916, August 2010.
- [AMH16] Mohiuddin Ahmed, Abdun Naser Mahmood, and Jiankun Hu. A Survey of Network Anomaly Detection Techniques. *Journal of Network and Computer Applications*, 60:19–31, January 2016.

- [AP98] Ross J. Anderson and Fabien A. P. Petitcolas. On the Limits of Steganography. *IEEE Journal on Selected Areas in Communications*, 16(4):474–481, May 1998.
- [ASKWS⁺21] Zeeshan Ahmad, Adnan Shahid Khan, Cheah Wai Shiang, Johari Abdullah, and Farhan Ahmad. Network Intrusion Detection System: A Systematic Study of Machine Learning and Deep Learning Approaches. *Transactions on Emerging Telecommunications Technologies*, 32(1):e4150, January 2021.
- [Bal17] Shumeet Baluja. Hiding Images in Plain Sight: Deep Steganography. In *Proc. of the 31st International Conference on Neural Information Processing Systems (NeurIPS)*, pages 2066–2076, Long Beach, CA, USA, December 2017.
- [BBC19] Daniel S. Berman, Anna L. Buczak, Jeffrey S. Chavis, and Cherita L. Corbett. A Survey of Deep Learning Methods for Cyber Security. *Information*, 10(4):122, February 2019.
- [BEK16] Anna Belozubova, Anna Epishkina, and Konstantin Kogos. Random Delays to Limit Timing Covert Channel. In *Proc. of the 2016 European Intelligence and Security Informatics Conference (EISIC)*, pages 188–191, Uppsala, Sweden, August 2016. IEEE.
- [Ber17] Gilberto Bertin. XDP in Practice: Integrating XDP Into our DDoS Mitigation Pipeline. In *Proc. of the Technical Conference on Linux Networking, Netdev*, volume 2, Montreal, Canada, April 2017.
- [BFZ21] Maximilian Bachl, Joachim Fabini, and Tanja Zseby. A Flow-based IDS Using Machine Learning in eBPF. *arXiv preprint arXiv:2102.09980*, February 2021.
- [BG15] Anna L. Buczak and Erhan Guven. A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection. *IEEE Communications Surveys & Tutorials*, 18(2):1153–1176, October 2015.
- [BGC05] Vincent Berk, Annarita Giani, and George Cybenko. Detection of Covert Channel Encoding in Network Packet Delays. Dartmouth Digital Commons, Dartmouth College, Hanover, NH, USA, January 2005.
- [BGN17] Arnab K. Biswas, Dipak Ghosal, and Shishir Nagaraja. A Survey of Timing Channels and Countermeasures. *ACM Computing Surveys*, 50(1):1–39, March 2017.
- [BHCdFR12] Jorge Blasco, Julio C. Hernandez-Castro, José M. de Fuentes, and Benjamin Ramos. A Framework for Avoiding Steganography Usage Over HTTP. *Journal of Network and Computer Applications*, 35(1):491–501, January 2012.

- [BHGG18] Andreas Bjørn-Hansen, Tor-Morten Grønli, and Gheorghita Ghinea. A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-platform Mobile Development. *ACM Computing Surveys*, 51(5):1–34, November 2018.
- [BKP20] Jonathan Berger, Amit Klein, and Benny Pinkas. Flaw Label: Exploiting IPv6 Flow Label. In *Proc. of the 2020 IEEE Symposium on Security and Privacy (SP)*, pages 1594–1611, San Francisco, CA, USA, May 2020. IEEE.
- [BLC13] Mathieu Bouet, Jérémie Leguay, and Vania Conan. Cost-based Placement of Virtualized Deep Packet Inspection Functions in SDN. In *Proc. of the 2013 IEEE Military Communications Conference (MILCOM)*, pages 992–997, San Diego, CA, USA, November 2013. IEEE.
- [BLM⁺18] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. High-speed Software Data Plane via Vectorized Packet Processing. *IEEE Communications Magazine*, 56(12):97–103, November 2018.
- [BLPL06] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy Layer-wise Training of Deep Networks. *Advances in Neural Information Processing Systems*, 19:153–160, September 2006.
- [Boe14] Benedikt Boehm. StegExpose - A Tool for Detecting LSB Steganography. *arXiv preprint arXiv:1410.6656*, October 2014.
- [BPK⁺16] Bernhards Blumbergs, Mauno Pihelgas, Markus Kont, Olaf Maennel, and Risto Vaarandi. Creating and Detecting IPv6 Transition Mechanism-based Information Exfiltration Covert Channels. In Billy B. Brumley and Juha Röning, editors, *Secure IT Systems*, Lecture Notes in Computer Science, pages 85–100, Oulu, Finland, October 2016. Springer, Springer International Publishing.
- [BvEC⁺17] Riccardo Bortolameotti, Thijs van Ede, Marco Caselli, Maarten H. Everts, Pieter Hartel, Rick Hofstede, Willem Jonker, and Andreas Peter. Decanter: Detection of Anomalous Outbound HTTP Traffic by Passive Application Fingerprinting. In *Proc. of the 33rd Annual Computer Security Applications Conference (ACSAC)*, pages 373–386, Orlando, FL, USA, December 2017. ACM.
- [Cav21] Luca Cavaglione. Trends and Challenges in Network Covert Channels Countermeasures. *Applied Sciences*, 11(4):1641, February 2021.
- [CBS04] Serdar Cabuk, Carla E. Brodley, and Clay Shields. IP Covert Timing Channels: Design and Detection. In *Proc. of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 178–187, Washington, DC, USA, October 2004. ACM.

- [CCC⁺20] Luca Caviglione, Michał Choraś, Igino Corona, Artur Janicki, Wojciech Mazurczyk, Marek Pawlicki, and Katarzyna Wasielewska. Tight Arms Race: Overview of Current Malware Threats and Trends in Their Detection. *IEEE Access*, 9:5371–5396, December 2020.
- [CCM⁺18] Krzysztof Cabaj, Luca Caviglione, Wojciech Mazurczyk, Steffen Wendzel, Alan Woodward, and Sebastian Zander. The new Threats of Information Hiding: the Road Ahead. *IT Professional*, 20(3):31–39, June 2018.
- [CCRZ20] Alessandro Carrega, Luca Caviglione, Matteo Repetto, and Marco Zuppelli. Programmable Data Gathering for Detecting Stegomalware. In *Proc. of the 2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pages 422–429, Ghent, Belgium (Online), June 2020. IEEE.
- [CGG⁺22] Luca Caviglione, Martin Grabowski, Kai Gutberlet, Adrian Marzecki, Marco Zuppelli, Andreas Schaffhauser, and Wojciech Mazurczyk. Detection of Malicious Images in Production-Quality Scenarios with the SIMARGL Toolkit. In *Proc. of the 17th International Conference on Availability, Reliability and Security (ARES)*, pages 1–7, Vienna, Austria, August 2022. ACM.
- [CLB⁺14] Yi-Chao Chen, Yong Liao, Mario Baldi, Sung-Ju Lee, and Lili Qiu. OS Fingerprinting and Tethering Detection in Mobile Networks. In *Proc. of the 2014 Conference on Internet Measurement Conference (IMC)*, pages 173–180, Vancouver, Canada, November 2014. ACM.
- [CLL⁺21] Shaojie Chen, Bo Lang, Hongyu Liu, Duokun Li, and Chuan Gao. DNS Covert Channel Detection Method Using the LSTM Model. *Computers & Security*, 104:102095, May 2021.
- [CLW⁺19] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. Android HIV: A Study of Repackaging Malware for Evading Machine-learning Detection. *IEEE Transactions on Information Forensics and Security*, 15:987–1001, July 2019.
- [CM22] Luca Caviglione and Wojciech Mazurczyk. Never Mind the Malware, Here’s the Stegomalware. *IEEE Security & Privacy*, 20(5):101–106, September 2022.
- [CMR⁺21] Luca Caviglione, Wojciech Mazurczyk, Matteo Repetto, Andreas Schaffhauser, and Marco Zuppelli. Kernel-level Tracing for Detecting Stegomalware and Covert Channels in Linux Environments. *Computer Networks*, 191:108010, May 2021.

- [CRR18] Stefan Covaci, Matteo Repetto, and Fulvio Rizzo. A new Paradigm to Address Threats for Virtualized Services. In *Proc. of the 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 689–694, Tokyo, Japan, July 2018. IEEE.
- [CSZM22] Luca Cavaglione, Andreas Schaffhauser, Marco Zuppelli, and Wojciech Mazurczyk. IPv6CC: IPv6 Covert Channels for Testing Networks Against Stegomalware and Data Exfiltration. *SoftwareX*, 17:100975, January 2022.
- [CTJS20] Cyril Cassagnes, Lucian Trestioreanu, Clement Joly, and Radu State. The Rise of eBPF for Non-intrusive Performance Monitoring. In *Proc. of the 2020 IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pages 1–7, Budapest, Hungary, 2020. IEEE.
- [DAFB⁺19] Omar Darwish, Ala Al-Fuqaha, Ghassen B. Brahim, Ilyes Jenhani, and Athanasios Vasilakos. Using Hierarchical Statistical Analysis and Deep Neural Networks to Detect Covert Timing Channels. *Applied Soft Computing*, 82:105546, September 2019.
- [DAMH12] Fatiha Djebbar, Beghdad Ayad, Karim A. Meraim, and Habib Hamam. Comparative Study of Digital Audio Steganography Techniques. *EURASIP Journal on Audio, Speech, and Music Processing*, 2012(1):1–16, October 2012.
- [DD15] Dhananjay M. Dakhane and Prashant R. Deshmukh. Active Warden for TCP Sequence Number Base Covert Channel. In *Proc. of the 2015 International Conference on Pervasive Computing (ICPC)*, pages 1–5, Pune, India, January 2015. IEEE.
- [Der04] Luca Deri. Improving Passive Packet Capture: Beyond Device Polling. In *Proc. of the 4th System Administration and Network Engineering Conference (SANE)*, volume 2004, pages 85–93, Amsterdam, The Netherlands, October 2004.
- [DFP20] Gianni D’Angelo, Massimo Ficco, and Francesco Palmieri. Malware Detection in Mobile Environments Based on Autoencoders and API-images. *Journal of Parallel and Distributed Computing*, 137:26–33, March 2020.
- [DHX⁺18] Derui Ding, Qing-Long Han, Yang Xiang, Xiaohua Ge, and Xian-Ming Zhang. A Survey on Security Control and Attack Detection for Industrial Cyber-physical Systems. *Neurocomputing*, 275:1674–1683, January 2018.
- [DR08] Tim Dierks and Eric Rescorla. RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor, August 2008.

- [DRF⁺11] Christian J. Dietrich, Christian Rossow, Felix C. Freiling, Herbert Bos, Maarten Van Steen, and Norbert Pohlmann. On Botnets That use DNS for Command and Control. In *Proc. of the 2011 7th European Conference on Computer Network Defense (EC2ND)*, pages 9–16, Gothenburg, Sweden, September 2011. IEEE.
- [DSM⁺19] Luca Deri, Samuele Sabella, Simone Mainardi, Pierpaolo Degano, and Roberto Zunino. Combining System Visibility and Security Using eBPF. In Pierpaolo Degano and Roberto Zunino, editors, *Proc. of The Italian Conference on Cyber-Security*, volume 2315 of *CEUR Workshop Proceedings*, Pisa, Italy, February 2019.
- [DWM02] Sorina Dumitrescu, Xiaolin Wu, and Nasir Memon. On Steganalysis of Random LSB Embedding in Continuous-tone Images. In *Proc. of the International Conference on Image Processing (ICIP)*, volume 3, pages 641–644, New York, NY, USA, 2002. IEEE.
- [DWW02] Sorina Dumitrescu, Xiaolin Wu, and Zhe Wang. Detection of LSB Steganography via Sample Pair Analysis. In Fabien A. P. Petitcolas, editor, *Information Hiding*, volume 2578 of *Lecture Notes in Computer Science*, pages 355–372, Berlin, Heidelberg, December 2002. Springer.
- [EKMV04] Cristian Estan, Ken Keys, David Moore, and George Varghese. Building a Better NetFlow. *ACM SIGCOMM Computer Communication Review*, 34(4):245–256, October 2004.
- [Eur20] European Telecommunications Standards Institute. System Architecture for the 5G System. Technical Specification 3GPP TS 23.501, October 2020.
- [FG02] Jessica Fridrich and Miroslav Goljan. Practical Steganalysis of Digital Images: State of the art. *Security and Watermarking of Multimedia Contents IV*, 4675:1–13, April 2002.
- [FGD01] Jessica Fridrich, Miroslav Goljan, and Rui Du. Reliable Detection of LSB Steganography in Color and Grayscale Images. In *Proc. of the 2001 Workshop on Multimedia & Security: new Challenges (MM&Sec)*, pages 27–30, New York, NY, USA, October 2001. ACM.
- [FGP19] Gianluigi Folino, Massimo Guarascio, and Giuseppe Papuzzo. Exploiting Fractal Dimension and a Distributed Evolutionary Approach to Classify Data Streams With Concept Drifts. *Applied Soft Computing*, 75:284–297, February 2019.
- [FMS12] Wojciech Fraczek, Wojciech Mazurczyk, and Krzysztof Szczypiorski. Hiding Information in a Stream Control Transmission Protocol. *Computer Communications*, 35(2):159–169, January 2012.

- [FPK07] Jessica Fridrich, Tomáš Pevný, and Jan Kodovský. Statistically Undetectable JPEG Steganography: Dead Ends Challenges, and Opportunities. In *Proc. of the 9th Workshop on Multimedia & Security (MM&Sec)*, pages 3–14, Dallas, TX, USA, September 2007. ACM.
- [GAG⁺19] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *Proc. of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1069–1084, Phoenix, AZ, USA, June 2019. ACM.
- [GGK⁺17] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In *Proc. of the 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 237–248, Denver, CO, USA, June 2017. IEEE.
- [Gir15] Ross Girshick. Fast R-CNN. In *Proc. of the 2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448, Santiago, Chile, December 2015. IEEE.
- [GMP20] Daniel Gibert, Carles Mateu, and Jordi Planes. The Rise of Machine Learning for Detection and Classification of Malware: Research Developments, Trends and Challenges. *Journal of Network and Computer Applications*, 153:102526, March 2020.
- [GMR18] Massimo Guarascio, Giuseppe Manco, and Ettore Ritacco. Deep Learning. *Encyclopedia of Bioinformatics and Computational Biology: ABC of Bioinformatics*, 1-3:634–647, 2018.
- [GRK13] Xun Gong, Mavis Rodrigues, and Negar Kiyavash. Invisible Flow Watermarks for Channels With Dependent Substitution, Deletion, and Bursty Insertion Errors. *IEEE Transactions on Information Forensics and Security*, 8(11):1850–1859, August 2013.
- [GW18] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Prentice Hall, 2018.
- [HJBB⁺18] Toke Høiland-Jørgensen, Jesper D. Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proc. of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, page 54–66, Heraklion, Greece, December 2018. ACM.

- [HJP⁺15] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, CA, USA, May 2015.
- [HJYH18] Jibum Hong, Seyeon Jeong, Jae-Hyoung Yoo, and James W.-K. Hong. Design and Implementation of eBPF-based Virtual TAP for Inter-VM Traffic Monitoring. In *Proc. of the 2018 14th International Conference on Network and Service Management (CNSM)*, Rome, Italy, November 2018. IEEE.
- [HKR18] Danny Hendler, Shay Kels, and Amir Rubin. Detecting Malicious PowerShell Commands Using Deep Neural Networks. In *Proc. of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS)*, pages 187–197, Nagasaki, Japan, May 2018. ACM.
- [HMC20] Corinna Heinz, Wojciech Mazurczyk, and Luca Cavaglione. Covert Channels in Transport Layer Security. In *Proc. of the European Interdisciplinary Cybersecurity Conference (EICC)*, pages 1–6, Rennes, France, November 2020. ACM.
- [HP09] John Heidemann and Christos Papadopoulos. Uses and Challenges for Network Datasets. In *Proc. of the 2009 Cybersecurity Applications & Technology Conference for Homeland Security (CATCH)*, pages 73–82, Washington, DC, USA, March 2009. IEEE.
- [HS96] Theodore G. Handel and Maxwell T. Sandford. Hiding Data in the OSI Network Model. In Ross Anderson, editor, *Information Hiding*, volume 1174 of *Lecture Notes in Computer Science*, pages 23–38, Cambridge, United Kingdom, January 1996. Springer, Springer Berlin Heidelberg.
- [HS06] Geoffrey E. Hinton and Ruslan R. Salakhutdinov. Reducing the Dimensionality of Data With Neural Networks. *Science*, 313(5786):504–507, July 2006.
- [HSK⁺12] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving Neural Networks by Preventing Co-adaptation of Feature Detectors. *arXiv preprint arXiv:1207.0580*, July 2012.
- [HWI⁺18] Mehdi Hussain, Ainuddin W. A. Wahab, Yamani I. B. Idris, Anthony T. S. Ho, and Ki-Hyun Jung. Image Steganography in Spatial Domain: A Survey. *Signal Processing: Image Communication*, 65:46–66, July 2018.
- [HY21] Abdelhakim Hannousse and Salima Yahiouche. Handling Webshell Attacks: A Systematic Mapping and Survey. *Computers & Security*, 108:102366, September 2021.

- [HYM⁺20] Ruidong Han, Chao Yang, Jianfeng Ma, Siqi Ma, Yunbo Wang, and Feng Li. IMShell-Dec: Pay More Attention to External Links in PowerShell. In Marko Hölbl, Kai Rannenberg, and Tatjana Welzer, editors, *ICT Systems Security and Privacy Protection*, IFIP Advances in Information and Communication Technology, pages 189–202, Maribor, Slovenia, September 2020. Springer, Springer International Publishing.
- [HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proc. of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, Las Vegas, NV, USA, June 2016. IEEE.
- [IE16] Alfonso Iacovazzi and Yuval Elovici. Network Flow Watermarking: A Survey. *IEEE Communications Surveys & Tutorials*, 19(1):512–530, September 2016.
- [IMAZ22] Félix Iglesias, Fares Meghdouri, Robert Annessi, and Tanja Zseby. CCgen: Injecting Covert Channels Into Network Traffic. *Security and Communication Networks*, 2022, May 2022.
- [IS15] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proc. of the 32nd International Conference on Machine Learning (ICML)*, volume 37, pages 448–456, Lille, France, July 2015. PMLR.
- [JLE20] Dong-Seob Jung, Sang-Joon Lee, and Ieek-Chae Euom. ImageDetox: Method for the Neutralization of Malicious Code Hidden in Image Files. *Symmetry*, 12(10):1621, September 2020.
- [Ker04] Andrew D. Ker. Quantitative Evaluation of Pairs and RS Steganalysis. In *Proc. of the Security, Steganography, and Watermarking of Multimedia Contents VI*, volume 5306, pages 83–97, San Jose, CA, USA, June 2004. SPIE.
- [KKP18] Konstantinos Karampidis, Ergina Kavallieratou, and Giorgos Papadourakis. A Review of Image Steganalysis Techniques for Digital Forensics. *Journal of Information Security and Applications*, 40:217–235, June 2018.
- [KM93] Myong H. Kang and Ira S. Moskowitz. A Pump for Rapid, Reliable, Secure Communication. In *Proc. of the 1st ACM Conference on Computer and Communications Security (CCS)*, pages 119–129, Fairfax, NY, USA, December 1993. ACM.
- [KMC⁺00] Eddie W. Kohler, Robert T. Morris, Benjie Chen, John Jannotti, and Frans M. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3), August 2000.

- [KML96] Myong H. Kang, Ira S. Moskowitz, and Daniel C. Lee. A Network Pump. *IEEE Transactions on Software Engineering*, 22(5):329–338, May 1996.
- [KTB17] R. Bala Krishnan, Prasanth K. Thandra, and M. Sai Baba. An Overview of Text Steganography. In *Proc. of the 2017 4th International Conference on Signal Processing, Communication and Networking (ICSCN)*, pages 1–6, Chennai, India, March 2017. IEEE.
- [KWE⁺16] Jaspreet Kaur, Steffen Wendzel, Omar Eissa, Jernej Tonejc, and Michael Meier. Covert Channel-internal Control Protocols: Attacks and Defense. *Security and Communication Networks*, 9(15):2986–2997, April 2016.
- [KWJ21] Tomasz Koziak, Katarzyna Wasielewska, and Artur Janicki. How to Make an Intrusion Detection System Aware of Steganographic Transmission. In *Proc. of the European Interdisciplinary Cybersecurity Conference (EICC)*, pages 77–82, Targu Mures, Romania (Online), November 2021. ACM.
- [KWZ⁺18] Ralf Keidel, Steffen Wendzel, Sebastian Zillien, Eric S. Conner, and Georg Haas. WoDiCoF-A Testbed for the Evaluation of (Parallel) Covert Channel Detection Algorithms. *Journal of Universal Computer Science*, 24(5):556–576, April 2018.
- [LAE⁺16] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single Shot MultiBox Detector. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – European Conference on Computer Vision 2016*, Lecture Notes in Computer Science, pages 21–37, Amsterdam, The Netherlands, October 2016. Springer, Springer International Publishing.
- [Lam73] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, October 1973.
- [LB95] Yann LeCun and Yoshua Bengio. Convolutional Networks for Images, Speech, and Time Series. *The Handbook of Brain Theory and Neural Networks*, 3361(10):1995, June 1995.
- [LBD⁺89] Yann LeCun, Bernhard Boser, John Denker, Donnie Henderson, Richard Howard, Wayne Hubbard, and Lawrence Jackel. Handwritten Digit Recognition with a Back-propagation Network. *Advances in Neural Information Processing Systems 2*, 2, November 1989.
- [LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 521(7553):436–444, May 2015.

- [LHH10] Weiqi Luo, Fangjun Huang, and Jiwu Huang. Edge Adaptive Image Steganography Based on LSB Matching Revisited. *IEEE Transactions on Information Forensics and Security*, 5(2):201–214, February 2010.
- [LLC05] Norka B. Lucena, Grzegorz Lewandowski, and Steve J. Chapin. Covert Channels in IPv6. In George Danezis and David Martin, editors, *Privacy Enhancing Technologies*, volume 3856 of *Lecture Notes in Computer Science*, pages 147–166, Cavtat, Croatia, May 2005. Springer, Springer Berlin Heidelberg.
- [LLL06] Wen-Nung Lie, T.C-L. Lin, and Chia-Wen Lin. Enhancing Video Error Resilience by Using Data-embedding Techniques. *IEEE Transactions on Circuits and Systems for Video Technology*, 16(2):300–308, February 2006.
- [LLW⁺12] Jianxin Li, Bo Li, Tianyu Wo, Chunming Hu, Jinpeng Huai, Lu Liu, and K.P. Lam. CyberGuarder: A Virtualization Security Assurance Architecture for Green Cloud Computing. *Future Generation Computer Systems*, 28(2):379–390, February 2012.
- [LSD15] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation. In *Proc. of the 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, Boston, MA, USA, June 2015. IEEE.
- [LW13] Jean-Francois Lalande and Steffen Wendzel. Hiding Privacy Leaks in Android Applications Using Low-attention Raising Covert Channels. In *Proc. of the 2013 International Conference on Availability, Reliability and Security (ARES)*, pages 701–710, Regensburg, Germany, September 2013. IEEE.
- [LWWL08] Xiang-Yang Luo, Dao-Shun Wang, Ping Wang, and Fen-Lin Liu. A Review on Blind Detection for Image Steganography. *Signal Processing*, 88(9):2138–2157, September 2008.
- [MAF⁺15] Khan Muhammad, Jamil Ahmad, Haleem Farman, Zahoor Jan, Muhammad Sajjad, and Sung W. Baik. A Secure Method for Color Image Steganography Using Gray-level Modification and Multi-level Encryption. *KSII Transactions on Internet and Information Systems*, 9(5):1938–1962, April 2015.
- [Maz13] Wojciech Mazurczyk. VoIP Steganography and its Detection - A Survey. *ACM Computing Surveys*, 46(2):1–21, November 2013.
- [MC14] Wojciech Mazurczyk and Luca Caviglione. Steganography in Modern Smartphones and Mitigation Techniques. *IEEE Communications Surveys & Tutorials*, 17(1):334–357, August 2014.

- [MC15] Wojciech Mazurczyk and Luca Caviglione. Information Hiding as a Challenge for Malware Detection. *IEEE Security & Privacy*, 13(2):89–93, March 2015.
- [MC21] Wojciech Mazurczyk and Luca Caviglione. Cyber Reconnaissance Techniques. *Communications of the ACM*, 64(3):86–95, March 2021.
- [ME21] A. Monika and R. Eswari. Ensemble-based Stegomalware Detection System for Hidden Ransomware Attack. In V. Suma, Joy I.-Z. Chen, Zubair Baig, and Haoxiang Wang, editors, *Inventive Systems and Control*, volume 204 of *Lecture Notes in Networks and Systems*, pages 599–619. Springer, Coimbatore, India, January 2021.
- [MEO05] Tayana Morkel, Jan H.P. Eloff, and Martin S. Olivier. An Overview of Image Steganography. In *Proc. of the ISSA 2005 New Knowledge Today Conference*, volume 1, pages 1–11, Sandton, South Africa, June 2005.
- [MG19a] Samaneh Mahdavifar and Ali A. Ghorbani. Application of Deep Learning to Cybersecurity: A Survey. *Neurocomputing*, 347:149–176, June 2019.
- [MG19b] Łukasz Makowski and Paola Grosso. Evaluation of Virtualization and Traffic Filtering Methods for Container Networks. *Future Generation Computer Systems*, 93:345–357, April 2019.
- [MJ15] Justin Merrill and Daryl Johnson. Covert Channels in SSL Session Negotiation Headers. In *Proc. of the 2015 International Conference on Security and Management (SAM)*, page 70, Las Vegas, NV, USA, July 2015.
- [MLR⁺07] Kieran Mansley, Greg Law, David Riddoch, Guido Barzini, Neil Turton, and Steven Pope. Getting 10 Gb/s from Xen: Safe and Fast Device Access From Unprivileged Domains. In Luc Bougé, Martti Forsell, Jesper L. Träff, Achim Streit, Wolfgang Ziegler, Michael Alexander, and Stephen Childs, editors, *EuroPar 2007 Workshops: Parallel Processing*, Lecture Notes in Computer Science, pages 224–233, Rennes, France, August 2007. Springer, Springer Berlin Heidelberg.
- [MP21] Julián D. Miranda and Diego J. Parada. LSB Steganography Detection in Monochromatic Still Images Using Artificial Neural Networks. *Multimedia Tools and Applications*, pages 1–21, September 2021.
- [MPC19] Wojciech Mazurczyk, Krystian Powójski, and Luca Caviglione. IPv6 Covert Channels in the Wild. In *Proc. of the 3rd Central European Cybersecurity Conference (CECC)*, pages 1–6, Munich, Germany, November 2019. ACM.

- [MRB17] Peter McLaren, Gordon Russell, and Bill Buchanan. Mining Malware Command and Control Traces. In *Proc. of the 2017 Computing Conference*, pages 788–794, London, United Kingdom, July 2017. IEEE.
- [MRFC12] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the Communication Between Colluding Applications on Modern Smartphones. In *Proc. of the 28th Annual Computer Security Applications Conference (ACSAC)*, pages 51–60, Orlando, FL, USA, December 2012. ACM.
- [MS08] Wojciech Mazurczyk and Krzysztof Szczypiorski. Covert Channels in SIP for VoIP signalling. In Hamid Jahankhani, Kenneth Revett, and Dominic Palmer-Brown, editors, *Global E-Security, Communications in Computer and Information Science*, pages 65–72, London, United Kingdom, June 2008. Springer, Springer Berlin Heidelberg.
- [MSWC19] Wojciech Mazurczyk, Przemysław Szary, Steffen Wendzel, and Luca Cavaglione. Towards Reversible Storage Network Covert Channels. In *Proc. of the 14th International Conference on Availability, Reliability and Security (ARES)*, pages 1–8, Canterbury, United Kingdom, August 2019. ACM.
- [MW17] Wojciech Mazurczyk and Steffen Wendzel. Information Hiding: Challenges for Forensic Experts. *Communications of the ACM*, 61(1):86–94, January 2017.
- [MWCK19] Wojciech Mazurczyk, Steffen Wendzel, Mehdi Chourib, and Jörg Keller. Countering Adaptive Network Covert Communication With Dynamic Wardens. *Future Generation Computer Systems*, 94:712–725, May 2019.
- [NBHC⁺19] Nataliia Neshenko, Elias Bou-Harb, Jorge Crichigno, Georges Kaddoum, and Nasir Ghani. Demystifying IoT Security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-scale IoT Exploitations. *IEEE Communications Surveys & Tutorials*, 21(3):2702–2733, April 2019.
- [NH10] Vinod Nair and Geoffrey E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proc. of the 27th International Conference on International Conference on Machine Learning (ICML)*, pages 807–814, Haifa, Israel, June 2010.
- [NHH15] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning Deconvolution Network for Semantic Segmentation. In *Proc. of the 2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1520–1528, Santiago, Chile, December 2015. IEEE.
- [NKK⁺11] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y. Ng. Multimodal Deep Learning. In *Proc. of the 28th International*

Conference on International Conference on Machine Learning (ICML), pages 689–696, Bellevue, WA, USA, July 2011.

- [NZCM20] Piotr Nowakowski, Piotr Zórawski, Krzysztof Cabaj, and Wojciech Mazurczyk. Network Covert Channels Detection Using Data Mining and Hierarchical Organisation of Frequent Sets: An Initial Study. In *Proc. of the 15th International Conference on Availability, Reliability and Security (ARES)*, pages 1–10, Dublin, Ireland (Online), August 2020. ACM.
- [OCL17] Eric Olson, Larry Carter, and Qingzhong Liu. A Comparison Study Using StegExpose for Steganalysis. *International Journal of Knowledge Engineering*, June 2017.
- [PC04] Vidyasagar M. Potdar and Elizabeth Chang. Grey Level Modification Steganography for Secret Communication. In *Proc. of the 2nd IEEE International Conference on Industrial Informatics (INDIN)*, pages 223–228, Berlin, Germany, June 2004. IEEE.
- [PCK⁺20] Damian Puchalski, Luca Cavaglione, Rafał Kozik, Adrian Marzecki, Sławomir Krawczyk, and Michał Choraś. Stegomalware Detection Through Structural Analysis of Media Files. In *Proc. of the 15th International Conference on Availability, Reliability and Security (ARES)*, pages 1–6, Dublin, Ireland (Online), August 2020. ACM.
- [PD12] Hardik Patel and Preeti Dave. Steganography Technique Based on DCT Coefficients. *International Journal of Engineering Research and Applications*, 2(1):713–717, January 2012.
- [PKKK16] Tomáš Pevný, Martin Kopp, Jakub Křoustek, and Andrew D. Ker. Malicons: Detecting Payload in Favicons. *Electronic Imaging*, 2016(8):1–9, February 2016.
- [PKP⁺19] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. Continual Lifelong Learning With Neural Networks: A Review. *Neural Networks*, 113:54–71, May 2019.
- [PKSJ22] Mikołaj Płachta, Marek Krzemień, Krzysztof Szczypiorski, and Artur Janicki. Detection of Image Steganography Using Deep Learning and Ensemble Classifiers. *Electronics*, 11(10):1565, May 2022.
- [PLL20] I-Hui Pan, Kung-Chin Liu, and Chiang-Lung Liu. Chi-Square Detection for PVD Steganography. In *Proc. of the 2020 International Symposium on Computer, Consumer and Control (IS3C)*, pages 30–33, Taichung City, Taiwan, November 2020. IEEE.

- [PNFR15] Michele Paolino, Nikolay Nikolaev, Jeremy Fanguede, and Daniel Raho. Snabb-Switch User Space Virtual Switch Benchmark and Performance Optimization for NFV. In *Proc. of the 2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pages 86–92, San Francisco, CA, USA, November 2015. IEEE.
- [PR11] Steve Pope and David Riddoch. Introduction to OpenOnload—building Application Transparency and Protocol Conformance Into Application Acceleration Middleware. Technical report, Solarflare Communication, April 2011.
- [PSW16] Simon Parkinson, Vassiliki Somaraki, and Rupert Ward. Auditing File System Permissions Using Association Rule Mining. *Expert Systems with Applications*, 55:274–283, August 2016.
- [RC21] Matteo Repetto and Alessandro Carrega. Efficient Flow Monitoring for Virtualized Applications With eBPF. Technical report, ASTRID project, July 2021.
- [RCL19] Matteo Repetto, Alessandro Carrega, and Guerino Lamanna. An Architecture to Manage Security Services for Cloud Applications. In *Proc. of the 2019 4th International Conference on Computing, Communications and Security (ICCCS)*, pages 1–8, Rome, Italy, October 2019. IEEE.
- [RCL⁺21] Matteo Repetto, Alessandro Carrega, Guerino Lamanna, Jaloliddin Yusupov, Orazio Toscano, Gianmarco Bruno, Michele Nuovo, and Marco Cappelli. Leveraging the 5G Architecture to Mitigate Amplification Attacks. In *Proc. of the 2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, pages 443–449, Tokyo, Japan, June 2021. IEEE.
- [RCR21] Matteo Repetto, Alessandro Carrega, and Riccardo Rapuzzi. An Architecture to Manage Security Operations for Digital Service Chains. *Future Generation Computer Systems*, 115:251–266, February 2021.
- [RCZ21] Matteo Repetto, Luca Caviglione, and Marco Zuppelli. bccstego: A Framework for Investigating Network Covert Channels. In *Proc. of the 16th International Conference on Availability, Reliability and Security (ARES)*, pages 1–7, Vienna, Austria, August 2021. ACM.
- [RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-time Object Detection. In *Proc. of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, Las Vegas, NV, USA, June 2016. IEEE.
- [RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Medical Image Computing*

and *Computer-Assisted Intervention*, Lecture Notes in Computer Science, pages 234–241, Munich, Germany, October 2015. Springer, Springer International Publishing.

- [RHS17] Fahimeh Rezaei, Michael Hempel, and Hamid Sharif. Towards a Reliable Detection of Covert Timing Channels Over Real-time Network Traffic. *IEEE Transactions on Dependable and Secure Computing*, 14(3):249–264, January 2017.
- [Riz12] Luigi Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *Proc. of the 21st USENIX Security Symposium (USENIX Security)*, pages 101–112, Bellevue, WA, USA, August 2012.
- [ROL13] Ruben Rios, Jose A. Onieva, and Javier Lopez. Covert Communications Through Network Configuration Messages. *Computers & Security*, 39:34–46, November 2013.
- [RR18] Riccardo Rapuzzi and Matteo Repetto. Building Situational Awareness for Network Threats in Fog/Edge Computing: Emerging Paradigms Beyond the Security Perimeter Model. *Future Generation Computer Systems*, 85:235–249, August 2018.
- [RRG19] Tabares-Soto Reinel, Ramos-Pollan Raul, and Isaza Gustavo. Deep Learning Applied to Steganalysis of Digital Images: A Systematic Review. *IEEE Access*, 7:68970–68990, May 2019.
- [RWS⁺19] Markus Ring, Sarah Wunderlich, Deniz Scheuring, Dieter Landes, and Andreas Hotho. A Survey of Network-based Intrusion Detection Data Sets. *Computers & Security*, 86:147–167, September 2019.
- [SCPA19] Nasrin Sultana, Naveen Chilamkurti, Wei Peng, and Rabei Alhadad. Survey on SDN Based Network Intrusion Detection System Using Machine Learning Approaches. *Peer-to-Peer Networking and Applications*, 12(2):493–501, January 2019.
- [SGL⁺18] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying IoT Devices in Smart Environments Using Network Traffic Characteristics. *IEEE Transactions on Mobile Computing*, 18(8):1745–1759, August 2018.
- [SHK⁺14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple way to Prevent Neural Networks From Overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, June 2014.

- [SHRS15] Pradhumna L. Shrestha, Michael Hempel, Fahimeh Rezaei, and Hamid Sharif. A Support Vector Machine-based Framework for Detection of Covert Timing Channels. *IEEE Transactions on Dependable and Secure Computing*, 13(2):274–283, April 2015.
- [Sim84] Gustavus J. Simmons. The Prisoners’ Problem and the Subliminal Channel. In David Chaum, editor, *Advances in Cryptology: Proceedings of Crypto 83*, pages 51–67, Boston, MA, USA, 1984. Springer, Springer US.
- [SK15] Sudhanshi Sharma and Umesh Kumar. Review of Transform Domain Techniques for Image Steganography. *International Journal of Science and Research*, 2(2):1, April 2015.
- [SKM15] Mennatallah M. Sadek, Amal S. Khalifa, and Mostafa G.M. Mostafa. Video Steganography: A Comprehensive Review. *Multimedia Tools and Applications*, 74(17):7063–7094, March 2015.
- [SL09] Marina Sokolova and Guy Lapalme. A Systematic Analysis of Performance Measures for Classification Tasks. *Information Processing & Management*, 45(4):427–437, July 2009.
- [SLJ⁺15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper With Convolutions. In *Proc. of the 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, Boston, MA, USA, June 2015. IEEE.
- [SMP17] Abdulrahman Salih, Xiaoqi Ma, and Evtim Peytchev. Implementation of Hybrid Artificial Intelligence Technique to Detect Covert Channels Attack in new Generation Internet Protocol IPv6. In Rachid Benlamri and Michael Sparer, editors, *Leadership, Innovation and Entrepreneurship as Driving Forces of the Global Economy*, Springer Proceedings in Business and Economics, pages 173–190. Springer International Publishing, Dubai, UAE, April 2017.
- [Sti08] Richard M. Stillman. Detecting IP Covert Timing Channels by Correlating Packet Timing With Memory Content. In *Proc. of the IEEE SoutheastCon 2008*, pages 204–209, Huntsville, AL, USA, April 2008. IEEE.
- [STTPL14] Guillermo Suarez-Tangil, Juan E. Tapiador, and Pedro Peris-Lopez. Stegomalware: Playing Hide and Seek With Malicious Components in Smartphone Apps. In Dongdai Lin, Moti Yung, and Jianying Zhou, editors, *Information Security and Cryptology*, volume 8957 of *Lecture Notes in Computer Science*, pages 496–515, Beijing, China, December 2014. Springer, Springer International Publishing.

- [SVS13] Steffen Schulz, Vijay Varadharajan, and Ahmad-Reza Sadeghi. The Silence of the LANs: Efficient Leakage Resilience for IPsec VPNs. *IEEE Transactions on Information Forensics and Security*, 9(2):221–232, November 2013.
- [SW22] Tobias Schmidbauer and Steffen Wendzel. SoK: A Survey Of Indirect Network-level Covert Channels. In *Proc. of the 2022 ACM on Asia Conference on Computer and Communications Security (ASIACCS)*, pages 546–560, Nagasaki, Japan, May 2022. ACM.
- [SZ14] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-scale Image Recognition. *arXiv preprint arXiv:1409.1556*, September 2014.
- [SZCR18] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. vNetTracer: Efficient and Programmable Packet Tracing in Virtualized Networks. In *Proc. of the IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 165–175, Vienna, Austria, July 2018. IEEE.
- [SZZ⁺11] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, volume 11, pages 17–33, San Diego, CA, USA, February 2011.
- [SZZW19] Yu Sun, Hao Zhang, Tao Zhang, and Ran Wang. Deep Neural Networks for Efficient Steganographic Payload Location. *Journal of Real-Time Image Processing*, 16(3):635–647, January 2019.
- [TA05] Eugene Tumoian and Maxim Anikeev. Network Based Detection of Passive Covert Channels in TCP/IP. In *Proc. of the IEEE Conference on Local Computer Networks (LCN)*, pages 802–809, Sydney, Australia, November 2005. IEEE.
- [TL07] Takehiro Takahashi and Wenke Lee. An Assessment of VoIP Covert Channel Threats. In *Proc. of the 2007 3rd International Conference on Security and Privacy in Communications Networks (SecureComm)*, pages 371–380, Nice, France, September 2007. IEEE.
- [TL14] Shunquan Tan and Bin Li. Stacked Convolutional Auto-encoders for Steganalysis of Digital Images. In *Proc. of the Signal and Information Processing Association Annual Summit and Conference (APSIPA)*, pages 1–4, Siem Reap, Cambodia, December 2014. IEEE.
- [TL20] Ankit Thakkar and Ritika Lohiya. A Review of the Advancement in Intrusion Detection Datasets. *Procedia Computer Science*, 167:636–645, April 2020.

- [UMLC17] Marcin Urbanski, Wojciech Mazurczyk, Jean-Francois Lalande, and Luca Caviglione. Detecting Local Covert Channels Using Process Activity Correlation on Android Smartphones. *International Journal of Computer Systems Science and Engineering*, 32(2):71–80, January 2017.
- [VLBM08] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and Composing Robust Features With Denoising Autoencoders. In *Proc. of the 25th International Conference on Machine Learning (ICML)*, pages 1096–1103, Helsinki, Finland, July 2008. ACM.
- [VMS19] Aleksandar Velinov, Aleksandra Mileva, and Done Stojanov. Power Consumption Analysis of the new Covert Channels in CoAP. *International Journal On Advances in Security*, 12(1 & 2):42–52, August 2019.
- [VMS22] Vinita Verma, Sunil K. Muttoo, and Vir B. Singh. Detecting Stegomalware: Malicious Image Steganography and Its Intrusion in Windows. In Udai Pratap Rao, Sankita J. Patel, Pethuru Raj, and Andrea Visconti, editors, *Security, Privacy and Data Analytics*, Lecture Notes in Electrical Engineering, pages 103–116. Springer Singapore, Surat, India, April 2022.
- [VMWM19] Aleksandar Velinov, Aleksandra Mileva, Steffen Wendzel, and Wojciech Mazurczyk. Covert Channels in the MQTT-based Internet of Things. *IEEE Access*, 7:161899–161915, November 2019.
- [WCM⁺21] Steffen Wendzel, Luca Caviglione, Wojciech Mazurczyk, Aleksandra Mileva, Jana Dittmann, Christian Krätzer, Kevin Lamshöft, Claus Vielhauer, Laura Hartmann, Jörg Keller, and Tom Neubert. A Revised Taxonomy of Steganography Embedding Patterns. In *Proc. of the 16th International Conference on Availability, Reliability and Security (ARES)*, pages 1–12, Vienna, Austria, August 2021. ACM.
- [WCM⁺22] Steffen Wendzel, Luca Caviglione, Wojciech Mazurczyk, Aleksandra Mileva, Jana Dittmann, Christian Krätzer, Kevin Lamshöft, Claus Vielhauer, Laura Hartmann, Jörg Keller, Tom Neubert, and Sebastian Zillien. A Generic Taxonomy for Steganography Methods. *10.36227/techrxiv.20215373.v2*, July 2022.
- [WCTS21] Shuhong Wu, Yonghong Chen, Hui Tian, and Chonggao Sun. Detection of Covert Timing Channel Based on Time Series Symbolization. *IEEE Open Journal of the Communications Society*, 2:2372–2382, October 2021.
- [WHWS20] Hui Wu, Haiting Han, Xiao Wang, and Shengli Sun. Research on Artificial Intelligence Enhancing Internet of Things Security: A Survey. *IEEE Access*, 8:153826–153848, August 2020.

- [WK12] Steffen Wendzel and Jörg Keller. Design and Implementation of an Active Warden Addressing Protocol Switching Covert Channels. In *Proc. of the 7th International Conference on Internet Monitoring and Protection (ICIMP)*, pages 1–6, Stuttgart, Germany, May 2012. IARIA.
- [WM16] Steffen Wendzel and Wojciech Mazurczyk. POSTER: An Educational Network Protocol for Covert Channel Analysis Using Patterns. In *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1739–1741, Vienna, Austria, October 2016.
- [WP99] Andreas Westfeld and Andreas Pfitzmann. Attacks on Steganographic Systems. In *Proc. of the 3rd International Workshop on Information Hiding*, pages 61–76, Dresden, Germany, September 1999. Springer.
- [WT03] Da-Chun Wu and Wen-Hsiang Tsai. A Steganographic Method for Images by Pixel-Value Differencing. *Pattern Recognition Letters*, 24(9-10):1613–1626, June 2003.
- [WYH16] Changda Wang, Yulin Yuan, and Lei Huang. Base Communication Model of IP Covert Timing Channels. *Frontiers of Computer Science*, 10(6):1130–1141, July 2016.
- [WZFH15] Steffen Wendzel, Sebastian Zander, Bernhard Fechner, and Christian Herdin. Pattern-based Survey and Categorization of Network Covert Channel Techniques. *ACM Computing Surveys*, 47(3):1–26, April 2015.
- [XKC20] Jiarong Xing, Qiao Kang, and Ang Chen. NetWarden: Mitigating Network Covert Channels While Preserving Performance. In *Proc. of the 29th USENIX Security Symposium (USENIX Security)*, pages 2039–2056, Boston, MA, USA, August 2020.
- [YA03] Alper Yilmaz and A. Aydin Alatan. Error Concealment of Video Sequences by Data Hiding. In *Proc. of the 2003 International Conference on Image Processing*, volume 2, pages II–679, Barcelona, Spain, September 2003. IEEE.
- [YY10] Ilsun You and Kangbin Yim. Malware Obfuscation Techniques: A Brief Survey. In *Proc. of the 2010 International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA)*, pages 297–300, Fukuoka, Japan, November 2010. IEEE.
- [YY18] Ping Yan and Zheng Yan. A Survey on Dynamic Mobile Malware Detection. *Software Quality Journal*, 26(3):891–919, May 2018.

- [ZA08] Sebastian Zander and Grenville Armitage. CCHEF–Covert Channels Evaluation Framework Design and Implementation. *CAIA Technical Report NO. 080530A*, September 2008.
- [ZAB06] Sebastian Zander, Grenville Armitage, and Philip Branch. Covert Channels in the IP Time to Live Field. In *Proc. of the Australian Telecommunication Networks and Application Conference (ATNAC)*, Melbourne, Australia, December 2006.
- [ZAB07] Sebastian Zander, Grenville Armitage, and Philip Branch. A Survey of Covert Channels and Countermeasures in Computer Network Protocols. *IEEE Communications Surveys & Tutorials*, 9(3):44–57, September 2007.
- [ZJRR12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their use to Extract Private Keys. In *Proc. of the 2012 ACM Conference on Computer and Communications Security (CCS)*, pages 305–316, Raleigh, NC, USA, October 2012. ACM.
- [ZLD10] Jiangtao Zhai, Guangjie Liu, and Yuewei Dai. A Covert Channel Detection Algorithm Based on TCP Markov Model. In *Proc. of the 2010 International Conference on Multimedia Information Networking and Security (MINES)*, pages 893–897, Nanjing, China, November 2010. IEEE.
- [ZMCG21] Marco Zuppelli, Giuseppe Manco, Luca Caviglione, and Massimo Guarascio. Sanitization of Images Containing Stegomalware via Machine Learning Approaches. In Michele Colajanni Alessandro Armando, editor, *Proc. of the Italian Conference on Cybersecurity*, CEUR Workshop Proceedings, pages 374–386, Online, April 2021.
- [ZNA12] Sebastian Zander, Thuy Nguyen, and Grenville Armitage. Sub-flow Packet Sampling for Scalable ML Classification of Interactive Traffic. In *Proc. of the 37th IEEE Conference on Local Computer Networks (LCN)*, pages 68–75, Clearwater Beach, FL, USA, October 2012. IEEE.
- [ZS13] Hong Zhao and Yun-Qing Shi. Detecting Covert Channels in Computer Networks Based on Chaos Theory. *IEEE Transactions on Information Forensics and Security*, 8(2):273–282, December 2013.

Appendix A

Software and Datasets

A.1 Software

The software developed during this Thesis is licensed under MIT, GNU GPL, and Creative Commons and it is available in the repository:

<https://github.com/Ocram95/Software-Thesis>.

Part of the software has been funded by the project SIMARGL (Grant Agreement No. 833042).

- [S1] `IPv6CC`: it allows to create storage network covert channels within IPv6 traffic in a Man-in-the-Middle flavor for pentesting purposes. This software is discussed in Chapter 2 and used in Chapters 3 and 4.
- [S2] `pcapStego`: it allows to create storage and timing network covert channels within IPv4, IPv6, ICMP, and ICMPv6 traffic, starting from pre-collected traffic traces. This software is discussed in Chapter 2 and used in Chapters 2, 5, and 6.
- [S3] `bccstego`: it allows to collect network data in a bin-based data structure via the eBPF technology. This software is discussed in Chapter 3 and used from Chapter 4 to Chapter 7.
- [S4] `eBPF Framework Simulator`: it allows to create datasets to reveal network covert channels. This software is used in Chapters 4 and 5.
- [S5] `Cabuk eBPF`: it implements an algorithm to detect timing covert channels via eBPF technology. This software is used in Chapters 5 and 6.
- [S6] `libpcap Filter`: it allows to collect network data via the libpcap library in a bin-based data structure to enable the detection of network covert channels. This software is used in Chapters 5 and 6.

[S7] `zeek-stego`: it is an extension for Zeek to collect per-packet network information in a bin-based data structure to enable the detection of network covert channels. This software is used in Chapters 5 and 6.

Additional software developed to test some ideas presented in this Thesis within the framework of the project SIMARGL (Grant Agreement No. 833042):

[S8] `BCCSIMARGLToolkit`: it allows to analyze and process data collected by the eBPF technology and enables the detection of network covert channels.

[S9] `Mavis`: it allows to reveal images containing PowerShell scripts embedded via the Invoke-PSImage technique.

A.2 Datasets

[D1] `Stego-Favicons-Dataset`: it is composed of 430,000 favicons (size: 32×32 pixels) containing malicious PHP and URLs payloads and embedded with LSB steganography. Produced in collaboration with the Institute for High Performance Computing and Networking (ICAR) of the National Research Council of Italy (CNR). This dataset is discussed and used in Chapter 8.

Available online at:

<https://github.com/Ocram95/Stego-Favicons-Dataset>

[D2] `Stego-Images-Dataset`: it is composed of 44,000 images (size: 512×512 pixels) containing malicious JavaScript, HTML, PowerShell, URLs and Ethereum addresses and embedded with LSB steganography. Produced in collaboration with ICAR-CNR. This dataset is discussed and used in Chapter 9.

Available online at:

<https://www.kaggle.com/datasets/marcozuppelli/stegoimagesdataset>

Appendix B

Publications Used in This Thesis

- [P1] N. Cassavia, L. Caviglione, M. Guarascio, A. Liguori, M. Zuppelli, “Ensembling Sparse Autoencoders for Network Covert Channel Detection in IoT Ecosystems”, in Proceedings of the 26th International Symposium on Intelligent Systems (ISMIS), Foundations of Intelligent Systems, in M. Ceci, S. Flesca, E. Masciari, G. Manco, Z.W. Raś (Eds.), Lecture Notes in Computer Science, Vol. 13515, pp. 209-218, Cosenza, Italy, October 2022.

Abstract. Network covert channels are becoming exploited by a wide-range of threats to avoid detection. Such offensive schemes are expected to be also used against IoT deployments, for instance to exfiltrate data or to covertly orchestrate botnets composed of simple devices. Therefore, we illustrate a solution based on Deep Learning for the detection of covert channels targeting the TTL field of IPv4 datagrams. To this aim, we take advantage of an Autoencoder ensemble to reveal anomalous traffic behaviors. An experimentation on realistic traffic traces demonstrates the effectiveness of our approach.

- [P2] N. Cassavia, L. Caviglione, M. Guarascio, G. Manco, M. Zuppelli, “Detection of Steganographic Threats Targeting Digital Images in Heterogeneous Ecosystems Through Machine Learning”, Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, Vol. 13, No. 3, pp. 50-67, September 2022.

Abstract. Steganography is increasingly exploited by malware to avoid detection and to implement different advanced offensive schemes. An attack paradigm expected to become widely used in the near future concerns cloaking data in innocent-looking pictures, which are normally used by several devices and applications, for instance to enhance the user experience. Therefore, with the increasing popularity of application stores, availability of cross-platform services, and the adoption of various devices for entertainment and business duties, the chances for hiding payloads in digital pictures multiply in an almost unbounded manner. To face such a new challenge, this paper presents an ecosystem exploiting a classifier based on Deep Neural Networks to reveal the presence of images embedding

malicious assets. Collected results indicated the effectiveness of the approach to detect malicious contents, even in the presence of an attacker trying to elude our framework via basic obfuscation techniques (i.e., zip compression) or the use of alternative encoding schemes (i.e., Base64). Specifically, the achieved accuracy is always $\sim 100\%$ with minor decays in terms of precision and recall caused by the presence of additional information caused by compression.

- [P3] M. Guarascio, M. Zuppelli, N. Cassavia, L. Caviglione, G. Manco, “Revealing MageCart-like Threats in Favicons via Artificial Intelligence”, in Proceedings of the 6th International Workshop on Criminal Use of Information Hiding (CUING), in conjunction of the 17th International Conference on Availability, Reliability and Security (ARES), Vienna, Austria, August 2022.

Abstract. Modern malware increasingly takes advantage of information hiding to avoid detection, spread infections, and obfuscate code. A major offensive strategy exploits steganography to conceal scripts or URLs, which can be used to steal credentials or retrieve additional payloads. A recent example is the attack campaign against the Magento e-commerce platform, where a web skimmer has been cloaked in favicons to steal payment information of users. In this paper, we propose an approach based on deep learning for detecting threats using least significant bit steganography to conceal malicious PHP scripts and URLs in favicons. Experimental results, conducted on a realistic dataset with both legitimate and compromised images, demonstrated the effectiveness of our solution. Specifically, our model detects $\sim 100\%$ of the compromised favicons when examples of various malicious payloads are provided in the learning phase. Instead, it achieves an overall accuracy of $\sim 90\%$ when in the presence of new payloads or alternative encoding schemes.

- [P4] M. Guarascio, M. Zuppelli, N. Cassavia, G. Manco, L. Caviglione, “Detection of Network Covert Channels in IoT Ecosystems Using Machine Learning”, in Proceedings of The Italian Conference on CyberSecurity (ITASEC), in C. Demetrescu, A. Mei (Eds.), CEUR Workshop Proceedings, Vol. 3260, pp. 102-113, Rome, Italy, June 2022.

Abstract. Steganographic techniques and especially covert channels are becoming prime mechanisms exploited by a wide-range of malware to avoid detection and to bypass network security tools. With the ubiquitous diffusion of IoT nodes, such offensive schemes are expected to be used to exfiltrate data or to covertly orchestrate botnets composed of resource-constrained nodes (e.g., as it happens in Mirai). Therefore, in this paper, we present a machine learning technique for the detection of network covert channels targeting the TTL field of IPv4 datagrams. Specifically, we propose to use Autoencoders to reveal anomalous traffic behaviors. The experimental evaluation performed over realistic traffic traces showcases the effectiveness of our approach.

- [P5] M. Zuppelli, M. Repetto, A. Schaffhauser, W. Mazurczyk, L. Caviglione, “Code Layering

for the Detection of Network Covert Channels in Agentless Systems”, IEEE Transactions on Network and Service Management, pp. 1-13, May 2022.

Abstract. The growing interest in agentless and serverless environments for the implementation of virtual/container network functions makes monitoring and inspection of network services challenging tasks. A major requirement concerns the agility of deploying security agents at runtime, especially to effectively address emerging and advanced attack patterns. This work investigates a framework leveraging the extended Berkeley Packet Filter to create ad-hoc security layers in virtualized architectures without the need of embedding additional agents. To prove the effectiveness of the approach, we focus on the detection of network covert channels, i.e., hidden/parasitic network conversations difficult to spot with legacy mechanisms. Experimental results demonstrate that different types of covert channels can be revealed with a good accuracy while using limited resources compared to existing cybersecurity tools (i.e., Zeek and libpcap).

- [P6] C. Heinz, M. Zuppelli, L. Caviglione, “Covert Channels in Transport Layer Security: Performance and Security Assessment”, Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, Vol. 12, No. 4, pp. 22-36, December 2021.

Abstract. The ability of creating covert channels within network traffic is now largely exploited by malware to elude detection, remain unnoticed while exfiltrating data or coordinating an attack. As a consequence, designing a network covert channel or anticipating its exploitation are prime goals to fully understand the security of modern network and computing environments. Due to its ubiquitous availability and large diffusion, Transport Layer Security (TLS) traffic may quickly become the target of malware or attackers wanting to establish a hidden communication path through the Internet. Therefore, this paper investigates mechanisms that can be used to create covert channels within TLS conversations. Experimental results also demonstrated the inability of de-facto standard network security tools to spot TLS-based covert channels out of the box.

- [P7] M. Zuppelli, A. Carrega, M. Repetto, “An Effective and Efficient Approach to Improve Visibility Over Network Communications”, Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, Vol. 12, No. 4, pp. 89-108, December 2021.

Abstract. Modern applications and services increasingly leverage network infrastructures, cyber-physical systems and distributed computing paradigms to offer unprecedented pervasive and immersive experience to users. Unfortunately, the massive usage of virtualization models, the mix of public and private infrastructures, and the large adoption of service-oriented architectures make the deployment and operation of traditional cyber-security appliances difficult. Although cyber-security architectures are already migrating towards distributed models and smarter detectors to account for ever-evolving forms of malware and attacks, they still miss effective and efficient mechanisms to programmatically inspect these new environments. In this paper, we investigate the use of the extended Berkeley

Packet Filter for inspecting network communications. We show how this framework can be employed to selectively gather various information describing a network conversation (e.g., packet headers), in order to spot emerging threats like malicious software taking advantage of hidden communications. Results indicate that our approach can be used to inspect network traffic in a more efficient way compared to other traditional mechanisms.

- [P8] M. Zuppelli, G. Manco, L. Caviglione, M. Guarascio, “Sanitization of Images Containing Stegomalware via Machine Learning Approaches”, in Proceedings of The Italian Conference on CyberSecurity (ITASEC), CEUR Workshop Proceedings, in A. Armando, M. Colajanni (Eds.), Vol. 2940, pp. 374-386, online, April 2021.

Abstract. In recent years, steganographic techniques have become increasingly exploited by malware to avoid detection and remain unnoticed for long periods. Among the various approaches observed in real attacks, a popular one exploits embedding malicious information within innocent-looking pictures. In this paper, we present a machine learning technique for sanitizing images containing malicious data injected via the Invoke-PSImage method. Specifically, we propose to use a deep neural network realized through a residual convolutional autoencoder to disrupt the malicious information hidden within an image without altering its visual quality. The experimental evaluation proves the effectiveness of our approach on a dataset of images injected with PowerShell scripts. Our tool removes the injected artifacts and inhibits the reconstruction of the scripts, partially recovering the initial image quality.

- [P9] L. Caviglione, W. Mazurczyk, M. Repetto, A. Schaffhauser, M. Zuppelli, “Kernel-level Tracing for Detecting Stegomalware and Covert Channels”, Special Issue on Novel CyberSecurity Paradigms for Software-defined and Virtualized Systems, Computer Networks, Vol. 191, pp. 1-12, May 2020.

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.

Appendix C

Publications Made During the PhD

C.1 International Journals

- [IJ1] N. Cassavia, L. Caviglione, M. Guarascio, G. Manco, M. Zuppelli, “Detection of Steganographic Threats Targeting Digital Images in Heterogeneous Ecosystems Through Machine Learning”, *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, Vol. 13, No. 3, pp. 50-67, September 2022 (doi:10.22667/JOWUA.2022.09.30.050).
- [IJ2] A. Schaffhauser, W. Mazurczyk, L. Caviglione, M. Zuppelli, J. Hernandez-Castro, “Efficient Detection and Recovery of Malicious PowerShell Scripts Embedded into Digital Images”, *Security and Communication Networks*, Vol. 2022, pp. 1-12, June 2022 (doi:10.1155/2022/4477317).
- [IJ3] M. Zuppelli, M. Repetto, A. Schaffhauser, W. Mazurczyk, L. Caviglione, “Code Layering for the Detection of Network Covert Channels in Agentless Systems”, *IEEE Transactions on Network and Service Management*, pp. 1-13, May 2022 (doi:10.1109/TNSM.2022.3176752).
- [IJ4] L. Caviglione, A. Schaffhauser, M. Zuppelli, M. Mazurczyk, “IPv6CC: IPv6 Covert Channels for Testing Networks Against Stegomalware and Data Exfiltration”, *SoftwareX*, Vol. 17, pp. 1-7, January 2022 (doi:10.1016/j.softx.2022.100975).
- [IJ5] C. Heinz, M. Zuppelli, L. Caviglione, “Covert Channels in Transport Layer Security: Performance and Security Assessment”, *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, Vol. 12, No. 4, pp. 22-36, December 2021 (doi:10.22667/JOWUA.2021.12.31.022).

- [IJ6] M. Zuppelli, A. Carrega, M. Repetto, “An Effective and Efficient Approach to Improve Visibility Over Network Communications”, *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, Vol. 12, No. 4, pp. 89-108, December 2021 (doi:10.22667/JOWUA.2021.12.31.089).
- [IJ7] L. Caviglione, W. Mazurczyk, M. Repetto, A. Schaffhauser, M. Zuppelli, “Kernel-level Tracing for Detecting Stegomalware and Covert Channels”, *Special Issue on Novel Cyber-Security Paradigms for Software-defined and Virtualized Systems, Computer Networks*, Vol. 191, pp. 1-12, May 2020 (doi:10.1016/j.comnet.2021.108010).

C.2 International Conferences

- [IC1] N. Cassavia, L. Caviglione, M. Guarascio, A. Liguori, M. Zuppelli, “Ensembling Sparse Autoencoders for Network Covert Channel Detection in IoT Ecosystems”, in *Proceedings of the 26th International Symposium on Intelligent Systems (ISMIS), Foundations of Intelligent Systems*, in M. Ceci, S. Flesca, E. Masciari, G. Manco, Z.W. Raś (Eds.), *Lecture Notes in Computer Science*, Vol. 13515, pp. 209-218, Cosenza, Italy, October 2022 (doi:10.1007/978-3-031-16564-1_20).
- [IC2] M. Guarascio, M. Zuppelli, N. Cassavia, L. Caviglione, G. Manco, “Revealing MageCart-like Threats in Favicons via Artificial Intelligence”, in *Proceedings of the 6th International Workshop on Criminal Use of Information Hiding (CUING), in conjunction of the 17th International Conference on Availability, Reliability and Security (ARES)*, Vienna, Austria, August 2022 (doi:10.1145/3538969.3544437).
- [IC3] L. Caviglione, M. Grabowski, A. Marzecki, M. Zuppelli, A. Schaffhauser, W. Mazurczyk, “Detection of Malicious Images in Production-Quality Scenarios with the SIMARGL Toolkit”, in *Proceedings of the 6th International Workshop on Criminal Use of Information Hiding (CUING), in conjunction of the 17th International Conference on Availability, Reliability and Security (ARES)*, Vienna, Austria, August 2022 (doi:10.1145/3538969.3544469).
- [IC4] M. Guarascio, M. Zuppelli, N. Cassavia, G. Manco, L. Caviglione, “Detection of Network Covert Channels in IoT Ecosystems Using Machine Learning”, in *Proceedings of The Italian Conference on CyberSecurity (ITASEC)*, in C. Demetrescu, A. Mei (Eds.), *CEUR Workshop Proceedings*, Vol. 3260, pp. 102-113, Rome, Italy, June 2022 (link:<https://ceur-ws.org/Vol-3260/paper7.pdf>).
- [IC5] M. Zuppelli, L. Caviglione, “pcapStego: A Tool for Generating Traffic Traces for Experimenting with Network Covert Channels”, in *Proceedings of the 3rd International Workshop on Information Security Methodology and Replication Studies (IWSMR)*, in conjunction

- of the 16th International Conference on Availability, Reliability and Security (ARES), online, pp. 1-8, August 2021 (doi:10.1145/3465481.3470067).
- [IC6] M. Repetto, L. Caviglione, M. Zuppelli, “bccstego: A Framework for Investigating Network Covert Channels”, in Proceedings of the 3rd International Workshop on Criminal Use of Information Hiding (CUING), in conjunction of the 16th International Conference on Availability, Reliability and Security (ARES), online, pp. 1-7, August 2021 (doi:10.1145/3465481.3470028).
- [IC7] L. Caviglione, M. Zuppelli, W. Mazurczyk, A. Schaffhauser, M. Repetto, “Code Augmentation for Detecting Covert Channels Targeting the IPv6 Flow Label”, in Proceedings of the 3rd International Workshop on Cyber-Security Threats, Trust and Privacy management in Software-defined and Virtualized Infrastructures (SecSoft), in conjunction of the 7th IEEE Conference on Network Softwarization (NetSoft), Tokyo, Japan, pp. 450-456, July 2021 (doi:10.1109/NetSoft51509.2021.9492661).
- [IC8] M. Zuppelli, L. Caviglione, M. Repetto, “Detecting Covert Channels Through Code Augmentation”, in Proceedings of The Italian Conference on CyberSecurity (ITASEC), online, in A. Armando, M. Colajanni (Eds.), CEUR Workshop Proceedings, Vol. 2940, pp. 133-144, April 2021 (link:<http://ceur-ws.org/Vol-2940/paper12.pdf>).
- [IC9] M. Zuppelli, G. Manco, L. Caviglione, M. Guarascio, “Sanitization of Images Containing Stegomalware via Machine Learning Approaches”, in Proceedings of The Italian Conference on CyberSecurity (ITASEC), online, in A. Armando, M. Colajanni (Eds.), CEUR Workshop Proceedings, Vol. 2940, pp. 374-386, April 2021 (link:<http://ceur-ws.org/Vol-2940/paper31.pdf>).
- [IC10] 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), in conjunction of the 6th IEEE Conference on Network Softwarization (NetSoft), Ghent, Belgium, pp. 422-429, June 2020 (doi:10.1109/NetSoft48620.2020.9165537).

C.3 Workshops and Posters

- [W1] M. Zuppelli, “Are We Protected Against Network Covert Channels?”, in Proceedings of the 3rd Computer Science Workshop (CWS), University of Genova, Department of Informatics, Bioengineering, Robotics and Systems Engineering (DIBRIS), Genoa, Italy, June 2022 (link:<https://docs-dibris.github.io/assets/theme/posters/Zuppelli.pdf>).

- [W2] S. Biasotti, N. Cassavia, L. Caviglione, G. Folino, M. Guarascio, G. Manco, F. S. Pisani, M. Zuppelli, “Strumenti Intelligenti per Threat Detection and Response”, in Proceedings of the workshop “AI per Cybersecurity”, in conjunction of the 2nd Convegno Nazionale CINI sull’Intelligenza Artificiale (Ital-AI22), online, January 2022.
- [W3] M. Zuppelli, L. Caviglione, C. Pizzi, M. Repetto, “An Efficient and Privacy-Aware Method for Revealing Network Covert Channels”, Edited Book of the 2021 GARR Conference, September 2021.