Code Layering for the Detection of Network Covert Channels in Agentless Systems

Marco Zuppelli, Matteo Repetto, Andreas Schaffhauser, Wojciech Mazurczyk, *Senior Member, IEEE*, and Luca Caviglione

***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).**

***Index Terms*—Code layering, network covert channels, eBPF, detection, agentless systems.**

I. Introduction

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 [2]. 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 [3]. To this aim, in this paper we explore the concept of *code layering* to instrument VNF/CNF entities with monitoring and inspection capabilities. We leverage the extended Berkeley Packet Filter (eBPF), a framework that allows the run-time injection of code in the Linux kernel. Though it was originally conceived for monitoring system performance, eBPF has been increasingly adopted to build network functions [4] and gain network insights. The framework has also been ported to Windows, and it is currently supported by Facebook, Google, Isolavent, Microsoft, and Netflix.[1]

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. Our 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 investigate the detection of *network covert channels*, i.e., parasitic communications cloaked in innocent-looking network activities [5], [6]. For instance, covert channels can be used to exfiltrate personal information, orchestrate nodes of a botnet, or implement multi-stage loading architectures to extend malware functionalities [7]. Since modern intrusion detection systems have major drawbacks when handling IPv6 traffic and seldom can detect covert channels out of the box [13], [14], 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 [15]. Unfortunately, embedding detection capabilities in resource-constrained devices is extremely challenging, therefore suggesting to address them within VNFs.

There are virtually unlimited opportunities to implement covert channels by altering protocol headers or packet timings, thus making their detection an open research question

Manuscript received 12 November 2021; revised 31 March 2022; accepted 18 May 2022. Date of publication 20 May 2022; date of current version 12 October 2022. This work has been supported by the EU Project ASTRID, Grant Agreement No 786922, and by the EU Project SIMARGL, Grant Agreement No 833042. The associate editor coordinating the review of this article and approving it for publication was C. Fung. *(Corresponding author: Matteo Repetto.)*

Marco Zuppelli, Matteo Repetto, and Luca Caviglione are with the National Research Council of Italy (CNR), Institute for Applied Mathematics and Information Technologies (IMATI), 16149 Genoa, Italy (e-mail: marco.zuppelli@ge.imati.cnr.it; matteo.repetto@ge.imati.cnr.it; luca.caviglione@ge.imati.cnr.it).

Andreas Schaffhauser is with the Chair of Parallelism & VLSI, FernUniversität in Hagen, 58097 Hagen, Germany (e-mail: andreas.schaffhauser@fernuni-hagen.de).

Wojciech Mazurczyk is with the Chair of Parallelism & VLSI, FernUniversität in Hagen, 58097 Hagen, Germany, and also with the Institute of Computer Science, Warsaw University of Technology, 00-665 Warsaw, Poland (e-mail: wojciech.mazurczyk@pw.edu.pl).

Digital Object Identifier 10.1109/TNSM.2022.3176752

[1] https://netflixtechblog.com/how-netflix-uses-ebpf-flow-logs-at-scale-for-network-insight-e3ea997dca96

[5], [7]–[10]. 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. Our framework 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 work are:

- a framework for the inspection of virtualized systems without the need of instrumenting VNF/CNF images or deploying additional sidecar containers;
- a scalable and privacy-preserving method to spot covert communications in protocol headers, with specific focus on IPv6;
- an analysis of code-layering schemes to detect timing channels via well-known techniques [12];
- 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 our investigation utilizes real traffic traces, differently from other works only focusing on theoretical analysis or data obtained in experimental setups (see, e.g., [14] and [16]).

Compared to the preliminary work [11], this paper has the following improvements: a broader scope, which also includes timing channels in addition to storage channels; extensive sensitivity and performance analyses to evaluate the detection, the resource consumption and the impact on packet processing; comparisons with de-facto standard tools for network monitoring; an architecture for monitoring services exploiting network virtualization in PaaS/serverless environments.

The rest of the paper is structured as follows. Section II showcases the reference architecture, Section III introduces the threat model and covert channels, while Section IV describes the experimental setup. Section V discusses the detection of storage covert channels, whereas Section VI considers timing channels. Section VII evaluates the performance of our approach compared to other tools and Section VIII reviews the related literature. Lastly, Section IX concludes the work.

II. 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-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, our approach exploits the eBPF technology to implement low-level inspection and tracing operations at run-time both in conventional or PaaS/serverless environments, with negligible impact on service continuity.

Figure 1 depicts the reference layered architecture of the proposed framework for monitoring and detection purposes.

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. In general, an eBPF program should be simple and with a reduced footprint, especially in terms of maximum number of instructions. Moreover, it should be "safe", e.g., it must be loop-free and not accessing memory out of bounds. In fact, for the case of inspecting network traffic, an eBPF program is triggered at the reception of each packet, thus resource-intensive behaviors could lead to hangs or scalability issues. Therefore, eBPF programs are preliminary verified and then executed via a virtual machine implemented as a part of the eBPF Runtime. Interaction with eBPF programs (including management operations and data exchange) is possible through a specific Kernel API.

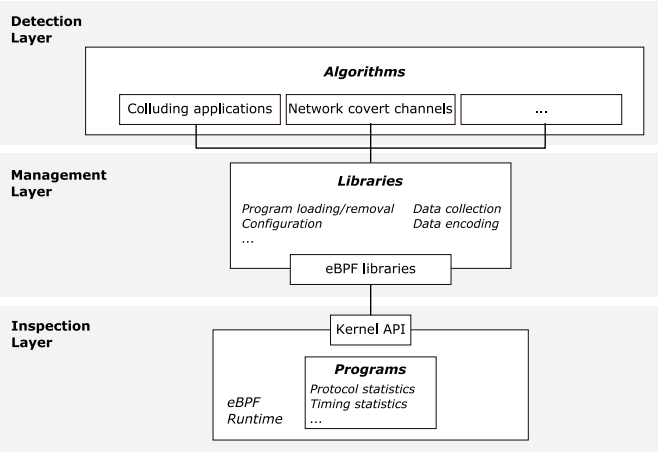


Fig. 1. Reference layered architecture for the agentless monitoring and detection of various threats.

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 Polycube.[2] 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

[2]https://polycube-network.readthedocs.io/en/latest/services/pcn-dynmon/dynmon.html.

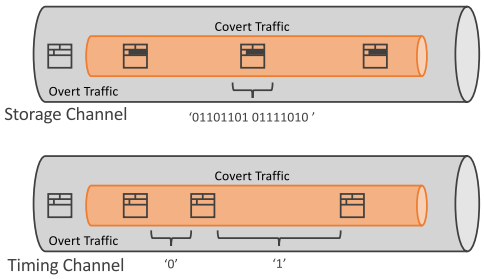


Fig. 2. Different types of network covert channels considered in this work.

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 [18], identification of processes or nodes contacting malicious servers without degrading the performance of the inspected traffic/processes [19], and support of deep packet inspection operations [3]. 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 [17].

III. Threat Model

The threat model considered in this work deals with two endpoints trying to remotely communicate in a cloaked manner. This template is usually exploited by an attacker wanting to exchange data from the host/device of the victim towards a remote Command & Control (C&C) server while avoiding detection or blockages. The attacker can inoculate a malware (e.g., via phishing) and then use the covert channel to exfiltrate sensitive information, orchestrate nodes of a botnet, implement multi-stage loading architectures to extend at runtime offensive functionalities, or bypass firewalls or filtering rules [7], [20]. To this aim, the covert sender hides information by altering an overt traffic flow and creates a cloaked communication path. By pre-sharing a hiding mechanism, the covert receiver can then extract the secret data. The overt traffic flow could be generated directly by the attacker co-located within the end node(s) or altered in a Man-in-the-Middle fashion. The process of hiding data should not disrupt the overt traffic or cause (too many) visible alterations, otherwise the hidden communication attempts would be spotted. The properties of a covert channel are usually tightly coupled. For instance, the higher the throughput of the covert communication, the higher the chance of revealing its presence due to alterations [9]. Two major classes of network covert channels exist as depicted in Figure 2 (see, [8] for a fine-grained taxonomy).

The first group consists of *storage channels*, which are created by directly hiding information in header fields, altering the structure of the packet, overwriting padding bits, or by re-arranging optional fields, just to mention the most popular techniques. Literature abounds in works exploring how to inject secret data in the TCP/IP suite [5], [8]. However, the use of IPv6 has been partially neglected and, with its increasing diffusion, it is expected to become a major target for covert communications in the future [17]. Therefore, in this work we consider the most effective storage network covert channels exploiting IPv6 traffic, especially those targeting the `Traffic Class`, `Flow Label`, and `Hop Limit` [16]. 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 values to encode '1' or '0'. In Section V 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 V-C.

The second group of covert channels consists of *timing channels*, which are created by encoding secret data through suitable alterations of the temporal evolution of network traffic. Possible encoding schemes are based upon the alteration of the throughput, introduction of statistical signatures in the jitter or the manipulation of the inter-packet time. Usually, timing channels are protocol-agnostic and mainly implemented at the network layer or by altering the error rates characterizing the data link [12], [21]. Since we are interested in covert channels with an Internet-wide scope, in Section VI 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 [5], [21]. Thus, we will resort to a known approach instead of proposing novel mechanisms.

IV. Experimental Setup

For the sake of evaluating code layering for the detection of network covert channels, we developed the reference implementation depicted in Figure 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[3] collecting data to address storage channels are based on a modified version of `bccstego`, i.e., a suite of tools able to generate filters for inspecting network and higher-level protocols like TCP/UDP [22]. Instead,

[3]https://github.com/mattereppe/bccstego

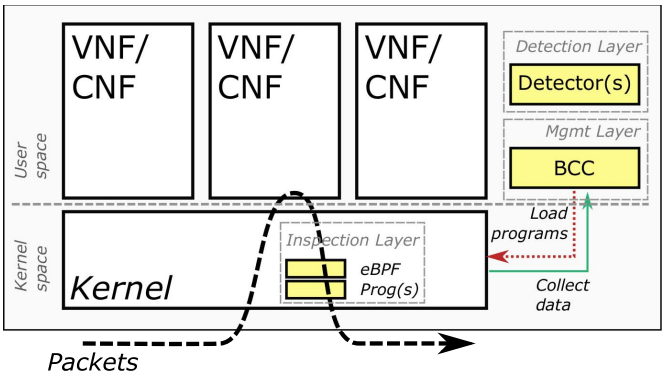


Fig. 3. Experimental setup leveraging run-time code augmentation for the detection of covert channels.

to address timing channels, we created a novel eBPF program[4] collecting time information and implementing the approach presented in [12] within the kernel;

- Management Layer: to load and unload eBPF programs as well as to collect measures, we implemented ad-hoc scripts and user-land utilities taking advantage of the BPF Compiler Collection (BCC) library.[5] This layer also includes functionalities for setting runtime parameters;
- 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 simply consider regularity metrics presented in [12]. Details on the detection methodology will be provided in Section V and Section VI, 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 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 performances. For the sake of comparison, the intermediate node has been also used to run a modified version of Zeek[6] 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` collection.[7]

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 the Center for Applied Internet Data Analysis (CAIDA).[8] 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 ∼15,000 TCP and UDP conversations. To implement storage covert channels, we directly injected various secret messages in the dumps provided by CAIDA [23]. Instead, for the case of timing channels, we used iPerf3[9] to generate ad-hoc flows. The approach in [23] 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[10] 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.

V. 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 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.

A. 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 [24], 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 intrusion detection systems.

[4]https://github.com/Ocram95/cabuk_eBPF

[5]BPF Compiler Collection. Available: https://github.com/iovisor/bcc.

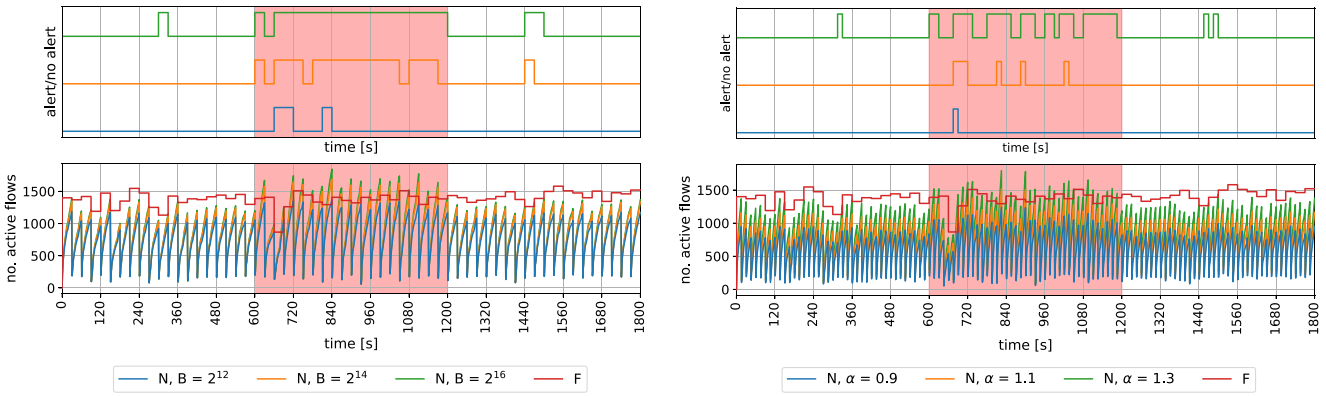
[6]The modification consists in a patch for inspecting IPv4/v6 headers. Available: https://github.com/mattereppe/zeek-stego

[7]http://sebastien.godard.pagesperso-orange.fr/index.html

[8]The CAIDA Anonymized Internet Traces Dataset (April 2008 - January 2019) - Available online: https://www.caida.org/data/monitors/passive-equinix-nyc.xml.

[9]https://iperf.fr/

[10]https://www.ntop.org/products/netflow/nprobe/



(a) $T = 30$ s, $\alpha = 0.9$ (b) $T = 15$ s, $B = 2^{14}$

Fig. 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.

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 engineered a lightweight packet inspection mechanism suitable for being implemented with eBPF. In essence, 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. The mapping is based on the first $\log_2 B$ bits of the `Flow Label`, which are used to index the array of bins. Data is then periodically collected by a user-space utility every Δt seconds and the bin-based structure is 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 4 showcases an example considering the exfiltration of 21.25 kbytes of data. In more detail, Figure 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 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 trade-off between the number of bins and the "frequency" of measures. Figure 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 V-B discusses in detail the design of the various parameters.

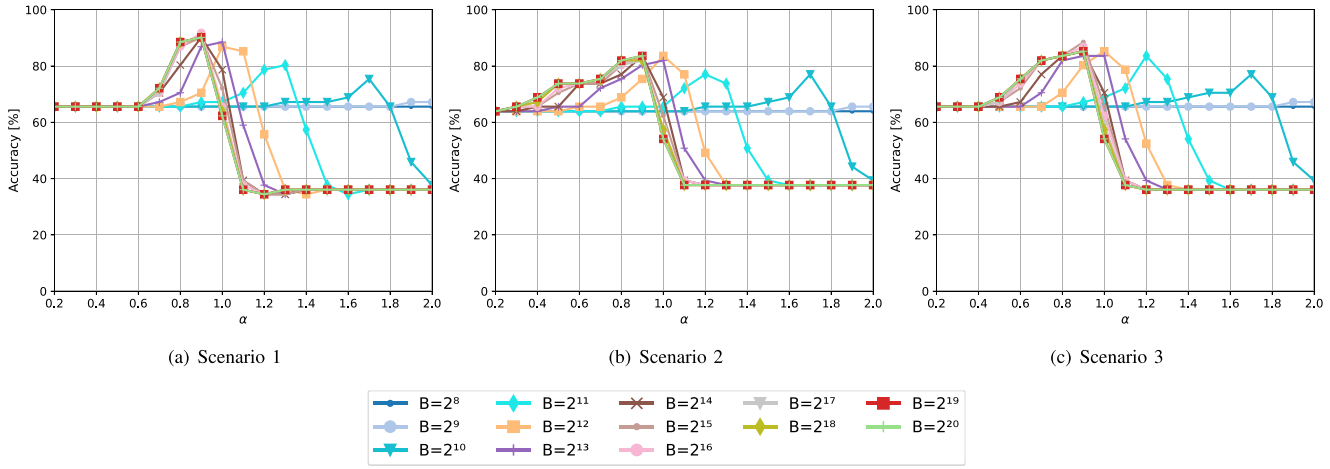
B. Sensitivity Analysis

Detecting storage covert channels is subject to many trade-offs. 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,

TABLE I
COMBINED CPU AND MEMORY USAGE FOR DIFFERENT VALUES OF B AND Δt

B		**CPU Usage** [%]					**Memory Usage** [Mbyte]				
	Δt [s]	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



(a) Scenario 1 (b) Scenario 2 (c) Scenario 3

Fig. 5. Accuracy of the bin-based detection mechanism with different values of B for all the considered scenarios.

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 [20]. 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, 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 I 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 user-land. 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., [25] for timing constraints for scalable classification). Therefore, for the sake of brevity, in the rest of the paper 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(d) provides a comprehensive overview for the impact of B on the accuracy. In general, as shown in Figure 5(a), best results are achieved for Scenario 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(b) and 5(c). Even if higher values of B typically lead to a better accuracy, the proposed approach is able to capture the presence

TABLE II
IMPACT OF B AND ξ OVER THE ACCURACY OF THE DETECTION (ACC), THE TRUE POSITIVE RATE (TPR), AND THE TRUE NEGATIVE RATE (TNR)

		Scenario 1			**Scenario 2**			**Scenario 3**		
B	ξ	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

of storage covert channels also with a reduced number of bins (see Figure 5(d) 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.

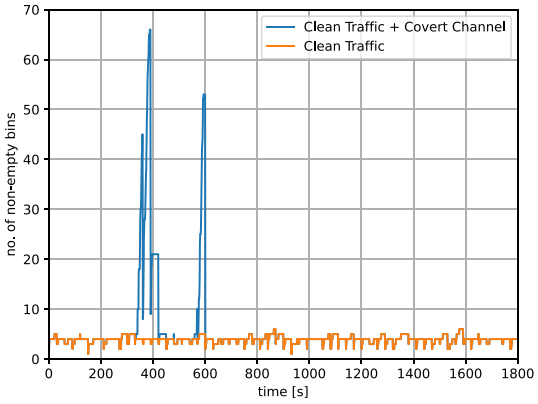
The accuracy may not be sufficient to capture the performance of the proposed approach in terms of false/true positive/negative events. Therefore, Table II reports the true positive rate (TPR) and the true negative rate (TNR) collected when using various values of B for $\alpha^* = 0.9$, i.e., the "optimal" α leading to the best performance. Moreover, as depicted in Figure 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 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.

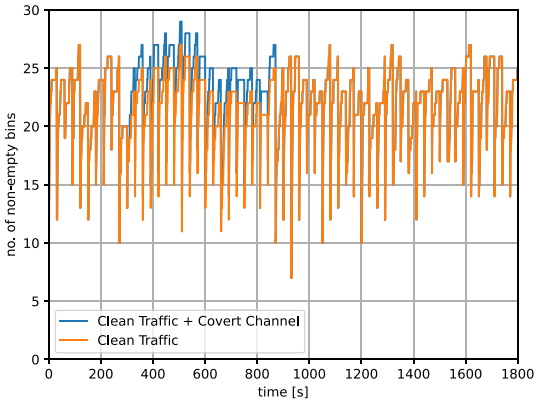
C. Channels Targeting Other IPv6 Fields

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 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 V-B. 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[11] 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.

[11]The used malicious payloads are part of the Fileless Command Lines public collection available online at: https://github.com/chenerlich/FCL/blob/master/Malwares/Silence.md.



(a) Covert channel within the `Traffic Class`.



(b) Covert channel within the `Hop Limit`.

Fig. 6. Number of "dirty" bins observed when inspecting the `Traffic Class` and `Hop Limit`.

Figure 6 depicts the collected results. Specifically, Figure 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 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.

VI. Detection of Timing 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 detection 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 [12], 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 [12] 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.

A. 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 ∼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 [12], 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 [12]. 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 ∼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 III contains the CPU utilization. In more detail, the code layering mechanism does not account for major overheads. Concerning

TABLE III
CPU USAGE FOR VARIOUS TRAFFIC RATES, W, Q, AND DIFFERENT PACKET SIZE

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

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.

VII. 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 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 forwarding.[12] 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 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 [26].

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`.

A. Impact on Packet Transmission

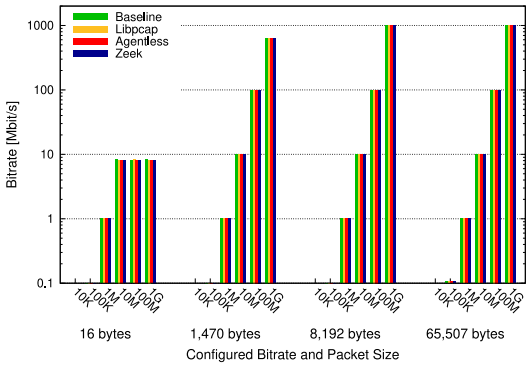
Figure 7 investigates how the inspection process behaves in the presence of different packet sizes and bitrates. In more detail, Figure 7(a) shows that the proposed method has a very limited impact on the transmission rate, for the whole range of relevant parameters. Specifically, libpcap-based tools 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 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 8 showcases the performances 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.

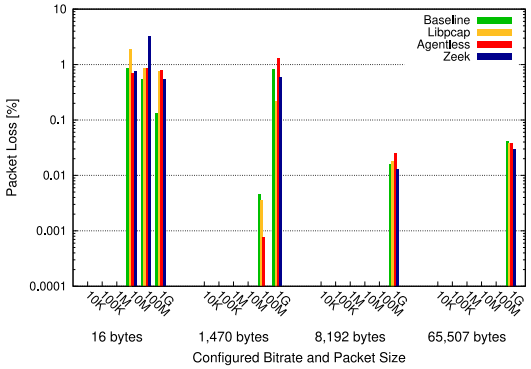
B. CPU and Memory Usage

CPU and memory utilizations are important to understand the footprint of the various frameworks used for the detection of network covert channels. Figure 9 reports a detailed

[12]The 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.



(a) Bitrate



(b) Packet loss

Fig. 7. Characteristics of the UDP flow after the inspection (the graph is in log scale).

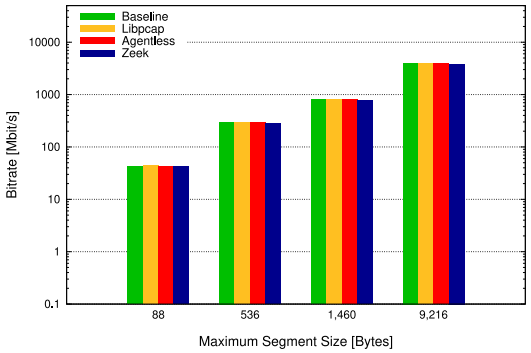


Fig. 8. Measured bitrate for a TCP flow after the inspection (the graph is in log scale).

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 our framework has a more "stable" demand. Similar considerations hold for the case of TCP as reported in Figure 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.

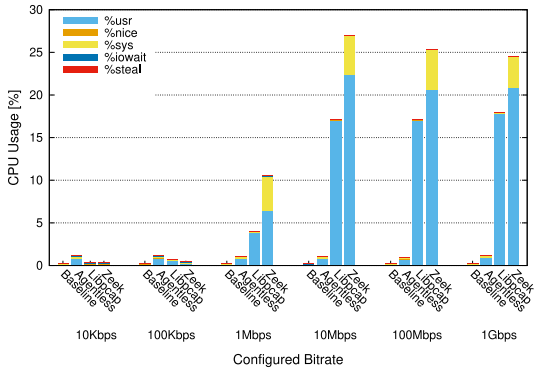
Concerning the used memory, in our trials we investigated the overall memory utilization including the Virtual Memory Size (VMS), the Resident Set Size (RSS) representing the size of physical memory including shared libraries, the Proportional Set Size (PSS) capturing the size of physical memory with proportional attribution to shared libraries, and the Anonymous utilization (Anon) containing the stack and other allocations. Figure 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 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.

VIII. Related Work

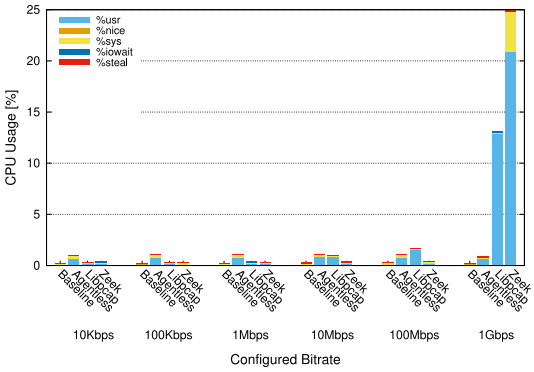
Even if eBPF is a relatively novel technology, it has been already considered for a variety of security-related tasks or to improve various software components. As an example, [19] investigates how to extend the `ntopng` network monitoring tool with events generated by the `libebpfflow`, which allows to enrich network-layer data with system metadata (e.g., source/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 eXpress Data Path, where ingress traffic can be processed before the allocation of kernel data structures, thus leading to performance benefits [27]. This paradigm can be used to provide a "first line of defense" against unwanted traffic such as flows with spoofed addresses or DoS/DDoS attacks [28].

In the context of network tracing, [29] 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 [30], the authors propose an eBPF-based implementation for monitoring the traffic exchanged between 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 [31] 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, [31] 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.

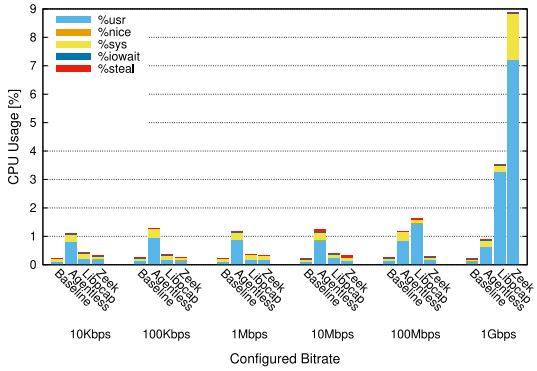
In general, detecting a network covert channel requires to develop protocol- and method-dependent metrics [5]. Therefore, a vast part of the literature focuses on specific injection mechanisms or protocols. Owing to its ubiquitous



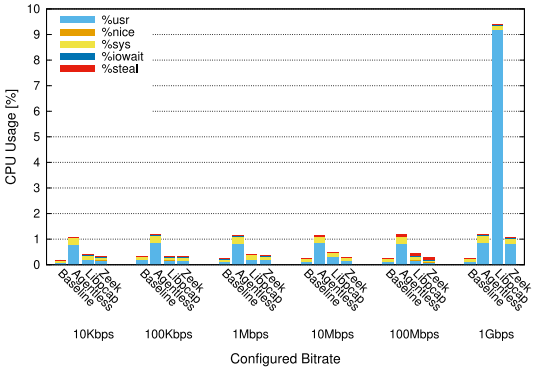
(a) Packet size = 16 bytes



(b) Packet size = 1,470 bytes



(c) Packet size = 8,192 bytes



(d) Packet size = 65,507 bytes

Fig. 9. Cumulative CPU usage of the intermediate node for various packet sizes and bitrates when inspecting a UDP flow.

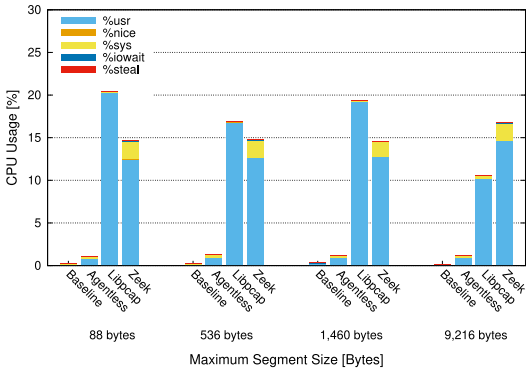


Fig. 10. Cumulative CPU usage for a TCP flow.

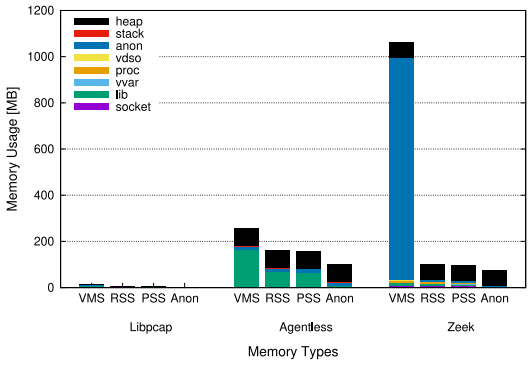


Fig. 11. Memory allocation for the different tools.

availability and multiple exploitable behaviors, many works address the problem of revealing hidden communication nested within the TCP. For instance, [32] proposes a statistical model to spot data injected in the Initial Sequence Number field used to synchronize peers. For similar reasons, 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 (see [33] and the references therein for a discussion on the real-time detection of end-to-end DNS covert channels). Even if there is no evidence of real-world attacks using Internet telephony to covertly exfiltrate data, a relevant amount of works investigated how to mitigate covert channels targeting VoIP conversations. As a possible example, [34] deals with the Session Initiation Protocol when exploited to move secret data and highlights the need of performing protocol-dependent parsing to spot the anomalous information. For the case of voice traffic, the work [35] reviews a plethora of mechanisms to tame covert communications within VoIP conversations, e.g., analyzing audio artifacts to spot data embedded in voice samples, search for anomalous traffic features to reveal encoding scheme using packet losses or manipulations of the delays, as well as deploy nodes buffering and padding traffic to disrupt parasitic information in a blind manner. Recent surveys [7], [10] highlighted the need of shifting towards more general indicators or exploiting features that can bring together different protocols, such as anomalous energy usages or signatures in the execution of processes running in end nodes.

For the specific case of detecting IPv6 covert channels, the literature already offers some previous attempts. In more detail, [36] proposes a machine-learning technique: unfortunately, this requires suitable datasets for training the detector. The work [13] deals with the leakage of hidden information through the manipulation of v4/v6 transitional mechanisms, which is definitely outside the scope of our work. Additionally, [17] and [14] provide preliminary assessment of mechanisms to detect IPv6 covert channels and also emphasize the inadequacy of standard network intrusion detection tools to handle such threat out of the box.

Lastly, the protocol-agnostic nature of timing channels leads to a more coherent and homogenous literature. Despite the wide-array of used methodologies (e.g., AI-capable framework or ad-hoc metrics), the problem of revealing hidden or parasitic conversations within timing feature has been better investigated compared to other type of channels, see [21] for a comprehensive survey on the topic.

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 storage 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.

IX. Conclusion and Future Work

In this paper, we have 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 the proposed approach to consider other channels and different threats (e.g., DDoS or cryptojacking campaigns). A part of our ongoing research concerns the utilization of eBPF for actively manipulating traffic, e.g., to sanitize flows and disrupt the channels by overwriting fields or restoring them to a standard value. To remove some limitations of the detection scheme, we are investigating the use of AI to face more sophisticated threats (e.g., modulation of values in the `Hop Limit`), also to reduce the dependance on parameters requiring a suitable design (e.g., α and ξ). 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. Obtained performance suggested the possibility to rewrite part of the framework in ANSI-C and take advantage of libbpf to prevent possible performance bottlenecks.

Acknowledgment

The authors would like to thank ntop (www.ntop.org) for providing a free license of `nProbe` used in experiments.

References

[1] "Cloud-native and verticals' services–5G-PPP projects analysis, v 1.0." 5G-PPP Softw. Netw. Working Group, Heidelberg, Germany, White Paper, Aug. 2019. [Online]. Available: https://5g-ppp.eu/wp-content/uploads/2019/09/5GPPP-Software-Network-WG-White-Paper-2019_FINAL.pdf

[2] *System Architecture for the 5G System, Version 16.6.0 Release 16*, 3GPP Standard TS 23.501, Oct. 2020. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/16.06.00_60/ts_123501v160600p.pdf

[3] M. Repetto *et al.*, "Leveraging the 5G architecture to mitigate amplification attacks," in *Proc. IEEE Int. Conf. Netw. Softwarization*, Tokyo, Japan, Feb.–Jun. 2021, pp. 443–449.

[4] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu, "A framework for eBPF-based network functions in an era of microservices," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 1, pp. 133–151, Mar. 2021.

[5] S. Zander, G. Armitage, and P. Branch, "A survey of covert channels and countermeasures in computer network protocols," *IEEE Commun. Surveys Tuts.*, vol. 9, no. 3, pp. 44–57, 3rd Quart., 2007.

[6] K. Cabaj, L. Caviglione, W. Mazurczyk, S. Wendzel, A. Woodward, and S. Zander, "The new threats of information hiding: The road ahead," *IT Prof.*, vol. 20, no. 3, pp. 31–39, May/Jun. 2018.

[7] L. Caviglione *et al.*, "Tight arms race: Overview of current malware threats and trends in their detection," *IEEE Access*, vol. 9, pp. 5371–5396, 2020.

[8] S. Wendzel, S. Zander, B. Fechner, and C. Herdin, "Pattern-based survey and categorization of network covert channel techniques," *ACM Comput. Surv.*, vol. 47, no. 3, pp. 1–26, 2015.

[9] W. Mazurczyk and L. Caviglione, "Steganography in modern smartphones and mitigation techniques," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 1, pp. 334–357, 1st Quart., 2014.

[10] L. Caviglione, "Trends and challenges in network covert channels countermeasures," *Appl. Sci.*, vol. 11, no. 4, p. 1641, 2021.

[11] L. Caviglione, M. Zuppelli, W. Mazurczyk, A. Schaffhauser, and M. Repetto, "Code augmentation for detecting covert channels targeting the IPv6 flow label," in *Proc. IEEE 7th Int. Conf. Netw. Softw. (NetSoft)*, 2021, pp. 450–456.

[12] S. Cabuk, C. E. Brodley, and C. Shields, "IP covert timing channels: Design and detection," in *Proc. 11th ACM Conf. Comput. Commun. Security*, 2004, pp. 178–187.

[13] B. Blumbergs, M. Pihelgas, M. Kont, O. Maennel, and R. Vaarandi, "Creating and detecting IPv6 transition mechanism-based information exfiltration covert channels," in *Proc. Nordic Conf. Secure IT Syst.*, 2016, pp. 85–100.

[14] W. Mazurczyk, K. Powójski, and L. Caviglione, "IPv6 covert channels in the wild," in *Proc. 3rd Central Eur. Cybersecurity Conf.*, 2019, pp. 1–6.

[15] T. Koziak, K. Wasielewska, and A. Janicki, "How to make an intrusion detection systemaware of steganographic transmission," in *Proc. Eur. Interdiscipl. Cybersecurity Conf.*, 2021, pp. 77–82.

[16] N. B. Lucena, G. Lewandowski, and S. J. Chapin, "Covert channels in IPv6," in *Proc. Int. Workshop Privacy Enhancing Technol.*, 2005, pp. 147–166.

[17] L. Caviglione, W. Mazurczyk, M. Repetto, A. Schaffhauser, and M. Zuppelli, "Kernel-level tracing for detecting stegomalware and covert channels in Linux environments," *Comput. Netw.*, vol. 191, May 2021, Art. no. 108010.

[18] M. Bachl, J. Fabini, and T. Zseby, "A flow-based IDS using machine learning in eBPF," 2021, *arXiv:2102.09980*.

[19] L. Deri, S. Sabella, S. Mainardi, P. Degano, and R. Zunino, "Combining system visibility and security using eBPF," in *Proc. ITASEC*, 2019, pp. 1–12.
[20] W. Mazurczyk and L. Caviglione, "Information hiding as a challenge for malware detection," *IEEE Security Privacy*, vol. 13, no. 2, pp. 89–93, Mar./Apr. 2015.
[21] A. K. Biswas, D. Ghosal, and S. Nagaraja, "A survey of timing channels and countermeasures," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 1–39, 2017.
[22] M. Repetto, L. Caviglione, and M. Zuppelli, "bccstego: A framework for investigating network covert channels," in *Proc. 16th Int. Conf. Availability Rel. Security*, 2021, pp. 1–7.
[23] M. Zuppelli and L. Caviglione, "pcapStego: A tool for generating traffic traces for experimenting with network covert channels," in *Proc. 16th Int. Conf. Availability Rel. Security*, 2021, pp. 1–8.
[24] S. Amante, B. Carpenter, S. Jiang, and J. Rajahalme, "IPv6 flow label specification," IETF, RFC 6437, Nov. 2011.
[25] S. Zander, T. Nguyen, and G. Armitage, "Sub-flow packet sampling for scalable ML classification of interactive traffic," in *Proc. 37th Annu. IEEE Conf. Local Comput. Netw.*, 2012, pp. 68–75.
[26] K. Mansley, G. Law, D. Riddoch, G. Barzini, N. Turton, and S. Pope, "Getting 10 gb/s from Xen: Safe and fast device access from unprivileged domains," in *Proc. Eur. Conf. Parallel Process.*, 2007, pp. 224–233.
[27] T. Høiland-Jørgensen *et al.*, "The eXpress data path: Fast programmable packet processing in the operating system kernel," in *Proc. 14th Int. Conf. Emerg. Netw. EXp. Technol. (CoNEXT)*, Heraklion, Greece, 2018, pp. 54–66.
[28] G. Bertin, "XDP in practice: Integrating XDP into our DDoS mitigation pipeline," in *Proc. Netdev 2.1 Techn. Conf. Linux Netw.*, Montreal, QC, Canada, 2017, pp. 1–5. [Online]. Available: https://netdevconf.info/2.1/papers/Gilberto_Bertin_XDP_in_practice.pdf
[29] K. Suo, Y. Zhao, W. Chen, and J. Rao, "vNetTracer: Efficient and programmable packet tracing in virtualized networks," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Vienna, Austria, 2018, pp. 165–175.
[30] J. Hong, S. Jeong, J.-H. Yoo, and J. W.-K. Hong, "Design and implementation of eBPF-based virtual TAP for inter-VM traffic monitoring," in *Proc. 14th Int. Conf. Netw. Service Manage. (CNSM)*, Rome, Italy, 2018, pp. 402–407.
[31] C. Cassagnes, L. Trestioreanu, C. Joly, and R. State, "The rise of eBPF for non-intrusive performance monitoring," in *Proc. IEEE/IFIP Netw. Oper. Manag. Symp. (NOMS)*, Budapest, Hungary, 2020, pp. 1–7.
[32] H. Zhao and Y.-Q. Shi, "Detecting covert channels in computer networks based on chaos theory," *IEEE Trans. Inf. Forensics Security*, vol. 8, pp. 273–282, 2013.
[33] S. Chen, B. Lang, H. Liu, D. Li, and C. Gao, "DNS covert channel detection method using the LSTM model," *Comput. Security*, vol. 104, May 2021, Art. no. 102095.
[34] W. Mazurczyk and K. Szczypiorski, "Covert channels in SIP for VoIP signalling," in *Proc. Int. Conf. Global e-Security*, 2008, pp. 65–72.
[35] W. Mazurczyk, "VoIP steganography and its detection—A survey," *ACM Comput. Surv.*, vol. 46, no. 2, pp. 1–21, 2013.
[36] A. Salih, X. Ma, and E. Peytchev, "Implementation of hybrid artificial intelligence technique to detect covert channels attack in new generation Internet protocol IPv6," in *Leadership, Innovation and Entrepreneurship as Driving Forces of the Global Economy*. Cham, Switzerland: Springer, 2017, pp. 173–190.

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

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

Andreas Schaffhauser is currently pursuing the Ph.D. degree with FernUniversität in Hagen, Germany. His research interests include information and network security.

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

Luca Caviglione received the Ph.D. degree in electronic and computer engineering from the University of Genoa, Italy. He is a Senior Research Scientist with the Institute for Applied Mathematics and Information Technologies, National Research Council of Italy. He is an author or coauthored more than 150 academic publications, and several patents in the field of p2p and energy-aware computing. He has been involved in Research Projects and Network of Excellences funded by the ESA, the EU and the MIUR. From 2019 to 2022, he was the Principal Investigator for IMATI of the Project "Secure Intelligent Methods for Advanced Recognition of Malware and Stegomalware" (SIMARGL - call H2020-SU-ICT-2018, grant no. 833042). His research interests include optimization of large-scale computing frameworks, traffic analysis, and network security. He is a Work Group Leader of the Italian IPv6 Task Force, a Contract Professor in the field of networking/security and a Professional Engineer. Since 2016, he has been an Associate Editor of the *International Journal of Computers and Applications* (Taylor & Francis). He is the Head of the IMATI Research Unit of the National Inter-University Consortium for Telecommunications and part of the Steering Committee of the Criminal Use of Information Hiding initiative.